Formal proof of dynamic memory isolation based on MMU

Narjes Jomaa, David Nowak, Gilles Grimaud, Samuel Hym

Formal Proof of Dynamic Memory Isolation Based on MMU[☆,☆☆]

Narjes Jomaa, David Nowak, Gilles Grimaud, Samuel Hym

CRIStAL, CNRS & Lille 1 University, France

Abstract

For security and safety reasons, it is essential to ensure memory isolation between processes. The memory manager is thus a critical part of the kernel of an operating system. It is common for kernels to ensure memory isolation through a piece of hardware called memory management unit (MMU). However an MMU by itself does not provide memory isolation. It is only a tool the kernel can use to ensure this property. In this paper we show how a proof assistant such as Coq can be used to model a hardware architecture with an MMU, and an abstract model of microkernel supporting preemptive scheduling and memory management. We proceed by making formally explicit the consistency properties that must be preserved in order for memory isolation to be preserved.

Keywords: Formal proof, Memory isolation, Microkernel, Coq.

1. Introduction

Modern operating-system kernels make it possible to share computer resources between untrusted processes, and to rapidly deal with external events, e.g., arrival of a network packet that would be lost if not dealt with immediately. In this context, both for safety and security reasons, it is important to respectively prevent accidental and malevolent access by a process to an address outside its own address space. On modern computers, kernels ensure memory isolation with the help of a piece of hardware called memory management unit (MMU). An MMU is a hardware component through which all memory accesses must go. It translates a virtual memory address to a physical address if there is indeed a corresponding one in the current setting. It also checks whether in the current setting accessing this address is allowed. It is indeed a common design to have the kernel space always mapped for efficiency reasons but not accessible while in user mode. For this to work properly, the kernel has to maintain page

[☆]This work was partially supported by the Celtic-Plus Project ODSI C2014/2-12, CNRS Action Spécifique Sécurité, and IRCICA USR 3380.

[☆☆]A preliminary version of this work appeared in the proceedings of the 10th International Symposium on Theoretical Aspects of Software Engineering (TASE 2016) [1].

tables which encode for each process the mapping between virtual addresses and physical addresses, and the access rights. It is important to note here that an MMU does not ensure memory isolation by itself, but it is only a tool the kernel can use to ensure the isolation property. A bug in the code of the kernel that deals with memory management (i.e. the memory manager) may lead to serious security and safety issues.

Since a kernel is executed in the so-called kernel mode (i.e. the privileged mode of the hardware), it is better from a security point of view to keep it as small as possible. This stems from the general principle that the trusted computing base (TCB) should be kept minimal. This is the reason why in this paper we focus on an abstract model of a microkernel [2] which supports preemptive scheduling and ensures memory isolation.

Contributions

Our main contribution is a formally proved model in the Coq proof assistant of dynamic memory isolation based on the MMU. More precisely, it consists of:

- A formal model of a hardware architecture as a monad: the parts that are important for memory isolation (e.g., the MMU and CPU) are modeled in all their relevant minutiae, while less relevant parts are abstracted away.
- A formal model of a microkernel supporting:
  - a memory management at an appropriate abstraction level so that it remains a realistic model without being linked to a particular implementation,
  - the basic principles of interrupts, in particular to support a preemptive scheduler.
- An explicit description of the consistency properties that must be preserved by a microkernel dealing with an MMU in order for the memory isolation to be preserved.

Related work

There have been many efforts to make formal proofs of security for kernels. Here are the most closely related to ours.

One of the most significant is the formal proof in the Isabelle/HOL proof assistant of the memory protection model of the microkernel seL4 [3]. It is proved that the assembly code emitted by the compiler is correct in the sense that it implements the abstract model of memory isolation [4]. This work was the first complete formal proof of an operating system kernel.

There is also CertiKOS which is a hypervisor dedicated to cloud computing that is formally verified [5, 6]. In particular, its memory manager BabyVMM is constructed in layers so as to allow for formal verification by a series of refinements that are formalized in the Coq proof assistant [7].

In contrast to those related work above, our goal is not to formally prove properties of a specific microkernel but to clarify what is needed to be assumed

by microkernels about the hardware architecture and what are the constraints a microkernel must respect in order for memory isolation to be guaranteed at all times.

Moreover, a novel framework is developed in [8]. It gives the possibility to extend a verified non-interruptible kernel to a verified interruptible kernel where device drivers are implemented inside the kernel, their approach was successfully applied on the kernel mCertiKOS. In our case, device drivers are kept outside the microkernel, and the isolation between these components, like any process user, is ensured by the virtual memory manager which is implemented inside the microkernel on top of the physical memory. In the mCertiKOS kernel a different isolation property is defined and verified in order to ensure separation between different device objects and the kernel inside the kernel without relying on the virtual memory manager.

In [9], an idealized model of a hypervisor was formalized in Coq and isolation properties were proved. While we also consider an abstract model, we are not treating isolation from the point of view of the information flow but at the lower level of page table management (information access). We are thus led to a model that includes an MMU and deals with page allocation.

In [10], the operations of allocation and deallocation of a microkernel were proved correct. However, those operations live in a higher layer of the operating system than the lower-level layer we consider here. Our work shows that the correct implementation of those operations is essential to ensure memory isolation.

Guo and Zhang proposed in [11] a verification framework for verifying preemption control operations in a preemptive kernel. In our case, we focus on proving memory isolation between processes.

Outline

We first describe in Section 2 our formal model of a microkernel. We then make explicit in Section 3 our formal definition of memory isolation and the consistency properties that are to be preserved in order for memory isolation to hold. In Section 4, we present our proof methodology and discuss the difficulties met and their solutions. We finally give a brief overview of our Coq code in Section 5, before concluding in Section 6.

2. Formal model of a microkernel

In this section, we first briefly recall some basic facts on how microkernels control hardware components in order to manage CPU time and MMU resources and then we describe our formal model of the dynamic evolution of the system based on interrupts.

2.1. Background on MMU-based microkernels

The purpose of a microkernel is to manage several executing programs known as runnable processes. To provide preemptive scheduling, the microkernel must

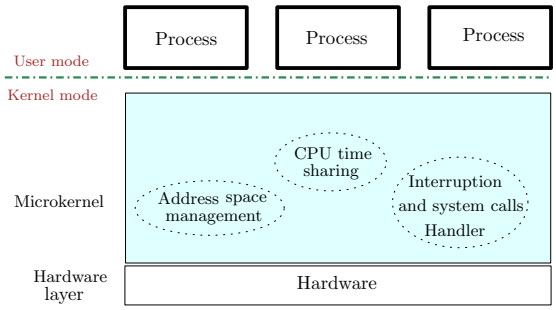

Figure 1: Architecture of a microkernel

share CPU time between them giving the illusion that the processes run simultaneously. Each runnable process should have access only to its own address space, so the microkernel should control execution to prevent illegal accesses. This architecture of a microkernel is, for instance, detailed on pages 17 and 18 in the MINIX book [12] and illustrated in Fig. 1. We have designed a model for a microkernel that includes these major mechanisms.

To ensure security (in particular memory isolation), processes cannot directly access the physical memory which is divided into fixed-length blocks called physical pages. All their accesses to memory use *virtual addresses* and go through the MMU that translates virtual addresses to physical addresses. We illustrate the internal operation of the MMU with one level of indirection (cf. Fig. 2 taken from [12]). This translation mechanism is implemented using *page tables*. A page table is a physical page which is managed by the memory manager of the kernel. Each process has an address space large enough to store its code and data. The memory manager should ensure that any physical page allocated to a given process is referenced only in its page table. Using the virtual address, *translate* starts by finding the corresponding entry in page table. It then checks whether accessing that virtual address is allowed, i.e. there is a mapped page in this entry, using the *present* bit. It also verifies whether this page is accessible or not using the *kernel_only* bit and the execution mode. When a process tries to violate these protection rules, the MMU raises an exception which will be handled by the microkernel.

Finally, in order to perform some operations that do require a higher level of privileges (read data from a file, get access to more physical memory, etc.), a process may request the microkernel to perform it on its behalf. To that end, the microkernel provides a set of *system calls*.

Since these mechanisms are clearly crucial to ensure memory isolation among processes, the memory manager and the scheduling mechanism used by the microkernel must be verified.

2.2. State of the system

In real implementations of operating systems, the state of the system is complex and includes the internal state of each hardware device and all the

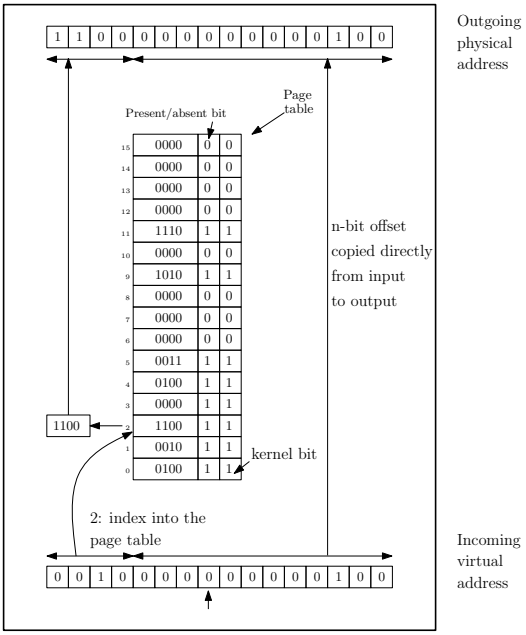

Figure 2: Memory management unit (MMU)

kernel data structures. In our formal model, the state s is a tuple containing many components; it is divided into two parts, the hardware state and the software state. For each part, we focus only on the components which are relevant to prove the properties we are interested in. The actual definitions of the *state* and its components are given in Appendix A.1 in Gallina syntax for reference. The formal definition of the type of the state is a record containing the following fields.

Hardware state. The following information about hardware devices (mostly processor and memory) is necessary to reason about memory isolation.

- *currentptp*(s) is the number of the physical page containing the page table of the current process.
- *kernel_mode*(s) is a boolean that will be true when the processor is executing in kernel mode and false, otherwise.
- *currentpc*(s) (current program counter) is the position of the current instruction to be executed.
- *memory*(s) is the physical memory that we model as a list; it contains in particular two crucial data structures for the memory management: the page tables of the current processes used by the MMU address space translation and the list of the free pages (which is detailed below).

- *interrupts*(s) models hardware interrupts in a simple way since modeling hardware itself is out-of-scope of this work; they are modeled as an infinite list (or *stream*) containing one element per clock *tick*; each element indicates whether a hardware interrupt is to be triggered at that tick and which interrupt it is, if any.

Software state. During execution, the system needs to store some information in physical memory and it must be accessible only in kernel mode. We modeled that information separately from the memory itself in order to simplify the proof of our property. Real implementations do ensure that it is kept separate by storing it in some part of memory which is reserved to the kernel and thus never accessible for processes. In the following, we provide some details concerning the most important fields :

- *processes*(s) is the list of runnable processes. Note that a process type is a record which contains information about a runnable process P in this list such as the reference to its page table *ptp*(P) and the address of the next instruction to execute *pc*(P). When we switch between processes the value of *currentptp*(s) should be updated with the value of *ptp*(P) of the selected process P and the value of *currentpc*(s) should be updated with the value of its *pc*(P).

- *code*(s) is a list that contains all the instructions of the system and of the user processes. The main property we are interested in here is to ensure data security. The proof of isolation for the code would be similar.

- *intr_table*(s) represents the interrupt descriptor table. It is a list associating to each interrupt number its handler code such that each position into the list corresponds to the interrupt number and the associated value represents the address (as a position) into *code*(s).

- *stack*(s) is the system stack used to store the context of the current runnable program when an interrupt occurs.

- *first_free_page*(s) is the first page of the list of free pages. A page is said *free* when it is not allocated to any process (either as its page table or as a regular page, to store its data). A memory manager is obviously required to determine which pages are available for allocation and which are not. In our model, this is done using the available pages themselves to store the linked list of free pages (cf. Fig. 3): since a free page does not contain any process data, we can use it to store the position of the next free page. That way, the microkernel needs to keep only the position of the first page of that linked list in its state *per se* to manage the free pages.

  On system startup, all available physical pages are initialized so that they are in the free-page list.

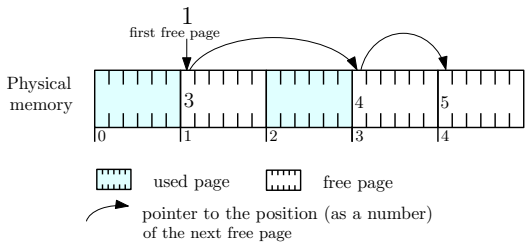

Figure 3: Memory model

2.3. Dynamic evolution of the system

During the lifetime of the system many processes may be created and executed and several events could occur generating state updates. So, the microkernel should provide an efficient mechanism to ensure correct state transitions. In this section we show how we formally model this dynamic evolution by using a monad.

H monad. Gallina, the specification language of Coq, is a purely functional language and thus does not provide imperative features such as updatable state, undefined behaviors and halting. In such a language, it is thus common to implement such features using a monad [13, 14]. For our model, we have defined a monad that we call *H monad* and that provides states (as described in Section 2.2) and support for undefined behaviors and halting.

Our H monad is a kind of state monad where $M(A)$ is the type of a *computation* that may have side effects and returns a result of type A: $M(A) =_{\text{def}} S \to \mathit{result}(A \times S)$ where S is the type of the state of the system and $\mathit{result}(X)$ is an inductive type with three constructors: one to return a result of type A and the new state of type S, and two others to denote an undefined behavior and halting. In the following, we will use s to denote a state in S. In our model, we identify three different kinds of *computations*:

- a *hardware component* models the behavior of a relevant piece of hardware; indeed, we need to model the pieces of hardware involved in the memory management (namely the operations performed by the MMU);
- an *instruction* corresponds to a single CPU instruction; it is modeled as a sequence of more elementary steps (for instance involving hardware components);
- a *subroutine* is an atomic sequence of instructions of the microkernel that cannot be interrupted.

System calls and interrupts. Our model of the CPU, in particular regarding how system calls and interrupts work, aims at being as general as possible.

Fig. 4 (taken from Stallings' book on *Computer Organization and Architecture* [15]) outlines the dynamic evolution of the system with and without interrupt and Fig. 5 represents the formal model implemented in Coq.

Our model of the kernel provides a very simple API, focused on our topic of interest: memory isolation of processes. So its API provides only the subroutines *create_process*, *switch_process*, *add_pte*, and *remove_pte* which will be detailed later in this paper. The actual definitions of the *step* hardware component in Gallina syntax is given for reference in Appendix A.2.

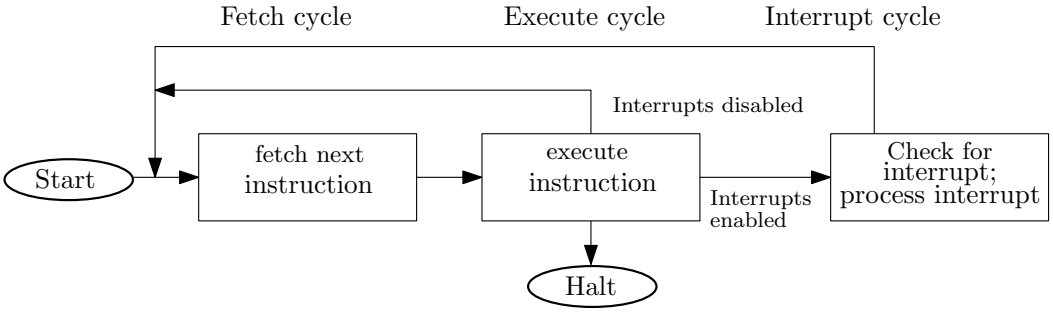

Figure 4: Dynamic evolution of the system

Hardware component step : $M(\text{unit})$
Action:
check for the presence of an interrupt ;
if *there is some interrupt* n **then**
  $interrupt(n)$;
else
  $i \leftarrow fetch\_instruction$;
  $execute(i)$;
end

Figure 5: The specification of the dynamic evolution of the system

We can see in Fig. 4 that instruction execution operates mainly in three steps:

1. Fetch instruction: read or extract the instruction from memory. Our hardware component *fetch_instruction* corresponds to this step.
2. Execute instruction: execute the sequence of elementary steps of that CPU instruction. For instance, the instruction may require the CPU to determine some effective (physical) address to load or store some data, such as when a process stores a value in memory using the instruction *write*. Before storing data in memory, using the hardware component *translate*, this instruction determines the physical address that corresponds to the virtual address provided by the process.

3. Check for interrupt: check whether there is an interrupt to handle before moving on to the next instruction. We modeled this operation by the hardware component *fetch_interrupt* (Fig. 6): it pops the head element of the hardware-interrupt stream in order to check if an interrupt has been triggered. If some interrupt needs to be dealt with, using the hardware component *interrupt* (Fig. 7), the current context is pushed on top of the system stack, does switching to kernel mode and branching to the code of the handler for the triggered interrupt. The hardware finds the right handler using the interrupt number as index in $intr\_table(s)$. To continue the execution of the current process properly after an interrupt, the hardware component *return_from_interrupt* (Fig. 8) needs to be executed to return from the interrupt handler.

Hardware component fetch_interrupt : M(option integer)
Action:
pop the head of the hardware interrupt stream;
return it;

Figure 6: Check for interrupt

Hardware component *interrupt* : integer $\rightarrow M$(unit)
Input : n: the number of the interrupt to execute
Action:
push the *currentpc* and the execution mode on the stack;
switch to kernel mode;
jump to the first instruction of the interrupt handler;

Figure 7: Handling the interrupt

Hardware component *return_from_interrupt* : M(unit)
Action:
pop the head of the stack;
restore the execution mode of the current program;
restore the *currentpc*;

Figure 8: Returning from an interrupt

Processes invoke system calls to interact with the microkernel by triggering *software* interrupts. In our model, thus, the instruction *trap* allows processes to trigger software interrupts which are then processed as explained in Fig. 7.

3. Isolation and consistency

In this section, we present our memory isolation property, then we introduce the consistency properties and motivate them with counterexamples that show how a simple breach of consistency would invalidate memory isolation.

Consistency is the conjunction of multiple properties that must be preserved in order for isolation to be preserved. Testing at runtime that these properties are preserved is not realistic since it would take too much time. Indeed, it would for instance require checking if all the entries of a set of tables match some condition on every system call. We rather characterize the consistency properties required for isolation and prove that they are always preserved.

3.1. Memory isolation

Each process has its own page table which is located in memory. Our model for page-table entries follows closely the description given in Section 2.1. Each entry of a page table corresponds to a virtual address and contains the corresponding physical-page number and some bits for access control such as the *present* bit and the *kernel_only* bit. The former should have the value *1* if there is a mapped page in this entry and the latter should have the value *0* if the mapped page should only be accessible to the microkernel. Unfortunately, the MMU cannot ensure separation between process address spaces all by itself.

Instruction write : integer → integer → M(unit)
Input : *val*: the value to be written
vaddr: the virtual address at which it should be written
Action:
paddr ← *translate*(*vaddr*) ;
if *paddr is not an exception* **then**
  write_phy(*val*, *paddr*) ; *(*write val at physical address paddr*)*
end

Figure 9: Writing a value in memory

Indeed, if the page tables are not configured correctly then the translation function could translate a virtual address to a physical address which is also used by another process. Let us consider how this could happen by looking at what happens when a process runs the *write* instruction. That instruction, illustrated in Fig. 9, writes a value *val* to some virtual address *vaddr*. In order to perform that action, it relies on the hardware component *translate* (cf. Fig. 10) to compute the physical address *paddr* for the virtual address *vaddr* according to the page table of the current process. Considering how these operations are performed, the MMU will translate addresses without raising any exception even if the physical page is also mapped in another process page table and so the current process will access and modify the content of that page. This clearly shows that the two processes are not properly isolated.

Hardware component translate : integer $\rightarrow M$(integer + exception)
Input : *vaddr*: a virtual address
Output: the corresponding physical address or an exception
Action :
if the *vaddr* size is valid **then**
  calculate the address of page table entry *pte* and the offset in this page corresponding to *vaddr*;
  if there is a mapped page in *pte* and (the *kernel_mode*(s) = *true* or *kernel_only*(*pte*) = 1) **then**
    calculate the physical address and return it;
  else
    return an exception
  end
else
  return an exception
end

Figure 10: Translating virtual addresses to physical addresses

To ensure isolation then, we need to guarantee that all page tables are always soundly configured. That is the aim of our memory isolation property (Def. 1). Intuitively, we want to show that for any state s there is no interference between any two runnable processes: if P_1 and P_2 are two runnable processes then any page which is used by P_1 is different from any page used by P_2.

Definition 1 (Memory isolation property)**.** *A state s is* isolated *iff for all $P_1, P_2 \in$ Processes(s) such that $P_1 \neq P_2$ (i.e. ptp(P_1) $\neq$ ptp(P_2)) and for all $p \in$ UsedPages(P_1), we have $p \notin$ UsedPages(P_2), where*

- *Processes(s) is the set of runnable processes in the state s,*
- *ptp(P_i) is the number of the physical page containing the page table of the process P_i,*
- *UsedPages(P_i) is the list of all the pages referenced in the page table ptp(P_i) plus ptp(P_i) itself.*

3.2. Consistency

In this section, we explain and justify the various properties composing the consistency. We organize them into two categories: software consistency and hardware consistency.

The consistency is required to prove the isolation property. It precisely capture some properties about the functional correctness of the system. The aim of our work is not to list and prove all possible consistency properties about the API and the hardware but only those which are necessary to prove that

isolation is preserved. In other words, the consistency properties give another view of what it actually means for isolation to be preserved.

Many of the consistency properties are related to the list of free pages. So we define the following notation:

Notation. *Given a state s, FreePageList(s) stands for the linked list of all physical pages available for allocation.*

That list is encoded in memory and in the microkernel state as described in Section 2.2.

3.2.1. Software consistency

All free pages are really free

The following consistency property REALLY_FREE ensures that all free pages are never mapped in any runnable-process page table.

Definition 2. *Given a state s,* REALLY_FREE(s) *holds iff for all $p \in$ FreePageList(s), $p \notin$ AllUsedPages(s) and $p <$ nb_pages, where nb_pages is the number of pages in physical memory and AllUsedPages(s) is the list of pages used by all processes in Processes(s).*

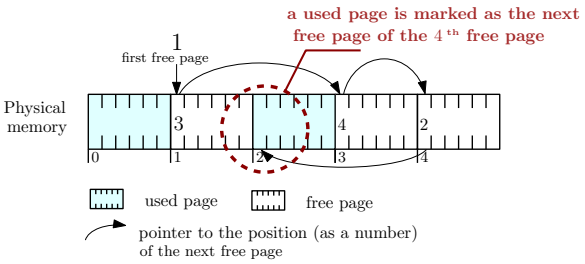

Figure 11: Counterexample for REALLY_FREE

By adding the property REALLY_FREE in consistency we must verify that on each system step, all free pages are not mapped in any runnable-process page table. For instance, without such a property, we cannot prove that the subroutine *add_pte* (which adds an entry to a page table, cf. Fig. 12) preserves isolation. Indeed, this subroutine allocates the first free page selected from *FreePageList*(s), then adds it into the process page table. If this page was already used by another process (cf. Fig. 11), the execution of *add_pte* would result in a state in which the page tables of two processes would reference the same page: consequently, the isolation property would no longer hold.

No cycle in the free-page list

The following consistency property NOT_CYCLIC means that no page appears more than once in the free-page list.

Definition 3. *Given a state s,* NOT_CYCLIC(s) *holds iff by traversing the FreePageList(s) list until reaching the end of the list, we never encounter the same node.*

Subroutine *add_pte* : integer → integer → M(unit)
Input : *permission*: access rights for the new mapped page
index: entry in the page table
Action:
if *permission* and *index* are valid **then**
  pte ← get the entry at position *index*;
  if there is a mapped page in *pte* **then**
    remove the entry content of *pte*;
  end
  allocate a new physical page p ;
  add a mapping in *pte* according to p and *permission*;
end

Figure 12: Adding an entry in page table

The search of the list is performed using an accumulator list called *seen*, by traversing the *FreePageList*(s) list until the end, we pop each encountered node into the list *seen*. If the current node is already seen (i.e. present into *seen*) that means the *FreePageList*(s) list is cyclic.

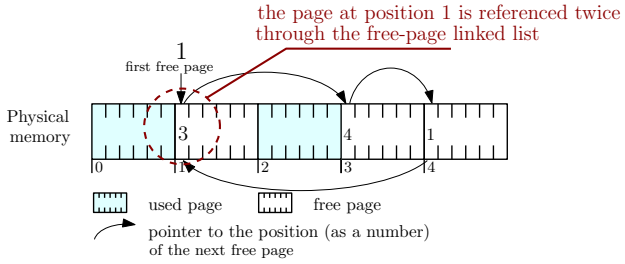

Figure 13: Counterexample for NOT_CYCLIC

As we have detailed above, free pages are referenced through a linked list in physical memory. If such a property did not hold, the subroutine *add_pte* could allocate twice the same physical page (to the same process or a different one): consequently, isolation property would no longer hold. Fig. 13 illustrates such counterexample.

No duplicate in process used pages

The following consistency property NODUPLIC_PROCESSPAGES property (Def. 4) ensures that for any process, all its used pages appear only once in its page table.

Definition 4. *The consistency property* NODUPLIC_PROCESSPAGES *holds of a state s iff for all process $P \in$ Processes(s), there is no duplicate in UsedPages(P).*

The need for this property arises when proving the isolation property of subroutine *remove_pte* (Fig. 14). This subroutine removes the content of a page-table entry and frees the page p of that entry by adding it to the free-page list. We use the *present* bit to check if the page-table entry is in use or free. After the execution of *remove_pte*, p must be really free. However, if there were another entry which mapped the same physical page p, after *remove_pte* p would be considered both used and free at the same time. Then another process might allocate p, and isolation property would no longer hold. Fig. 15 illustrates an inconsistent state produced after the execution of *remove_pte* when a physical page is mapped twice by the current process.

Subroutine *remove_pte* : integer $\rightarrow M$(unit)
Input : *vaddr*: virtual address to be removed
Action:
index $\leftarrow$ the entry position corresponding to *vaddr*;
if *index* is valid and there is a mapped page at *index* **then**
  Remove the entry content and return the page number p which was mapped in this entry;
  At the first word of p write the value of the first free page of s then return p ;
  Update the first free page of s with the value p;
end

Figure 14: Remove a page table entry content

The current page table is of a process

The following consistency property currProcess_inProcessList (Def. 5) ensures that the number *currentptp*(s) of the physical page storing the page table of the current process is indeed the *ptp* of one of the runnable processes.

Definition 5. *Given a state s, the property* currProcess_inProcessList(s) *holds iff there exists $P \in Processes(s)$ such that ptp(P) = currentptp(s).*

This property is required to preserve the isolation property for some subroutines which depend on this part of the state, the current page table. For instance, when the scheduler switches between processes, it calls the subroutine *switch_process* which sequentially execute the subroutines *save_process* (Fig. 16) and *restore_process* (Fig. 17). The first one removes the first process (which is the currently running process) from the runnable-process list to then add it at the end of the list with its *ptp*(P) value and the updated current pc. On the other hand, the second subroutine sets the next process into the runnable-process list as the new current process by mainly updating the *currentptp*(s) with the corresponding process page table and jumping to the next instruction of that process.

The isolation property requires that the used pages should be different, thus when the *save_process* subroutine adds a process to the process list, each of its

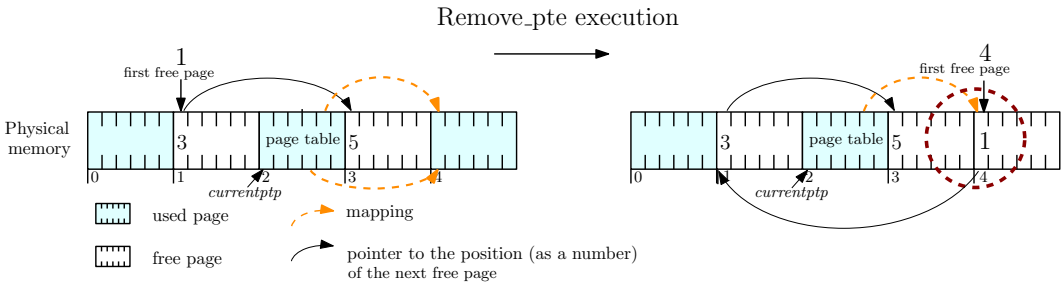

Figure 15: Counterexample for noDuplic_processPages

mapped page must be different from any mapped page of other processes. Consequently, to prove isolation, adding the current process to process list requires that it matches a process in $Processes(s)$, precisely the first one which has been removed previously. Note that because of this property the list of processes can never be empty even in the initial state.

Subroutine *save_process* : $M(\text{unit})$
Action:
remove the first process from the runnable-process list;
add the current process to the end of the runnable-process list;

Figure 16: Saving the current process to the process list

Subroutine *restore_process* : $M(\text{unit})$
Action:
set the first process of the runnable-process list as the current process;
update the current page table with the value of the page table of the new current process;
replace the head of the kernel stack by the next instruction to execute by the new current process and its execution mode;

Figure 17: Restoring the state of a process

3.2.2. Hardware consistency

Page 0 is never used or free

Physical memory may contain several kinds of pages such as used pages, free pages and pages which are reserved by the kernel (and therefore not available for allocation to any process). The latter is very important to isolate some part of the memory from all processes during all possible executions, for instance to store the code of the microkernel and its data. Thus, this information must be stated in consistency properties. So we need to make sure that the pages which

are not available for allocation are indeed never used by a process or considered free. In our model, we chose page 0 as a simple example of this kind of pages. Of course, it could readily be generalized to any set of memory locations which should never be allocated to process.

Two specific consistency properties, FREE_NOTZERO (Def. 7) and USED_NOTZERO (Def. 6), serve to check that page 0 stays unavailable for allocation. In a generalization to any set of unavailable locations, those properties would check that used and free pages stay within the range of valid page numbers.

Definition 6. *Given a state s,* USED_NOTZERO(s) *holds iff for all process $P \in$ Processes(s), for all page $p \in$ UsedPages(P) then $0 < p <$ nb_pages.*

Definition 7. *Given a state s,* FREE_NOTZERO(s) *holds iff for all $p \in$ FreePageList(s), $p \neq 0$.*

Physical memory large enough

The following consistency property ensures that the memory is large enough.

Definition 8. *Given a state s,* MEMORY_LENGTH(s) *holds iff*

$$nb\_pages \times page\_size \leq length(data(s))$$

where length(memory(s)) to denote the size (in bytes) of the physical memory and page_size to denote the size of a page in memory.

Obviously, an undefined hardware behavior can cause vulnerabilities and hence render a proof of security impossible. Commonly, it is the programmer that should ensure that the code never invokes any undefined hardware behavior. In particular, we cannot determine the result of accessing a physical page which is not defined in memory. Consequently, we need to define some property that ensures that all available physical pages are valid and we prove that our model never causes this security issue.

4. Isolation proof

4.1. Hoare logic on top of the H monad

In order to reason about our code, we define a Hoare logic [16] on top of our H monad. A similar approach was used in [17, 18]. Properties of computations are specified by Hoare triples of the form $\{P\}\, c\, \{Q\}$ where:

- P is a precondition, i.e. a unary predicate on the starting state;
- c is a computation returning a result of type A, i.e. the computation c is of type $M(A)$;
- Q is a postcondition, i.e. a binary predicate on the returned value and on the ending state.

By definition, a triple $\{P\}\, c\, \{Q\}$ holds iff: for all state s, if P holds for s then either $c(s)$ denotes the halting of the system or it denotes a pair (a, s') where a is a returned value and s' is an ending state such that the postcondition Q holds for this pair. In the case of $c(s)$ denoting an undefined behavior, the triple does not hold.

The weakest precondition for a computation c and a postcondition Q is the unary predicate on state $wp(Q, c)$ such that:

- the triple $\{wp(Q, c)\}\, c\, \{Q\}$ holds, and
- for any precondition P such that $\{P\}\, c\, \{Q\}$ holds we have, for all state s, $P(s)$ implies $wp(Q, c)(s)$.

4.2. Preservation of isolation and consistency

We have formally proved in Coq that all the instructions, subroutines and hardware components that we model preserve the isolation and the consistency properties. For the most basic computations used as building blocks for our instructions and subroutines, we first prove their weakest precondition triples, and then use it to prove their invariant triples that state preservation of isolation and consistency.

Then we combine those basic invariant triples to obtain invariant triples for the more complex instructions, subroutines and hardware components. We start this section by detailing the formal proof sketch of the basic instruction *trap*, followed by a more involved one, *write*, and give some details about our approach.

Detailed example: the trap instruction

Of course, processes are running in a low-privileged mode called *user mode*, while the microkernel is running in a high-privileged mode called *kernel mode*. In our model, a process can invoke a system call by triggering a software interrupt *trap* (cf. Fig. 18) in order to request the microkernel to perform some operations that do require a higher level of privilege. The intended behavior of this instruction is to increment the *currentpc* position then call the instruction *interrupt* (cf. Fig. 7) which save the context of the current running program (the next instruction to execute and the execution mode) in the *stack*, switch to the kernel mode, then branches to the involved interrupt handler instruction identified by the *interrupt* argument. The proof of this instruction is fairly straightforward. So, we start by proving its weakest precondition then we prove that isolation and consistency properties are preserved after the execution of *trap*. None of these properties depend on the state fields updated by this instruction such as the system *stack*, the execution mode *kernel_mode* and the current instruction to execute. Consequently, this invariant is trivial.

Detailed example: writing in memory

The intended behavior of *write* is to store a given value at a given virtual address in memory. First, this instruction invokes the hardware instruction

Instruction *trap* : integer $\rightarrow M(\text{unit})$
Input : n: the number of the interrupt to execute
Action:
increment the current instruction to execute;
interrupt(n);

Figure 18: triggering a software interrupt

translate. If there is a mapping that corresponds to the given virtual address *vaddr*, *translate* returns the physical address *paddr*, otherwise it returns an exception. In the first case, *write* then executes the instruction *write_phy* (cf. Fig. 19) which stores a value v at the memory address *paddr* of the current process.

Hardware component *write_phy* : integer $\rightarrow$ integer $\rightarrow M(\text{unit})$
Input : v: value to be stored at *paddr*
paddr: physical address
Action :
$p \leftarrow$ the page of *paddr*;
$i \leftarrow$ the position of *paddr* in p;
update_memory(v, i, p);

Figure 19: writing a value at a physical address

Our aim is to ensure that the instruction *write* preserves isolation and consistency. So, we must prove the correctness of the Hoare triple *write_invariant*.

Proposition (write_invariant)**.** *If the isolation property I and the consistency property C hold for the state before the execution of* write*, then I and C also hold afterwards. Formally, we write:*

$$\{I \wedge C\}\ write(v, vaddr)\ \{I \wedge C\}$$

translate is invoked first. It can return an exception. In that case, *write* ends and the final state will be identical to the initial state: isolation and consistency are then trivially preserved.

Let us then consider when *translate* succeeds. Since *translate* is invoked first, its precondition must be the same as the precondition of the instruction *write*. The instruction *write_phy* is the last instruction invoked by *write* so its postcondition must be the same postcondition as *write*.

Since the whole instruction *write* is the sequence of those two functions *translate* and *write_phy*, the postcondition of the first must match the precondition of the second. *translate* should preserve isolation and consistency, so its postcondition will include both these properties.

Another relevant point is that *write_phy* uses the value *paddr* returned by *translate* as a parameter, so to prove that isolation and consistency hold after

the execution of *write_phy*, we must define some property R that depends on this value and the state produced by *translate*. Therefore, the challenge here is to determine the property R and prove the Hoare triples for *translate_invariant* (Lemma 1) and *write_phy_invariant* (Lemma 2).

Lemma 1 (translate_invariant)**.** *If the isolation property I and the consistency property C hold for the state before the execution of* translate*, then I, C and R also hold afterwards. Formally, we write:*

$$\{I \wedge C\}\ translate(vaddr)\ \{I \wedge C \wedge R\}$$

Lemma 2 (write_phy_invariant)**.** *If the isolation property I, the consistency property C and the property R hold for the state before the execution of* write_phy*, then I and C also hold afterwards. Formally, we write:*

$$\{I \wedge C \wedge R\}\ write\_phy(v, paddr)\ \{I \wedge C\}$$

Determining the intermediate property R. Before storing a value in memory, *write_phy* (Fig. 19) must calculate the corresponding page number p of the physical address *paddr* and the offset i of *paddr* in page p, then store v at the position i in the page p. To preserve isolation and consistency, the page p must be mapped in the current process and i must be an offset in this page. So, the property R is defined as follows:

Given a physical address paddr, R holds of a state s and a physical address paddr iff there exists a physical page p and an offset i such that:

- $paddr = p \times page\_size + i$,
- $p \in MappedPages(ptp(s))$, *and*
- $i < page\_size$.

where $MappedPages(ptp(s))$ is the list of all the pages referenced in the page table $ptp(s)$.

After the execution of *translate*, the new state is equal to the previous state. So, it is straightforward to prove that isolation and consistency are preserved. On the other hand, we have to prove that $R(paddr, s)$ holds afterwards (where *paddr* is the physical address returned by *translate*). For such needs, we use its weakest precondition triple.

Contrary to *translate*, *write_phy* modifies the current state and does not return any value. Consequently, the postcondition will depend on the new state that we denote s'. This instruction stores the value v in physical memory. Thus, only $memory(s)$ will be changed.

Therefore we have to prove that if isolation I, consistency C and R hold of the parameter *paddr* and the state before the execution of *write_phy* then isolation and consistency hold afterwards. This proof requires eight cases, one for isolation, and one per consistency property. In the following we sketch the proof of all properties.

- ISOLATION(s) : This case amounts to the fact that if the current process writes a value in physical memory, it cannot modify a page table of a runnable process. The challenge is to prove that the position of the page p (cf. Fig. 19) is different from all positions of runnable-process page tables and mapped pages into these tables.

  Let P_1 and P_2 be two distinct processes from $Processes(s)$. The consistency property CURRPROCESS_INPROCESSLIST(s) (Def. 5) ensures that the page table of the current process is a page table of a process in $Processes(s)$. Consequently we have three cases (two of which are symmetric). The first case is when $currentptp(s)$ is different from both $ptp(P_1)$ and $ptp(P_2)$. Isolation and consistency are trivially preserved in this case. The two other cases are respectively when $currentptp(s)$ is equal to $ptp(P_1)$ or, symmetrically, $ptp(P_2)$. In those cases, we need the property R to ensure that the page p is a mapped page in the current-process page table and i is a position in this page. In addition we need the consistency property NODUPLIC_PROCESSPAGES to ensure that the position of a mapped page is different from the position of the current page table and thus that the page table will not be modified. Also, we use the isolation property of the previous state to prove that this instruction cannot modify any other runnable-process page table.

- REALLY_FREE(s) : This property depends on the free-page list and process page tables. So, in this case we prove that if a process writes a value in physical memory, it cannot modify this kernel data. We use the property R to ensure that the page p which will be modified is a mapped page and thus using the property REALLY_FREE on the previous state, we ensure that it is not a free page. Also, we use the property NODUPLIC_PROCESSPAGES to ensure that this instruction cannot modify the current process page table. Finally we use the isolation property on the previous state to prove that it cannot modify any runnable-process page table.

- NOT_CYCLIC(s) : To prove that there is no cycle in the free-page list on the ending state, we first use the property NOT_CYCLIC of the previous state. Then, using a reasoning similar to the one used for REALLY_FREE, we prove that this instruction cannot modify the free-page list using mainly R and REALLY_FREE.

- NODUPLIC_PROCESSPAGES(s): To prove that there is no duplication in the pages used by the processes on the ending state, we use NODUPLIC_PROCESSPAGES to ensure that this instruction cannot modify the current-process page table and the isolation property in the previous state to ensure that this instruction cannot modify any runnable-process page table.

- FREE_NOTZERO(s): To prove that all free pages remain different from the page 0 during the execution of *write_phy*, we use a reasoning similar to

the one used for NOT_CYCLIC (which depends only upon this list, too) to prove that the free-page list is unchanged on the ending state.

- USED_NOTZERO(s): To prove that the page 0 is never used by any process during the execution of *write_phy*, we have to prove that the page table of any process is not modified on the ending state. Both, ISOLATION(s) and USED_NOTZERO(s) depend on this data, consequently the proof is the same as in ISOLATION(s).

- MEMORY_LENGTH(s): In this case we need to prove that the physical-memory length does not vary during the execution of *write_phy*.

- CURRPROCESS_INPROCESSLIST(s): The proof of this property is trivial because it does not depend on $memory(s)$.

Other example: adding a new PTE

The proof sketch of *write_invariant* above was set out to explore the most relevant points necessary to understand our approach to establish the expected properties of our model. There are however more involved subroutines that need more effort to prove their expected properties because of their complexity. In this section, we will briefly discuss another example.

The expected behavior of the subroutine *add_pte* (Fig. 12) is to add a new entry to the page table of the currently-running process: it maps a new physical page at a given index in the page table of the currently-running process. More precisely, if there is no mapping yet at that index, it invokes a subroutine called *alloc_page*. This subroutine allocates a new page then adds a new mapping corresponding to this page and to a given permission. The latter one requires a precondition ensuring that there is no mapping in the involved entry which is, notably, ensured by the second test in Fig. 12. The difficulty is that between this test and the second instruction which adds the mapping in *pte*, the subroutine *alloc_page* changes the state. Consequently, we have to propagate this property by proving that if it holds at the state before the execution of *alloc_page* then it holds afterwards. In our model, *alloc_page* is used in several subroutines. Therefore, we have defined and proved a new invariant for *alloc_page* which preserves isolation and consistency and propagates the necessary property.

4.3. Initial state and process creation

In our approach we consider that the isolation and consistency properties are invariant. So, when the system starts up, its first task is to reach the first, initial, state in which those properties are verified. This is the goal of the *boot* subroutine.

An example of a consistent initial state could be the initialization of the list of processes with a single process P which is also the current process. The value of $currentpc(s)$ would be initialized with the value of $pc(P)$ (i.e. the pointer to the first instruction to execute by P) and the value of $currentptp(s)$ would be initialized with the value of $ptp(P)$ (i.e. the page table of the process P).

Similarly, physical memory should be initialized so that the consistency of the free pages linked list (as detailed in Fig. 3) is ensured and so that the value of *first_free_page*(s) corresponds to the value of the first free page. Finally, the *intr_table*(s) list should contain the entry points of all the interrupt handlers in *code*(s). An example of this list is to set the *switch_process* subroutine as the interrupt number 0 and the *create_process* subroutine as the interrupt number 1.

In order to verify this initial state we prove the Hoare triple for *boot_invariant* (Lemma 3).

Lemma 3 (boot_invariant)**.** *The isolation property I and the consistency property C hold for the state after the execution of* boot

$$\{True\}\, boot\, \{I \wedge C\}$$

After booting up, any process can create new processes using the *create_process* subroutine which allocates a new page as the page table of this process and adds it to the list of processes.

Subroutine *create_process* : integer $\rightarrow M$(unit)
Input : *pc*: the first instruction to execute by the process
Action :
table $\leftarrow$ allocate new physical page;
set *table* as the page table of the process;
set *pc* as the pointer to the first instruction to execute by the process;
add the new process to *processes*(s);

Figure 20: create a new process

It is also during start-up that the system configures the timer to trigger preemptive scheduling. This can be implemented in our model by building (during the *boot* subroutine) an initial state such that:

- some interrupt number (the one used by the timer) is associated in *intr_table* to the scheduler,
- and the hardware *interrupts* stream periodically contains the timer interrupt number.

5. Overview of the formalization

In this section we describe the structure of our development in Coq. The Coq code of our formalization is available at: `https://github.com/jomaa/MIMIC`. It can be compiled with the version `8.5pl2` of Coq. It consists of about 2400 lines of specification and 9700 lines of proof. The size of each file can be found in Table 1. Our development is organized in the following way:

- The definition of our hardware monad that provides states is split in the two files StateMonad.v and HMonad.v.

Category	Filename	lines of spec	lines of proof
Hardware monad	StateMonad.v	101	70
	HMonad.v	145	70
	Subtotal	**246**	**140**
Hardware architecture	MMU.v	84	170
	MemoryManager.v	93	76
	Access.v	107	42
	Instructions.v	168	117
	Step.v	32	0
	Subtotal	**484**	**405**
Microkernel	PageTableManager.v	207	76
	Scheduler.v	115	25
	ProcessManager.v	115	6
	Subtotal	**437**	**107**
Definition and proof of isolation and consistency	Properties.v	58	0
	MMU invariant.v	18	273
	Access_invariants.v	117	1487
	Scheduler_invariant.v	140	347
	Instructions_invariants.v	55	108
	ProcessManager_invariant.v	123	596
	Step_invariant.v	8	37
	Alloc_invariants.v	37	102
	Addpte_invariant.v	122	1749
	Removepte_invariant.v	323	3763
	Subtotal	**1001**	**8462**
Miscellaneous files	Lib.v	182	525
	LibOs.v	29	69
	Example.v	83	89
	Subtotal	**294**	**683**
Total		**2462**	**9797**

Table 1: Project organization

- Our model of a hardware architecture is split into different files:
  - The hardware component *translate* for the MMU is singled out in MMU.v.
  - Memory allocation is modeled in MemoryManager.v by the *alloc_page* subroutine.
  - Access to physical memory is modeled in Access.v where the instructions *write* and *read* are defined.
  - The dynamic evolution of the system (i.e. *step*), including interrupt management (*interrupt*, *fetch_interrupt*, *fetch_instruction* and *return_from_interrupt*), is modeled in Instructions.v and Step.v.

- Our model of a microkernel is split into different files:
  - Subroutines *add_pte* and *remove_pte* to modify process page tables are in PageTableManager.v.
  - Subroutines *save_process*, *restore_process* and *switch_process* for preemptive CPU-time sharing are in Scheduler.v.
  - The process creation subroutine *create_process* is defined in ProcessManager.v.
- The file Properties.v contains the definitions of isolation and consistency.
- The proofs of preservation of isolation and consistency are spread between the files MMU_invariant.v, Access_invariants.v, Scheduler_invariant.v, Instructions_invariants.v, ProcessManager_invariant.v, Step_invariant.v, Alloc_invariants.v, Addpte_invariant.v, Removepte_invariant.v.

6. Conclusions and future work

In this paper we developed an approach based on a variant of the Hoare logic in order to verify an abstract model of a microkernel in the Coq proof assistant. This formal model is implemented using a monadic style in order to represent stateful programs. The physical memory of user processes is guaranteed to be isolated. Indeed, we first formalize the model of the relevant part of both MMU and CPU behaviors that are required for the memory isolation property. Then we implement an abstract model of a virtual memory manager and the basic principles of interrupts in order to support preemptive scheduling.

Several consistency properties about the microkernel kernel behavior and the corresponding hardware architecture were discovered incrementally during the proof, that must be preserved by the microkernel in order for the memory isolation to be preserved. Our proofs consist in proving that both isolation and consistency properties are preserved along the execution of the microkernel.

One conclusion we can draw from this formalization is that many details about the architecture and the microkernel are to be taken into account in order to prove memory isolation between processes. This is thus a typical example of a proof that would be hard for a human to conduct without a proof assistant, because there would be too many details to keep in mind at all times. But with the help of the Coq proof assistant that keeps track rigorously of all the minutiae of the proof, one can be sure not to overlook any corner case.

Also we arrived at the current list of consistency properties listed in Section 3 after a few iterations. And each time we were extending this list, we had to go through all the invariant proofs again to prove that consistency was preserved. We benefited from the simple but useful mechanism called bullets which allows to structure proof script and thus easily find where to insert the additional cases to be dealt with.

One possible future work is to use the insights we gained from this formalization to design kernels that are more amenable to formal proof. We are

in particular interested in exokernels [19] because they push much further the minimality principle [2] while still ensuring fundamental security properties that would be interesting to formally prove.

References

[1] N. Jomaa, D. Nowak, G. Grimaud, S. Hym, Formal proof of dynamic memory isolation based on MMU, in: 10th International Symposium on Theoretical Aspects of Software Engineering, TASE 2016, Shanghai, China, July 17-19, 2016, IEEE Computer Society, 2016, pp. 73–80. doi:10.1109/TASE.2016.28.

[2] J. Liedtke, On micro-kernel construction, in: Jones [20], pp. 237–250. doi:10.1145/224056.224075.

[3] D. Elkaduwe, G. Klein, K. Elphinstone, Verified protection model of the sel4 microkernel, in: N. Shankar, J. Woodcock (Eds.), Verified Software: Theories, Tools, Experiments, Second International Conference, VSTTE 2008, Toronto, Canada, October 6-9, 2008. Proceedings, Vol. 5295 of Lecture Notes in Computer Science, Springer, 2008, pp. 99–114. doi:10.1007/978-3-540-87873-5_11.

[4] G. Klein, J. Andronick, K. Elphinstone, T. C. Murray, T. Sewell, R. Kolanski, G. Heiser, Comprehensive formal verification of an OS microkernel, ACM Trans. Comput. Syst. 32 (1) (2014) 2:1–2:70. doi:10.1145/2560537.

[5] L. Gu, A. Vaynberg, B. Ford, Z. Shao, D. Costanzo, Certikos: a certified kernel for secure cloud computing, in: H. Chen, Z. Zhang, S. Moon, Y. Zhou (Eds.), APSys '11 Asia Pacific Workshop on Systems, Shanghai, China, July 11-12, 2011, ACM, 2011, p. 3. doi:10.1145/2103799.2103803.

[6] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, D. Costanzo, Certikos: An extensible architecture for building certified concurrent OS kernels, in: K. Keeton, T. Roscoe (Eds.), 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016., USENIX Association, 2016, pp. 653–669.

[7] A. Vaynberg, Z. Shao, Compositional verification of a baby virtual memory manager, in: C. Hawblitzel, D. Miller (Eds.), Certified Programs and Proofs - Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings, Vol. 7679 of Lecture Notes in Computer Science, Springer, 2012, pp. 143–159. doi:10.1007/978-3-642-35308-6_13.

[8] H. Chen, X. N. Wu, Z. Shao, J. Lockerman, R. Gu, Toward compositional verification of interruptible OS kernels and device drivers, in: C. Krintz, E. Berger (Eds.), Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016, ACM, 2016, pp. 431–447. doi:10.1145/2908080.2908101.

[9] G. Barthe, G. Betarte, J. D. Campo, J. M. Chimento, C. Luna, Formally verified implementation of an idealized model of virtualization, in: R. Matthes, A. Schubert (Eds.), 19th International Conference on Types for Proofs and Programs, TYPES 2013, April 22-26, 2013, Toulouse, France, Vol. 26 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013, pp. 45–63. doi:10.4230/LIPIcs.TYPES.2013.45.

[10] N. Marti, R. Affeldt, A. Yonezawa, Formal verification of the heap manager of an operating system using separation logic, in: Z. Liu, J. He (Eds.), Formal Methods and Software Engineering, 8th International Conference on Formal Engineering Methods, ICFEM 2006, Macao, China, November 1-3, 2006, Proceedings, Vol. 4260 of Lecture Notes in Computer Science, Springer, 2006, pp. 400–419. doi:10.1007/11901433_22.

[11] Y. Guo, H. Zhang, Verifying preemptive kernel code with preemption control support, in: 2014 Theoretical Aspects of Software Engineering Conference, TASE 2014, Changsha, China, September 1-3, 2014, IEEE Computer Society, 2014, pp. 26–33. doi:10.1109/TASE.2014.29.

[12] A. S. Tanenbaum, A. S. Woodhull, Operating systems - design and implementation, 3rd Edition, Pearson Education, 2006.

[13] E. Moggi, Notions of computation and monads, Inf. Comput. 93 (1) (1991) 55–92. doi:10.1016/0890-5401(91)90052-4.

[14] P. Wadler, Comprehending monads, Mathematical Structures in Computer Science 2 (4) (1992) 461–493. doi:10.1017/S0960129500001560.

[15] W. Stallings, P. Zeno, Computer Organization and Architecture: Designing for Performance, 10th Edition, Always learning, Pearson, 2015, Ch. 3, p. 116.

[16] C. A. R. Hoare, An axiomatic basis for computer programming, Commun. ACM 12 (10) (1969) 576–580. doi:10.1145/363235.363259.

[17] D. Cock, G. Klein, T. Sewell, Secure microkernels, state monads and scalable refinement, in: O. A. Mohamed, C. A. Muñoz, S. Tahar (Eds.), Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings, Vol. 5170 of Lecture Notes in Computer Science, Springer, 2008, pp. 167–182. doi:10.1007/978-3-540-71067-7_16.

[18] W. Swierstra, A hoare logic for the state monad, in: S. Berghofer, T. Nipkow, C. Urban, M. Wenzel (Eds.), Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings, Vol. 5674 of Lecture Notes in Computer Science, Springer, 2009, pp. 440–451. doi:10.1007/978-3-642-03359-9_30.

[19] D. R. Engler, M. F. Kaashoek, J. O'Toole, Exokernel: An operating system architecture for application-level resource management, in: Jones [20], pp. 251–266. doi:10.1145/224056.224076.

[20] M. B. Jones (Ed.), Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995, ACM, 1995. doi:10.1145/224056.

Appendix A. The formal model in Coq

Appendix A.1. The state of the system

```
Inductive instr : Type :=
(** Hardware component and instructions  *)
| Halt
| Trap (n : nat)
| Iret
| Reset
| Write (val vaddr : nat)
| Load  (addr : nat)
| Nop
| Exit
(** kernel subroutines *)
| Create_process (pc : nat)
| Switch_process
| Add_pte (permission index: nat)
| Remove_pte (page : nat).

Record process : Type := {
  pc : nat;
  process_kernel_mode : bool;
  ptp : nat;
  stack_process : list nat
}.

Record state : Type := {
  process_list : list process;
  current_process : process;
  currentptp : nat ;
  code : list instr ;
  intr_table : list nat;
  interruptions : Stream (option nat);
  kernel_mode : bool;
  currentpc : nat;
  stack : list (bool * nat);
```

```
  register : nat;
  memory : list nat ;
  first_free_page : nat
}.
```

Appendix A.2. The dynamic evolution of the system

In the formal definition for the *step* hardware component, we used notations for the monadic operations.

```
perform b := M in N
```

stands for the standard monadic *bind* (binding the result of the computation M to the name b in N); and

```
M ;; N
```

stands for the monadic *bind* that discards the result of the computation M.

Using those notations, *step* is defined as follows.

```
Definition step : M unit :=
  perform b := fetch_interruption in
  match b with
  | None => perform i := fetch_instruction in
    match i with
(** Hardware component and instructions *)
    | Halt => halt
    | Trap n => incr_pc ;; interrupt n
    | Iret => return_from_interrupt
    | Reset => incr_pc ;; reset
    | Write v vaddr => incr_pc ;; write v vaddr
    | Load addr => incr_pc ;; load addr
    | Nop => incr_pc
    | Exit => ret tt
(** kernel subroutines *)
    | Create_process pc => incr_pc ;; create_process pc
    | Switch_process => incr_pc ;; switch_process
    | Add_pte permission index => incr_pc ;; add_pte permission index
    | Remove_pte vaddr => incr_pc ;; remove_pte  vaddr
    end
  | Some n => interrupt n
  end.
```