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 and 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 manager. 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; Micro-kernel; Coq.**

I. Introduction

Modern operating-system kernels allow 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, for both 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 which all memory accesses must go through. It translates a virtual memory address to a physical address if there is indeed a corresponding one for 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 but not accessible while in user mode. For this to work properly, the kernel has to maintain page 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 it. 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 minimal as possible. This stems from the general principle that the trusted computing base (TCB) should be kept minimal. This is the reason why we focus in this paper on an abstract model of a microkernel [1] which supports preemptive scheduling and ensures memory isolation.

Contributions: Our main contribution is a formal proof 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) are modeled in all their relevant minutiae, while less relevant parts are abstracted away.
- A formal model of a microkernel supporting preemptive scheduling and memory management at an appropriate abstraction level so that it remains a realistic model without being linked to a particular implementation.
- 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. Below, we compare our work with those that appear most closely related to ours.

One of the most significant is the formal proof in the Isabelle/HOL proof assistant of the functional correctness and security properties of the microkernel seL4 [2]. There is also CertiKOS which is a hypervisor dedicated to cloud computing that is formally verified [3]. 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 [4]. In contrast to those work above, our goal is not to prove formally properties of a specific microkernel but to clarify what is assumed by microkernels about the hardware architecture and what are the constraints a microkernel must follow in order for memory isolation to be guaranteed at all times.

In [5], 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 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 that deals with page allocation.

In [6], the operations of allocation and deallocation of a microkernel were proved correct. However, those operations

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

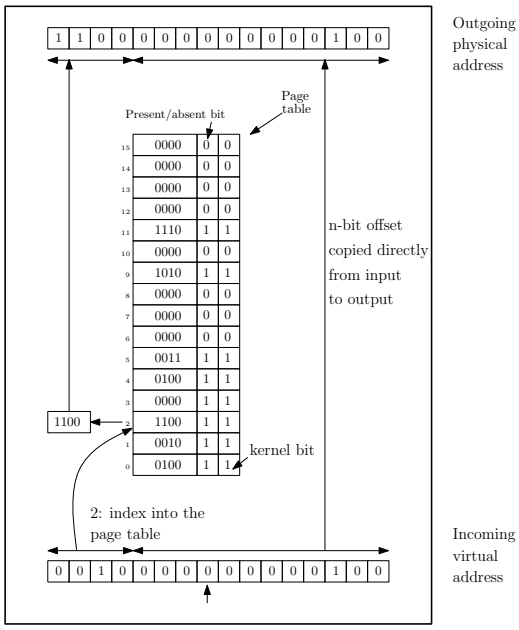

Figure 1. Memory management unit (MMU)

live in a higher layer of the operating system than the lower-level layer we assume here. Moreover our work show that the correct implementation of those operations is essential to ensure memory isolation.

Guo and Zhang proposed in [7] a verification framework for verifying preemption control operations in a preemptive kernel. While, in our case, we focus our verification effort on proving memory isolation between processes.

Outline: We first describe in Section II our formal model of a microkernel and our formal definition of the memory isolation. We then make explicit in Section III the consistency properties that are to be preserved in order for memory isolation to be always guaranteed. In Section IV, we present our proof methodology and discuss the difficulties met and their solutions. We finally conclude Section V.

II. Formal model of a microkernel

In this section, we first briefly recall some basic facts about microkernels and then describe our formal model for a microkernel and our formal definition of the memory isolation.

A. Background on MMU-based microkernels

The purpose of a microkernel is to manage several executing programs known as runnable processes. To ensure security (in particular memory isolation), processes cannot directly access the physical memory. 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. 1 taken from [8]). This translation mechanism is implemented using *page tables*. These page tables are 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 all the physical pages allocated to a given process are 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 to 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 *kernel_mode* of the current state s. When a process tries to violate these protection rules, the MMU raises an exception which will be handled by the microkernel. Moreover, to provide preemptive scheduling, the microkernel must share CPU time between all the currently runnable processes giving the illusion of executing all processes at the same time. 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*. A process can then invoke a system call by triggering an interruption.

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.

B. 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 language, it is thus common to implement such features by using a monad [9]. For our model, we have defined a monad that we call *H monad* and that provides states (as described below in Section II-C) 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 the 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 classify computations into three groups: a *hardware component* models the behavior of a piece of hadware; an *instruction* is code for an atomic CPU instruction; a *subroutine* is a piece of code that should not be interrupted.

C. 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 kernel data structures. In

our formal verification, we focus only on the part of the state that is relevant to prove the properties we are interested in. We split accordingly our formal state into its hardware and software parts, and the formal definition of the type of the state is a record whose fields are the functions listed under the following hardware state and software state.

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 in user mode.
- *data*(s) is the physical memory; it contains in particular the links to the free-page list and the page tables of processes which are essential for the MMU address space translation.

The hardware state contains more information, such as the position of the current instruction to execute, the stream of interruptions, etc. We will not detail them here since they are not necessary to understand the rest of this paper.

Software state : During execution, the system needs to store some information in physical memory and which 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 memory which is reserved by the kernel and thus never available for allocation. 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 like the reference to its page table *ptp*(P) and the address to the next instruction to execute. When we switch between processes the value of *currentptp*(s) should be updated with the value of *ptp*(P) of the selected process P.
- *first_free_page*(s) is the first page of the free-pages linked list containing all the pages that can be allocated. Therefore, a memory manager is required to determine which pages are available for allocation and which are not. In our model, it consists of using the available pages themselves to store the linked list of available pages. We illustrate this memory model in (Fig 2). On system startup, available physical pages are initialized in such a way that the value of the first byte of the page corresponds to the position of the next free page; the microkernel has only to keep in its state the position of the first page of that linked list.
- *code*(s) denotes the sequences of system and process instructions. The main property we are interested in here is to ensure data security. The proof of isolation for the code would be similar.

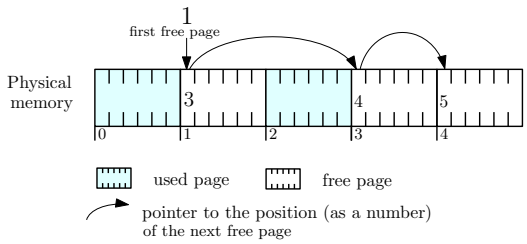

Figure 2. Memory model

Just as for the hardware state, the software state contains some more information that will not be detailed in this paper, such as the system stack.

D. Memory isolation

As explained above, 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 II-A. 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.

```
1 Instruction write : integer → integer → M(unit)
  Input : val: the value to be written
          vaddr: the virtual address at which it should
  be written
  Action:
2 paddr ← translate(vaddr) ;
3 if paddr is not an exception then
4 |   write val at physical address paddr
5 end
```

Figure 3. 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. For instance, we have formalized the hardware instruction *write* (Fig. 3) that writes a value *val* at some physical address *paddr*. This instruction needs the hardware function *translate* (Fig. 4) to compute the physical address *paddr* for the virtual address *vaddr* according to the page table of the current process. In such a case, even if the physical page is mapped in two processes page tables, the MMU will translate addresses without raising any exception and the current process will access and modify the content of

```
1 Hardware component
  translate : integer → M(integer + exception)
  Input  : vaddr: a virtual address
  Output: the corresponding physical address or an
          exception
  Action :
2 if the vaddr size is valid then
3 |   calculate the address of page table entry pte and
  |   the offset in this page corresponding to vaddr;
4 |   if there is a mapped page in pte and (the
  |    kernel_mode(s) = true or kernel_only(pte) = 0)
  |   then
5 |   |   calculate the physical address and return it;
6 |   end
7 end
```

Figure 4. Translating virtual addresses to physical addresses

that page. This clearly shows that the two processes are not properly isolated.

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 all the pages which are used by P_1 are different from all pages 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 ptp$(P_1) \neq$ ptp(P_2) and for all $p \in$ UsedPages(P_1), $p \notin$ 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.*

III. Consistency

In this section, we introduce consistency properties and motivate them with counterexamples that show how a simple breach of consistency would invalidate memory isolation. Various of these properties rely on the list of free-pages.

Definition 2. *Given a state s, FreePagesLinkedList(s) is the linked list of all physical pages available for allocation.*

A. Software consistency

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

Definition 3. *Given a state s,* REALLY_FREE(s) *holds iff for all $p \in$ FreePagesLinkedList(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).*

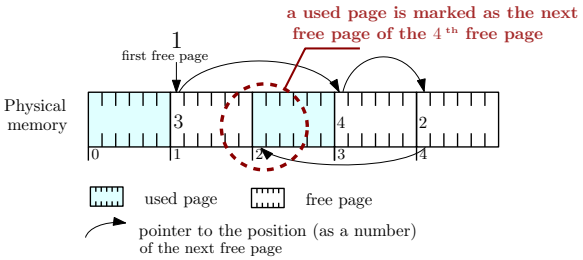

Figure 5. Counterexample for REALLY_FREE

By adding the property REALLY_FREE in consistency we must verify that on each system step, all marked-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* (cf. Fig. 6) preserves isolation. Indeed, this subroutine allocates the first free page selected from *FreePagesLinkedList(s)*, then adds it into the process page table. If this page was already used by another process (cf. Fig. 5), 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.

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

Figure 6. Adding an entry in page table

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

Definition 4. *Given a state s,* NOT_CYCLIC(s) *holds iff for all page p, nb_occurrences$(p,$ FreePagesLinkedList$(s)) \leq 1$*

where nb_occurrences(p, l) is the number of occurrences of p in list l.

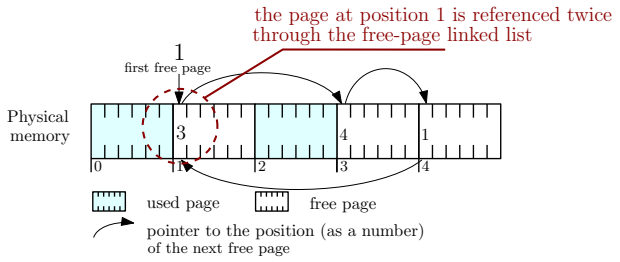

Figure 7. Counterexample for NOT_CYCLIC

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. 7 illustrates such counterexample.

No duplication in process used pages: When defining memory isolation we have explained that all used pages of any process in the runnable-process list must be different from all used pages of all the other runnable processes. The following consistency property NODUPLIC_PROCESSPAGES property (Def. 5) ensures that for any process, all its used pages appear only once in its page table.

Definition 5. *The consistency property* NODUPLIC_PROCESSPAGES *holds of a state s iff for all process $P \in$ Processes(s), there is no duplication in UsedPages(P).*

The need for this property arises when proving the isolation property of subroutine *remove_pte* (Fig. 8). 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. 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 marked both used and free at the same time. Then another process might allocate p, and isolation property would no longer hold. Fig. 9 illustrates an inconsistent state produced after the execution of *remove_pte* when a physical page is mapped twice by the current process.

The current page table is of a process: The following consistency property CURRPROCESS_INPROCESSLIST (Def. 6) 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 6. *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 *save_process* (Fig. 10) to remove 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 the *currentptp*(s) value and the new current instruction. The isolation property requires that the used pages should be different, thus when we add a process to the process list, all its mapped pages must be different from other mapped pages. 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. This property do ensure that the current page table is the page table of a process in *Processes*(s).

```
1 Subroutine remove_pte : integer → M(unit)
  Input : vaddr: virtual address to be removed
  Action:
2 index ← the entry position corresponding to vaddr;
3 if index is valid and there is a mapped page at index
  then
4     Remove the entry content and return the page p
      which was mapped in this entry;
5     At the first byte of p write the value of the first
      free page of s then return p ;
6     Update the first free page of s with the value p;
7 end
```

Figure 8. Remove a page table entry content

B. Hardware consistency

Page 0 is never used or marked-free: Physical memory may contain several kinds of pages such as used pages, free pages and pages which are not available for allocation. The latter is very interesting 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 are unavailable for allocation.

Two specific consistency properties, FREE_NOTZERO (Def. 8) and USED_NOTZERO (Def. 7), 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 stays within the range of valid page numbers.

Definition 7. *Given a state s,* USED_NOTZERO(s) *holds*

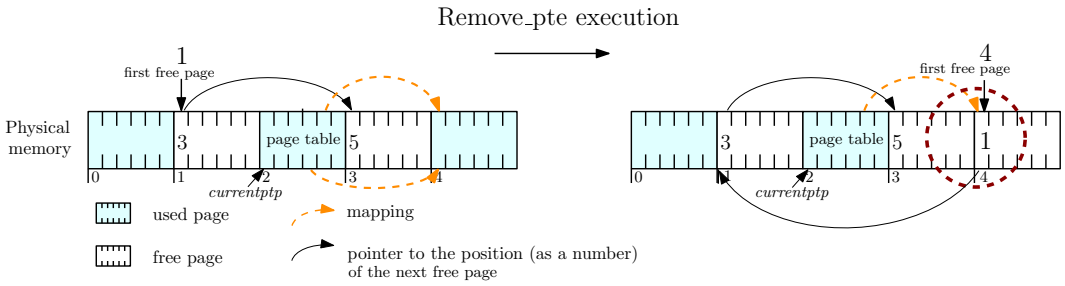

Figure 9. Counterexample for NODUPLIC_PROCESSPAGES

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

Figure 10. Saving the current process to the process list

iff for all process $P \in Processes(s)$*, for all page* $p \in UsedPages(P)$ *then* $0 < p < nb\_pages$.

Definition 8. *Given a state s,* FREE_NOTZERO(s) *holds iff for all* $p \in FreePagesLinkedList(s)$, $p \neq 0$.

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

Definition 9. *Given a state s,* MEMORY_LENGTH(s) *holds iff*

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

where $length(data(s))$ *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 undefined hardware behaviour. Precisely here we cannot determine the result of accessing to a physical page which is not defined in memory. Consequently, we need to define some property that ensure that all available physical pages are valid and we prove that our model never causes this security issue.

C. Conclusion

To conclude, 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. Instead, we characterize the consistency properties required for isolation and prove that they are always preserved.

IV. ISOLATION PROOF

A. Hoare logic on monad

In order to reason about our code, we define a Hoare logic [10] on top of our H monad. A similar approach was used in [11], [12]. 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 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 $wp(Q, c)(s)$.

B. 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. The following examples illustrate our approach and the difficulties we encountered.

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 *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. 11) which stores a value v at the memory address *paddr* of the current process.

1 Instruction *write_phy* : integer $\rightarrow$ integer $\rightarrow M$(unit)
Input : v: value to be stored at *paddr*
paddr: physical address
Action :
2 $p \leftarrow$ the page of *paddr*;
3 $i \leftarrow$ the position of *paddr* in p;
4 *update_memory*(v, i, p);

Figure 11. 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:*

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

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 return value *paddr* of *translate* as a parameter, so we must define some property *R* that depends on this value and the state produced by *translate*. This property is required to prove that isolation and consistency hold after the execution of *write_phy*. Therefore, the challenge here is to determine the property *R* then 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:*

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

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:*

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

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*$(\textit{paddr}, s)$ 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 *data*(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 only sketch the proof of the case for isolation. 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. 11) is different from all positions of runnable-process page tables and mapped pages into these tables.

Let P_1 and P_2 be two different processes from *Processes*(s). The consistency property CURRPROCESS_INPROCESSLIST(s) (Def. 6) 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 can not modify any other runnable-process page table.

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 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. 6) 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. 6. 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.

V. Conclusions and future work

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 III 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.

Our formalization in Coq is available at:

http://www.cristal.univ-lille.fr/~nowakd/microkernel/

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 [13] because they push much further the minimality principle [1] while still ensuring fundamental security properties that would be interesting to formally prove.

References

[1] J. Liedtke, "On micro-kernel construction," in *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995*.

[2] G. Klein, J. Andronick, K. Elphinstone, T. C. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive formal verification of an OS microkernel," *ACM Trans. Comput. Syst.*, 2014.

[3] L. Gu, A. Vaynberg, B. Ford, Z. Shao, and D. Costanzo, "Certikos: a certified kernel for secure cloud computing," in *APSys '11 Asia Pacific Workshop on Systems, Shanghai, China, July 11-12, 2011*.

[4] A. Vaynberg and Z. Shao, "Compositional verification of a baby virtual memory manager," in *Certified Programs and Proofs - Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings*.

[5] G. Barthe, G. Betarte, J. D. Campo, J. M. Chimento, and C. Luna, "Formally verified implementation of an idealized model of virtualization," in *19th International Conference on Types for Proofs and Programs, TYPES 2013, April 22-26, 2013, Toulouse, France*.

[6] N. Marti, R. Affeldt, and A. Yonezawa, "Formal verification of the heap manager of an operating system using separation logic," in *Formal Methods and Software Engineering, 8th International Conference on Formal Engineering Methods, ICFEM 2006, Macao, China, November 1-3, 2006, Proceedings*.

[7] Y. Guo and 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*.

[8] A. S. Tanenbaum and A. S. Woodhull, *Operating systems - design and implementation (3. ed.)*. Pearson Education, 2006.

[9] P. Wadler, "Comprehending monads," *Mathematical Structures in Computer Science*, 1992.

[10] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, 1969.

[11] D. Cock, G. Klein, and T. Sewell, "Secure microkernels, state monads and scalable refinement," in *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*.

[12] W. Swierstra, "A hoare logic for the state monad," in *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*.

[13] D. R. Engler, M. F. Kaashoek, and J. O'Toole, "Exokernel: An operating system architecture for application-level resource management," in *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995*.