Formal Proof of Dynamic Memory Isolation Based on MMU

David Nowak

CRIStAL, CNRS & Lille 1 University, France

SoSySec
September 23, 2016

Joint work with
Narjes Jomaa, Gilles Grimaud, and Samuel Hym

Memory isolation between processes

Why? For safety and security

How? By software (kernel of an OS), and
by hardware (MMU, kernel mode)

Correct? Ensured by a formal proof

Doable? Yes, by reducing the Trusting Computing Base

Outline

Formally proving memory isolation

- Microkernel and MMU
- Monad and Hoare logic
- Memory isolation and consistency

A proof-oriented minimal kernel (work in progress)

- Memory isolation in Pip
- Implementation of Pip

Outline

Reduced TCB with a microkernel

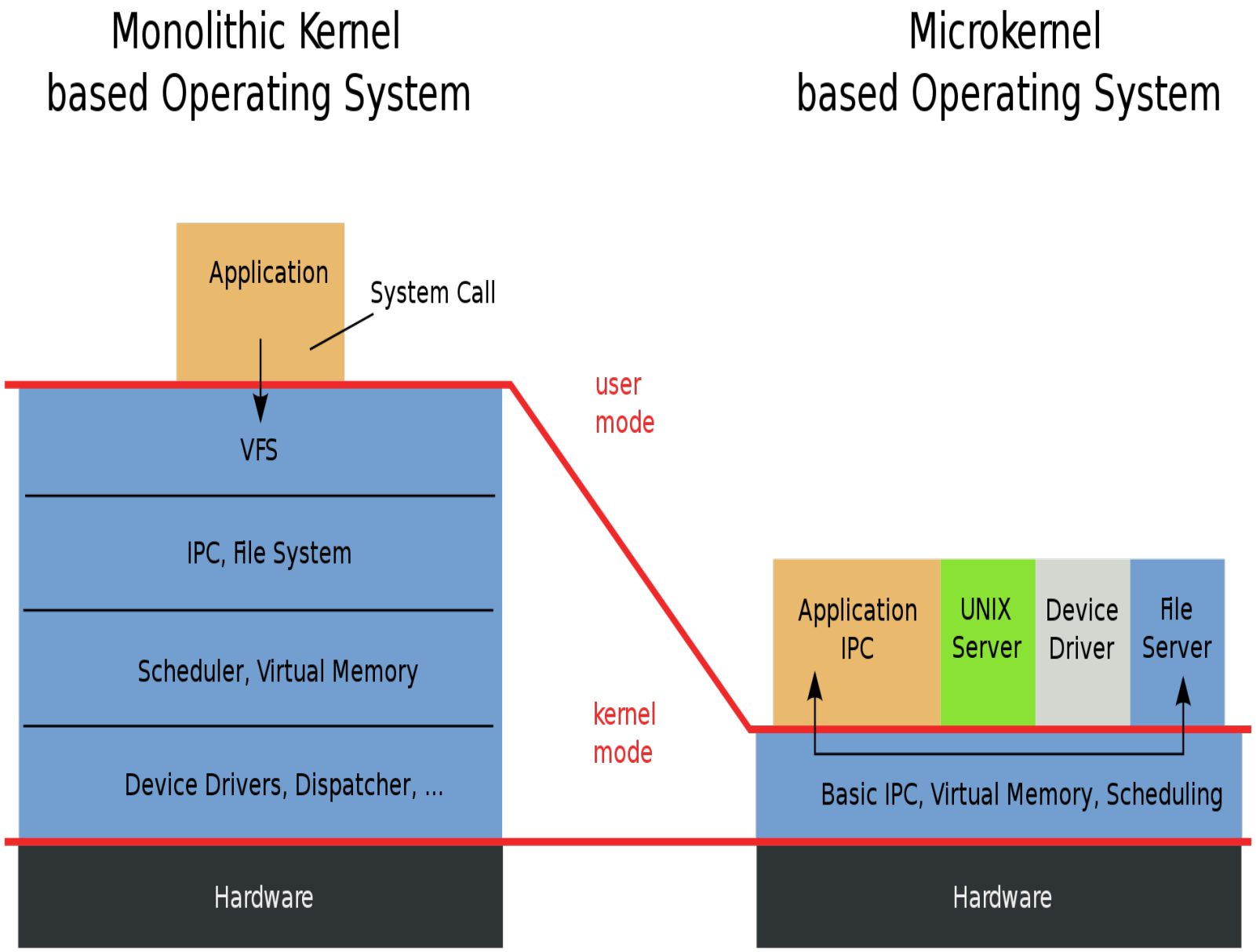


Image from Wikipedia — ©Wooptoo

Virtual memory

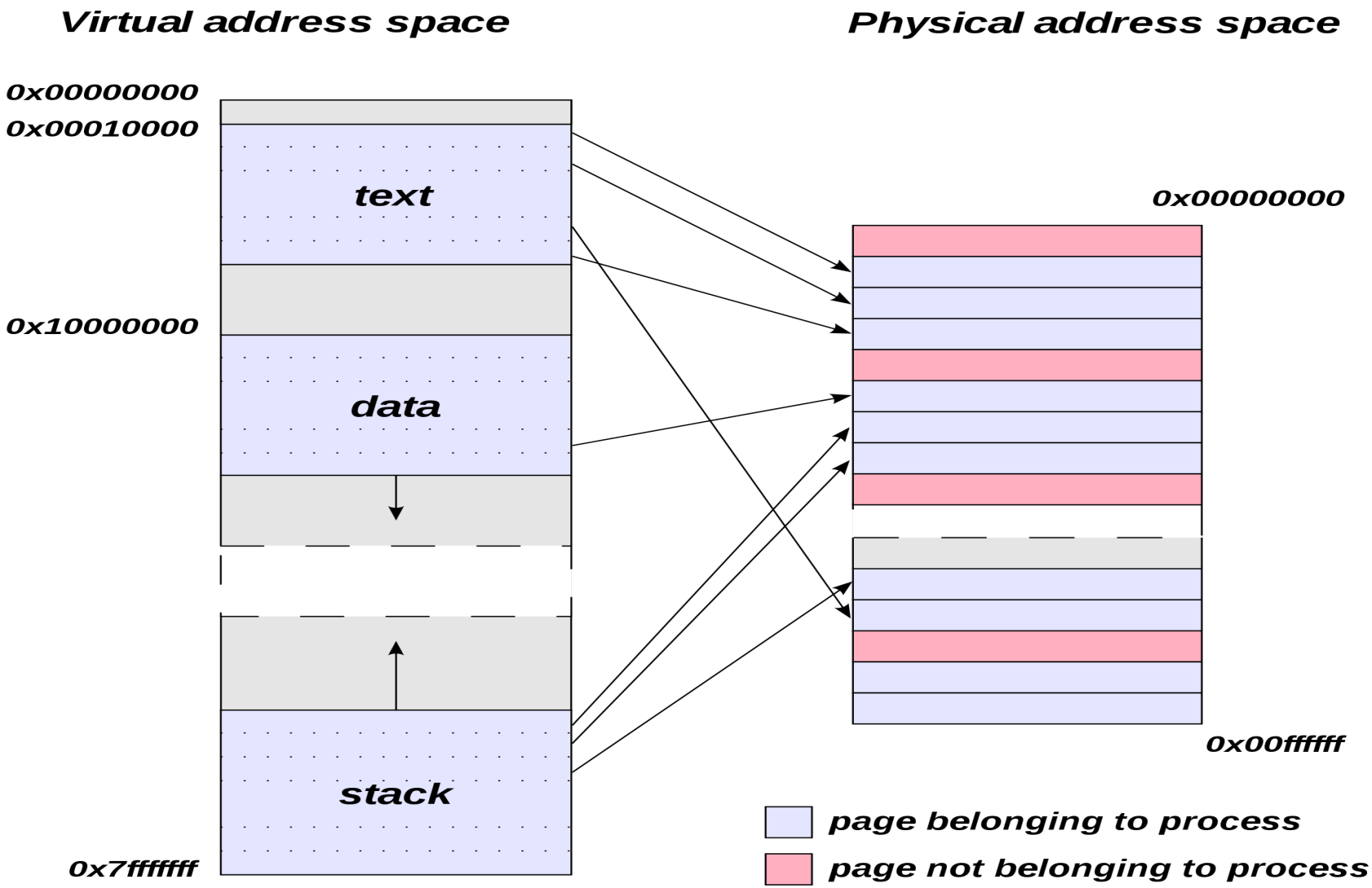
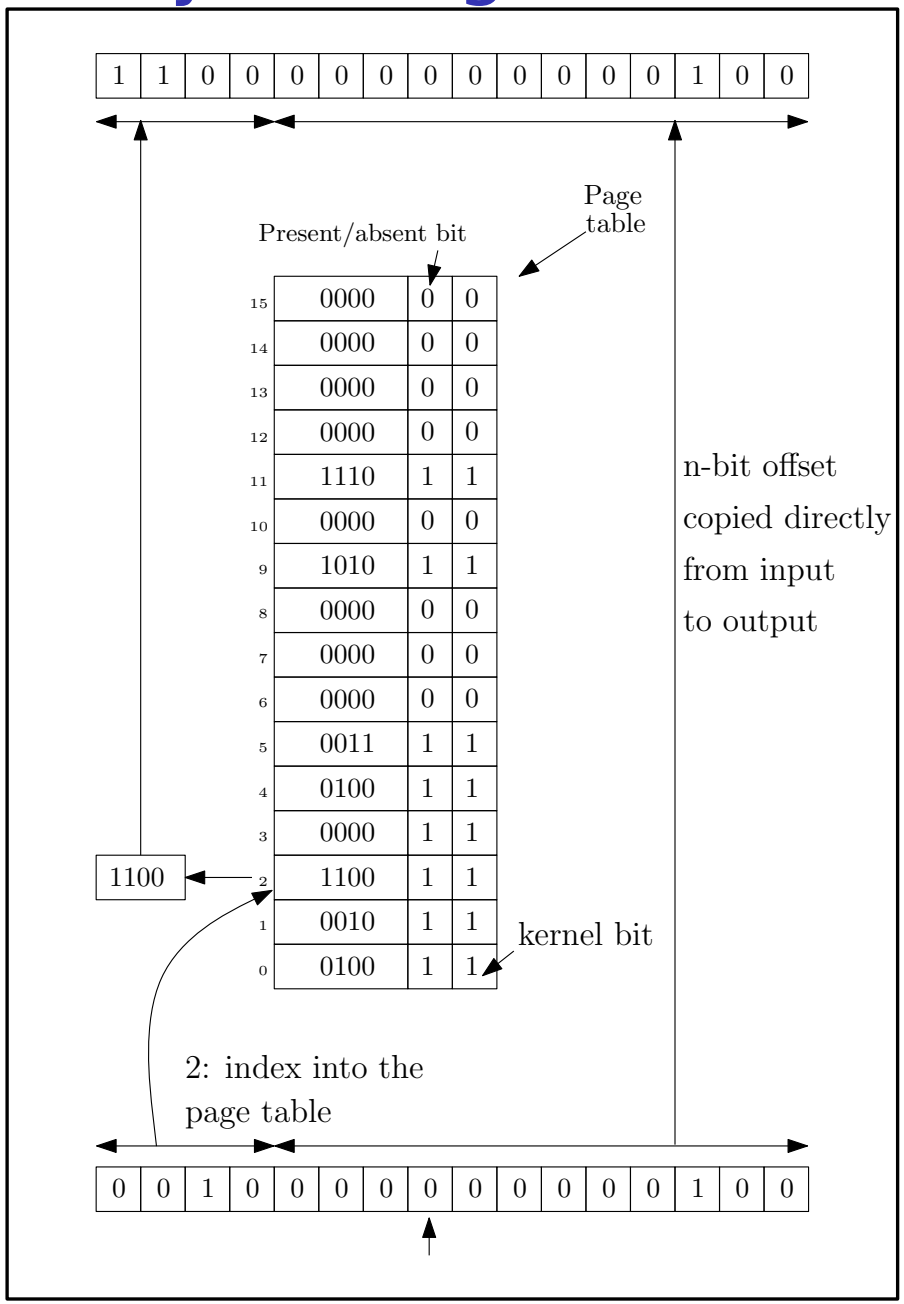


Image from Wikipedia — ©en:User:Dysprosia

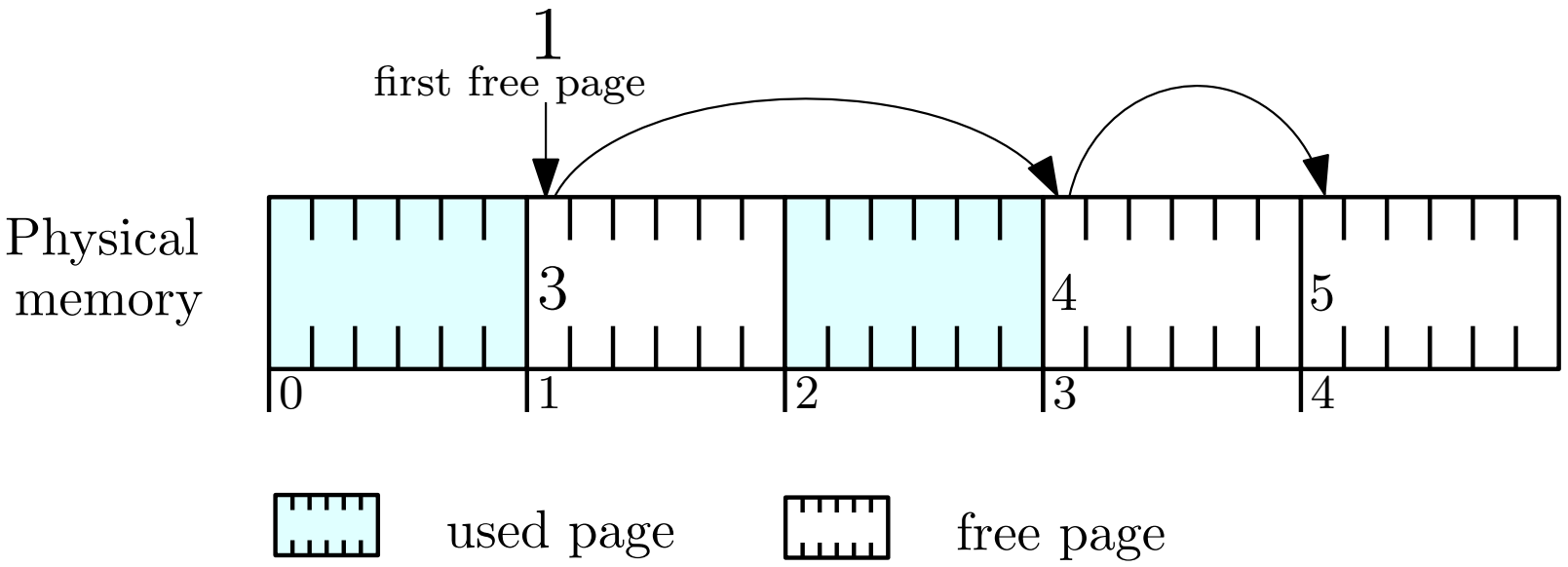
Memory Management Unit (MMU)



- The MMU is a hardware component.
- It translates virtual addresses into physical addresses.
- It is reconfigured when the running process changes.
- ⚠ It does not ensure by itself memory isolation between processes.

Memory management by the microkernel

We apply our approach on a simple but realistic memory manager.



The first word of a free memory page is used as a pointer to the next free memory page (linked list).

- Allocated page: taken at the beginning of the list
- Freed page: put back at the beginning of the list

Undefined hardware behavior

- Examples:
  - Accessing a reserved flag in a register
  - Using an unspecified assembly instruction
  - Accessing an out-of-bound physical address
  - Store a value of a certain type and access it as if it had another type
- An undefined hardware behavior can cause vulnerabilities.
- They must be dealt with in formal proof of security.

Outline

Modeling hardware and software

- Gallina, the specification language of the proof assistant Coq, is a purely functional language.
- But, in order model hardware (MMU, kernel mode) and software (microkernel), we need imperative features:
  - updatable state,
  - undefined behaviors, and
  - halting.
- We model those imperative features with a monad.

The H monad (1/2)

- A term of type `M A` is called a computation:
  It depends on the current state and can change it.

```
Definition M (A :Type) : Type :=
  state → result (A * state).

Inductive result (X : Type) : Type :=
| val : X → result X
| hlt : result X
| undef : result X.
```

- In our model the are three kinds of computations:
  - 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.
- `M` is an abstract datatype.

The H monad (2/2)

The H monad is equipped with 6 primitives:

```
(* trivial computation *)
Definition ret {A : Type}(a : A) : M A := ...

(* sequence *)
Definition bind {A B : Type} (m : M A) (f : A → M B) : M B := ...

(* writing the state *)
Definition put (s : state) : M unit := ...

(* reading the state *)
Definition get : M state := ...

(* halting *)
Definition halt {A : Type} : M A := ...

(* undefined behavior *)
Definition undefined {A : Type} : M A := ...
```

Hoare logic on top of the H monad (1/2)

We define a variant of Hoare logic.

```
Definition hoare_triple {A : Type}
  (P : state → Prop) (c : M A) (Q : A → state → Prop) : Prop :=
  ...

Notation "{{ P }} c {{ Q }}" := (hoare_triple P c Q)
```

"When the precondition `P` is met, executing the command `c` establishes the postcondition `Q`."

- `P` is a unary predicate on the starting state;
- `c` is a computation;
- `Q` is a binary predicate on the returned value and the resulting state.

Hoare logic on top of the H monad (2/2)

```
Definition hoare_triple {A : Type}
  (P : state → Prop) (c : M A) (Q : A → state → Prop) : Prop :=
  ∀ s, P s → match c s with
    | val (a, s') ⇒ Q a s'
    | hlt         ⇒ True
    | undef       ⇒ False
    end.
```

If a a triple holds then:

- either the postcondition holds or the computer halts; and
- there is no undefined behavior.

Weakest precondition

```
Definition wp
  {A : Type} (Q : A → state → Prop) (c : M A) : state → Prop :=
  fun s ⇒ match c s with
  | val (a, s') ⇒ Q a s'
  | hlt         ⇒ True
  | undef       ⇒ False
  end.

Lemma wp_is_precondition
  (A : Type) (Q : A → state → Prop) (c : M A) :
  {{ wp Q c }} c {{ Q }}.

Lemma wp_is_weakest_precondition (A : Type)
  (P : state → Prop) (Q : A → state → Prop) (c : M A) :
  {{ P }} c {{ Q }} → ∀ s, P s → (wp Q c) s.
```

Outline

Memory isolation (1/2)

- not from the point of view of information flow
- but at the lower level of page table management
- A state is **isolated** iff, for any two distinct processes P_1 and P_2, any page used by P_1 is not used by P_2.
  - By *pages used by a process* P_i, we mean the pages referenced in its page table $ptp(P_i)$ and the page $ptp(P_i)$ itself.
  - By *two distinct processes* P_1 *and* P_2, we mean $ptp(P_1) \neq ptp(P_2)$
- Our goal is to show that this property is preserved.

Memory isolation (2/2)

- We would be satisfied if we could prove the following triple for each command c:

```
{{ fun s ⇒ Isolated s }} c {{ fun a s' ⇒ Isolated s' }}
```

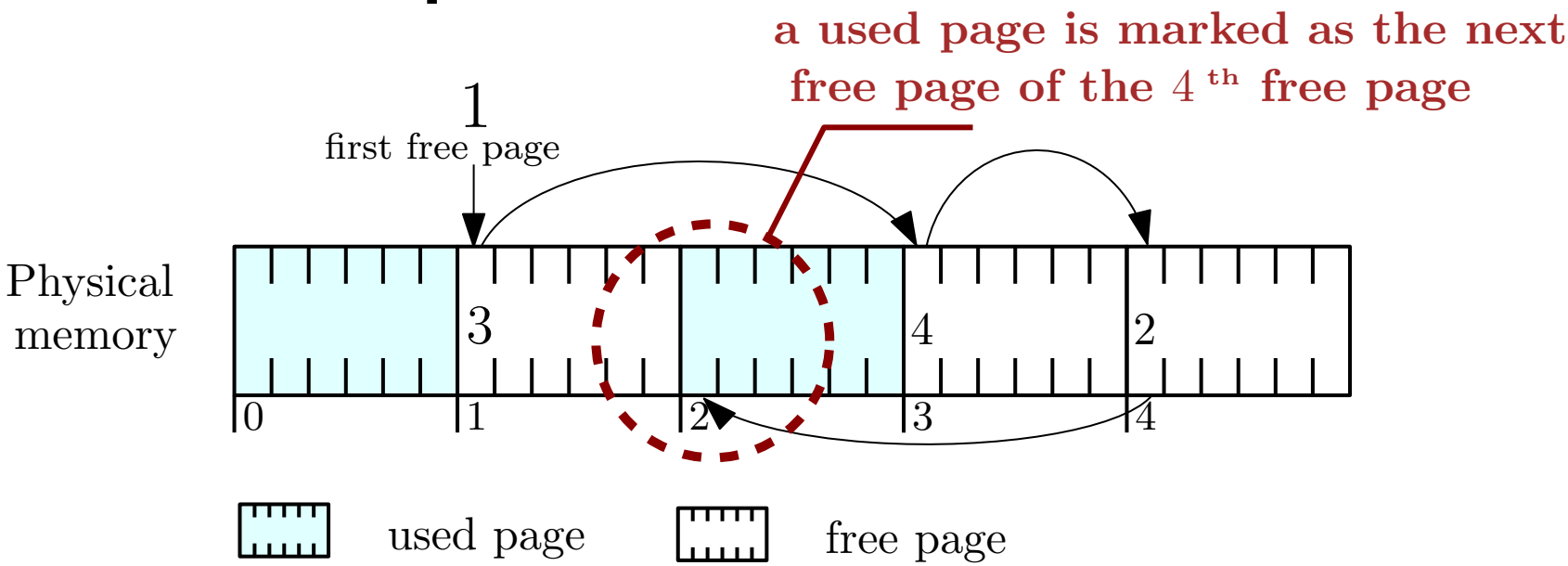
- But it is false in general:
  - The precondition must be strenghened with consistency properties.
  - Those consistency properties must also be preserved

```
{{ fun s    ⇒ Isolated s ∧ Consistent s }}
c
{{ fun a s' ⇒ Isolated s' ∧ Consistent s' }}
```

All marked-free pages should be really free

Without this consistency property, we cannot prove that the subroutine for allocating a page preserves isolation.

Counterexample:

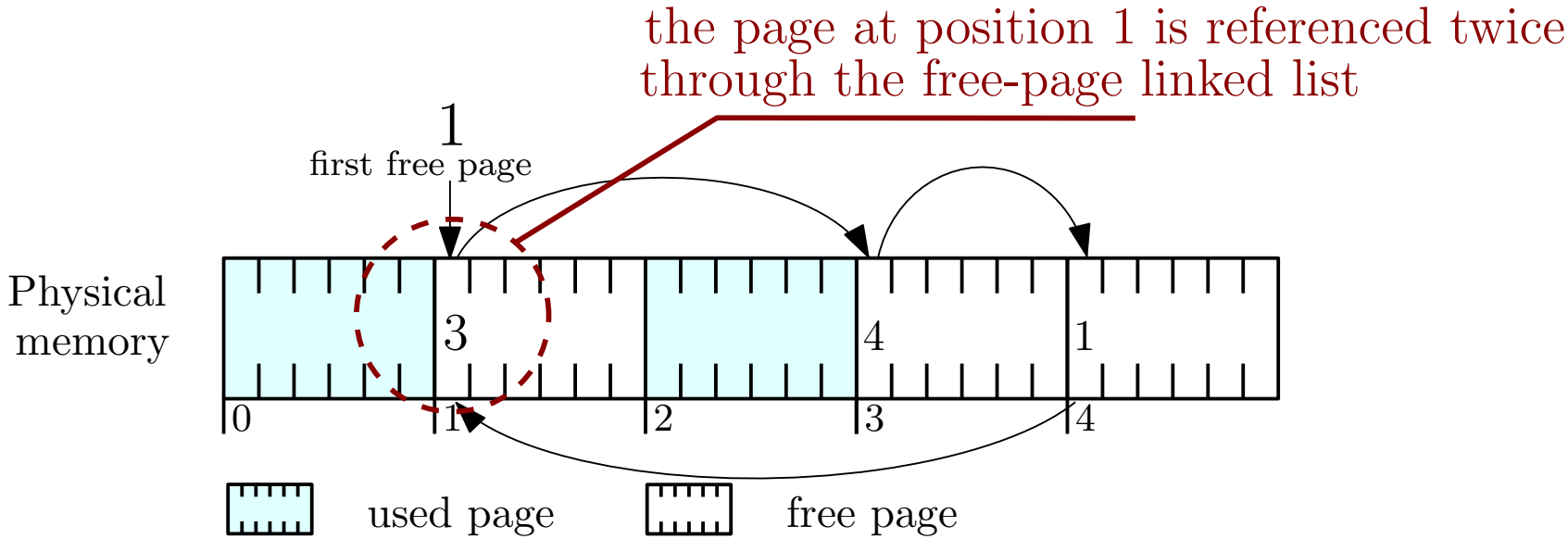


An already allocated page could be re-allocated to a different process.

No cycle in free-pages list

Without this consistency property, we cannot prove that the subroutine for allocating a page preserves isolation.

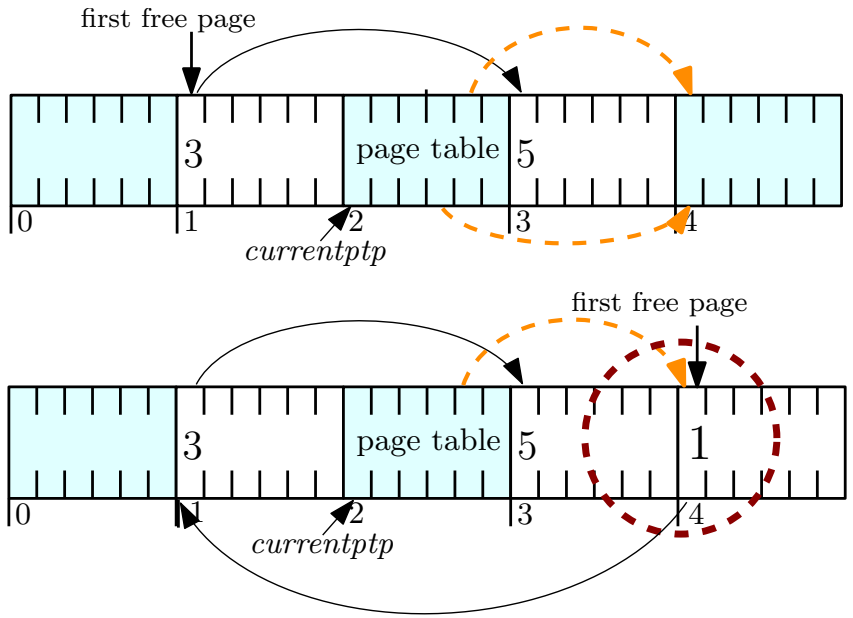
Counterexample:



The same page could be allocated to different processes.

No duplication in process used pages

- For any process, all its used pages must be referenced only once in its page table.
- **Counterexample:**



The subroutine that deallocates a page would only deallocate the first one.

- **Alternative:** Have the subroutine scan for all the references of the page to be deallocated

The current page table is of a process

- The number $currentptp(s)$ of the physical page storing the page table of the current process must be the ptp of one of the runnable processes.

- **counterexample:** The scheduler would not preserve isolation when it put the current process at the end of its queue.

Page 0 is never used or marked-free

- 0 is used to mark empty entries in page tables.
- **Counterexample:** The page 0 would appear to be shared by many processes, thus breaking isolation.
- Other pages must be neither used nor marked-free
  - to isolate some part of the memory from all processes
  - to store the code of the microkernel and its data

Physical memory large enough

- All physical addresses must exist.
- **Counterexample:** If a virtual address were mapped to a physical address that does not exist, then it would cause an undefined hardware behavior.

  ⚠ This would be a vulnerability.

Outline

Pip: a minimal kernel

- Here, we verify an implementation, not a model.

  source code in Gallina (the language of Coq)

- The TCB is minimal: smaller than an exokernel.
  - Scheduling and IPC are pushed into user mode.
  - Multiplexing is also pushed into user mode.
  - Kernel mode is only for:
    - multi-level MMU configuration,
    - catching and forwarding interruptions.
  - Pip does not provide any hardware abstraction.

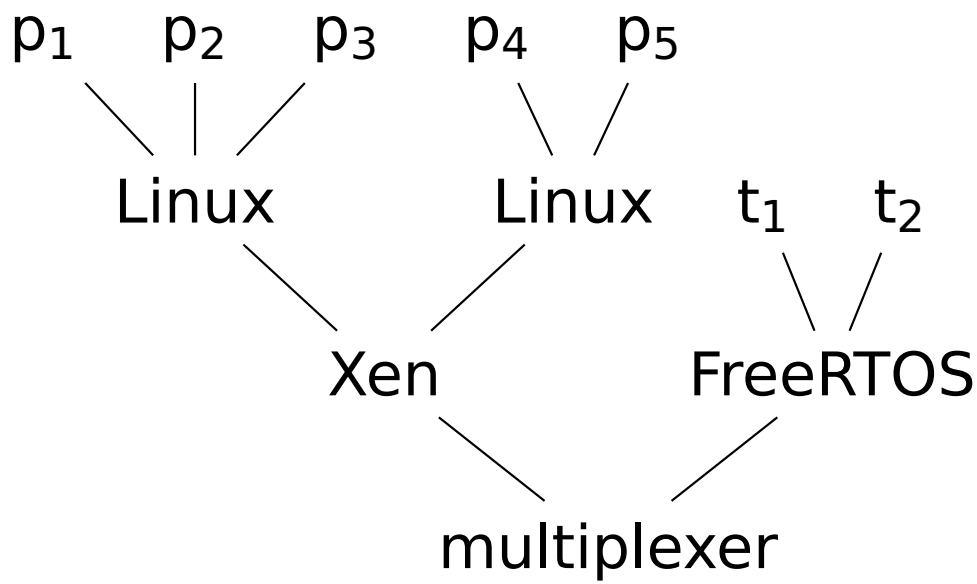
    They are provided by a user-level library.

Outline

Partitions tree (1/2)

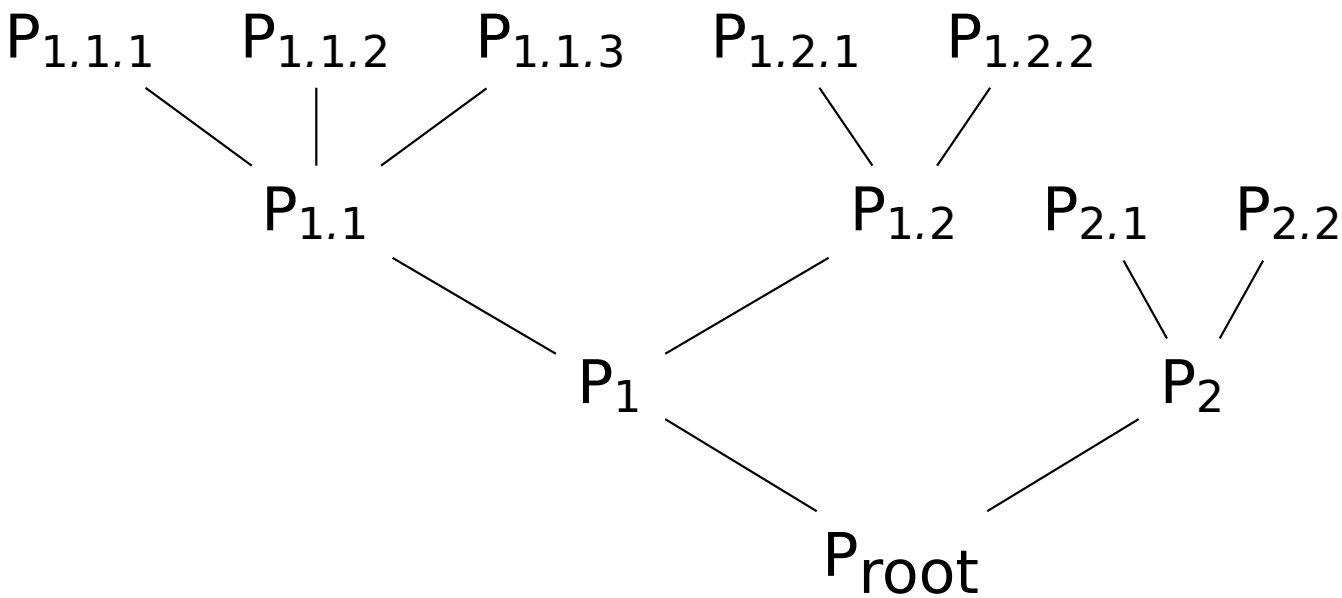
The memory is organized into hierarchical partitions.

Example

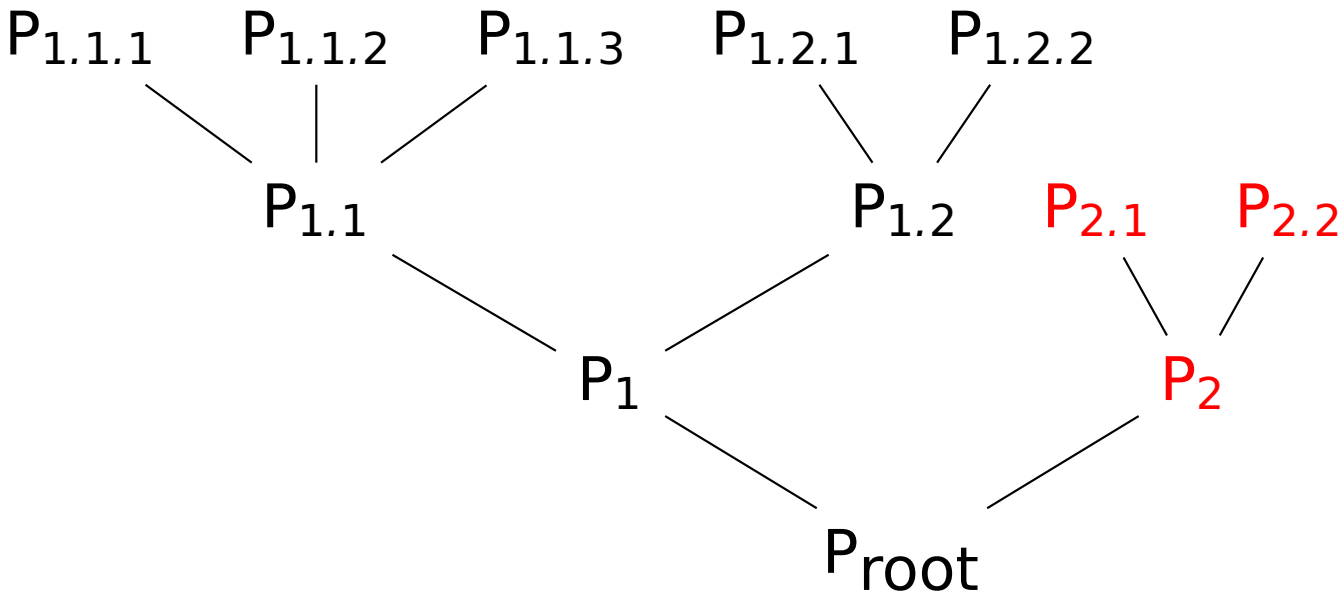


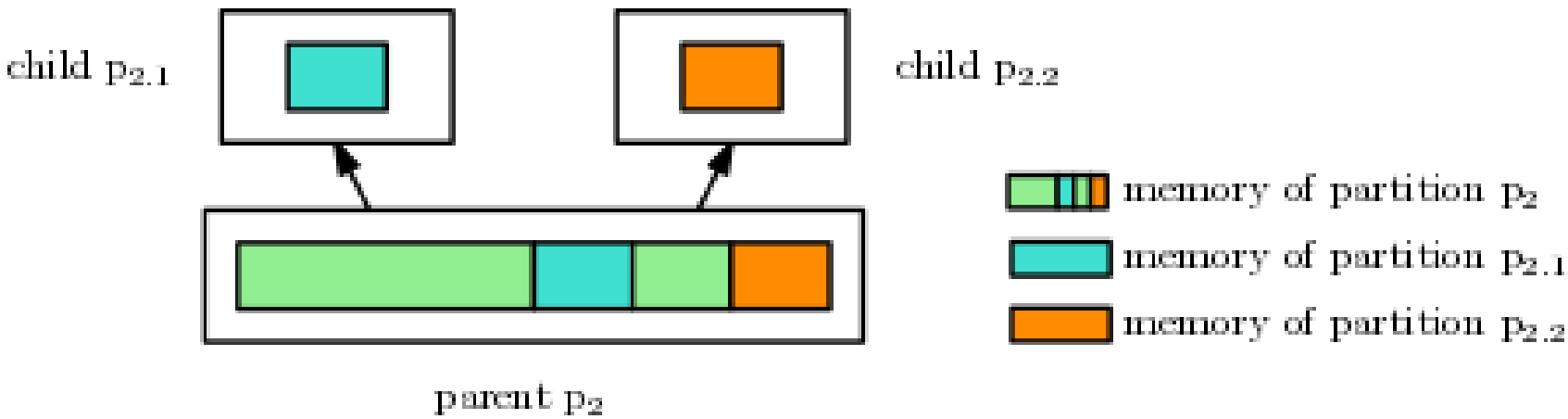
Partitions tree (2/2)

Pip does not know what is in each partition.



Horizontal isolation and vertical sharing





Horizontal isolation

Different children have distinct used pages.

```
Definition partitionsIsolation s : Prop :=

∀ parent child1 child2 : page,

  In parent (getPartitions root s) →

  In child1 (getChildren parent s) →

  In child2 (getChildren parent s) →

  child1 <> child2 →

  disjoint (getUsedPages child1 s) (getUsedPages child2 s).
```

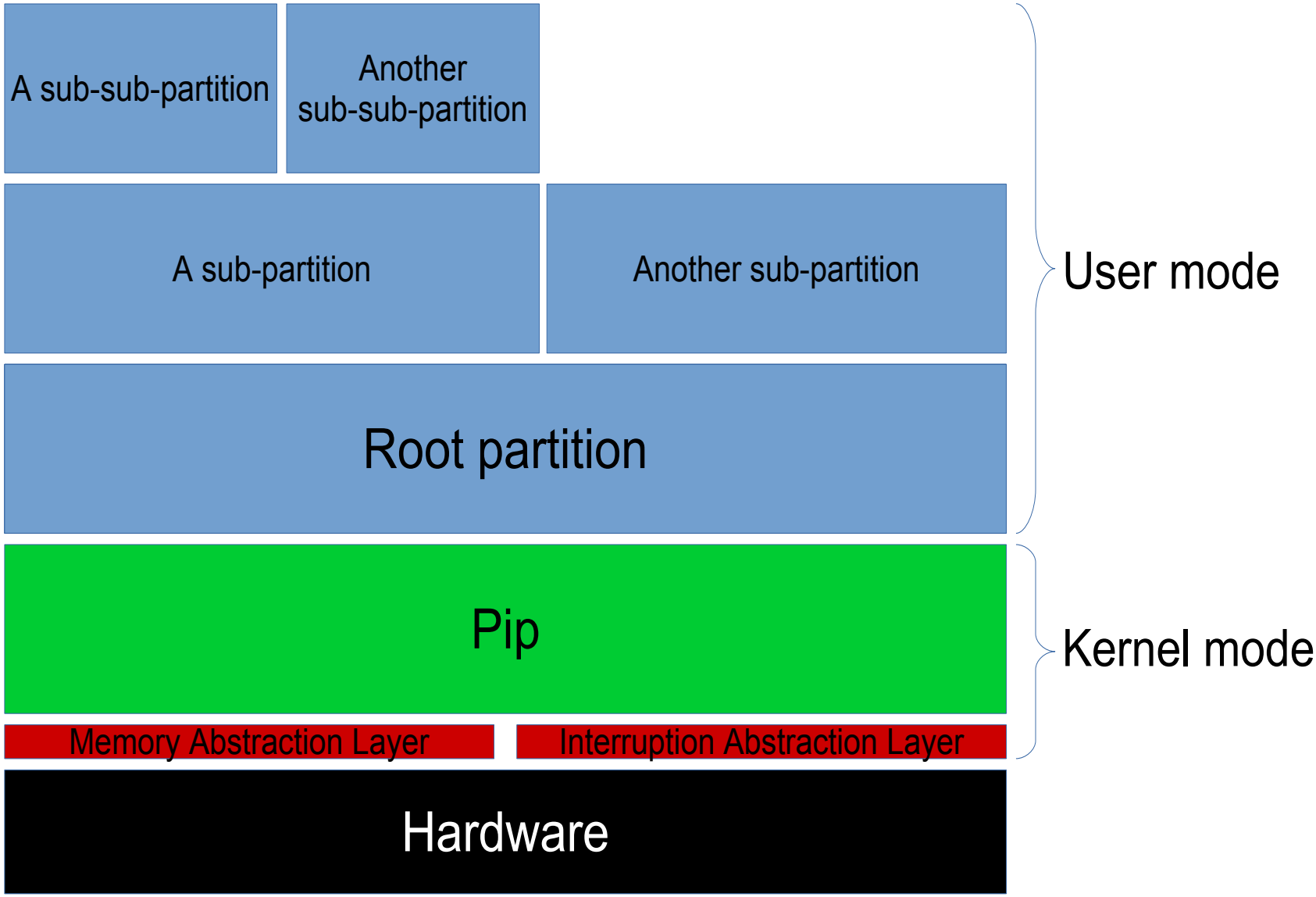
Vertical sharing

All the used pages (configuration tables and mapped pages) of a partition are mapped into its parent partition.

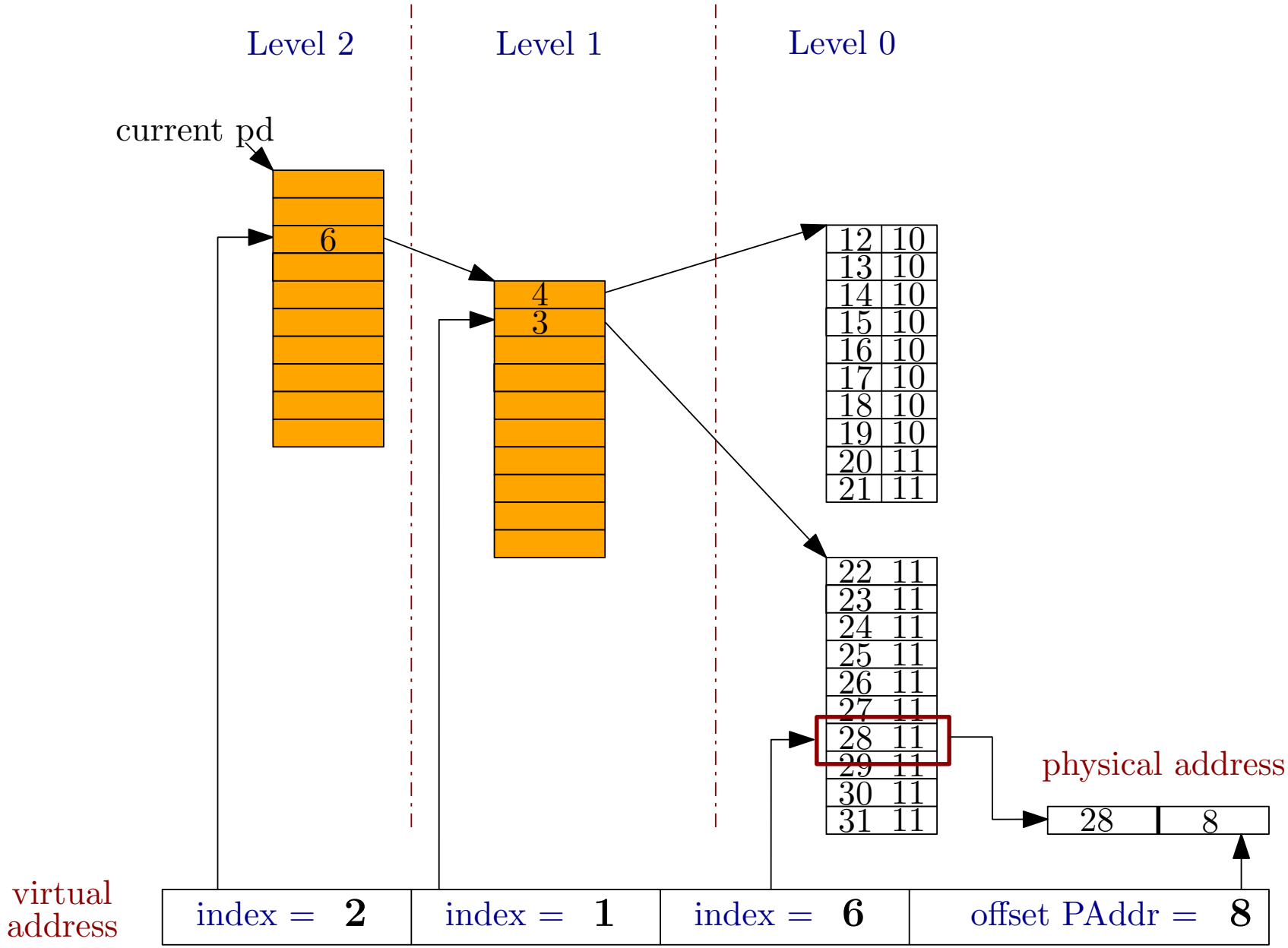
```
Definition verticalSharing s : Prop :=

∀ parent child : page,

  In parent (getPartitions root s) →

  In child (getChildren parent s) →

  incl (getUsedPages child s) (getMappedPages parent s).
```

Outline

Software layers



Example: An MMU with three-level page table



Pip system calls for memory management

`createPartition`	create a partition
`removePartition`	delete a partition
`addVaddr`	map an address
`removeVaddr`	remove a mapping
`pageCount`	return the number of indirections to map an address
`prepare`	add the indirections to map an address
`collect`	delete all empty indirections

Pip system calls for control flow

Pip redirects:

- a sofware interrupt to the parent of the caller,
- a hardware interrupt to the root partition.

`dispatch`	give control to another partition
`resume`	return control to another partition

Internal data structure

- The MMU pages tables (used by Pip and MMU)

- Two shadow MMUs (used only by Pip):
  - same structure than MMU page tables, but different data at leafs.
  - Shadow 1: flags related to vertical sharing
    ⇒ To ensure horizontal isolation
  - Shadow 2: virtual address in the parent of a page
    ⇒ To optimize `removeVaddr`

- A linked list of pairs of:
  - physical address of a configuration page, and
  - its virtual address in the parent partition.

  ⇒ To optimize `collect`

Conclusions

- Part 1: A preliminary study on microkernels
  - A formal model of a hardware architecture
  - A formal model of microkernels
  - A proof that they ensure memory isolation
- Part 2: Lesson learned: we design the Pip kernel
  - smallest possible TCB:
    even multiplexing is pushed outside the kernel.
  - Source code in Gallina (the language of Coq)
  - Same proof technique as in the preliminary study