Comprendre la Virtualisation

Julien Sopena - julien.sopena@lip6.fr
Gauthier Voron - gauthier.voron@lip6.fr

Virtualisation système : pourquoi

- La virtualisation système est une technologie largement répandue
  - Cloud computing (location de serveurs, Amazon EC2, ...)
  - Systèmes embarqués (séparation temps réel / multimédia)
  - Plateformes de test (travis, model checking contest)
- La virtualisation système impacte les performances
  - Vitesse d'exécution CPU (instructions par cycle)
  - Vitesse d'entrées / sorties (disque, réseau)
  - Effets mémoire (pollution de cache, effets NUMA)
- Comprendre la virtualisation et ses effets permet
  - La conception de systèmes virtualisation efficaces
  - Un usage raisonné de la virtualisation

Virtualisation système : quoi

- La virtualisation est le procédé qui consiste à exposer au logiciel une ressource virtuelle différente de la ressource physique
  - Mémoire virtuelle $\rightarrow$ adresses virtuelles $\neq$ adresses physiques
  - Virtualisation système $\rightarrow$ machine virtuelle $\neq$ machine physique

- Usages d'une machine virtuelle
  - Simuler du nouveau matériel $\rightarrow$ conception de puce
  - Porter des logiciels sur différentes architectures $\rightarrow$ émulateur gameboy
  - Partage de ressources $\rightarrow$ exécution simultanée de Linux et Windows
  - Isolation de services $\rightarrow$ exécution simultanée de plusieurs Linux

Virtualisation système : quoi

- La virtualisation est le procédé qui consiste à exposer au logiciel une ressource virtuelle différente de la ressource physique
  - Mémoire virtuelle $\rightarrow$ adresses virtuelles $\neq$ adresses physiques
  - Virtualisation système $\rightarrow$ machine virtuelle $\neq$ machine physique

- Usages d'une machine virtuelle
  - Simuler du nouveau matériel $\rightarrow$ conception de puce
  - Porter des logiciels sur différentes architectures $\rightarrow$ émulateur gameboy
  - Partage de ressources $\rightarrow$ exécution simultanée de Linux et Windows
  - Isolation de services $\rightarrow$ exécution simultanée de plusieurs Linux

Plan du cours

1. Virtualisation d'instructions
   - Simulation *cycle accurate* et émulation
   - Émulation d'instructions et état virtuel
   - *Dynamic Binary Translation* et *basic blocks*

2. Virtualisation de ressources
   - Virtualisation mémoire et *shadowing*
   - Paravirtualisation et interface matérielle

3. Virtualisation comme isolation
   - Exécution directe et interception
   - Support materiel à la virtualisation

Plan du cours

1. Virtualisation d'instructions
   - Simulation *cycle accurate* et émulation
   - Émulation d'instructions et état virtuel
   - *Dynamic Binary Translation* et *basic blocks*
2. Virtualisation de ressources
   - Virtualisation mémoire et *shadowing*
   - Paravirtualisation et interface matérielle
3. Virtualisation comme isolation
   - Exécution directe et interception
   - Support materiel à la virtualisation

Simulation *cycle accurate*

- Usages d'une machine virtuelle
  - Simuler du nouveau matériel → conception de puce
  - Porter des logiciels sur différentes architectures → émulateur gameboy
  - Partage de ressources → exécution simultanée de Linux et Windows
  - Isolation de services → exécution simultanée de plusieurs Linux
- Objectif : simuler une machine virtuelle
- Solution : simuler chaque composant d'une machine virtuelle

```
#include "systemc.h"

SC_MODULE(dff) {
    sc_in<bool> din;
    sc_in<bool> clock;
    sc_out<bool> dout;

    void sample() {
        dout = din;
    };

    SC_CTOR(dff) {
        SC_METHOD(sample);
        sensitive << clock.pos();
    }
};
```

- Langages / bibliothèques dédiées
  - VHDL : *domain specific language*
  - SystemC : bibliothèque C++
- Simulation de chaque
  - signal entrant
  - signal sortant
  - tick d'horloge
- Simulation précise au cycle près

Simulation *cycle accurate*

- Usages d'une machine virtuelle
  - Simuler du nouveau matériel → conception de puce
  - Porter des logiciels sur différentes architectures → émulateur gameboy
  - Partage de ressources → exécution simultanée de Linux et Windows
  - Isolation de services → exécution simultanée de plusieurs Linux
- Objectif : simuler une machine virtuelle
- Solution : simuler chaque composant d'une machine virtuelle

```
#include "systemc.h"

SC_MODULE(dff) {
    sc_in<bool> din;
    sc_in<bool> clock;
    sc_out<bool> dout;

    void sample() {
        dout = din;
    };

    SC_CTOR(dff) {
        SC_METHOD(sample);
        sensitive << clock.pos();
    }
};
```

- Langages / bibliothèques dédiées
  - VHDL : *domain specific language*
  - SystemC : bibliothèque C++
- Simulation de chaque
  - signal entrant
  - signal sortant
  - tick d'horloge
- Simulation précise au cycle près

Usages de la simulation *cycle accurate*

- La simulation *cycle accurate* permet une description arbitrairement précise du matériel simulé

- Cette précision permet une observation fine des exécutions
  - Estimation des performances au cycle près
  - Interactions logiciel/matériel observables et reproductibles
  - Test de nouveau matériel avant mise en production
  - Expérience sur du matériel expérimental / difficile d'accès

- Beaucoup de calculs nécessaires $\rightarrow$ simulation lente (facteur x100)
- Implique de connaître l'implémentation des circuits testés $\rightarrow$ pas toujours possible

De la simulation à l'émulation

- Usages d'une machine virtuelle
  - Simuler du nouveau matériel → conception de puce
  - Porter des logiciels sur différentes architectures → émulateur gameboy
  - Partage de ressources → exécution simultanée de Linux et Windows
  - Isolation de services → exécution simultanée de plusieurs Linux

- Objectif : simuler une machine virtuelle

De la simulation à l'émulation

- Usages d'une machine virtuelle
  - Simuler du nouveau matériel → conception de puce
  - Porter des logiciels sur différentes architectures → émulateur gameboy
  - Partage de ressources → exécution simultanée de Linux et Windows
  - Isolation de services → exécution simultanée de plusieurs Linux

- Objectif : simuler l'interface d'une machine virtuelle
  - Pas besoin de reproduire le fonctionnement d'une machine physique
  - On veut juste une interface avec un comportement similaire

De la simulation à l'émulation

- Usages d'une machine virtuelle
  - Simuler du nouveau matériel → conception de puce
  - Porter des logiciels sur différentes architectures → émulateur gameboy
  - Partage de ressources → exécution simultanée de Linux et Windows
  - Isolation de services → exécution simultanée de plusieurs Linux

- Objectif : simuler l'interface d'une machine virtuelle
  - Pas besoin de reproduire le fonctionnement d'une machine physique
  - On veut juste une interface avec un comportement similaire

- Solution : émuler les instructions du système virtualisé
  - Du point de vue du système virtualisé, les instructions donnent les mêmes résultats que sur une machine physique

- Défi
  - Changement de jeu d'instruction
  - Isolation des contextes

- Exemple

```
mov     %rax, %rcx
```

Copier le contenu du registre `rax` dans le registre `rcx`

De la simulation à l'émulation

- Usages d'une machine virtuelle
  - Simuler du nouveau matériel → conception de puce
  - Porter des logiciels sur différentes architectures → émulateur gameboy
  - Partage de ressources → exécution simultanée de Linux et Windows
  - Isolation de services → exécution simultanée de plusieurs Linux

- Objectif : simuler l'interface d'une machine virtuelle
  - Pas besoin de reproduire le fonctionnement d'une machine physique
  - On veut juste une interface avec un comportement similaire

- Solution : émuler les instructions du système virtualisé
  - Du point de vue du système virtualisé, les instructions donnent les mêmes résultats que sur une machine physique

- Défi
  - Changement de jeu d'instruction
  - Isolation des contextes
  - Exécution d'instructions privilégiées

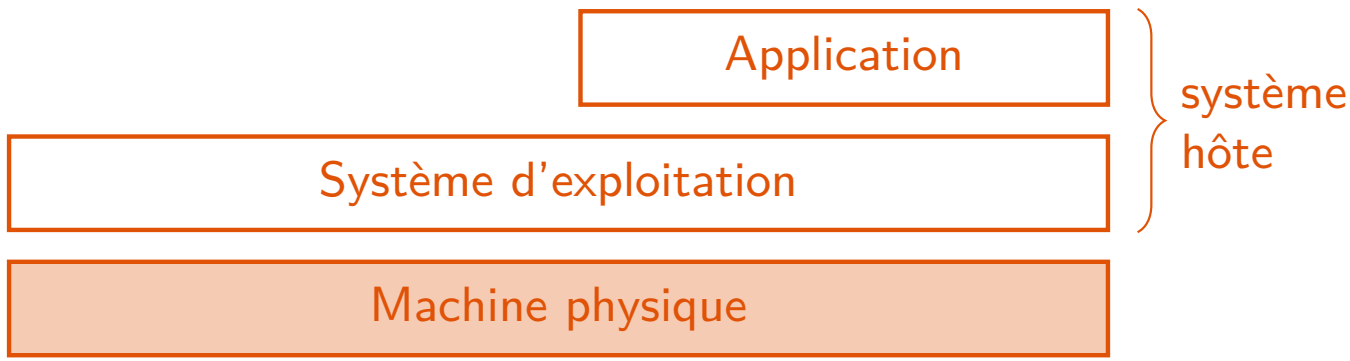
- Exemple

```
cli
```

Masquer les interruptions

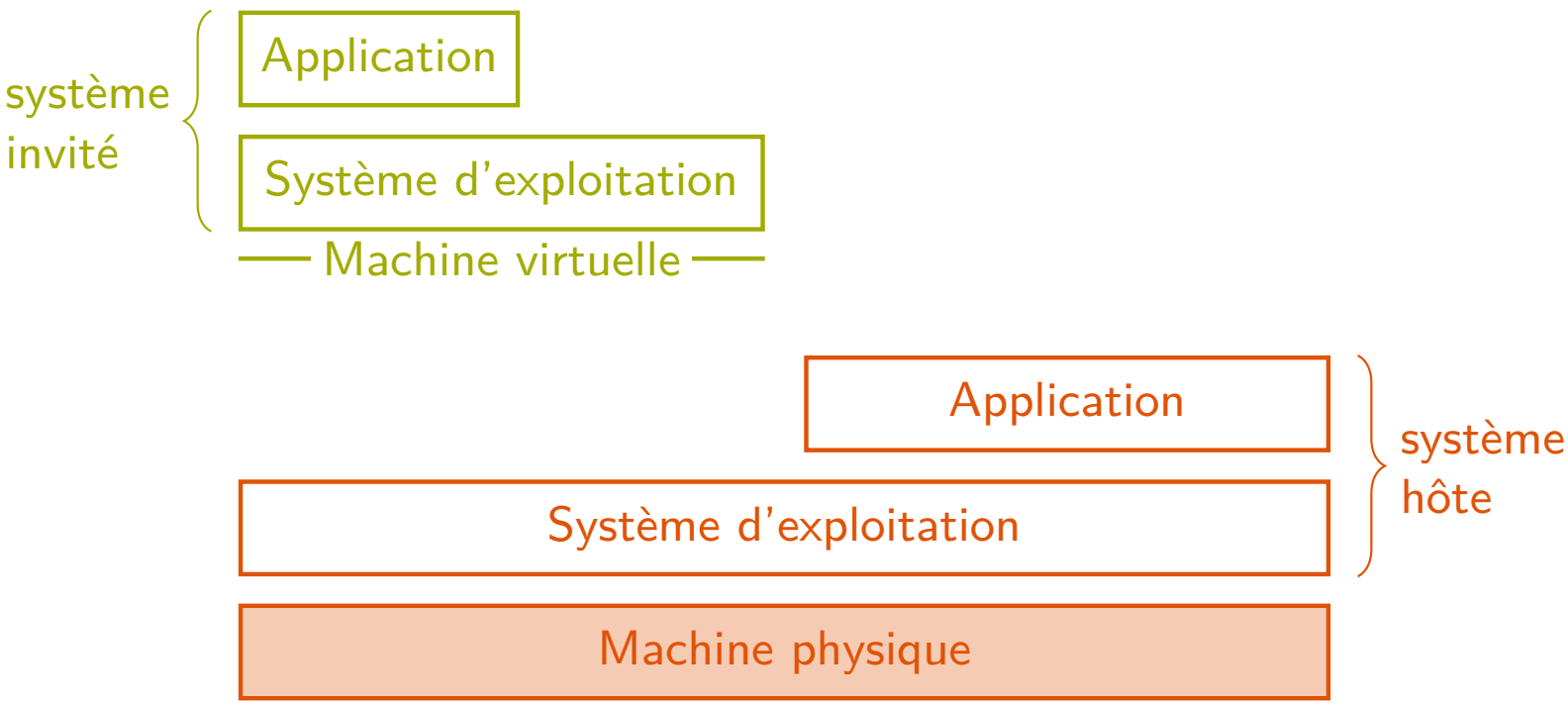
Système invité et moniteur de machines virtuelles

- Le système hôte est l'ensemble des programmes qui utilisent directement les ressources de la machine



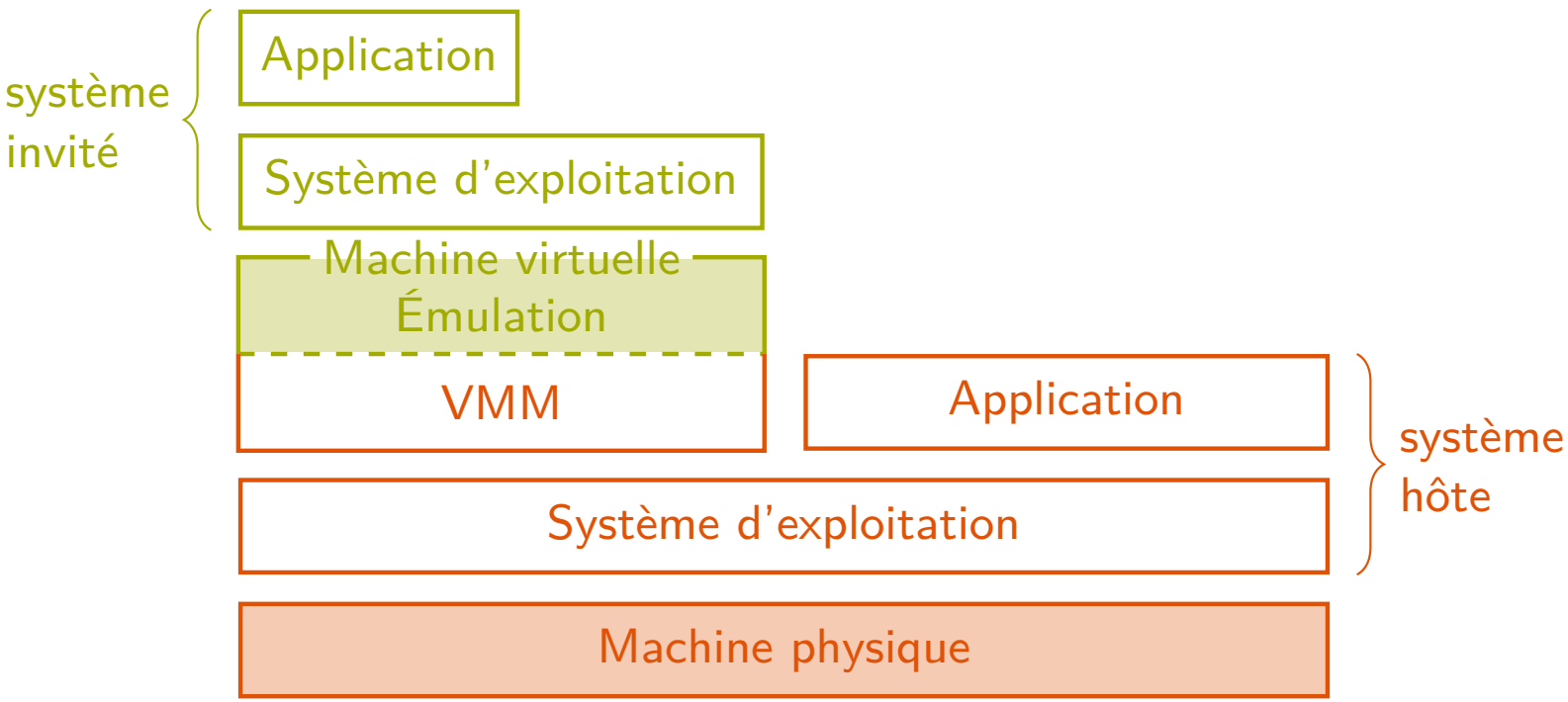
Système invité et moniteur de machines virtuelles

- Le système hôte est l'ensemble des programmes qui utilisent directement les ressources de la machine
- Un système invité est l'ensemble des programmes qui utilisent les ressources de la machine via l'interface d'une machine virtuelle



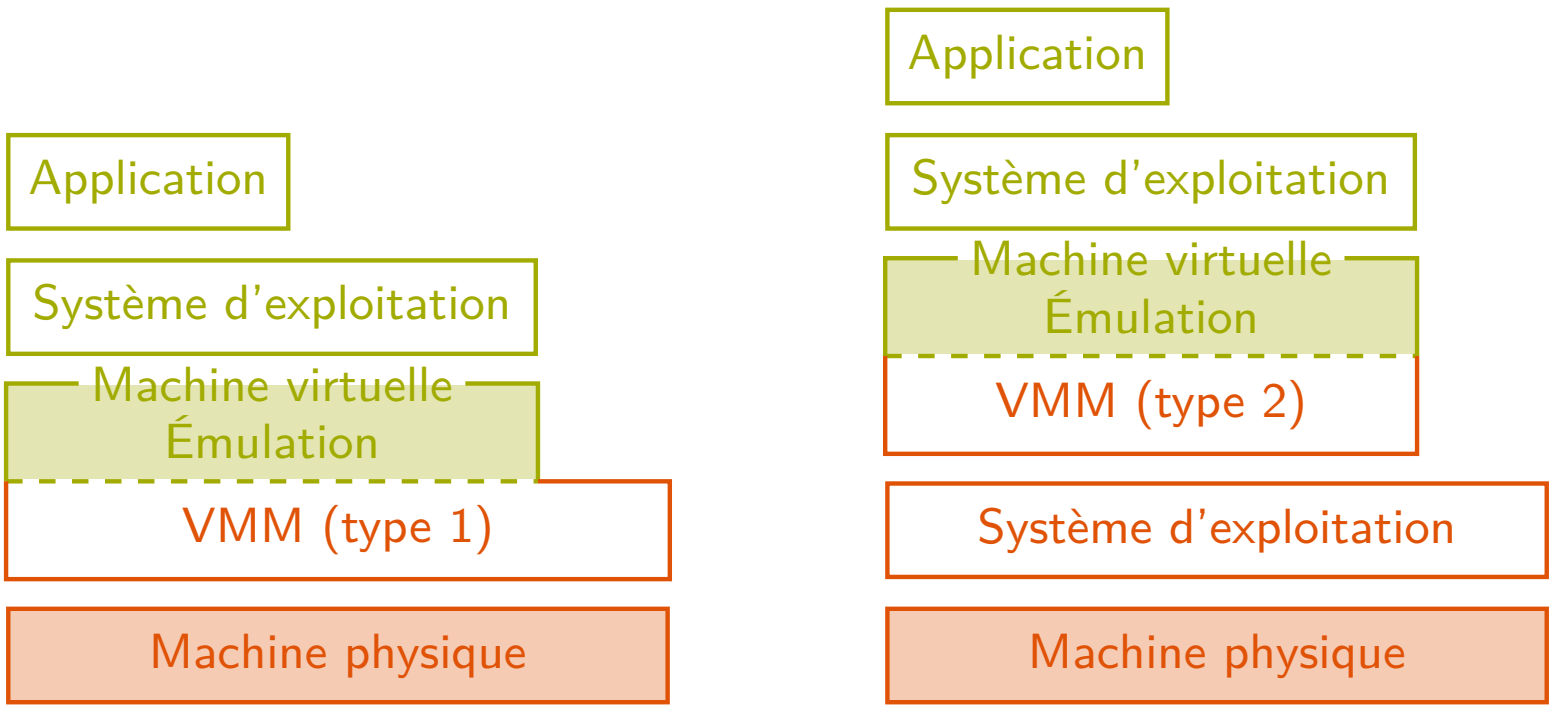
Système invité et moniteur de machines virtuelles

- Le système hôte est l'ensemble des programmes qui utilisent directement les ressources de la machine
- Un système invité est l'ensemble des programmes qui utilisent les ressources de la machine via l'interface d'une machine virtuelle
- Un moniteur de machines virtuelles (VMM) est un logiciel qui présente une interface de machine virtuelle aux systèmes invités



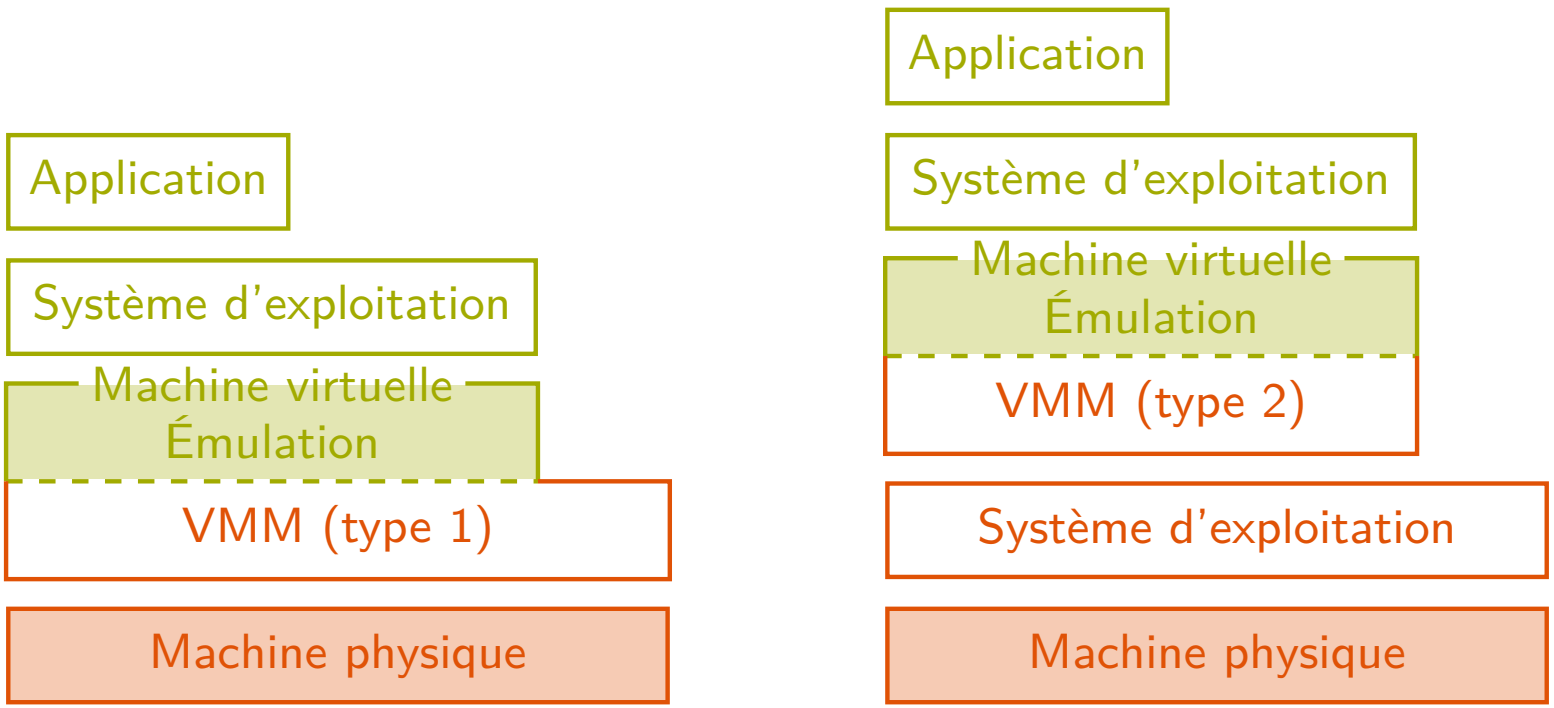
Système invité et moniteur de machines virtuelles

- Un moniteur de type 1 s'exécute en mode privilégié, il est l'unique programme hôte → exemple : VMware ESX
- Un Moniteur de type 2 s'exécute en mode utilisateur, il est en concurrence avec d'autres programmes hôtes → exemple : VirtualBox



Système invité et moniteur de machines virtuelles

- Un moniteur de type 1 s'exécute en mode privilégié, il est l'unique programme hôte → exemple : VMware ESX
- Un Moniteur de type 2 s'exécute en mode utilisateur, il est en concurrence avec d'autres programmes hôtes → exemple : VirtualBox
- Question : quel est le type de Qemu-KVM ?



Simulation *cycle accurate* et émulation : résumé

- La simulation *cycle accurate* reproduit l'implémentation d'une machine physique en simulant le comportement de chaque composant de la machine virtuelle
  - Permet de tester de nouveaux composants matériels
  - Permet la mesure précise des performances d'un programme
  - Plus lente d'un facteur 100 que l'exécution sur machine physique

- L'émulation est la méthode qui consiste à présenter l'interface d'une machine à un système logiciel sans en reproduire l'implémentation
  - Le moniteur de machines virtuelles (VMM) est un logiciel du système hôte qui présente une interface de machine à chaque système invité
  - Un VMM de type 1 s'exécute en mode privilégié
  - Un VMM de type 2 s'exécute en mode utilisateur

Plan du cours

1. Virtualisation d'instructions
   - Simulation *cycle accurate* et émulation
   - Émulation d'instructions et état virtuel
   - *Dynamic Binary Translation* et *basic blocks*
2. Virtualisation de ressources
   - Virtualisation mémoire et *shadowing*
   - Paravirtualisation et interface matérielle
3. Virtualisation comme isolation
   - Exécution directe et interception
   - Support materiel à la virtualisation

Émulation d'instructions

- Objectif : la machine virtuelle exécute les instructions du système invité comme le ferait une machine physique
- Solution : le VMM décode et interprète chaque instruction du système invité

Code système invité

```
...
48 89 c1
...
```

Code VMM : main_loop()

```
while (TRUE) {
    bin = guest.next_instruction();
    inst = decode_instruction(bin);
    guest.execute_instruction(inst);
}
```

Émulation d'instructions

- Objectif : la machine virtuelle exécute les instructions du système invité comme le ferait une machine physique
- Solution : le VMM décode et interprète chaque instruction du système invité

Code système invité

```
...
48 89 c1
...
```

Code VMM : main_loop()

```
while (TRUE) {
    bin = guest.next_instruction();
    inst = decode_instruction(bin);
    guest.execute_instruction(inst);
}
```

Émulation d'instructions

- Objectif : la machine virtuelle exécute les instructions du système invité comme le ferait une machine physique
- Solution : le VMM décode et interprète chaque instruction du système invité

Code système invité

```
...
48 89 c1
...
```

Code VMM : main_loop()

```
while (TRUE) {
    bin = guest.next_instruction();
    inst = decode_instruction(bin);
    guest.execute_instruction(inst);
}
```

Émulation d'instructions

- Objectif : la machine virtuelle exécute les instructions du système invité comme le ferait une machine physique
- Solution : le VMM décode et interprète chaque instruction du système invité

Code système invité

```
...
48 89 c1
...
```

Code VMM : main_loop()

```
while (TRUE) {
    bin = guest.next_instruction();
    inst = decode_instruction(bin);
    guest.execute_instruction(inst);
}
```

- Instructions décodées logiciellement selon l'opcode de la machine virtuelle

Émulation d'instructions

- Objectif : la machine virtuelle exécute les instructions du système invité comme le ferait une machine physique
- Solution : le VMM décode et interprète chaque instruction du système invité

Code système invité

```
...
mov     %rax, %rcx
...
```

Code VMM : main_loop()

```
while (TRUE) {
    bin = guest.next_instruction();
    inst = decode_instruction(bin);
    guest.execute_instruction(inst);
}
```

- Instructions décodées logiciellement selon l'opcode de la machine virtuelle

Émulation d'instructions

- Objectif : la machine virtuelle exécute les instructions du système invité comme le ferait une machine physique
- Solution : le VMM décode et interprète chaque instruction du système invité

Code système invité

```
...
mov     %rax, %rcx
...
```

Code VMM : main_loop()

```
while (TRUE) {
    bin = guest.next_instruction();
    inst = decode_instruction(bin);
    guest.execute_instruction(inst);
}
```

- Instructions décodées logiciellement selon l'opcode de la machine virtuelle
- L'instruction décodée indique quelle opération effectuer

Émulation d'instructions

- Objectif : la machine virtuelle exécute les instructions du système invité comme le ferait une machine physique
- Solution : le VMM décode et interprète chaque instruction du système invité

Code système invité

```
...
mov     %rax, %rcx
...
```

Code VMM : `execute_mov()`

```
src = get_src_register(inst);
dest = get_dest_register(inst);
*dest = *src;
```

- Instructions décodées logiciellement selon l'opcode de la machine virtuelle
- L'instruction décodée indique quelle opération effectuer

Émulation d'instructions

- Objectif : la machine virtuelle exécute les instructions du système invité comme le ferait une machine physique
- Solution : le VMM décode et interprète chaque instruction du système invité

Code système invité

```
...
mov     %rax, %rcx
...
```

Code VMM : `execute_mov()`

```
src = get_src_register(inst);
dest = get_dest_register(inst);
*dest = *src;
```

- Instructions décodées logiciellement selon l'opcode de la machine virtuelle
- L'instruction décodée indique quelle opération effectuer

Émulation d'instructions

- Objectif : la machine virtuelle exécute les instructions du système invité comme le ferait une machine physique
- Solution : le VMM décode et interprète chaque instruction du système invité

Code système invité

```
...
mov     %rax, %rcx
...
```

Code VMM : `execute_mov()`

```
src = get_src_register(inst);
dest = get_dest_register(inst);
*dest = *src;
```

- Instructions décodées logiciellement selon l'opcode de la machine virtuelle
- L'instruction décodée indique quelle opération effectuer

Émulation d'instructions

- Objectif : la machine virtuelle exécute les instructions du système invité comme le ferait une machine physique
- Solution : le VMM décode et interprète chaque instruction du système invité

Code système invité

```
...
mov     %rax, %rcx
...
```

Code VMM : `execute_mov()`

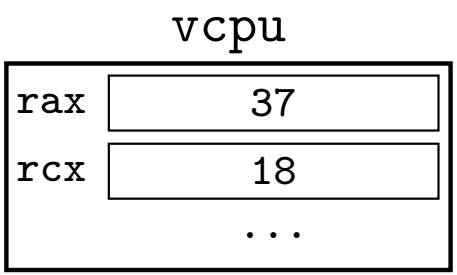
```
src = get_src_register(inst);
dest = get_dest_register(inst);
*dest = *src;
```

- Instructions décodées logiciellement selon l'opcode de la machine virtuelle
- L'instruction décodée indique quelle opération effectuer

État virtuel du processeur

- Les opérations du système invité agissent sur des ressouces virtuelles

Mémoire VMM



Code VMM

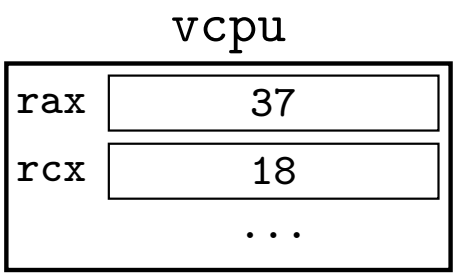
```
struct vcpu {
    uint64_t rax;
    uint64_t rcx;
    ...
};
```

- L'état du CPU virtuel (vCPU) est une structure en mémoire

État virtuel du processeur

- Les opérations du système invité agissent sur des ressouces virtuelles

Mémoire VMM



Code VMM : `execute_mov()`

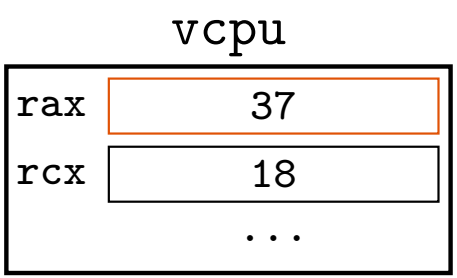
```
src = get_src_register(inst);
dest = get_dest_register(inst);
*dest = *src;
```

- L'état du CPU virtuel (vCPU) est une structure en mémoire

État virtuel du processeur

- Les opérations du système invité agissent sur des ressouces virtuelles

Mémoire VMM



Code VMM : `execute_mov()`

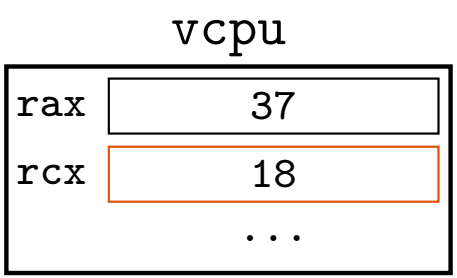
```
src = get_src_register(inst);
dest = get_dest_register(inst);
*dest = *src;
```

- L'état du CPU virtuel (vCPU) est une structure en mémoire
  - Accessible par le VMM comme n'importe quelle donnée

État virtuel du processeur

- Les opérations du système invité agissent sur des ressouces virtuelles

Mémoire VMM



Code VMM : `execute_mov()`

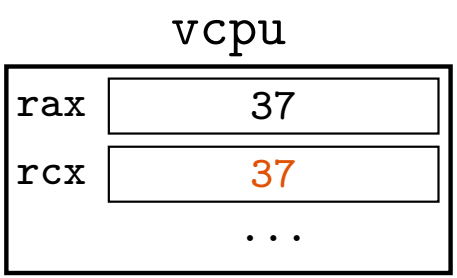
```
src = get_src_register(inst);
dest = get_dest_register(inst);
*dest = *src;
```

- L'état du CPU virtuel (vCPU) est une structure en mémoire
  - Accessible par le VMM comme n'importe quelle donnée

État virtuel du processeur

- Les opérations du système invité agissent sur des ressouces virtuelles

Mémoire VMM



Code VMM : `execute_mov()`

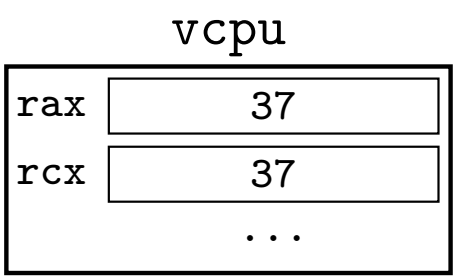
```
src = get_src_register(inst);
dest = get_dest_register(inst);
*dest = *src;
```

- L'état du CPU virtuel (vCPU) est une structure en mémoire
  - Accessible par le VMM comme n'importe quelle donnée
  - Accessible par le système invité par l'interface de machine virtuelle

État virtuel du processeur

- Les opérations du système invité agissent sur des ressouces virtuelles
  - Isolation du système invité et du système hôte (%rax invité $\neq$ %rax hôte)
  - Émulation de ressources d'une autre architecture (%rax sur ARMv7)

Mémoire VMM



Code VMM : `execute_mov()`

```
src = get_src_register(inst);
dest = get_dest_register(inst);
*dest = *src;
```

- L'état du CPU virtuel (vCPU) est une structure en mémoire
  - Accessible par le VMM comme n'importe quelle donnée
  - Accessible par le système invité par l'interface de machine virtuelle

Émulation d'instructions privilégiées

- Le VMM utilise uniquement des instructions non privilégiées pour modifier l'état de la machine virtuelle

Émulation d'instructions privilégiées

- Le VMM utilise uniquement des instructions non privilégiées pour modifier l'état de la machine virtuelle
- Exemple : l'instruction x86 `cli` masque les interruptions en désactivant le flag `IF` dans le registre spécial `rflags`.

Code système invité

```
...
cli
...
```

Émulation d'instructions privilégiées

- Le VMM utilise uniquement des instructions non privilégiées pour modifier l'état de la machine virtuelle
- Exemple : l'instruction x86 `cli` masque les interruptions en désactivant le flag `IF` dans le registre spécial `rflags`.

Code système invité

```
...
cli
...
```

Code VMM : `execute_cli()`

```
if (get_mode() != KERNEL)
    inject_exception(GPF);
else
    vcpu.rflags &= ~RFLAGS_IF;
```

Émulation d'instructions privilégiées

- Le VMM utilise uniquement des instructions non privilégiées pour modifier l'état de la machine virtuelle
- Exemple : l'instruction x86 `cli` masque les interruptions en désactivant le flag `IF` dans le registre spécial `rflags`.

Code système invité

```
...
cli
...
```

Code VMM : `execute_cli()`

```
if (get_mode() != KERNEL)
    inject_exception(GPF);
else
    vcpu.rflags &= ~RFLAGS_IF;
```

Émulation d'instructions privilégiées

- Le VMM utilise uniquement des instructions non privilégiées pour modifier l'état de la machine virtuelle
- Exemple : l'instruction x86 `cli` masque les interruptions en désactivant le flag `IF` dans le registre spécial `rflags`.

Code système invité

```
...
cli
...
```

Code VMM : `execute_cli()`

```
if (get_mode() != KERNEL)
    inject_exception(GPF);
else
    vcpu.rflags &= ~RFLAGS_IF;
```

Émulation d'instructions privilégiées

- Le VMM utilise uniquement des instructions non privilégiées pour modifier l'état de la machine virtuelle
- Exemple : l'instruction x86 `cli` masque les interruptions en désactivant le flag `IF` dans le registre spécial `rflags`.

Code système invité

```
...
cli
...
```

Code VMM : `execute_cli()`

```
if (get_mode() != KERNEL)
    inject_exception(GPF);
else
    vcpu.rflags &= ~RFLAGS_IF;
```

Émulation d'instructions privilégiées

- Le VMM utilise uniquement des instructions non privilégiées pour modifier l'état de la machine virtuelle
- Exemple : l'instruction x86 `cli` masque les interruptions en désactivant le flag `IF` dans le registre spécial `rflags`.
  - Tant que ce flag est désactivé, les interruptions ne sont pas délivrées

Code système invité

```
...
cli
...
```

Code VMM : `execute_cli()`

```
if (get_mode() != KERNEL)
    inject_exception(GPF);
else
    vcpu.rflags &= ~RFLAGS_IF;
```

Code VMM : `deliver_interrupt()`

```
if (vcpu.rflags & RFLAGS_IF) {
    inject_interrupt(inum);
    acknowledge_interrupt(inum);
}
```

Émulation d'instructions privilégiées

- Le VMM utilise uniquement des instructions non privilégiées pour modifier l'état de la machine virtuelle
- Exemple : l'instruction x86 `cli` masque les interruptions en désactivant le flag `IF` dans le registre spécial `rflags`.
  - Tant que ce flag est désactivé, les interruptions ne sont pas délivrées

Code système invité

```
...
cli
...
```

Code VMM : `execute_cli()`

```
if (get_mode() != KERNEL)
    inject_exception(GPF);
else
    vcpu.rflags &= ~RFLAGS_IF;
```

Code VMM : `deliver_interrupt()`

```
if (vcpu.rflags & RFLAGS_IF) {
    inject_interrupt(inum);
    acknowledge_interrupt(inum);
}
```

Émulation d'instructions et état virtuel : résumé

- L'émulation d'instructions est une technique de virtualisation où le VMM décode et interprète chaque instruction du système invité
  - Le VMM reproduit logiciellement toutes les comportements du matériel
- Le VMM exécute les instructions décodées en utilisant un état virtuel de la machine exposée au système invité
- L'émulation d'instruction est plus rapide que la simulation mais reste beaucoup plus lente qu'une exécution physique (facteur x10)
- L'utilisateur peut inspecter facilement l'état de la machine virtuelle $\Rightarrow$ pratique pour le débogage
- Exemples de VMM qui utilisent l'émulation d'instructions
  - Bochs
  - Gearboy

Plan du cours

1. Virtualisation d'instructions
   - Simulation *cycle accurate* et émulation
   - Émulation d'instructions et état virtuel
   - *Dynamic Binary Translation* et *basic blocks*
2. Virtualisation de ressources
   - Virtualisation mémoire et *shadowing*
   - Paravirtualisation et interface matérielle
3. Virtualisation comme isolation
   - Exécution directe et interception
   - Support materiel à la virtualisation

Émulation d'instructions optimisée

- L'émulation d'instruction consomme beaucoup de cycles CPU
  - Le décodage d'instruction peut être complexe selon l'architecture
  - L'interprètation d'une instruction implique plusieurs micro-opérations

Code système invité

```
...
for (i = 0; i < 100; i++)
    arr[i] += arr[i - 1];
...
```

Code VMM

```
while (TRUE) {
    fetch();
    decode();
    execute();
}
```

Émulation d'instructions optimisée

- L'émulation d'instruction consomme beaucoup de cycles CPU
  - Le décodage d'instruction peut être complexe selon l'architecture
  - L'interprètation d'une instruction implique plusieurs micro-opérations

Code système invité

```
...
mov     -8(%rax), %rdx
add     %rdx, (%rax)
add     8, %rax
cmp     %rcx, %rax
jne     -0x30
...
```

Code VMM

```
while (TRUE) {
    fetch();
    decode();
    execute();
}
```

Émulation d'instructions optimisée

- L'émulation d'instruction consomme beaucoup de cycles CPU
  - Le décodage d'instruction peut être complexe selon l'architecture
  - L'interprètation d'une instruction implique plusieurs micro-opérations

Code système invité

```
...
mov     -8(%rax), %rdx
add     %rdx, (%rax)
add     8, %rax
cmp     %rcx, %rax
jne     -0x30
...
```

Code VMM

```
while (TRUE) {
    fetch();
    decode();
    execute();
}
```

Émulation d'instructions optimisée

- L'émulation d'instruction consomme beaucoup de cycles CPU
  - Le décodage d'instruction peut être complexe selon l'architecture
  - L'interprètation d'une instruction implique plusieurs micro-opérations

Code système invité

```
...
mov     -8(%rax), %rdx
add     %rdx, (%rax)
add     8, %rax
cmp     %rcx, %rax
jne     -0x30
...
```

Code VMM

```
while (TRUE) {
    fetch();
    decode();
    execute();
}
```

Émulation d'instructions optimisée

- L'émulation d'instruction consomme beaucoup de cycles CPU
  - Le décodage d'instruction peut être complexe selon l'architecture
  - L'interprètation d'une instruction implique plusieurs micro-opérations

Code système invité

```
...
mov     -8(%rax), %rdx
add     %rdx, (%rax)
add     8, %rax
cmp     %rcx, %rax
jne     -0x30
...
```

Code VMM

```
while (TRUE) {
    fetch();
    decode();
    execute();
}
```

Émulation d'instructions optimisée

- L'émulation d'instruction consomme beaucoup de cycles CPU
  - Le décodage d'instruction peut être complexe selon l'architecture
  - L'interprètation d'une instruction implique plusieurs micro-opérations

Code système invité

```
...
mov     -8(%rax), %rdx
add     %rdx, (%rax)
add     8, %rax
cmp     %rcx, %rax
jne     -0x30
...
```

Code VMM

```
while (TRUE) {
    fetch();
    decode();
    execute();
}
```

Émulation d'instructions optimisée

- L'émulation d'instruction consomme beaucoup de cycles CPU
  - Le décodage d'instruction peut être complexe selon l'architecture
  - L'interprètation d'une instruction implique plusieurs micro-opérations

Code système invité

```
...
mov     -8(%rax), %rdx
add     %rdx, (%rax)
add     8, %rax
cmp     %rcx, %rax
jne     -0x30
...
```

Code VMM

```
while (TRUE) {
    fetch();
    decode();
    execute();
}
```

Émulation d'instructions optimisée

- L'émulation d'instruction consomme beaucoup de cycles CPU
  - Le décodage d'instruction peut être complexe selon l'architecture
  - L'interprètation d'une instruction implique plusieurs micro-opérations

Code système invité

```
...
mov     -8(%rax), %rdx
add     %rdx, (%rax)
add     8, %rax
cmp     %rcx, %rax
jne     -0x30
...
```

Code VMM

```
while (TRUE) {
    fetch();
    decode();
    execute();
}
```

Émulation d'instructions optimisée

- L'émulation d'instruction consomme beaucoup de cycles CPU
  - Le décodage d'instruction peut être complexe selon l'architecture
  - L'interprètation d'une instruction implique plusieurs micro-opérations

Code système invité

```
...
mov     -8(%rax), %rdx
add     %rdx, (%rax)
add     8, %rax
cmp     %rcx, %rax
jne     -0x30
...
```

Code VMM

```
while (TRUE) {
    fetch();
    decode();
    execute();
}
```

Émulation d'instructions optimisée

- L'émulation d'instruction consomme beaucoup de cycles CPU
  - Le décodage d'instruction peut être complexe selon l'architecture
  - L'interprètation d'une instruction implique plusieurs micro-opérations

Code système invité

```
...
mov     -8(%rax), %rdx
add     %rdx, (%rax)
add     8, %rax
cmp     %rcx, %rax
jne     -0x30
...
```

Code VMM

```
while (TRUE) {
    fetch();
    decode();
    execute();
}
```

Émulation d'instructions optimisée

- L'émulation d'instruction consomme beaucoup de cycles CPU
  - Le décodage d'instruction peut être complexe selon l'architecture
  - L'interprètation d'une instruction implique plusieurs micro-opérations

Code système invité

```
...
mov     -8(%rax), %rdx
add     %rdx, (%rax)
add     8, %rax
cmp     %rcx, %rax
jne     -0x30
...
```

Code VMM

```
while (TRUE) {
    fetch();
    decode();
    execute();
}
```

Émulation d'instructions optimisée

- L'émulation d'instruction consomme beaucoup de cycles CPU
  - Le décodage d'instruction peut être complexe selon l'architecture
  - L'interprètation d'une instruction implique plusieurs micro-opérations

Code système invité

```
...
mov     -8(%rax), %rdx
add     %rdx, (%rax)
add     8, %rax
cmp     %rcx, %rax
jne     -0x30
...
```

Code VMM

```
while (TRUE) {
    fetch();
    decode();
    execute();
}
```

Émulation d'instructions optimisée

- L'émulation d'instruction consomme beaucoup de cycles CPU
  - Le décodage d'instruction peut être complexe selon l'architecture
  - L'interprètation d'une instruction implique plusieurs micro-opérations
- Certaines instructions sont exécutées à de nombreuses reprises
  - Le travail de décodage pourrait être réutilisé

Code système invité

```
...
mov     -8(%rax), %rdx
add     %rdx, (%rax)
add     8, %rax
cmp     %rcx, %rax
jne     -0x30
...
```

Code VMM

```
while (TRUE) {
    fetch();
    decode();
    execute();
}
```

Émulation d'instructions optimisée

- L'émulation d'instruction consomme beaucoup de cycles CPU
  - Le décodage d'instruction peut être complexe selon l'architecture
  - L'interprètation d'une instruction implique plusieurs micro-opérations
- Certaines instructions sont exécutées à de nombreuses reprises
  - Le travail de décodage pourrait être réutilisé
- Certaines instructions sont toujours exécutées ensemble
  - Certaines micro-opérations pourraient être évitées

Code système invité

```
...
mov     -8(%rax), %rdx
add     %rdx, (%rax)
add     8, %rax
cmp     %rcx, %rax
jne     -0x30
...
```

Code VMM

```
while (TRUE) {
    fetch();
    decode();
    execute();
}
```

Émulation d'instructions optimisée

- L'émulation d'instruction consomme beaucoup de cycles CPU
  - Le décodage d'instruction peut être complexe selon l'architecture
  - L'interprètation d'une instruction implique plusieurs micro-opérations
- Certaines instructions sont exécutées à de nombreuses reprises
  - Le travail de décodage pourrait être réutilisé
- Certaines instructions sont toujours exécutées ensemble
  - Certaines micro-opérations pourraient être évitées

Code système invité

```
...
mov     -8(%rax), %rdx
add     %rdx, (%rax)
add     8, %rax
cmp     %rcx, %rax
jne     -0x30
...
```

Code VMM

```
while (TRUE) {
    fetch();
    decode();
    execute();
}
```

- Optimisation classique des interprèteurs : compilation à la volée (JIT)

Compilation à la volée

- Abandon de la boucle `fetch-decode-execute`
- Utilisation d'une boucle `fetch-compile-branch`

Code système invité

```
xor
jz
mov
add
cmp
jne
shl
```

Code VMM

```
while (TRUE) {
    block = fetch_basic_block();
    compd = compile_basic_block(block);
    branch_basic_block(compd);
}
```

Compilation à la volée

- Abandon de la boucle `fetch-decode-execute`
- Utilisation d'une boucle `fetch-compile-branch`

Code système invité

```
xor
jz
mov  ⟵ rip
add
cmp
jne
shl
```

Code VMM

```
while (TRUE) {
    block = fetch_basic_block();
    compd = compile_basic_block(block);
    branch_basic_block(compd);
}
```

- Un *basic block* est une séquence indivisible d'instructions
  - Délimités par les instructions de branchement

Compilation à la volée

- Abandon de la boucle `fetch-decode-execute`
- Utilisation d'une boucle `fetch-compile-branch`

Code système invité

```
xor
jz
mov  ⟵ rip
add
cmp
jne  ⟵ prochain saut
shl
```

Code VMM

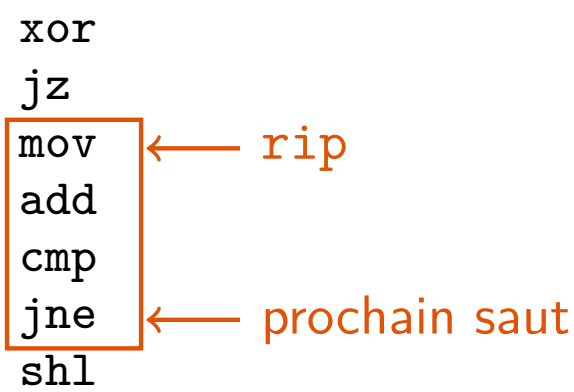
```
while (TRUE) {
    block = fetch_basic_block();
    compd = compile_basic_block(block);
    branch_basic_block(compd);
}
```

- Un *basic block* est une séquence indivisible d'instructions
  - Délimités par les instructions de branchement

Compilation à la volée

- Abandon de la boucle `fetch-decode-execute`
- Utilisation d'une boucle `fetch-compile-branch`

Code système invité



Code VMM

```
while (TRUE) {
    block = fetch_basic_block();
    compd = compile_basic_block(block);
    branch_basic_block(compd);
}
```

- Un *basic block* est une séquence indivisible d'instructions
  - Délimités par les instructions de branchement

~~Compilation à la volée~~

- Abandon de la boucle `fetch-decode-execute`
- Utilisation d'une boucle `fetch-compile-branch`

Code système invité

```
xor
jz
mov
add
cmp
jne
shl
```

Code VMM

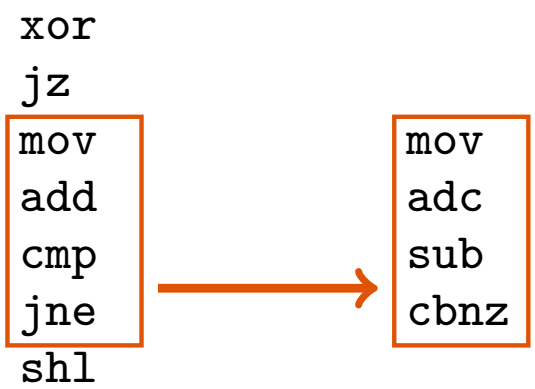
```
while (TRUE) {
    block = fetch_basic_block();
    compd = compile_basic_block(block);
    branch_basic_block(compd);
}
```

- Un *basic block* est une séquence indivisible d'instructions
  - Délimités par les instructions de branchement
- Les *basic blocks* sont compilés vers le jeu d'instructions de l'hôte
  - Cette compilation s'appelle une *Dynamic Binary Translation* (DBT)

Dynamic Binary Translation

- Abandon de la boucle `fetch-decode-execute`
- Utilisation d'une boucle `fetch-compile-branch`

Code système invité



Code VMM

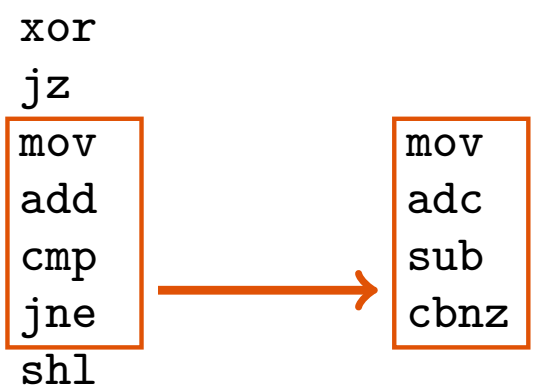
```
while (TRUE) {
    block = fetch_basic_block();
    compd = compile_basic_block(block);
    branch_basic_block(compd);
}
```

- Un *basic block* est une séquence indivisible d'instructions
  - Délimités par les instructions de branchement
- Les *basic blocks* sont compilés vers le jeu d'instructions de l'hôte
  - Cette compilation s'appelle une *Dynamic Binary Translation* (DBT)

Dynamic Binary Translation

- Abandon de la boucle `fetch-decode-execute`
- Utilisation d'une boucle `fetch-compile-branch`

Code système invité



Code VMM

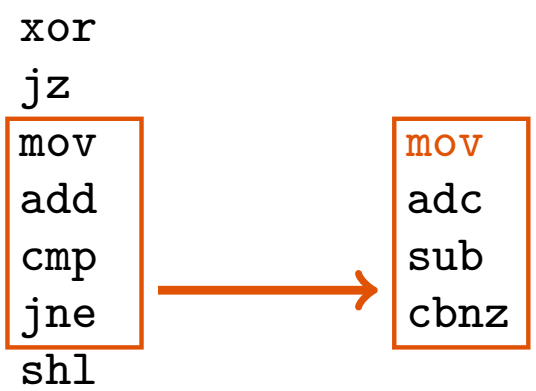
```
while (TRUE) {
    block = fetch_basic_block();
    compd = compile_basic_block(block);
    branch_basic_block(compd);
}
```

- Un *basic block* est une séquence indivisible d'instructions
  - Délimités par les instructions de branchement
- Les *basic blocks* sont compilés vers le jeu d'instructions de l'hôte
  - Cette compilation s'appelle une *Dynamic Binary Translation* (DBT)
- Le bloc résultat est exécuté directement par la machine physique

Dynamic Binary Translation

- Abandon de la boucle `fetch-decode-execute`
- Utilisation d'une boucle `fetch-compile-branch`

Code système invité



Code VMM

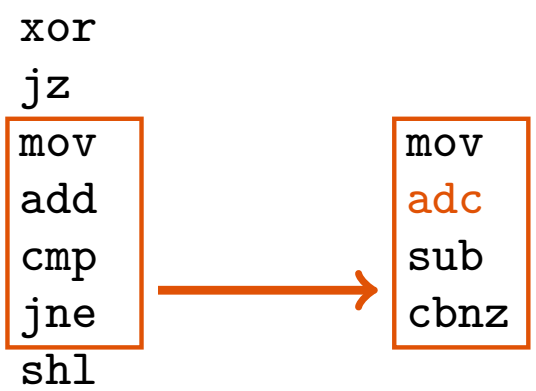
```
while (TRUE) {
    block = fetch_basic_block();
    compd = compile_basic_block(block);
    branch_basic_block(compd);
}
```

- Un *basic block* est une séquence indivisible d'instructions
  - Délimités par les instructions de branchement
- Les *basic blocks* sont compilés vers le jeu d'instructions de l'hôte
  - Cette compilation s'appelle une *Dynamic Binary Translation* (DBT)
- Le bloc résultat est exécuté directement par la machine physique

Dynamic Binary Translation

- Abandon de la boucle `fetch-decode-execute`
- Utilisation d'une boucle `fetch-compile-branch`

Code système invité



Code VMM

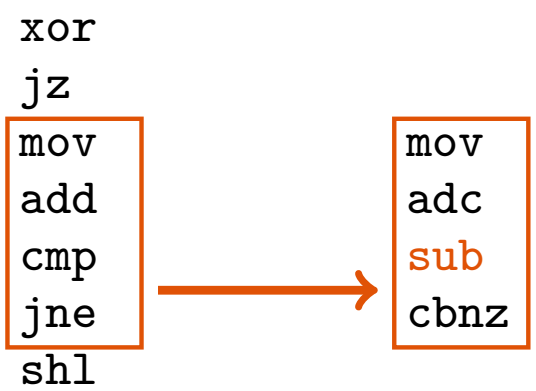
```
while (TRUE) {
    block = fetch_basic_block();
    compd = compile_basic_block(block);
    branch_basic_block(compd);
}
```

- Un *basic block* est une séquence indivisible d'instructions
  - Délimités par les instructions de branchement
- Les *basic blocks* sont compilés vers le jeu d'instructions de l'hôte
  - Cette compilation s'appelle une *Dynamic Binary Translation* (DBT)
- Le bloc résultat est exécuté directement par la machine physique

Dynamic Binary Translation

- Abandon de la boucle `fetch-decode-execute`
- Utilisation d'une boucle `fetch-compile-branch`

Code système invité



Code VMM

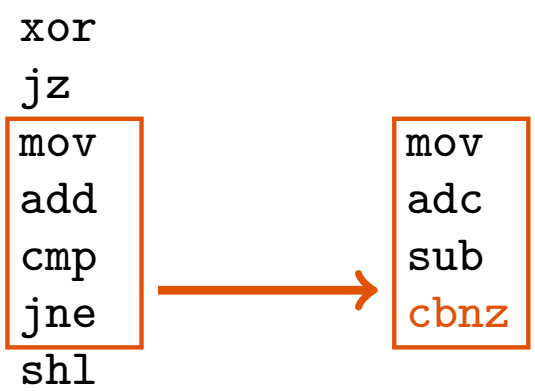
```
while (TRUE) {
    block = fetch_basic_block();
    compd = compile_basic_block(block);
    branch_basic_block(compd);
}
```

- Un *basic block* est une séquence indivisible d'instructions
  - Délimités par les instructions de branchement
- Les *basic blocks* sont compilés vers le jeu d'instructions de l'hôte
  - Cette compilation s'appelle une *Dynamic Binary Translation* (DBT)
- Le bloc résultat est exécuté directement par la machine physique

Dynamic Binary Translation

- Abandon de la boucle `fetch-decode-execute`
- Utilisation d'une boucle `fetch-compile-branch`

Code système invité



Code VMM

```
while (TRUE) {
    block = fetch_basic_block();
    compd = compile_basic_block(block);
    branch_basic_block(compd);
}
```

- Un *basic block* est une séquence indivisible d'instructions
  - Délimités par les instructions de branchement
- Les *basic blocks* sont compilés vers le jeu d'instructions de l'hôte
  - Cette compilation s'appelle une *Dynamic Binary Translation* (DBT)
- Le bloc résultat est exécuté directement par la machine physique

Traduction d'instructions et état virtuel

- Instructions utilisateur traduites vers des instructions hôtes équivalentes
  - Pas toujours de correspondance 1:1 entre les jeux d'instructions

Code invité source (x86)

```
mov     %rax, %rcx
```

Code résultat (ARMv7)

```
mov     $R4, $R0
mov     $R5, $R1
```

Traduction d'instructions et état virtuel

- Instructions utilisateur traduites vers des instructions hôtes équivalentes
  - Pas toujours de correspondance 1:1 entre les jeux d'instructions

Code invité source (x86)

```
mov     %rax, %rcx
```

Code résultat (ARMv7)

```
mov     $R4, $R0
mov     $R5, $R1
```

Traduction d'instructions et état virtuel

- Instructions utilisateur traduites vers des instructions hôtes équivalentes
  - Pas toujours de correspondance 1:1 entre les jeux d'instructions
- Instructions privilégiées traduites par des appels aux fonctions d'émulation
  - Même fonctionnement que pour l'émulation d'instruction

Code invité source (x86)

```
mov     %rax, %rcx

cli
```

Code résultat (ARMv7)

```
mov     $R4, $R0
mov     $R5, $R1

bl      <__vmm_emulate_cli>
```

Traduction d'instructions et état virtuel

- Instructions utilisateur traduites vers des instructions hôtes équivalentes
  - Pas toujours de correspondance 1 : 1 entre les jeux d'instructions
- Instructions privilégiées traduites par des appels aux fonctions d'émulation
  - Même fonctionnement que pour l'émulation d'instruction

Code invité source (x86)

```
mov     %rax, %rcx

cli
```

Code résultat (ARMv7)

```
mov     $R0, <vcpu.rax.low>
...

mov     $R4, $R0
mov     $R5, $R1

bl      <__vmm_emulate_cli>

...
mov     <vcpu.rax.low>, $R0
```

- Le VMM maintient toujours un état virtuel de la machine
  - Chargé dans le processeur physique en prologue de chaque *basic block*
  - Sauvegardé en mémoire en épilogue de chaque *basic block*

Branchements et chaînage de blocs

- Les blocs source sont branchés les uns sur les autres

```
0x3bf: sub   ...
       call  0x104
0x400: add   ...
       cmp   ...
       jne   0x3fb
0x40a: shl   ...
       j     0x2200
```

Branchements et chaînage de blocs

- Les blocs source sont branchés les uns sur les autres

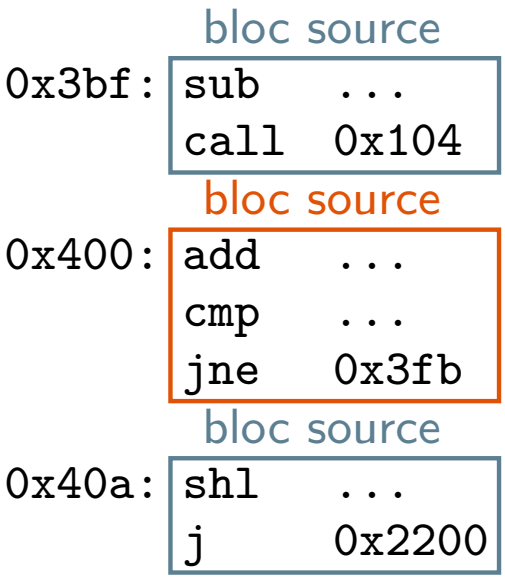
```
0x3bf: sub   ...
       call  0x104

0x400: add   ...
       cmp   ...
       jne   0x3fb

0x40a: shl   ...
       j     0x2200
```

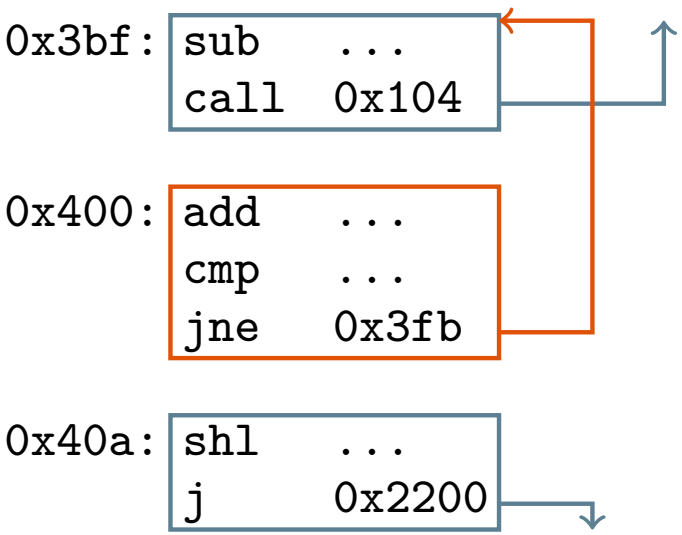
Branchements et chaînage de blocs

- Les blocs source sont branchés les uns sur les autres



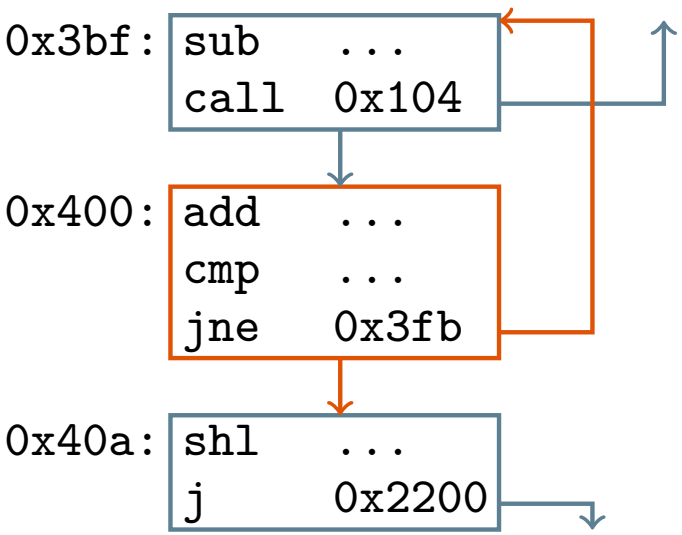
Branchements et chaînage de blocs

- Les blocs source sont branchés les uns sur les autres



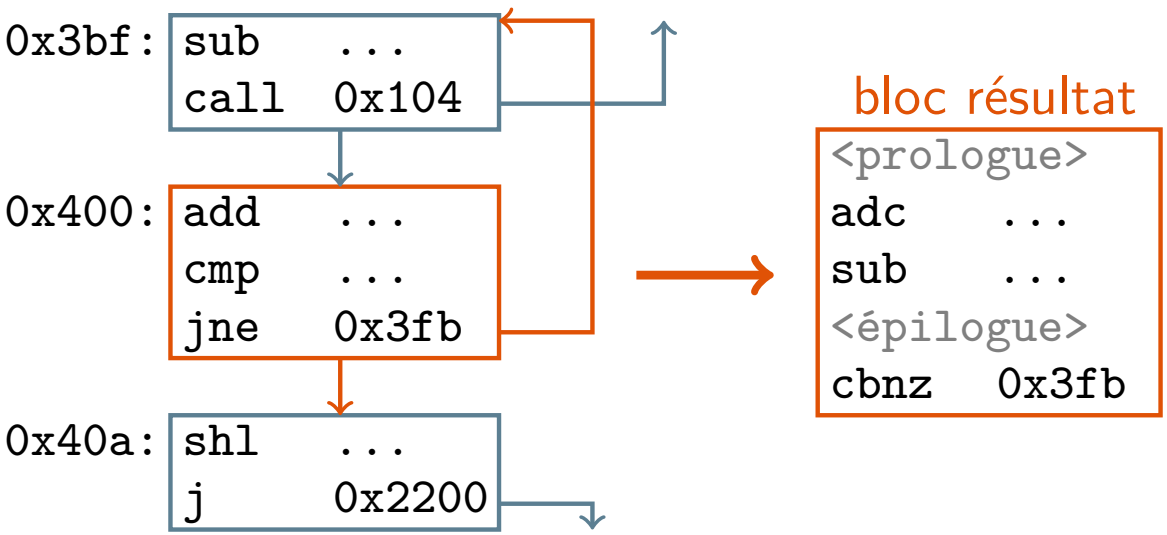
Branchements et chaînage de blocs

- Les blocs source sont branchés les uns sur les autres



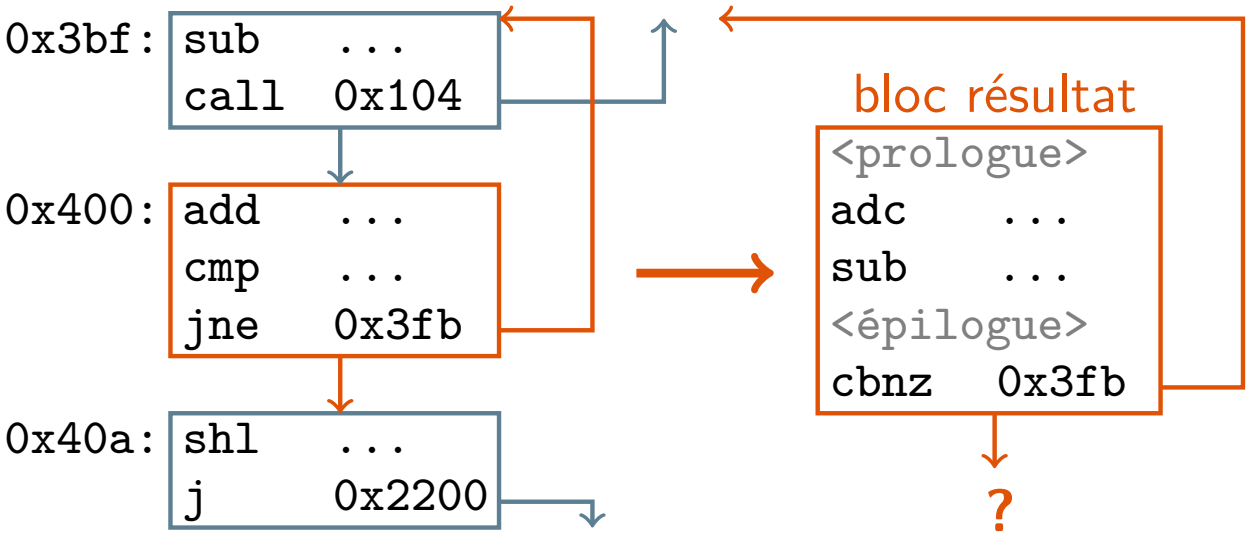
Branchements et chaînage de blocs

- Les blocs source sont branchés les uns sur les autres



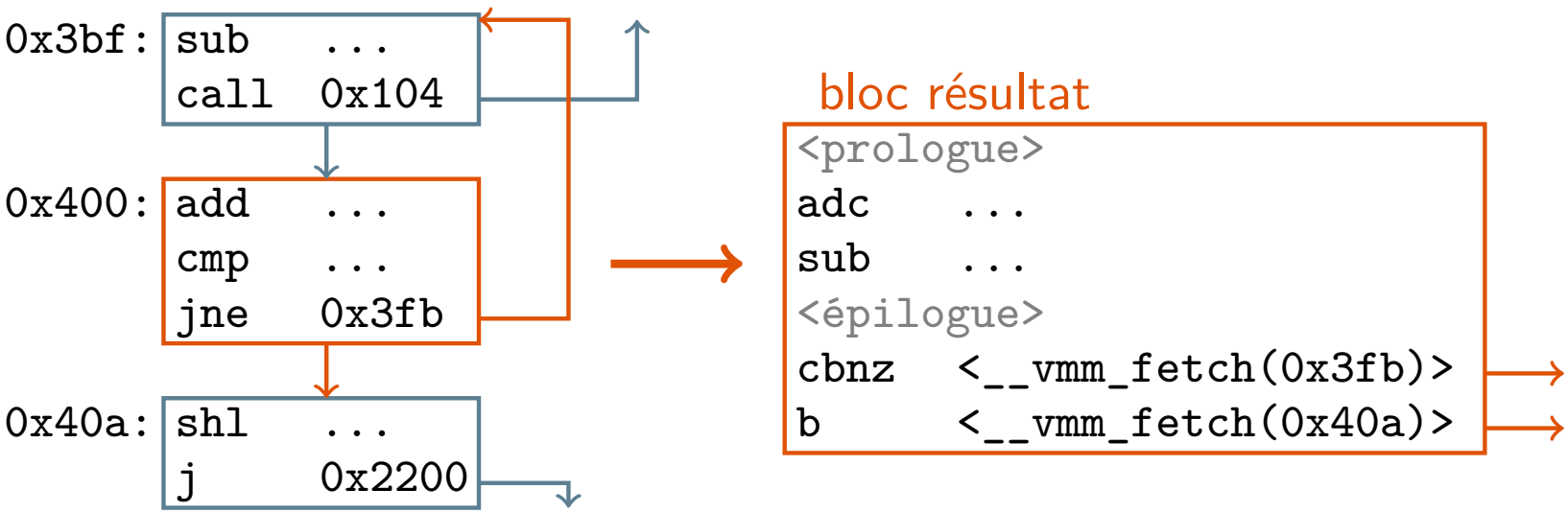
Branchements et chaînage de blocs

- Les blocs source sont branchés les uns sur les autres
  - Un bloc résultat ne doit pas se brancher sur un bloc source
  - Un bloc résultat ne doit pas se brancher sur un bloc inexistant



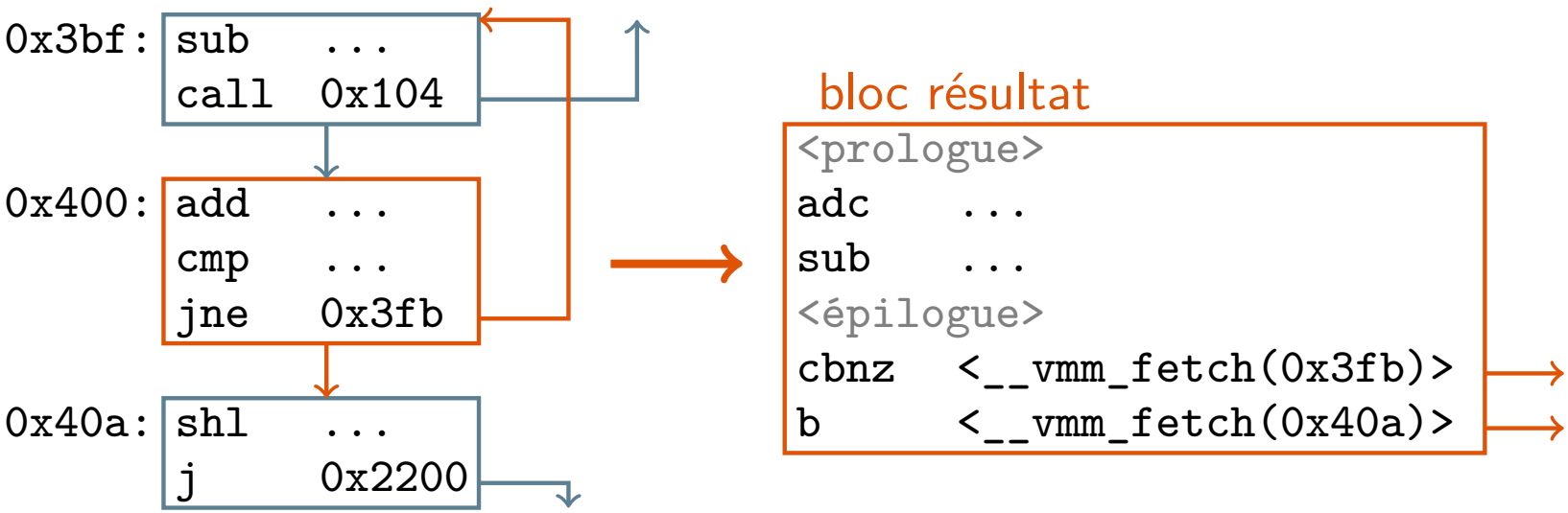
Branchements et chaînage de blocs

- Les blocs source sont branchés les uns sur les autres
  - Un bloc résultat ne doit pas se brancher sur un bloc source
  - Un bloc résultat ne doit pas se brancher sur un bloc inexistant
  - Les blocs résultats se branchent sur la boucle principale du VMM



Branchements et chaînage de blocs

- Les blocs source sont branchés les uns sur les autres
  - Un bloc résultat ne doit pas se brancher sur un bloc source
  - Un bloc résultat ne doit pas se brancher sur un bloc inexistant
  - Les blocs résultats se branchent sur la boucle principale du VMM



- Les blocs résultats sont mis en cache pour être réutilisés
  - Peuvent être directement branchés les uns sur les autres par le VMM

Dynamic Binary Translation : résumé

- La *Dynamic Binary Translation* est une technique de virtualisation où le VMM compile et exécute chaque *basic block* du système invité
- Un *basic block* est une liste d'instruction dont la dernière est l'unique instruction de saut

- Les instructions non privilégiées sont traduites par une ou plusieurs instructions non privilégiées équivalentes
- Les instructions privilégiées sont traduites par un appel à la fonction d'émulation correspondante

- Les blocs résultats se branchent sur le VMM ou sur d'autres blocs résultats quand ils sont disponibles en cache

- Technique plus efficace que l'émulation (facteur x2)
- Exemples de VMM qui utilisent la *Dynamic Binary Translation*
  - VMware Workstation
  - Qemu

Plan du cours

1. Virtualisation d'instructions
   - Simulation *cycle accurate* et émulation
   - Émulation d'instructions et état virtuel
   - *Dynamic Binary Translation* et *basic blocks*

2. Virtualisation de ressources
   - Virtualisation mémoire et *shadowing*
   - Paravirtualisation et interface matérielle

3. Virtualisation comme isolation
   - Exécution directe et interception
   - Support materiel à la virtualisation

Virtualisation et mémoire virtuelle

- Le système invité accède à sa propre mémoire

Code système invité

```
...
mov     %rax, (%rdx)
...
```

Code VMM : `execute_mov_mem()`

```
rax = get_src_register(inst);
rdx = get_dest_register(inst);
addr = *rdx;
*addr = *rax;       // erreur
```

Virtualisation et mémoire virtuelle

- Le système invité accède à sa propre mémoire

Code système invité

```
...
mov     %rax, (%rdx)
...
```

Code VMM : `execute_mov_mem()`

```
rax = get_src_register(inst);
rdx = get_dest_register(inst);
addr = *rdx;
*addr = *rax;        // erreur
```

Virtualisation et mémoire virtuelle

- Le système invité accède à sa propre mémoire

Code système invité

```
...
mov      %rax, (%rdx)
...
```

Code VMM : `execute_mov_mem()`

```
rax = get_src_register(inst);
rdx = get_dest_register(inst);
addr = *rdx;
*addr = *rax;        // erreur
```

Virtualisation et mémoire virtuelle

- Le système invité accède à sa propre mémoire

Code système invité

```
...
mov      %rax, (%rdx)
...
```

Code VMM : execute_mov_mem()

```
rax = get_src_register(inst);
rdx = get_dest_register(inst);
addr = *rdx;
*addr = *rax;        // erreur
```

Virtualisation et mémoire virtuelle

- Le système invité accède à sa propre mémoire

Code système invité

```
...
mov     %rax, (%rdx)
...
```

Code VMM : `execute_mov_mem()`

```
rax = get_src_register(inst);
rdx = get_dest_register(inst);
addr = *rdx;
*addr = *rax;        // erreur
```

Virtualisation et mémoire virtuelle

- Le système invité accède à sa propre mémoire
  - L'interface des machines actuelles comprend une MMU
  - Le système invité veut définir un mapping @ virtuelle $\leftrightarrow$ @ physique

Code système invité

```
...
mov     %rax, (%rdx)
...
```

Code VMM : `execute_mov_mem()`

```
rax = get_src_register(inst);
rdx = get_dest_register(inst);
addr = *rdx;
*addr = *rax;        // erreur
```

- Toute adresse virtuelle émise par le CPU est traduite par la MMU
  - La MMU physique utilise une table des pages du système hôte

Virtualisation et mémoire virtuelle

- Le système invité accède à sa propre mémoire
  - L'interface des machines actuelles comprend une MMU
  - Le système invité veut définir un mapping @ virtuelle $\leftrightarrow$ @ physique

Code système invité

```
...
mov     %rax, (%rdx)
...
```

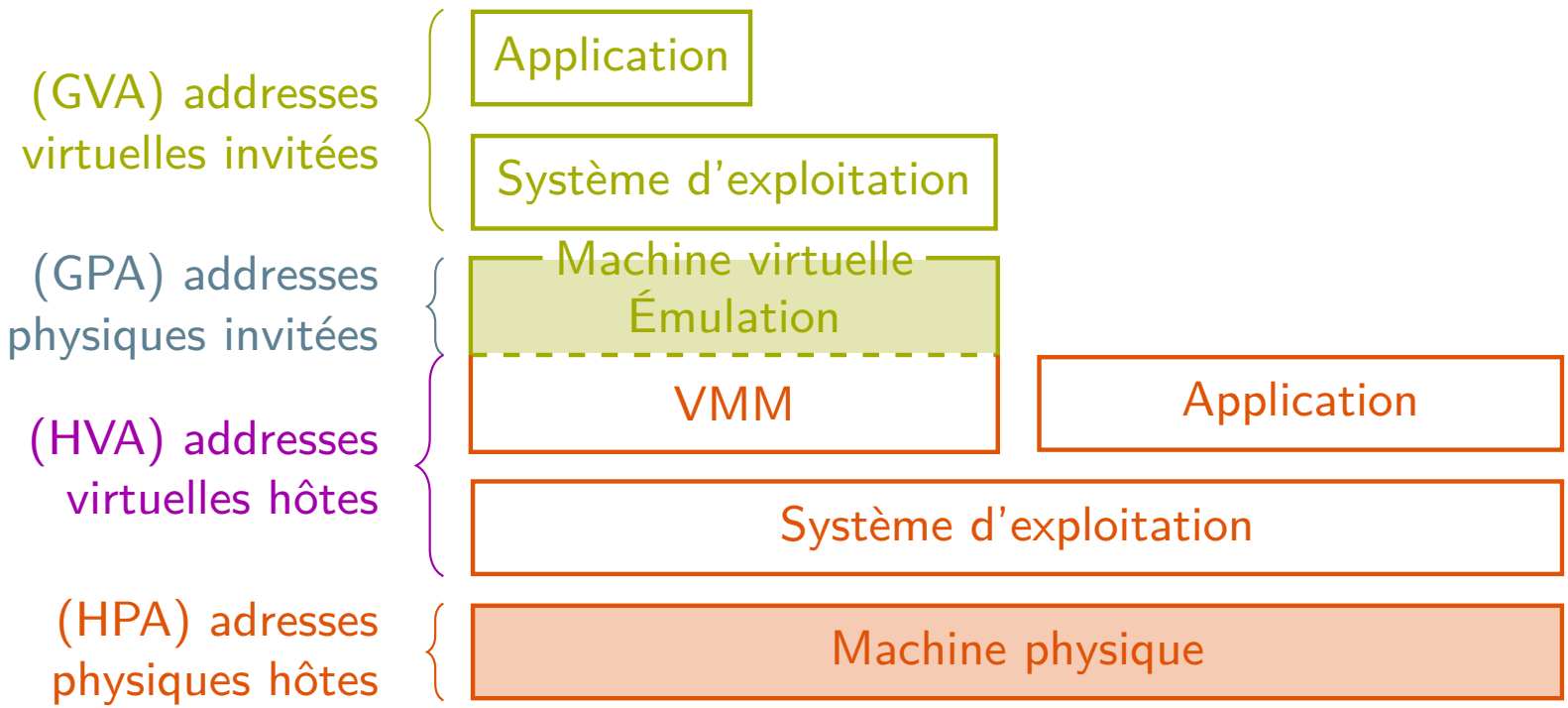
Code VMM : `execute_mov_mem()`

```
rax = get_src_register(inst);
rdx = get_dest_register(inst);
addr = *rdx;
*addr = *rax;        // erreur
```

- Toute adresse virtuelle émise par le CPU est traduite par la MMU
  - La MMU physique utilise une table des pages du système hôte

Virtualisation et adressage

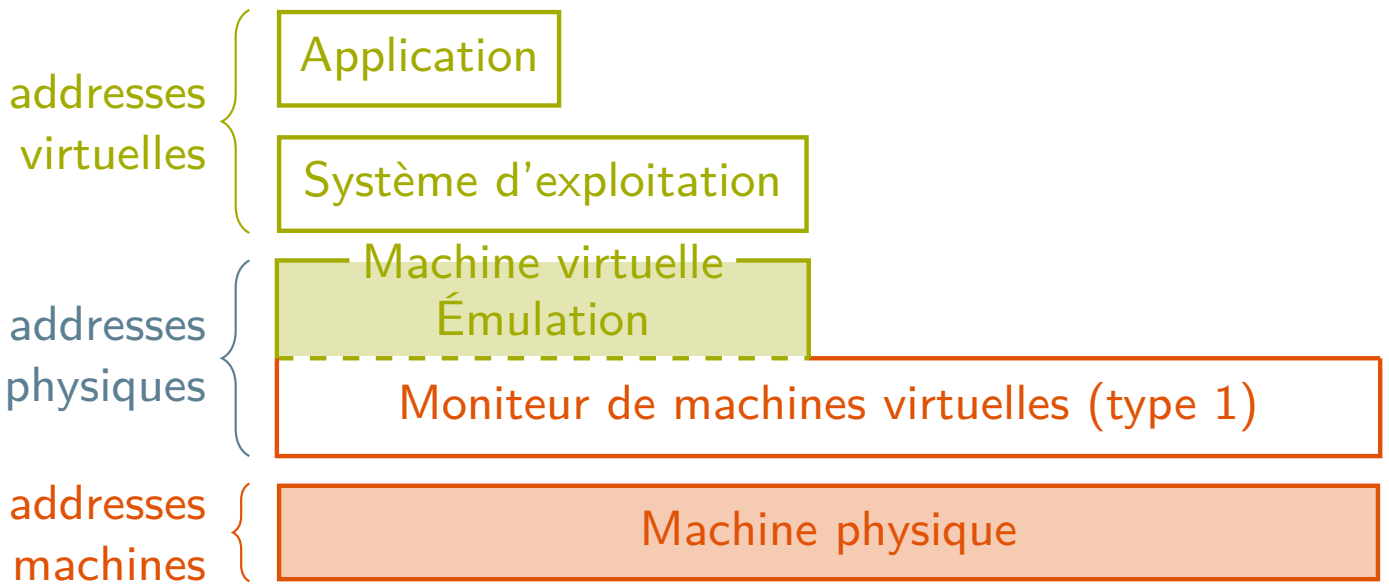
- Une adresse mémoire est caractérisée par deux propriétés
  - Physique / virtuelle : utilisable par le logiciel ou par le matériel
  - Hôte / invitée : valable pour le système hôte ou invité



Virtualisation et adressage

- Une adresse mémoire est caractérisée par deux propriétés
  - Physique / virtuelle : utilisable par le logiciel ou par le matériel
  - Hôte / invitée : valable pour le système hôte ou invité

- Pour les VMM de type 1, on ne distingue pas les (GPA) des (HVA)
  - On utilise la terminologie : virtuelle / physique / machine



Virtualisation et MMU logicielle

- Le VMM utilise une structure logicielle quelconque pour associer une GPA à une HVA

Code système invité

```
...
mov     %rax, (%rdx)
...
```

Code VMM : `execute_mov_mem()`

```
...
gpa = ...
hva = __translate_guest_to_host(gpa);
...
```

Virtualisation et MMU logicielle

- Le VMM utilise une structure logicielle quelconque pour associer une GPA à une HVA
- La MMU physique utilise la table des pages du système hôte pour traduire la HVA obtenue en HPA

Code système invité

```
...
mov     %rax, (%rdx)
...
```

Code VMM : `execute_mov_mem()`

```
 ...
 gpa = ...
 hva = __translate_guest_to_host(gpa);
*hva = *rax;  // la MMU fait hva -> hpa
```

Virtualisation et MMU logicielle

- Le VMM utilise une structure logicielle quelconque pour associer une GPA à une HVA
- La MMU physique utilise la table des pages du système hôte pour traduire la HVA obtenue en HPA

Code système invité

```
...
mov     %rax, (%rdx)
...
```

Code VMM : `execute_mov_mem()`

```
 ...
 gva = *rdx;
 gpa = ...
 hva = __translate_guest_to_host(gpa);
*hva = *rax;  // la MMU fait hva -> hpa
```

- Le système invité attend que la MMU traduise GVA $\rightarrow$ GPA

Virtualisation et MMU logicielle

- Le VMM utilise une structure logicielle quelconque pour associer une GPA à une HVA
- La MMU physique utilise la table des pages du système hôte pour traduire la HVA obtenue en HPA

Code système invité

```
...
mov     %rax, (%rdx)
...
```

Code VMM : `execute_mov_mem()`

```
 ...
 gva = *rdx;
 gpa = __software_mmu_walk(gva);
 hva = __translate_guest_to_host(gpa);
*hva = *rax;  // la MMU fait hva -> hpa
```

- Le système invité attend que la MMU traduise GVA $\rightarrow$ GPA
  - Solution : émuler l'action de la MMU à chaque accès mémoire invité
  - Le VMM parcours la table des pages invité en partant du CR3 virtuel
  - Possibilité d'utiliser un cache de traduction (TLB virtuel)

Virtualisation et MMU logicielle

- Le VMM utilise une structure logicielle quelconque pour associer une GPA à une HVA
- La MMU physique utilise la table des pages du système hôte pour traduire la HVA obtenue en HPA

Code système invité

```
...
mov     %rax, (%rdx)
...
```

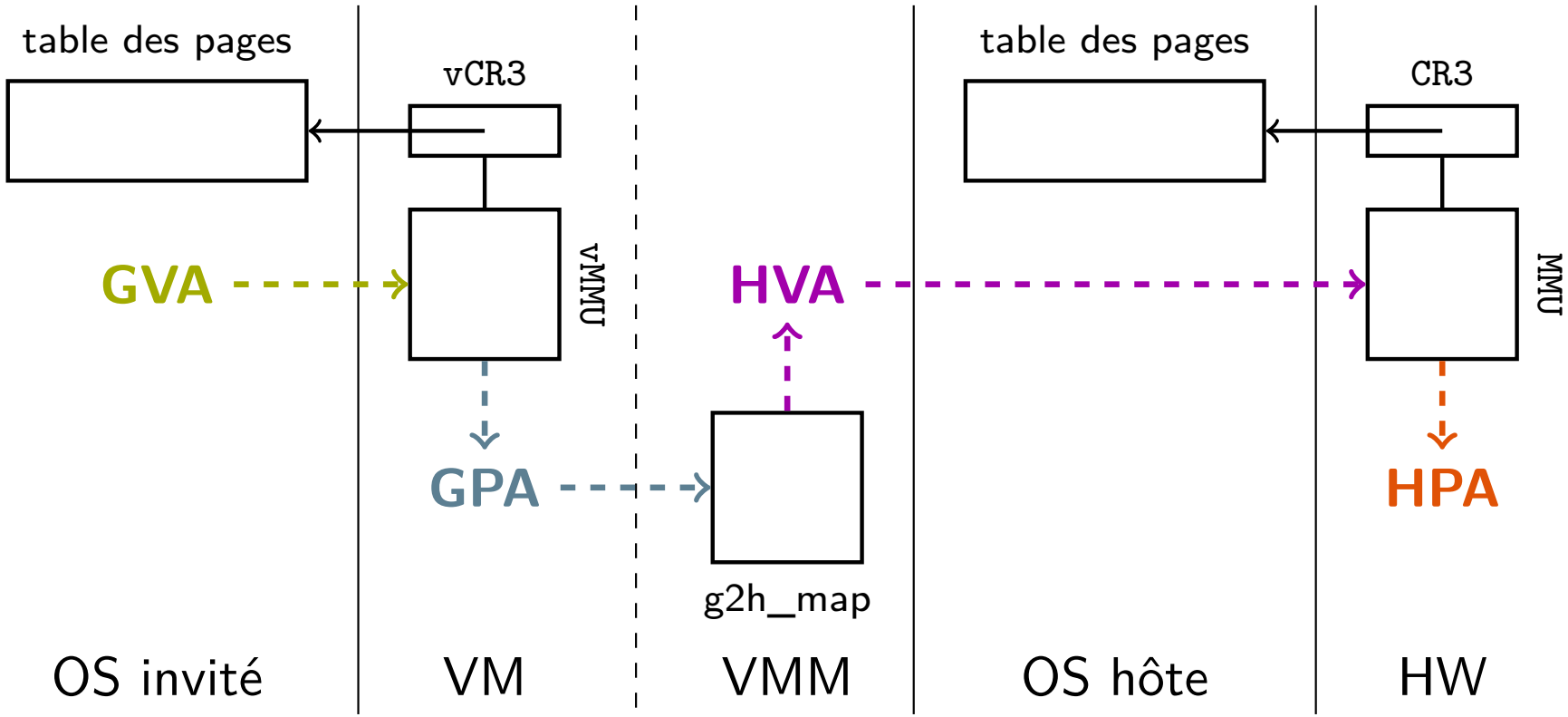
Code résultat (DBT)

```
...
b       <__vmm_software_mmu($R6, $R7)>
b       <__vmm_translate_gva($R0, $R1)>
strd          // la MMU fait hva -> hpa
```

- Le système invité attend que la MMU traduise GVA $\rightarrow$ GPA
  - Solution : émuler l'action de la MMU à chaque accès mémoire invité
  - Le VMM parcours la table des pages invité en partant du CR3 virtuel
  - Possibilité d'utiliser un cache de traduction (TLB virtuel)

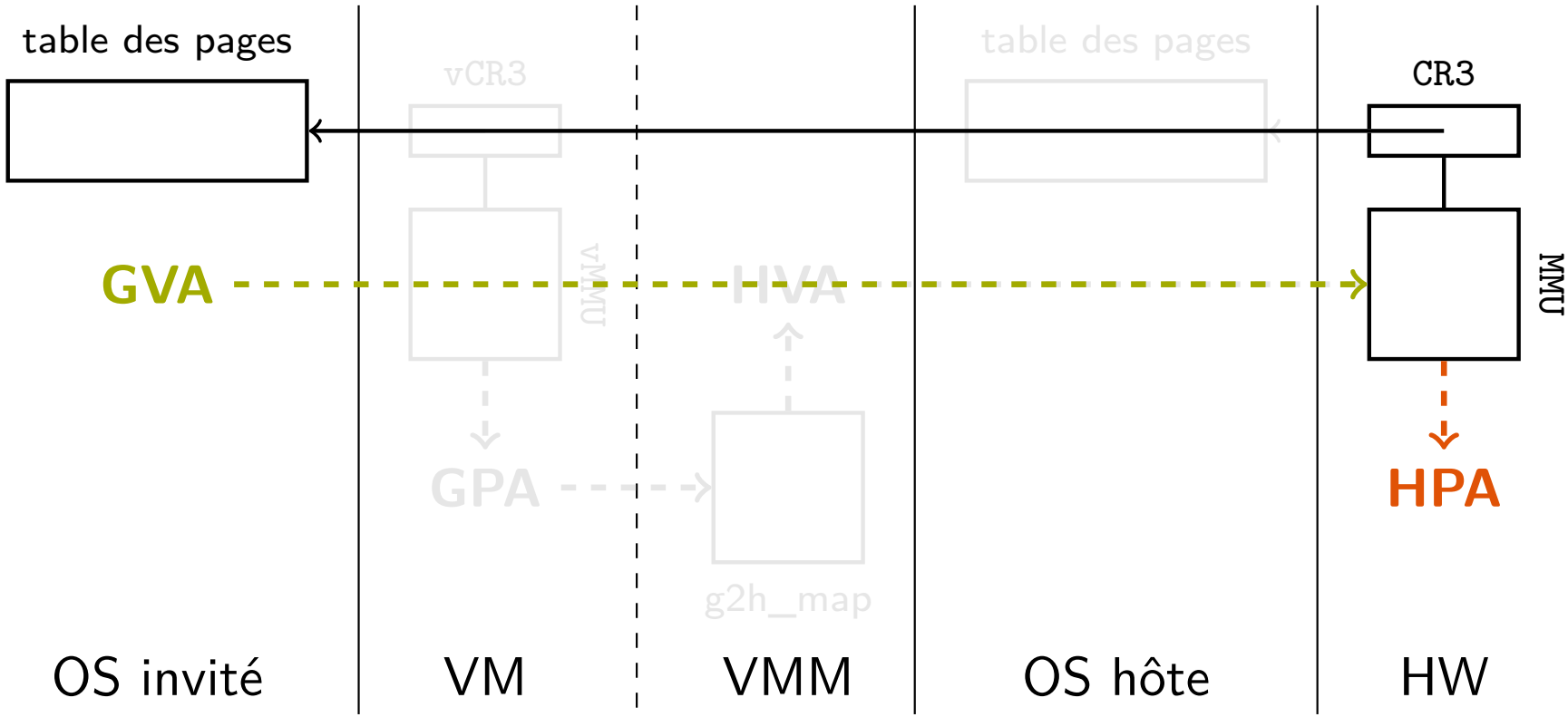
Shadowing mémoire : principe

- La traduction logicielle des adresses est un mécanisme coûteux



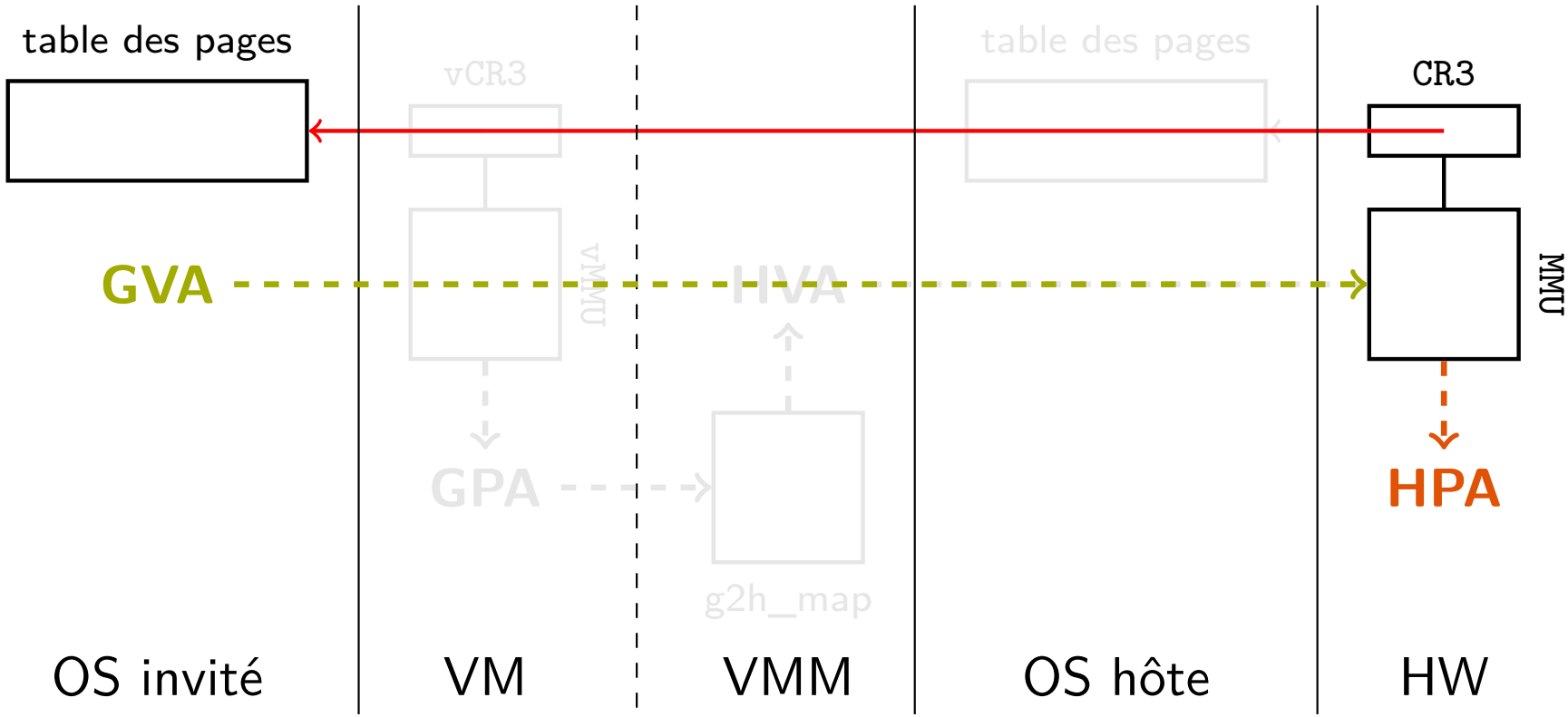
Shadowing mémoire : principe

- La traduction logicielle des adresses est un mécanisme coûteux
  - On voudrait utiliser la MMU physique pour traduire GVA $\rightarrow$ HPA



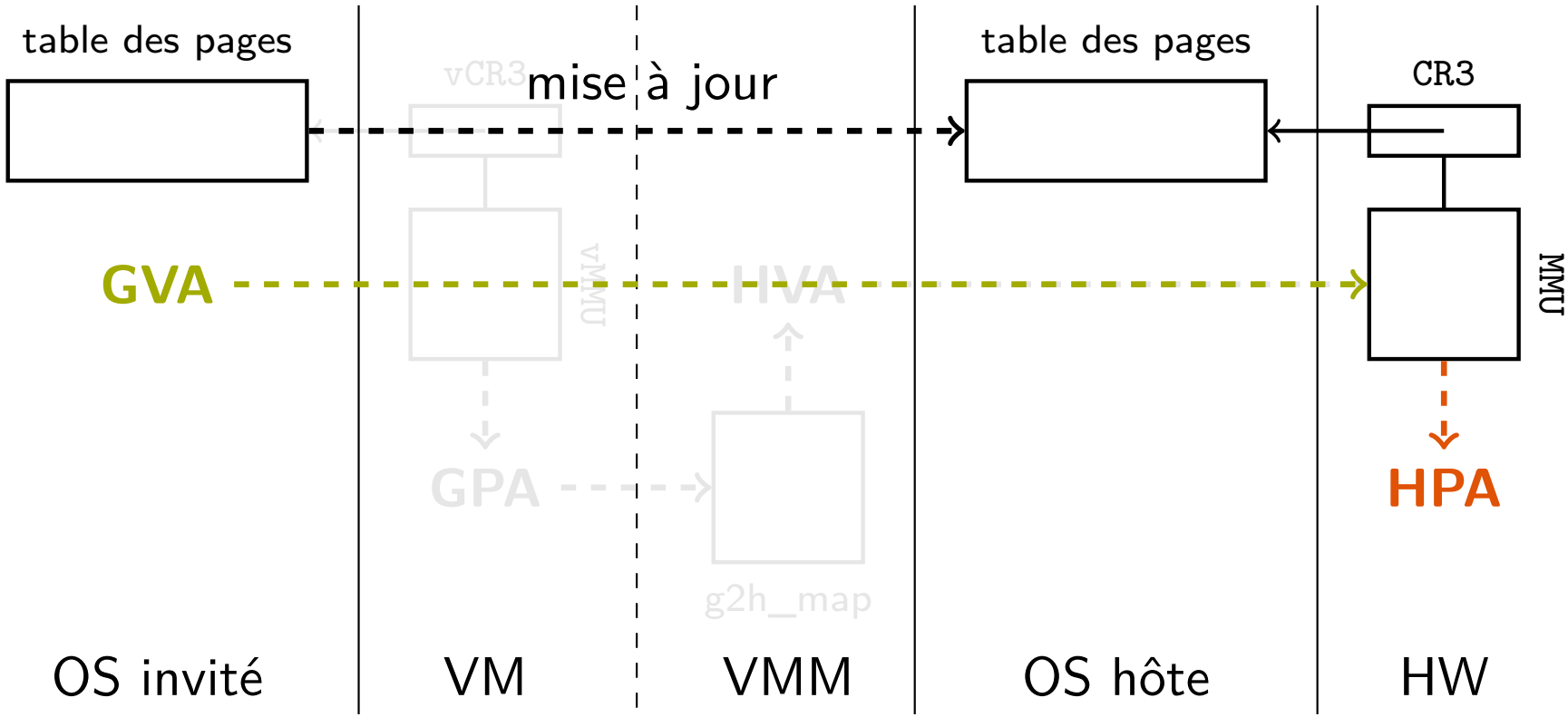
Shadowing mémoire : principe

- La traduction logicielle des adresses est un mécanisme coûteux
  - On voudrait utiliser la MMU physique pour traduire GVA $\rightarrow$ HPA
  - On ne peut pas laisser le système invité controller la MMU physique

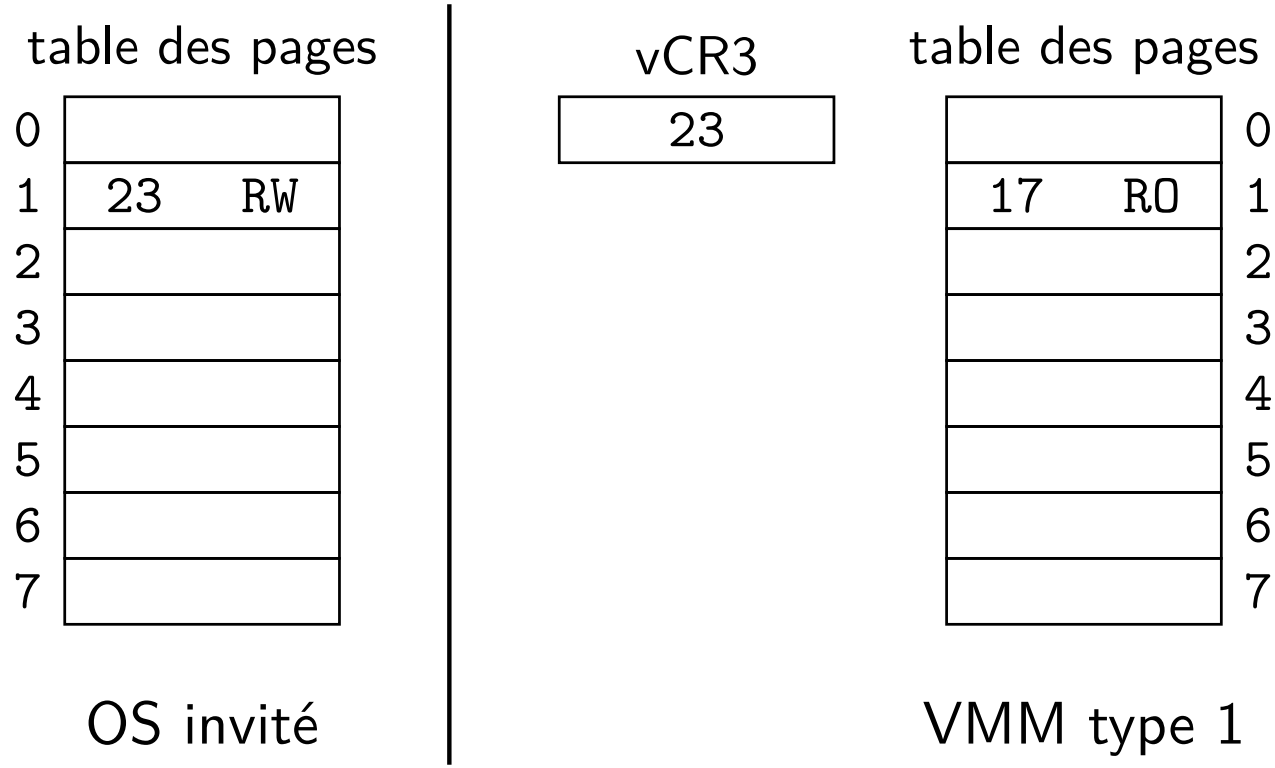


Shadowing mémoire : principe

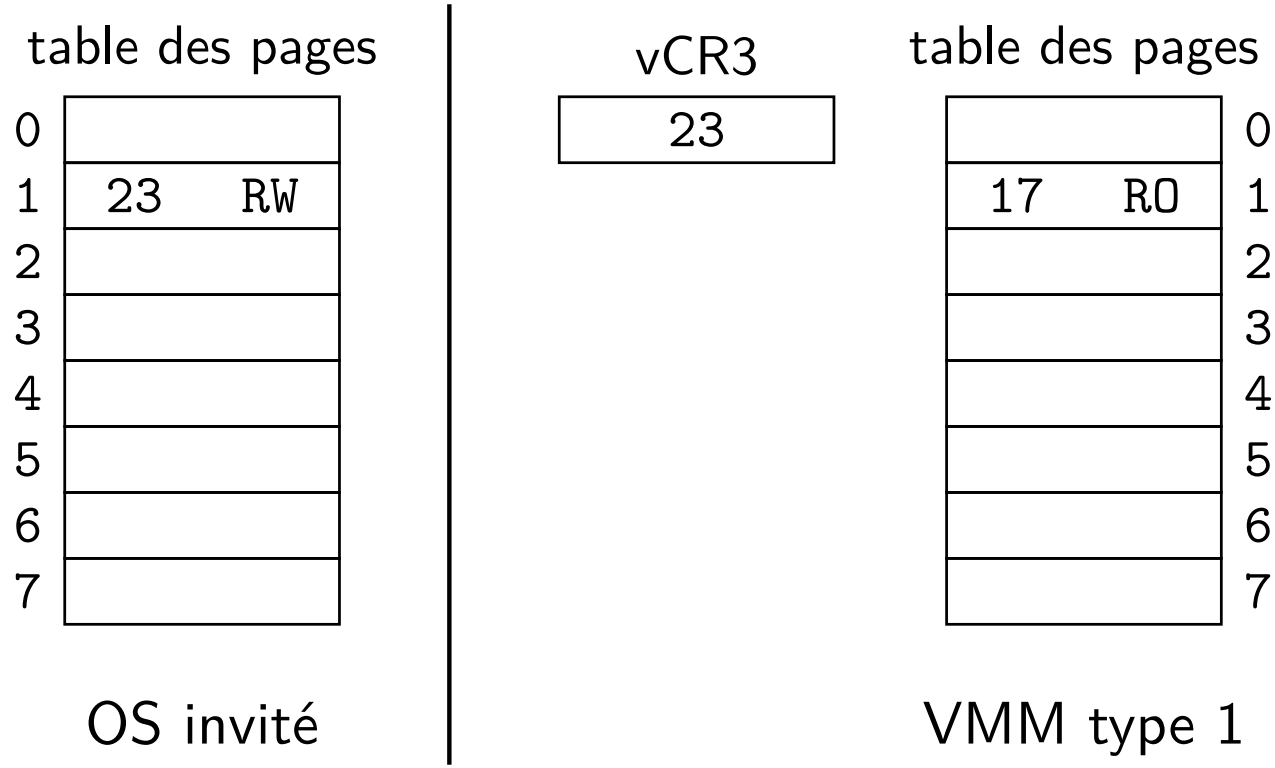
- La traduction logicielle des adresses est un mécanisme coûteux
  - On voudrait utiliser la MMU physique pour traduire GVA $\rightarrow$ HPA
  - On ne peut pas laisser le système invité controller la MMU physique
  - On peut mettre à jour la table hôte en fonction des actions de l'invité



Shadowing de table de pages : VMM type 1

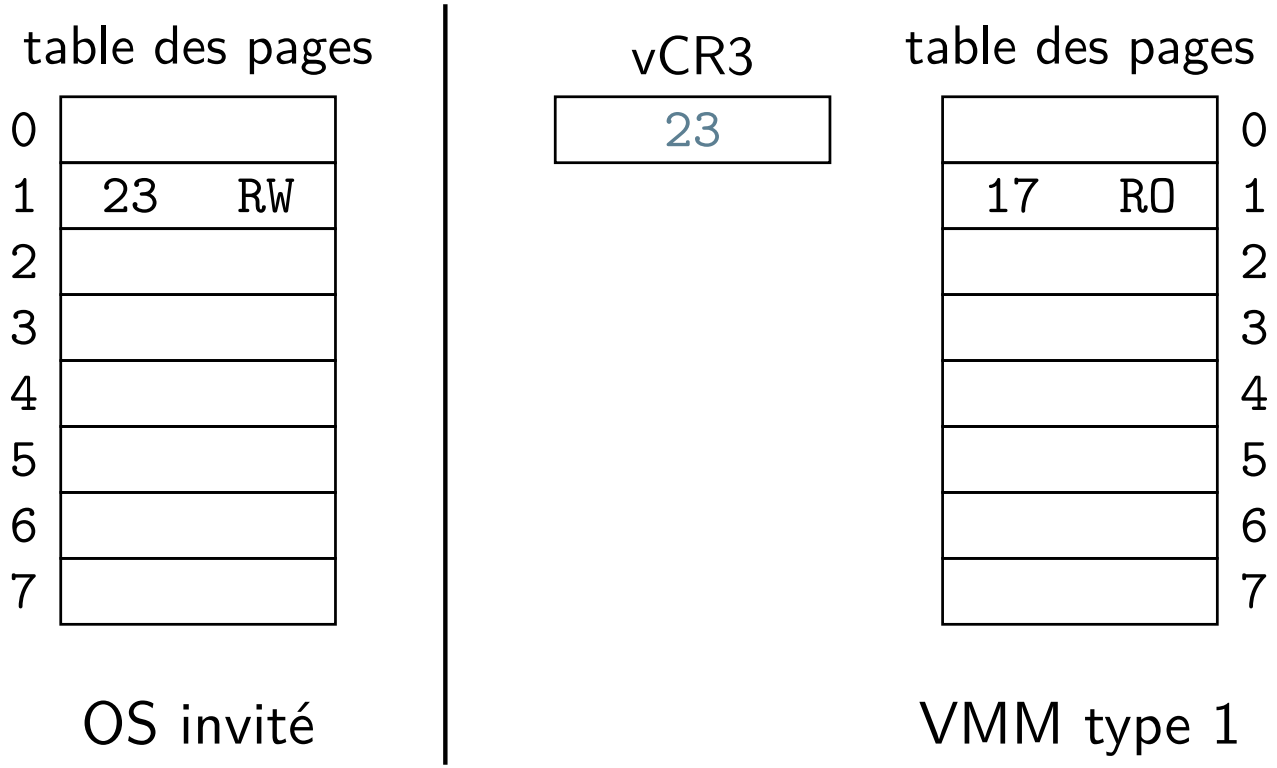


Shadowing de table de pages : VMM type 1



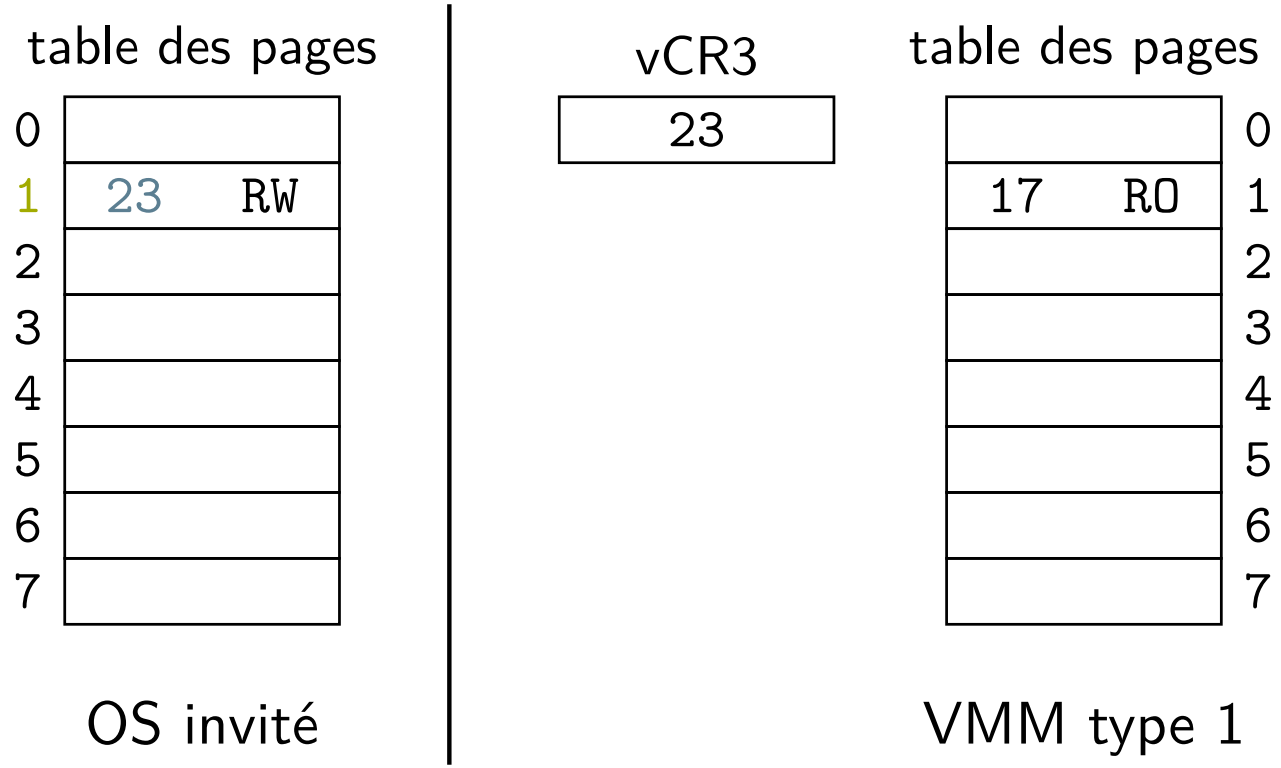
- Exercice : donnez pour la table des pages invitée
  - L'adresse physique :
  - L'adresse virtuelle :
  - L'adresse machine :

Shadowing de table de pages : VMM type 1



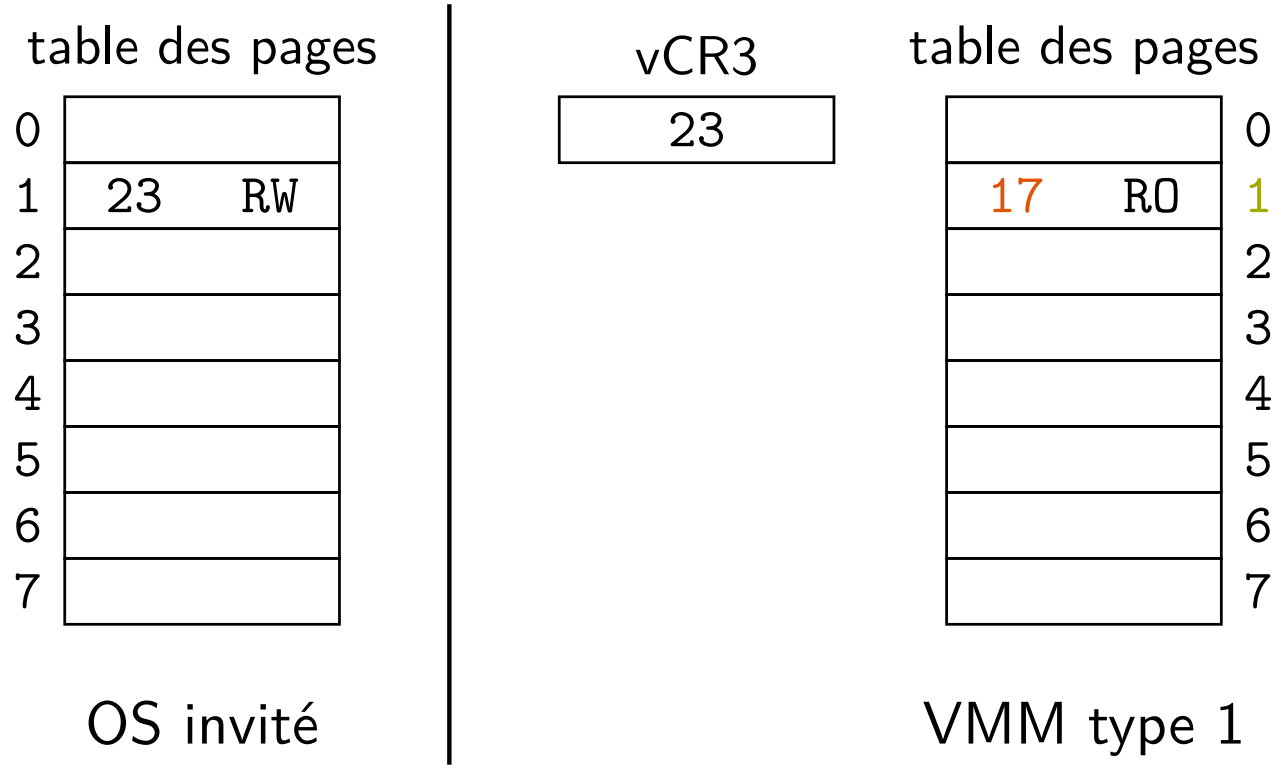
- Exercice : donnez pour la table des pages invitée
  - L'adresse physique : `0x23000`
  - L'adresse virtuelle :
  - L'adresse machine :

Shadowing de table de pages : VMM type 1



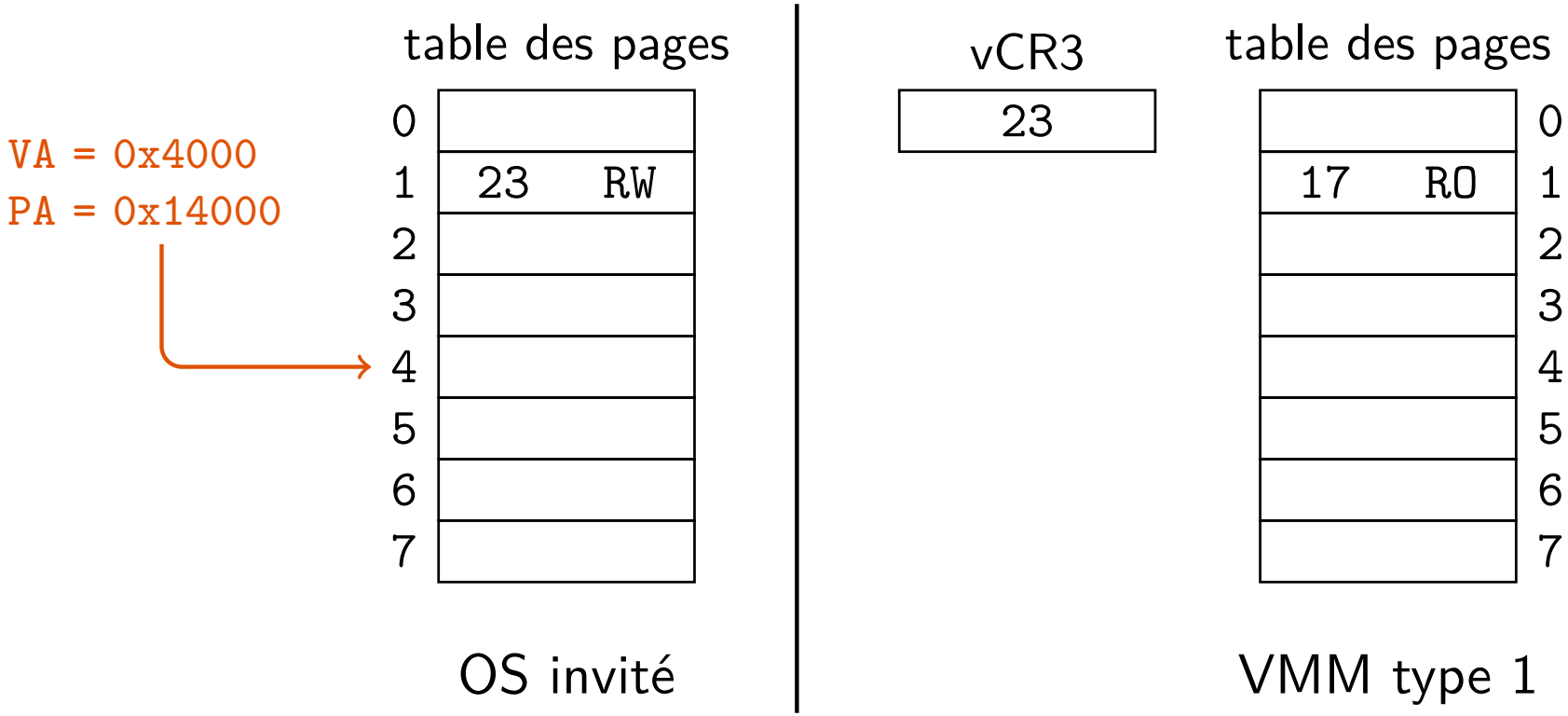
- Exercice : donnez pour la table des pages invitée
  - L'adresse physique : `0x23000`
  - L'adresse virtuelle : `0x1000`
  - L'adresse machine :

Shadowing de table de pages : VMM type 1



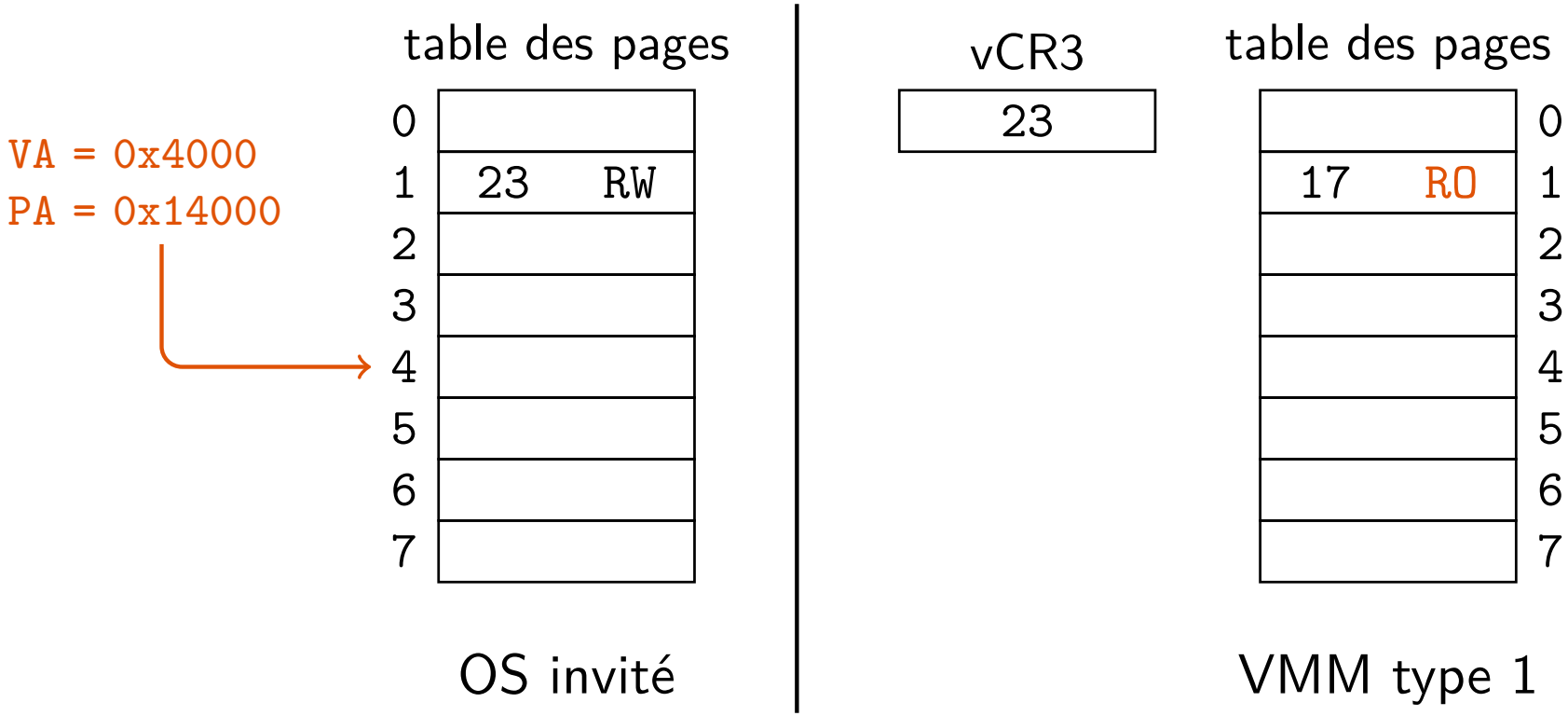
- Exercice : donnez pour la table des pages invitée
  - L'adresse physique : `0x23000`
  - L'adresse virtuelle : `0x1000`
  - L'adresse machine : `0x17000`

Shadowing de table de pages : VMM type 1



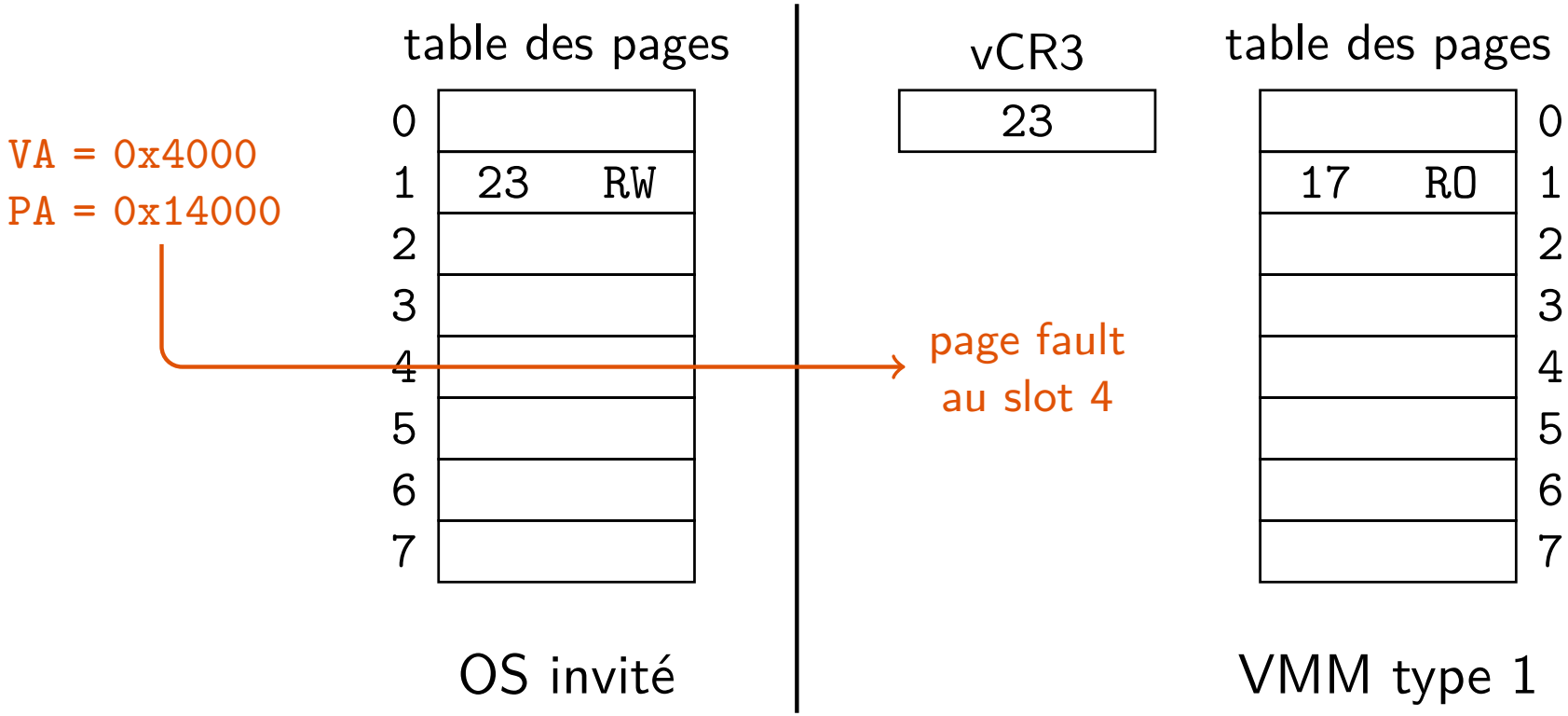
- L'invité crée un nouveau mapping VA $\rightarrow$ PA en modifiant sa table des pages

Shadowing de table de pages : VMM type 1



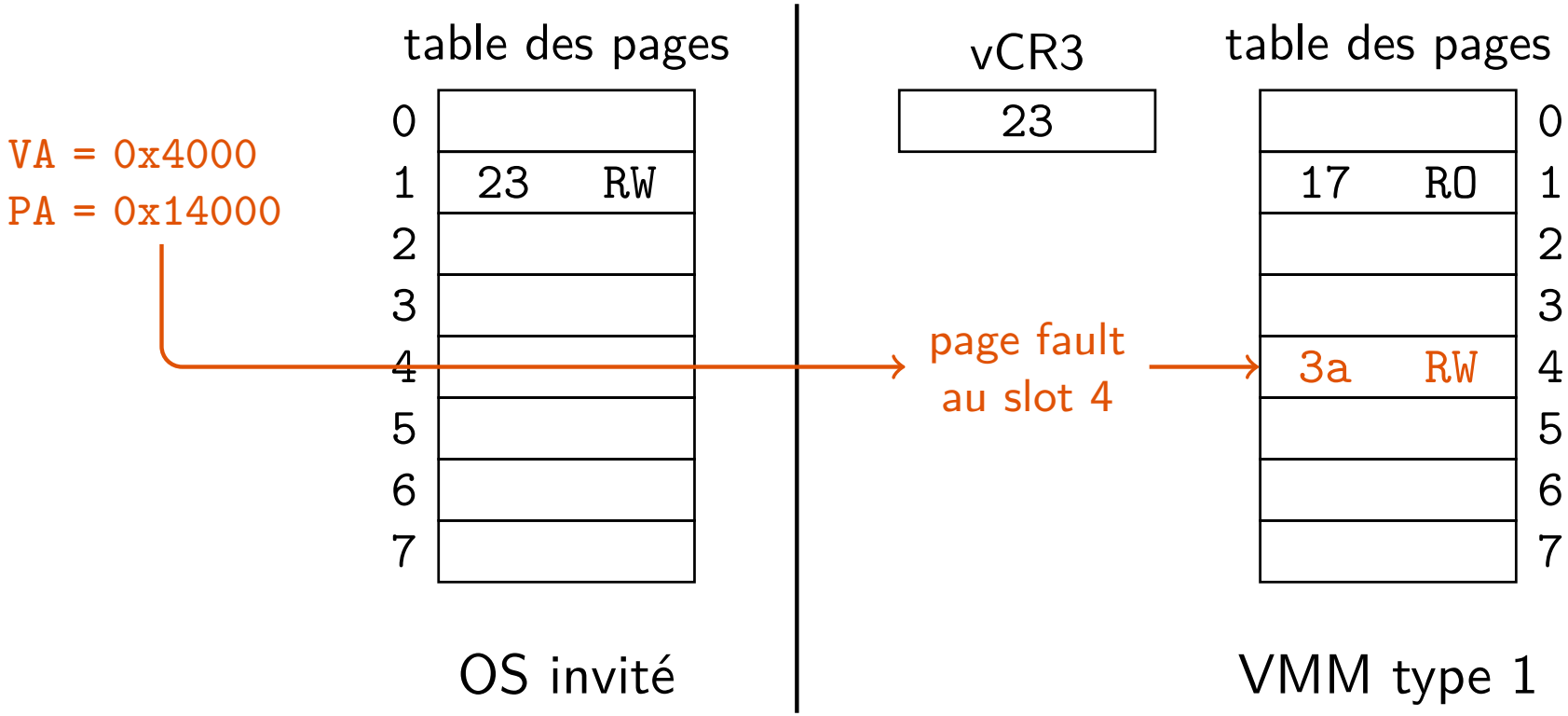
- L'invité crée un nouveau mapping VA $\rightarrow$ PA en modifiant sa table des pages
  - Pour intercepter cette écriture, le VMM protège la table invitée en écriture

Shadowing de table de pages : VMM type 1



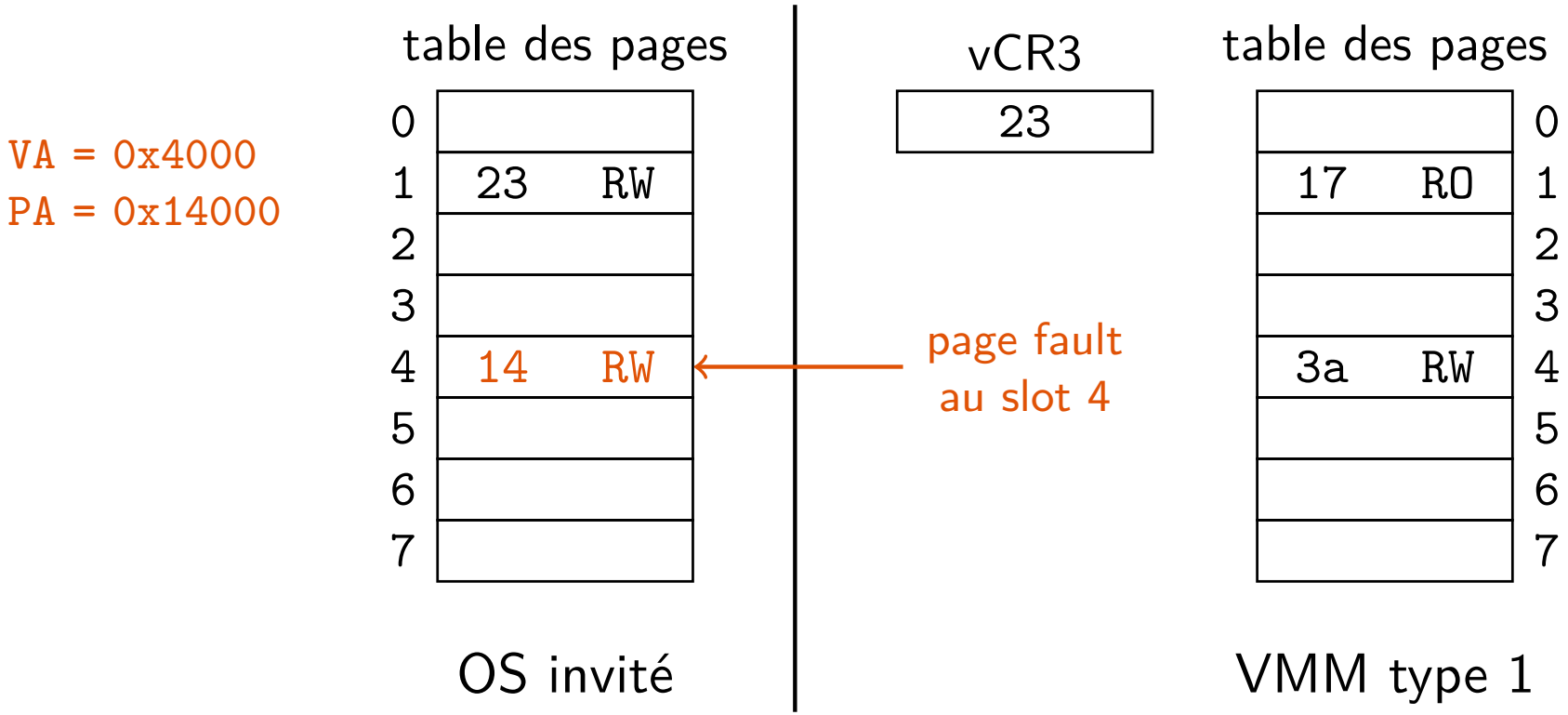
- L'invité crée un nouveau mapping VA $\rightarrow$ PA en modifiant sa table des pages
  - Pour intercepter cette écriture, le VMM protège la table invitée en écriture
  - Le VMM est avertit de la tentative d'écriture par une faute de page

Shadowing de table de pages : VMM type 1



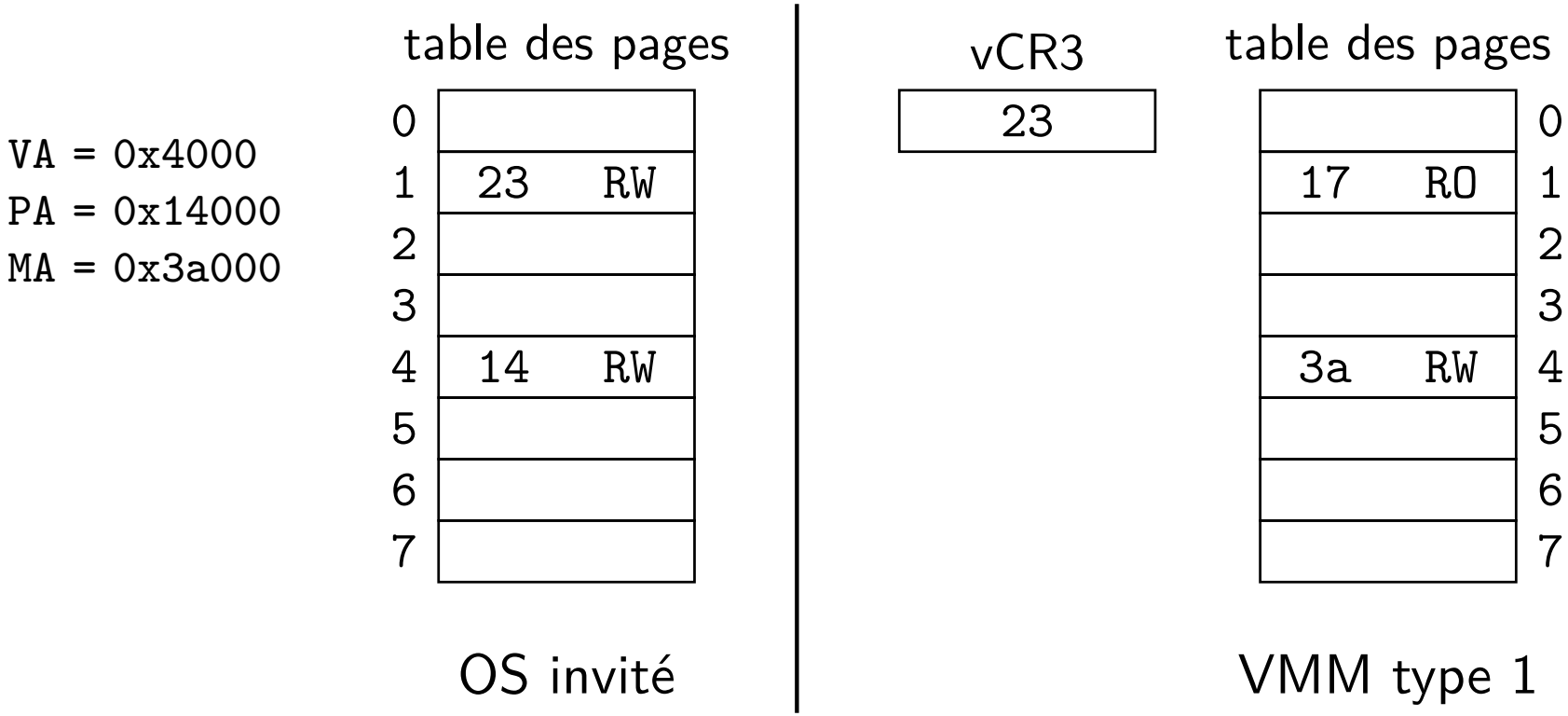
- L'invité crée un nouveau mapping VA $\rightarrow$ PA en modifiant sa table des pages
  - Pour intercepter cette écriture, le VMM protège la table invitée en écriture
  - Le VMM est avertit de la tentative d'écriture par une faute de page
  - Le VMM associe l'adresse virtuelle fautive à l'adresse machine de son choix

Shadowing de table de pages : VMM type 1



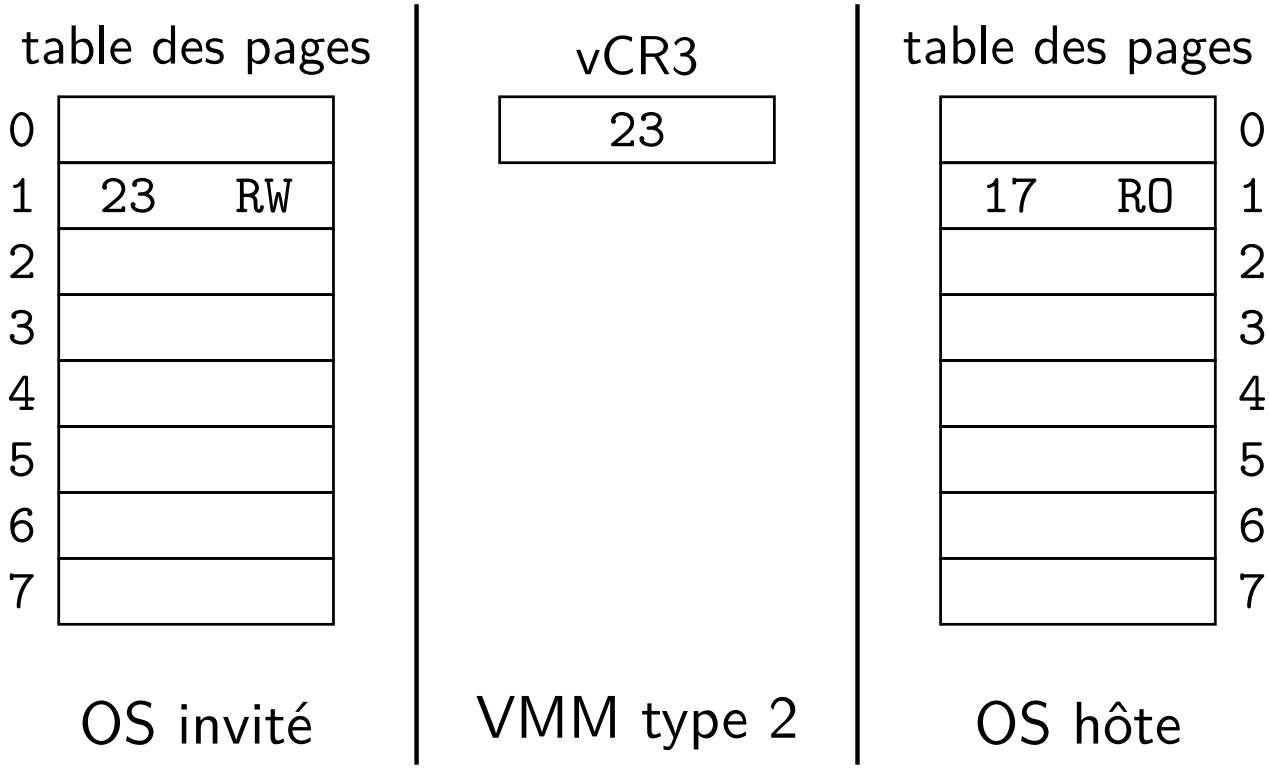
- L'invité crée un nouveau mapping VA $\rightarrow$ PA en modifiant sa table des pages
  - Pour intercepter cette écriture, le VMM protège la table invitée en écriture
  - Le VMM est avertit de la tentative d'écriture par une faute de page
  - Le VMM associe l'adresse virtuelle fautive à l'adresse machine de son choix
  - Le VMM modifie la table des pages invitée pour écrire l'adresse physique

Shadowing de table de pages : VMM type 1



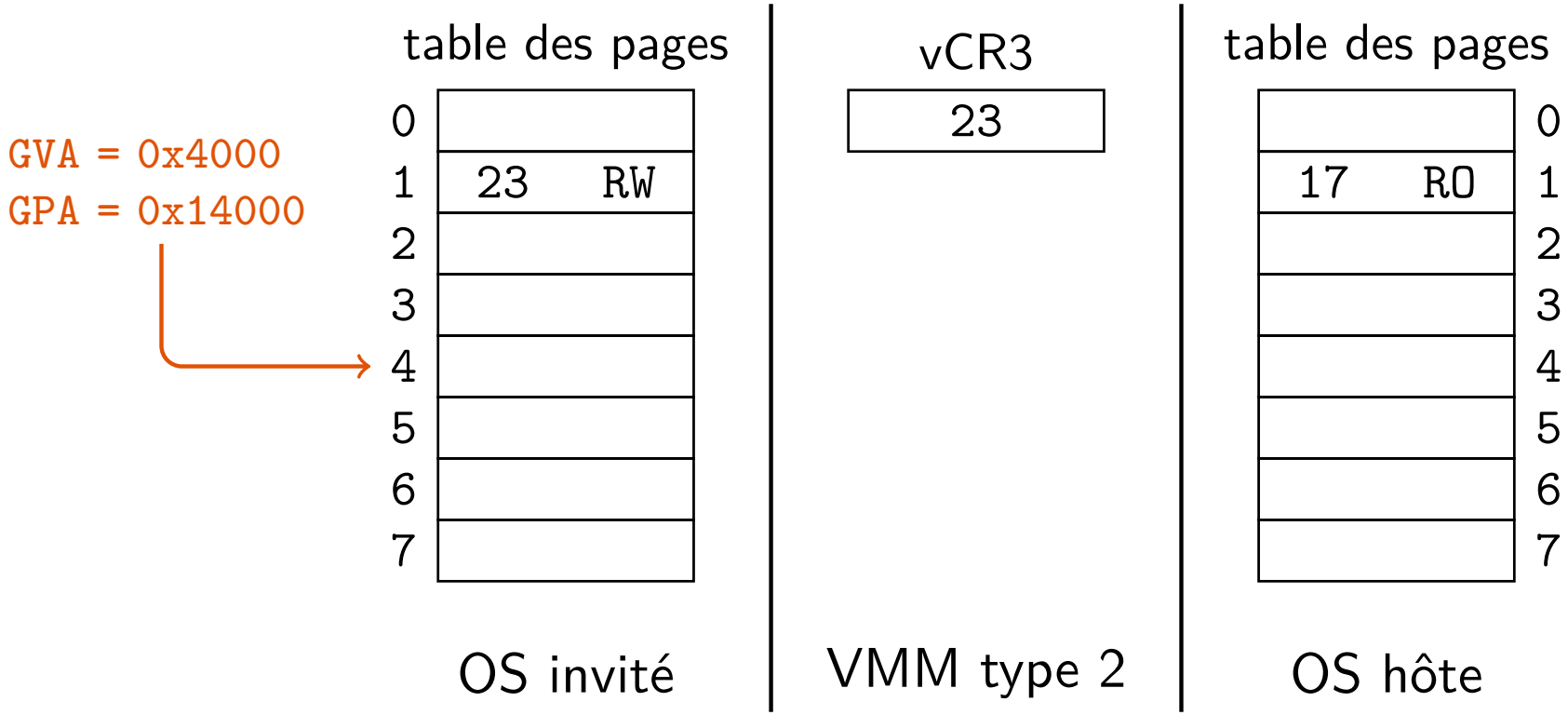
- L'invité crée un nouveau mapping VA $\rightarrow$ PA en modifiant sa table des pages
  - Pour intercepter cette écriture, le VMM protège la table invitée en écriture
  - Le VMM est avertit de la tentative d'écriture par une faute de page
  - Le VMM associe l'adresse virtuelle fautive à l'adresse machine de son choix
  - Le VMM modifie la table des pages invitée pour écrire l'adresse physique

Shadowing de table de pages : VMM type 2



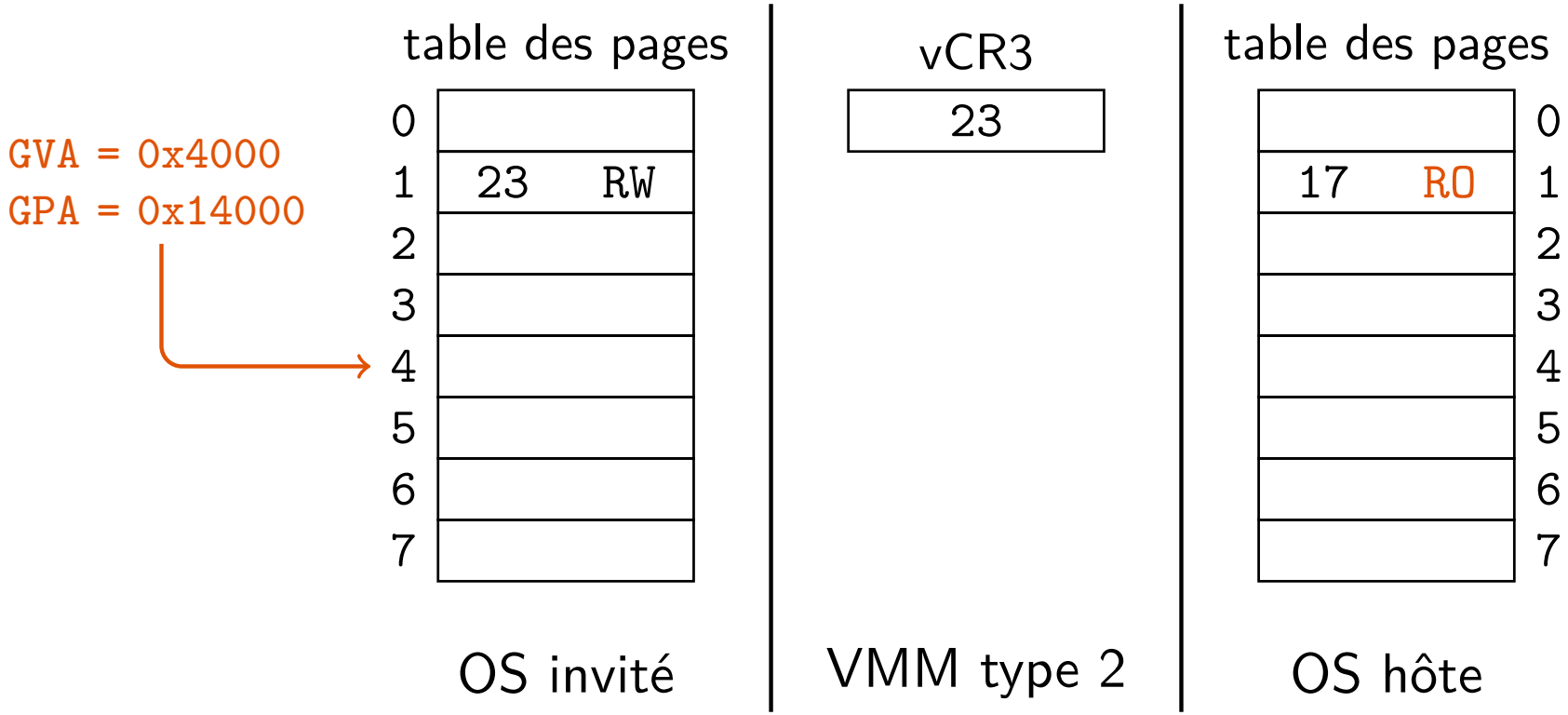
- Le fonctionnement d'une *shadow page table* en type 2 est très similaire

Shadowing de table de pages : VMM type 2



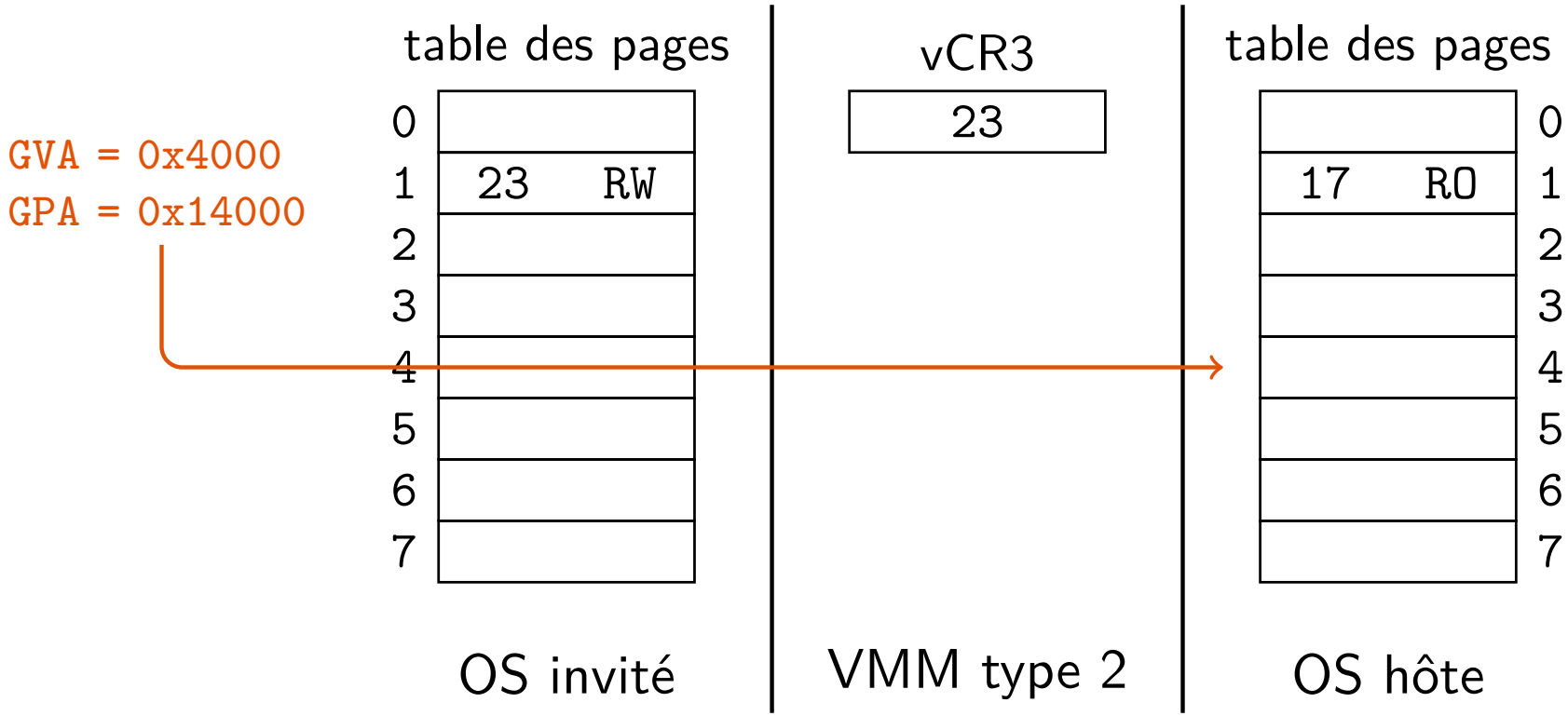
- Le fonctionnement d'une *shadow page table* en type 2 est très similaire

Shadowing de table de pages : VMM type 2



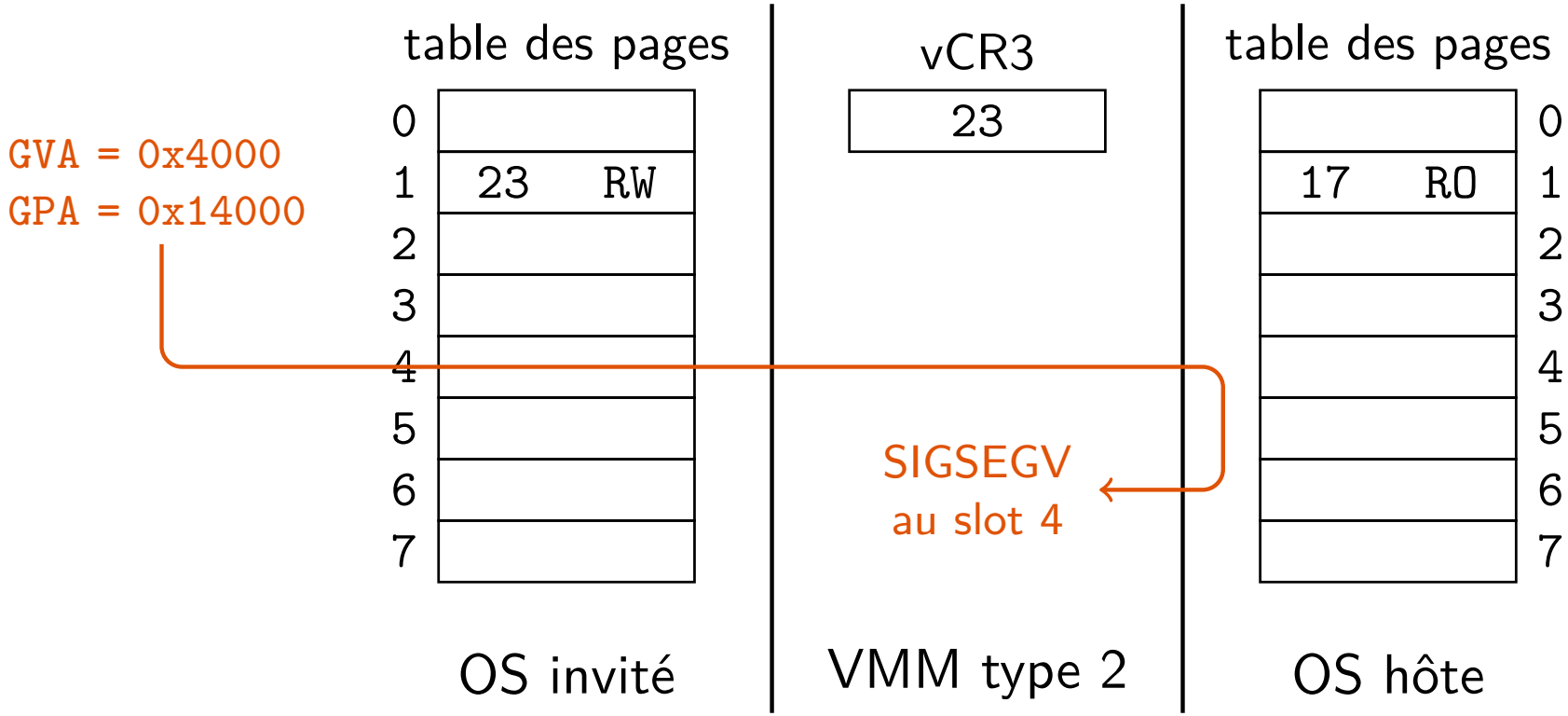
- Le fonctionnement d'une *shadow page table* en type 2 est très similaire
  - Le VMM protège la table invitée (avec `mprotect`)

Shadowing de table de pages : VMM type 2



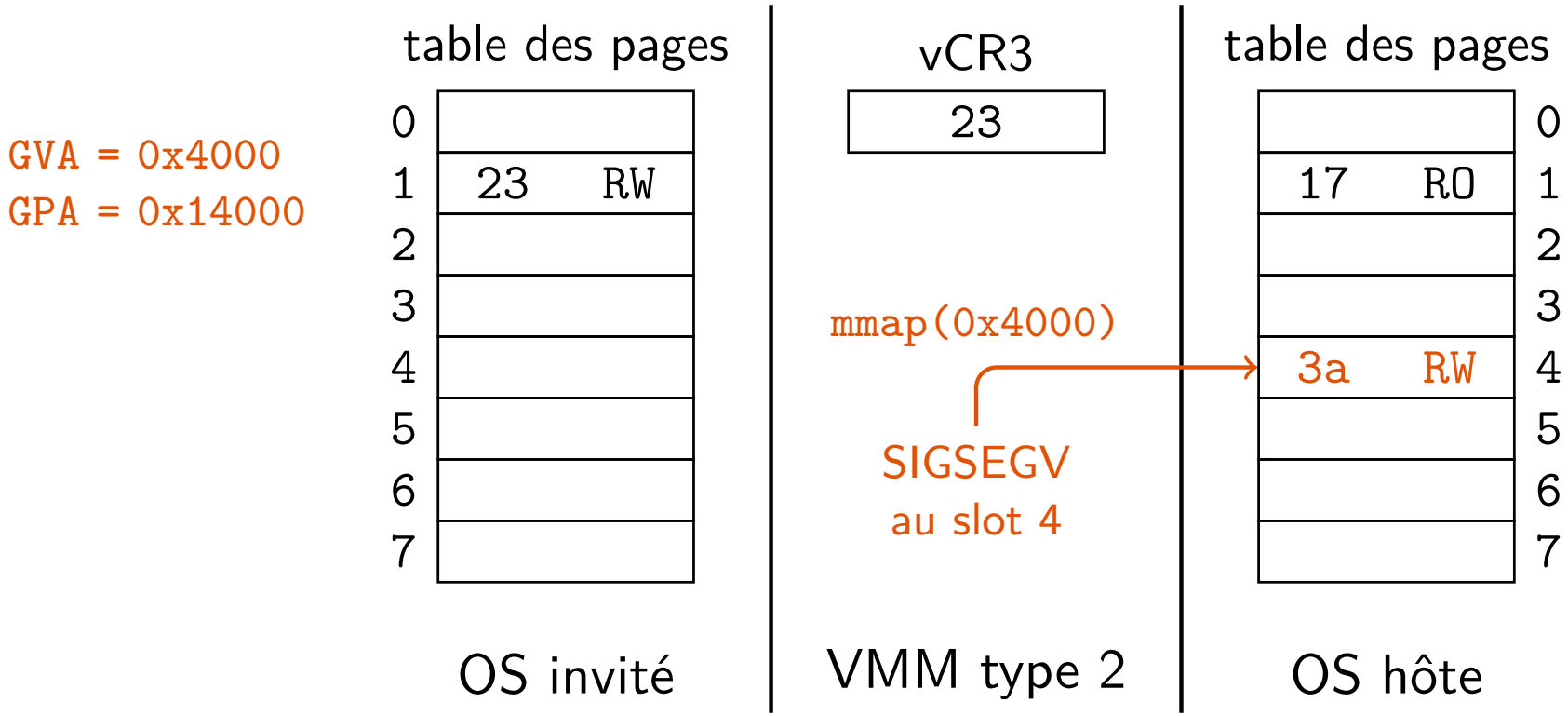
- Le fonctionnement d'une *shadow page table* en type 2 est très similaire
  - Le VMM protège la table invitée (avec `mprotect`)
  - La faute est interceptée par l'OS hôte

Shadowing de table de pages : VMM type 2



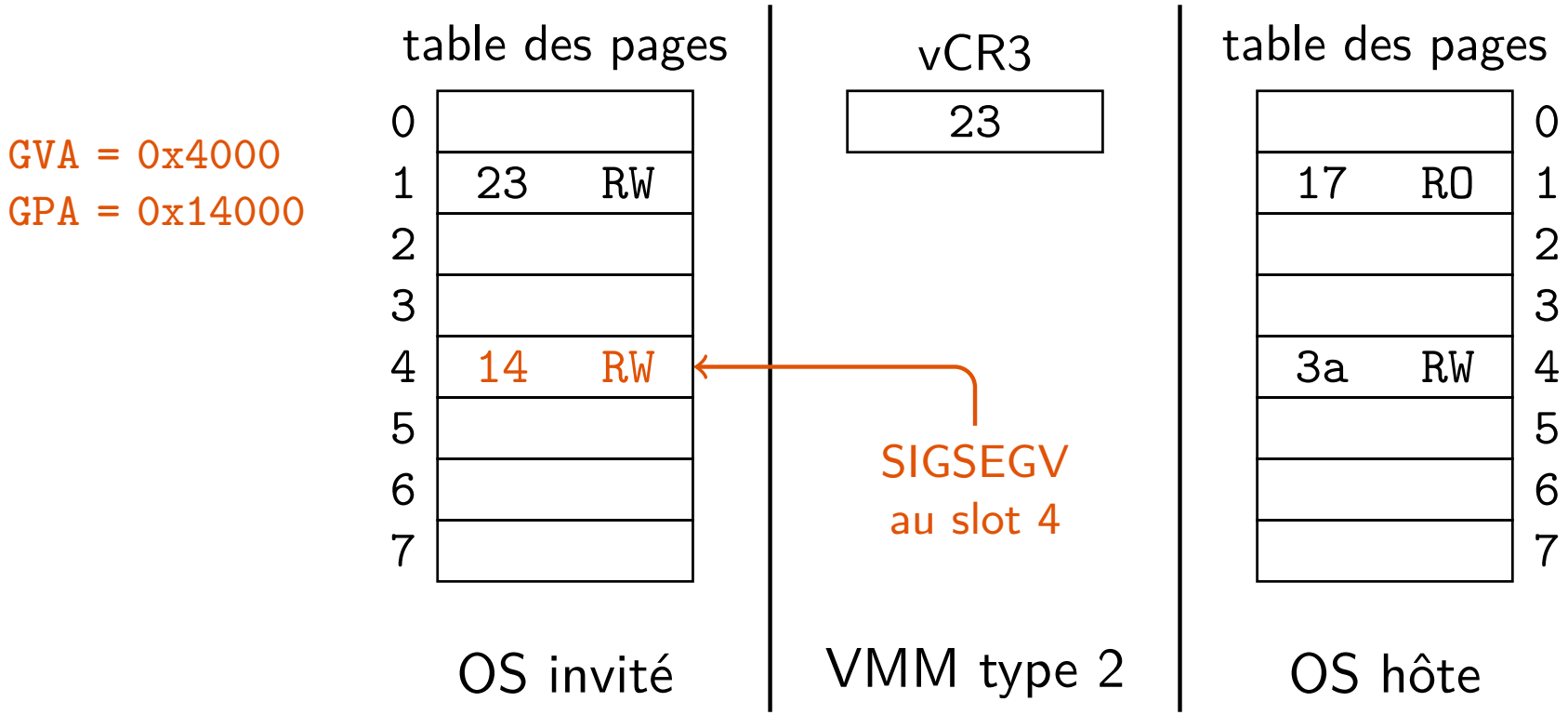
- Le fonctionnement d'une *shadow page table* en type 2 est très similaire
  - Le VMM protège la table invitée (avec `mprotect`)
  - La faute est interceptée par l'OS hôte puis redirigée vers le VMM

Shadowing de table de pages : VMM type 2



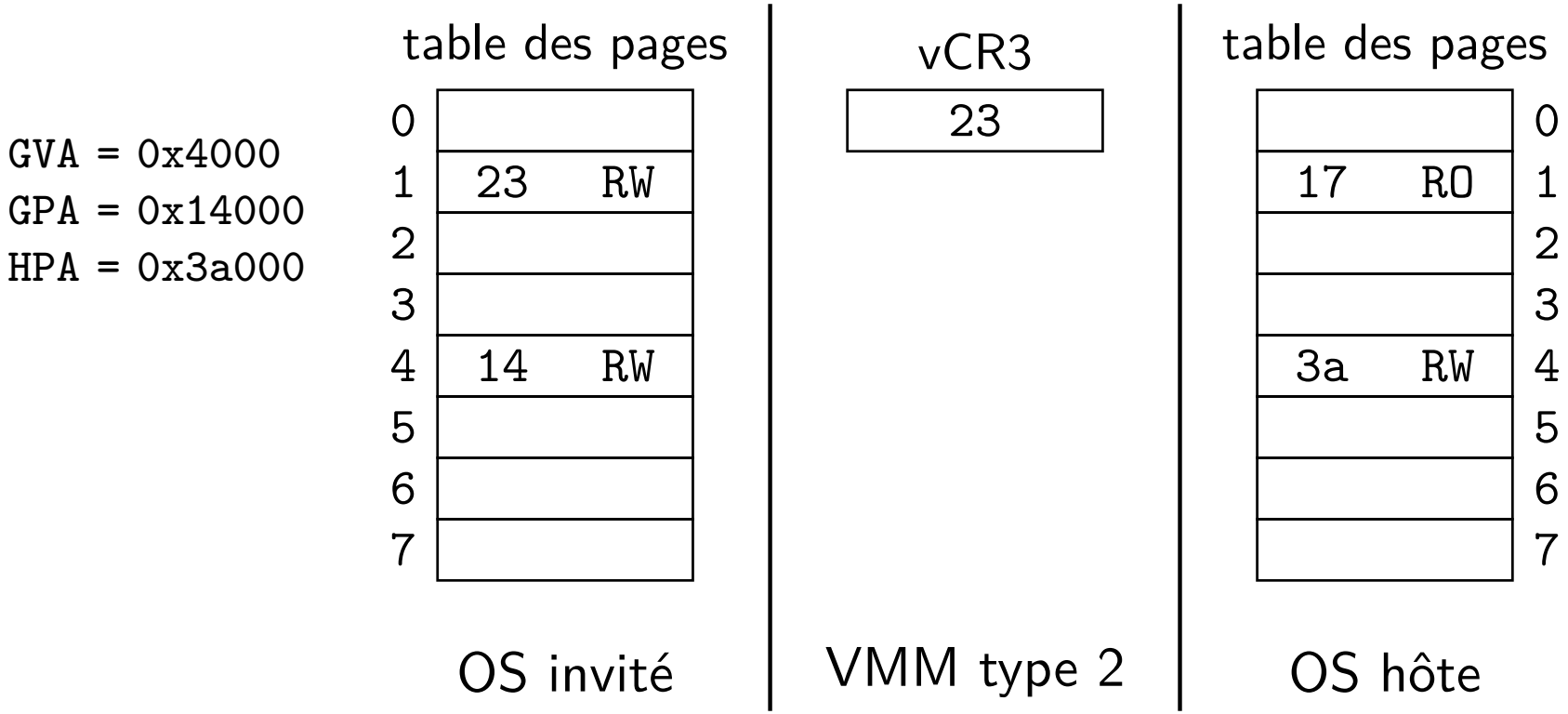
- Le fonctionnement d'une *shadow page table* en type 2 est très similaire
  - Le VMM protège la table invitée (avec `mprotect`)
  - La faute est interceptée par l'OS hôte puis redirigée vers le VMM
  - Le VMM demande à l'OS invité de mapper l'adresse virtuelle fautive

Shadowing de table de pages : VMM type 2



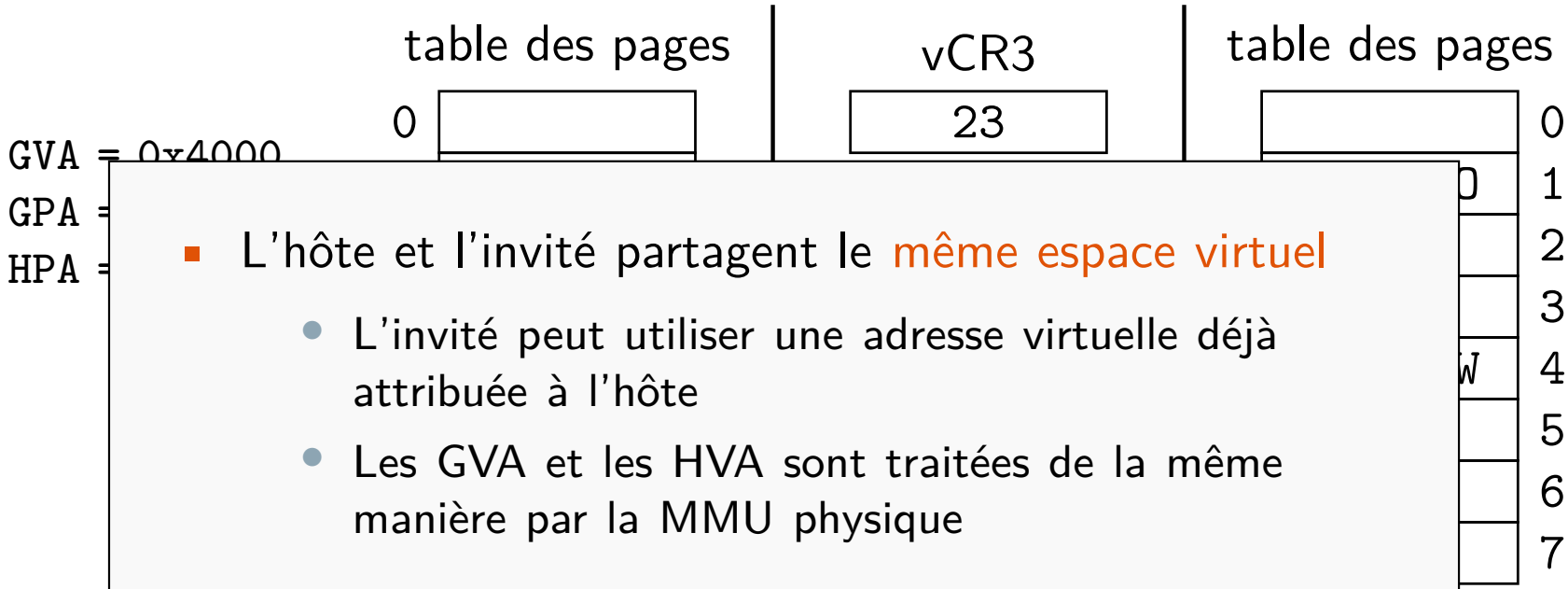
- Le fonctionnement d'une *shadow page table* en type 2 est très similaire
  - Le VMM protège la table invitée (avec `mprotect`)
  - La faute est interceptée par l'OS hôte puis redirigée vers le VMM
  - Le VMM demande à l'OS invité de mapper l'adresse virtuelle fautive
  - Le VMM modifie la table des pages invitée pour écrire l'adresse physique

Shadowing de table de pages : VMM type 2



- Le fonctionnement d'une *shadow page table* en type 2 est très similaire
  - Le VMM protège la table invitée (avec `mprotect`)
  - La faute est interceptée par l'OS hôte puis redirigée vers le VMM
  - Le VMM demande à l'OS invité de mapper l'adresse virtuelle fautive
  - Le VMM modifie la table des pages invitée pour écrire l'adresse physique

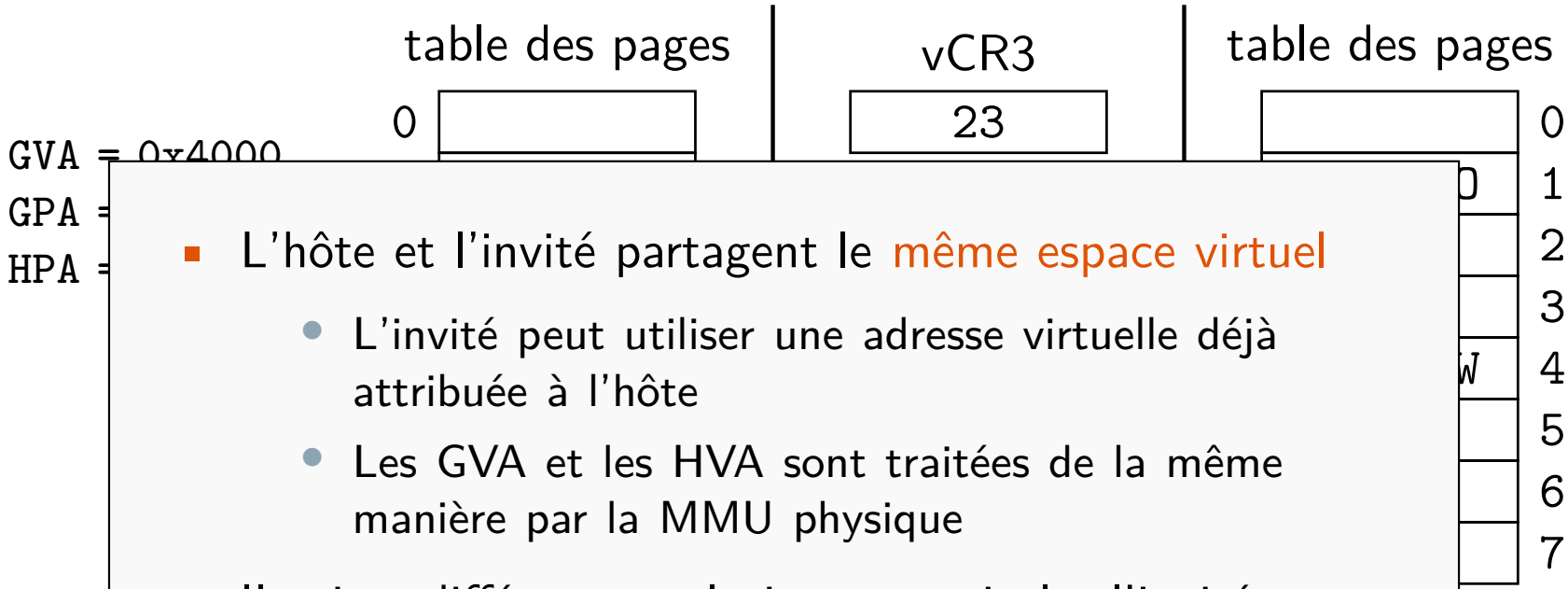
Shadowing de table de pages : VMM type 2



- L'hôte et l'invité partagent le même espace virtuel
  - L'invité peut utiliser une adresse virtuelle déjà attribuée à l'hôte
  - Les GVA et les HVA sont traitées de la même manière par la MMU physique

- Le f laire
  - La faute est interceptée par l'OS hôte puis redirigée vers le VMM
  - Le VMM demande à l'OS invité de mapper l'adresse virtuelle fautive
  - Le VMM modifie la table des pages invitée pour écrire l'adresse physique

Shadowing de table de pages : VMM type 2



- L'hôte et l'invité partagent le même espace virtuel
  - L'invité peut utiliser une adresse virtuelle déjà attribuée à l'hôte
  - Les GVA et les HVA sont traitées de la même manière par la MMU physique
- Il existe différentes solutions pour isoler l'invité
  - Utiliser un espace virtuel dédié pour l'invité
  - Émuler tout accès à une zone réservée à l'hôte
  - Déplacer l'hôte en cas de conflit

- Le f laire
  - La faute est interceptée par l'OS hôte puis redirigée vers le VMM
  - Le VMM demande à l'OS invité de mapper l'adresse virtuelle fautive
  - Le VMM modifie la table des pages invitée pour écrire l'adresse physique

Virtualisation mémoire et *shadowing* : résumé

- Le système invité définit une correspondance entre GVA et GPA
- Le système hôte utilise la MMU pour associer HVA et HPA
- Le VMM assure une traduction de GVA vers HPA sans collision
  - P1 Entre les différents espaces virtuels invités
  - P2 Entre les espaces virtuels invités et l'espace virtuel hôte
- La première méthode est la traduction logicielle
  - P1 Le VMM émule la traduction via une MMU logicielle
  - P2 Les GPA sont traduites une seconde fois pour éviter les collisions
- La deuxième méthode est la *shadow page table*
  - P1 Le VMM met à jour la table des pages hôte en fonction des actions de l'invité $\rightarrow$ la MMU physique traduit le GVA en HPA
  - P2 Le VMM utilise une combinaisons d'autres méthodes pour éviter les collisions entre l'hôte et l'invité

Plan du cours

1. Virtualisation d'instructions
   - Simulation *cycle accurate* et émulation
   - Émulation d'instructions et état virtuel
   - *Dynamic Binary Translation* et *basic blocks*

2. Virtualisation de ressources
   - Virtualisation mémoire et *shadowing*
   - Paravirtualisation et interface matérielle

3. Virtualisation comme isolation
   - Exécution directe et interception
   - Support materiel à la virtualisation

Interface matérielle et coût d'interception

- Le *shadowing* de table des pages évite la traduction logicielle des adresse $\rightarrow$ accès mémoire plus efficace
  - Surcoût pour toute écriture dans la table des pages
  - Surcoût pour toute modification du `vCR3` (*context switch* invité)

- Le *shadowing* est possible pour du matériel configuré par l'hôte mais dont l'exécution est automatique $\rightarrow$ MMU, contrôleur d'interruptions, ...

- Impossible pour le matériel aux actions commandées explicitement par l'hôte $\rightarrow$ Contrôleur d'entrées/sorties, instructions privilégiées, ...

Interface matérielle et coût d'interception

- Le *shadowing* de table des pages évite la traduction logicielle des adresse → accès mémoire plus efficace
  - Surcoût pour toute écriture dans la table des pages
  - Surcoût pour toute modification du `vCR3` (*context switch* invité)

- Le *shadowing* est possible pour du matériel configuré par l'hôte mais dont l'exécution est automatique → MMU, contrôleur d'interruptions, ...

- Impossible pour le matériel aux actions commandées explicitement par l'hôte → Contrôleur d'entrées/sorties, instructions privilégiées, ...

- Exemple : contrôleur disque ATA

```
// lecture d'un secteur en mode DMA (out = instr. privilégiée)
out(BUS_CMD, 0);            // réinitialiser le bus maître
out(DEV_PRDT, &prdt);       // sélectionner une adresse cible
out(DEV_DRIVE, drivenum); // sélectionner un numéro de disque
out(DEV_COUNT, 1);          // sélectionner le nombre de secteurs
out(DEV_CMD, 0xc8);         // envoyer la commande DMA au disque
out(BUS_CMD, 0x09);         // activer le bus maître
```

Interface matérielle et coût d'interception

- Le *shadowing* de table des pages évite la traduction logicielle des adresse → accès mémoire plus efficace
  - Surcoût pour toute écriture dans la table des pages
  - Surcoût pour toute modification du `vCR3` (*context switch* invité)

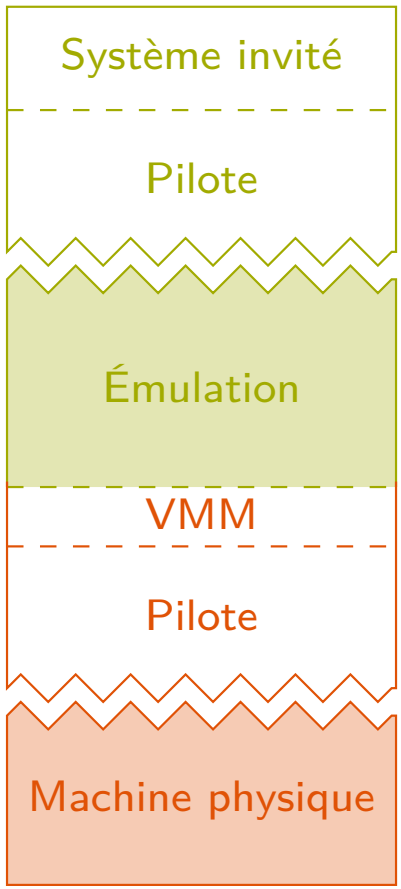
- Le *shadowing* est possible pour du matériel configuré par l'hôte mais dont l'exécution est automatique → MMU, contrôleur d'interruptions, ...

- Impossible pour le matériel aux actions commandées explicitement par l'hôte → Contrôleur d'entrées/sorties, instructions privilégiées, ...

- Exemple : contrôleur disque ATA
- Chaque commande **`out`** est une commande privilégiée d'entrée/sortie
  - Le VMM intercepte et décode chacune d'elle
  - Le VMM met à jour l'état du contrôleur de disque virtuel
  - À la fin de la séquence, le VMM démarre un transfert disque physique

Paravirtualisation : principe

- La communication entre l'invité et le VMM est complexe
  - Interface machine complexe
  - Donc pilotes complexes
- Cette complexité a un impact sur les performances

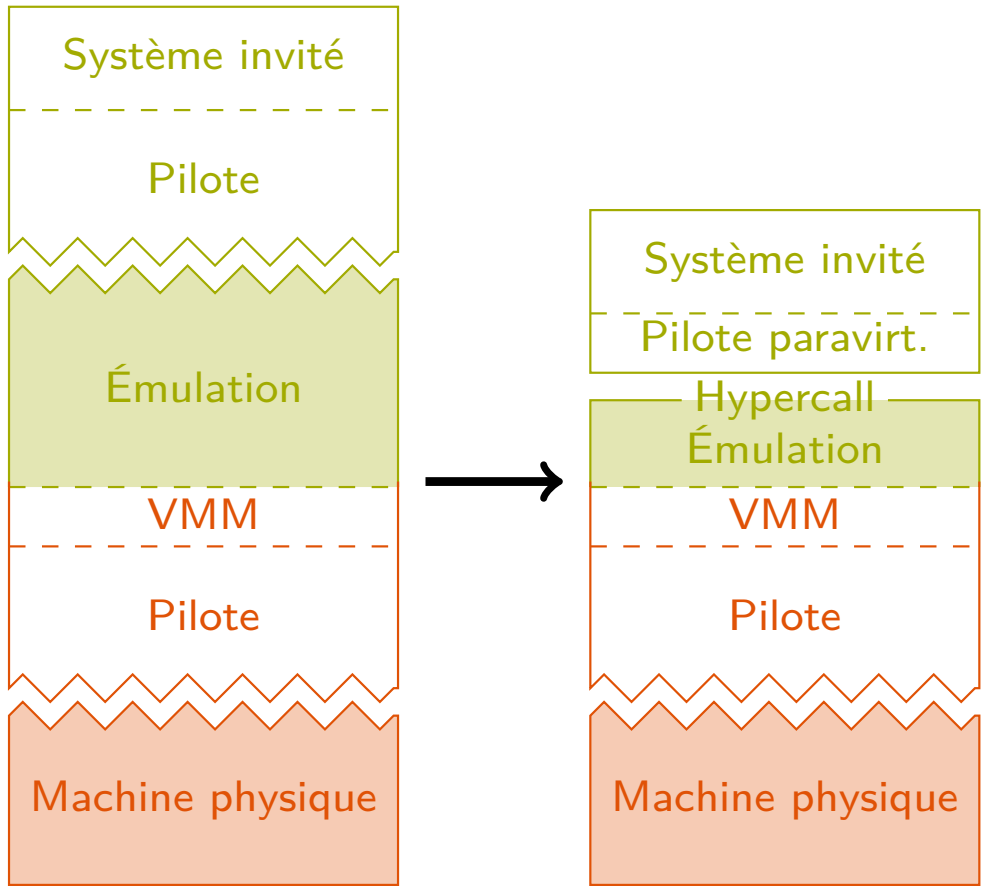


Émulation

Paravirtualisation : principe

- La communication entre l'invité et le VMM est complexe
  - Interface machine complexe
  - Donc pilotes complexes
- Cette complexité a un impact sur les performances

- Solution : supprimer l'interface machine virtuelle
  - Modifier l'invité pour faire directement appel au VMM
  - L'invité sait qu'il est virtualisé
  - Communication par *hypercall*



Paravirtualisation et *hypercall*

Code système invité

```
...
out    $BUS_CMD, %rax
out    $DEV_PDRT, %rdi
out    $DEV_DRIVE, %rsi
...
```

Code résultat

```
...
bl   <__vmm_emulate_out(BUS_CMD, $R0)>
bl   <__vmm_emulate_out(DEV_PRDT, $R8)>
bl   <__vmm_emulate_out(BUS_DRIVE, $R10)>
...
```

- Pour un invité sans paravirtualisation

Paravirtualisation et *hypercall*

Code système invité

```
...
out    $BUS_CMD, %rax
out    $DEV_PDRT, %rdi
out    $DEV_DRIVE, %rsi
...
```

Code résultat

```
...
bl   <__vmm_emulate_out(BUS_CMD, $R0)>
bl   <__vmm_emulate_out(DEV_PRDT, $R8)>
bl   <__vmm_emulate_out(BUS_DRIVE, $R10)>
...
```

- Pour un invité sans paravirtualisation
  - Le VMM doit émuler chaque instruction privilégiée

Paravirtualisation et *hypercall*

Code système invité

```
...
out    $BUS_CMD, %rax
out    $DEV_PDRT, %rdi
out    $DEV_DRIVE, %rsi
...
```

Code VMM : `__vmm_emulate_out()`

```
if (get_mode() != KERNEL)
    inject_exception(GPF);
if (out_port == BUS_CMD)
    emulate_update_bus(out_val);
} else ...
```

- Pour un invité sans paravirtualisation
  - Le VMM doit émuler chaque instruction privilégiée

Paravirtualisation et *hypercall*

Code système invité

```
...
out    $BUS_CMD, %rax
out    $DEV_PDRT, %rdi
out    $DEV_DRIVE, %rsi
...
```

Code VMM : `__vmm_emulate_out()`

```
if (get_mode() != KERNEL)
    inject_exception(GPF);
if (out_port == BUS_CMD)
    emulate_update_bus(out_val);
} else ...
```

- Pour un invité sans paravirtualisation
  - Le VMM doit émuler chaque instruction privilégiée

Paravirtualisation et *hypercall*

Code paravirtualisé

```
...
mov    $0, %rax
mov    $HYPER_CODE, (%rax)
...
```

Code résultat

```
...
mov   $R0, 0
mov   $R1, HYPER_CODE
str   $R1, $R0
...
```

- Pour un invité sans paravirtualisation
  - Le VMM doit émuler chaque instruction privilégiée
- Un système paravirtualisé utilise un unique *hypercall* à la place
  - Peut être n'importe quelle instruction que le VMM intercepte et qui ne correspond à aucune action légitime pour le matériel

Paravirtualisation et *hypercall*

Code paravirtualisé

```
...
mov    $0, %rax
mov    $HYPER_CODE, (%rax)
...
```

Code résultat

```
...
mov   $R0, 0
mov   $R1, HYPER_CODE
str   $R1, $R0
...
```

- Pour un invité sans paravirtualisation
  - Le VMM doit émuler chaque instruction privilégiée
- Un système paravirtualisé utilise un unique *hypercall* à la place
  - Peut être n'importe quelle instruction que le VMM intercepte et qui ne correspond à aucune action légitime pour le matériel

Paravirtualisation et *hypercall*

Code paravirtualisé

```
...
mov    $0, %rax
mov    $HYPER_CODE, (%rax)
...
```

Code résultat

```
...
mov   $R0, 0
mov   $R1, HYPER_CODE
str   $R1, $R0
...
```

- Pour un invité sans paravirtualisation
  - Le VMM doit émuler chaque instruction privilégiée
- Un système paravirtualisé utilise un unique *hypercall* à la place
  - Peut être n'importe quelle instruction que le VMM intercepte et qui ne correspond à aucune action légitime pour le matériel

Paravirtualisation et *hypercall*

Code paravirtualisé

```
...
mov    $0, %rax
mov    $HYPER_CODE, (%rax)
...
```

Code résultat

```
...
mov   $R0, 0
mov   $R1, HYPER_CODE
str   $R1, $R0
...
```

- Pour un invité sans paravirtualisation
  - Le VMM doit émuler chaque instruction privilégiée
- Un système paravirtualisé utilise un unique *hypercall* à la place
  - Peut être n'importe quelle instruction que le VMM intercepte et qui ne correspond à aucune action légitime pour le matériel

Paravirtualisation et *hypercall*

Code paravirtualisé

```
...
mov    $0, %rax
mov    $HYPER_CODE, (%rax)
...
```

Code VMM : `__vmm_page_fault()`

```
if (detect_hypercall()) {
    if (get_rcx() == DMA_READ) {
        dest_addr = get_rdi();
        src_sector = get_rsi();
        ...
    } else ...
} else {
    handle_guest_page_fault();
}
```

- Pour un invité sans paravirtualisation
  - Le VMM doit émuler chaque instruction privilégiée
- Un système paravirtualisé utilise un unique *hypercall* à la place
  - Peut être n'importe quelle instruction que le VMM intercepte et qui ne correspond à aucune action légitime pour le matériel
  - Le VMM discrimine les *hypercalls* des fautes avec une ABI prédéfinie

Paravirtualisation et *hypercall*

Code paravirtualisé

```
...
mov    $0, %rax
mov    $HYPER_CODE, (%rax)
...
```

Code VMM : `__vmm_page_fault()`

```
if (detect_hypercall()) {
    if (get_rcx() == DMA_READ) {
        dest_addr = get_rdi();
        src_sector = get_rsi();
        ...
    } else ...
} else {
    handle_guest_page_fault();
}
```

- Pour un invité sans paravirtualisation
  - Le VMM doit émuler chaque instruction privilégiée
- Un système paravirtualisé utilise un unique *hypercall* à la place
  - Peut être n'importe quelle instruction que le VMM intercepte et qui ne correspond à aucune action légitime pour le matériel
  - Le VMM discrimine les *hypercalls* des fautes avec une ABI prédéfinie
  - Si c'est un *hypercall*, le VMM décode selon l'ABI définie de l'*hypercall* et traite la demande si elle est légitime

Paravirtualisation et interface matérielle : résumé

- L'interface matérielle des périphériques est complexe
  - S'adapte bien aux contraintes d'un circuit matériel
  - Pilotes logiciels peu efficaces et difficiles à maintenir

- En temps normal, le VMM doit émuler cette interface matérielle
  - Interface complexe à émuler, mal adaptée au logiciel
  - Pilotes invités peu efficaces et difficiles à maintenir

- Le VMM et le système invité peuvent coopérer pour être plus efficaces

- Les interfaces matérielles complexes sont remplacées par des interfaces paravirtualisées basées sur les *hypercalls*
  - *Hypercall* simple et rapide à décoder par le VMM, facile à maintenir
  - Pilotes invités paravirtualisés simples et efficaces
  - Le système invité est modifié pour s'exécuter dans un VMM donné

Plan du cours

1. Virtualisation d'instructions
   - Simulation *cycle accurate* et émulation
   - Émulation d'instructions et état virtuel
   - *Dynamic Binary Translation* et *basic blocks*

2. Virtualisation de ressources
   - Virtualisation mémoire et *shadowing*
   - Paravirtualisation et interface matérielle

3. Virtualisation comme isolation
   - Exécution directe et interception
   - Support materiel à la virtualisation

Des émulateurs aux hyperviseurs

- Usages d'une machine virtuelle
  - Simuler du nouveau matériel → conception de puce
  - Porter des logiciels sur différentes architectures → émulateur gameboy
  - Partage de ressources → exécution simultanée de Linux et Windows
  - Isolation de services → exécution simultanée de plusieurs Linux

- Objectif : simuler l'interface d'une machine virtuelle

Des émulateurs aux hyperviseurs

- Usages d'une machine virtuelle
  - Simuler du nouveau matériel → conception de puce
  - Porter des logiciels sur différentes architectures → émulateur gameboy
  - Partage de ressources → exécution simultanée de Linux et Windows
  - Isolation de services → exécution simultanée de plusieurs Linux

- Objectif : simuler l'interface d'une machine virtuelle semblable à la machine physique
  - Pas besoin de traduire les instructions non privilégiées
  - Besoin d'intercepter les instructions privilégiées et les sauts

Code système invité

```
add     $8, %rax
cli
add     $16, %rcx
mov     %rax, 0(%rdx)
cmp     %rax, %rcx
jne     0x14a9
```

Code résultat

```
add     $8, %rax
call    <__vmm_emulate_cli>
add     $16, %rcx
mov     %rax, 0(%rdx)
cmp     %rax, %rcx
jne     <__vmm_fetch(0x14a9)>
jmp     <__vmm_fetch(0x14d3)>
```

Des émulateurs aux hyperviseurs

- Usages d'une machine virtuelle
  - Simuler du nouveau matériel → conception de puce
  - Porter des logiciels sur différentes architectures → émulateur gameboy
  - Partage de ressources → exécution simultanée de Linux et Windows
  - Isolation de services → exécution simultanée de plusieurs Linux

- Objectif : simuler l'interface d'une machine virtuelle semblable à la machine physique
  - Pas besoin de traduire les instructions non privilégiées
  - Besoin d'intercepter les instructions privilégiées et les sauts

Code système invité

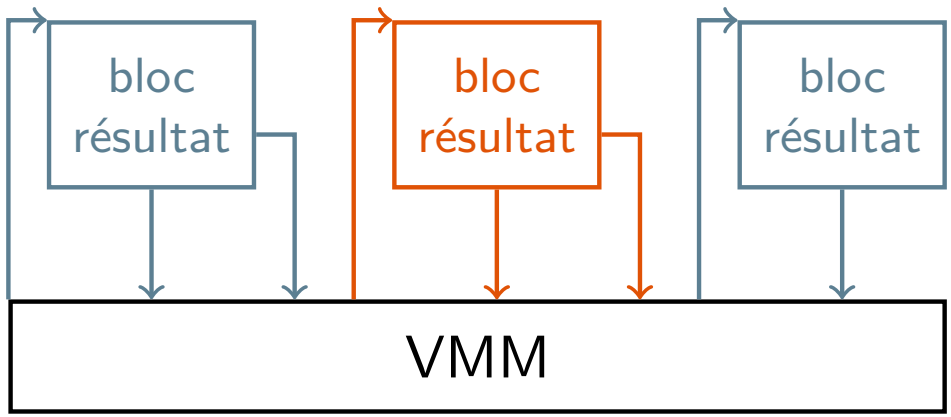
```
add     $8, %rax
cli
add     $16, %rcx
mov     %rax, 0(%rdx)
cmp     %rax, %rcx
jne     0x14a9
```

Code résultat

```
add     $8, %rax
call    <__vmm_emulate_cli>
add     $16, %rcx
mov     %rax, 0(%rdx)
cmp     %rax, %rcx
jne     <__vmm_fetch(0x14a9)>
jmp     <__vmm_fetch(0x14d3)>
```

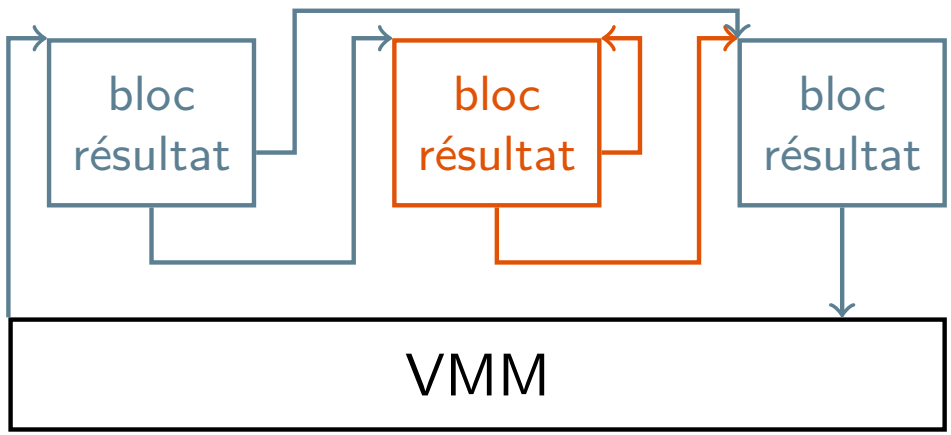
Exécution directe du code invité

- La *Dynamic Binary Translation* est plus rapide quand les architectures cibles et sources sont les mêmes
  - Les blocs de code non privilégié n'ont qu'un surcoût de chaînage



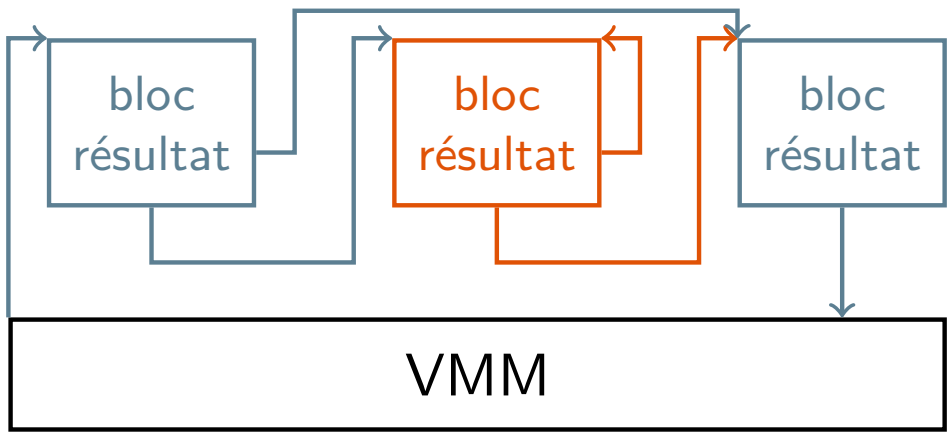
Exécution directe du code invité

- La *Dynamic Binary Translation* est plus rapide quand les architectures cibles et sources sont les mêmes
  - Les blocs de code non privilégié n'ont qu'un surcoût de chaînage
  - Ce surcoût disparait quand tous les blocs sont traduits (chaînage direct)
  - Les codes source et résultat sont identiques



Exécution directe du code invité

- La *Dynamic Binary Translation* est plus rapide quand les architectures cibles et sources sont les mêmes
  - Les blocs de code non privilégié n'ont qu'un surcoût de chaînage
  - Ce surcoût disparait quand tous les blocs sont traduits (chaînage direct)
  - Les codes source et résultat sont identiques



- La traduction est inutile pour le code non privilégié
  - Nouvelle stratégie : exécuter directement le code du système invité
  - Intercepter uniquement les instructions privilégiées et les émuler

Interception d'instructions privilégiées

Code système invité

```
mov     %rax, (%rdx)
cli
```

Code VMM : `main`

```
setup_handlers();
branch_to_guest();
```

- Après sa phase d'initialisation, l'hyperviseur se branche directement sur le code du système invité en mode utilisateur

Interception d'instructions privilégiées

Code système invité

```
mov     %rax, (%rdx)
cli
```

Code VMM : `main`

```
setup_handlers();
branch_to_guest();
```

- Après sa phase d'initialisation, l'hyperviseur se branche directement sur le code du système invité en mode utilisateur

Interception d'instructions privilégiées

Code système invité

```
mov     %rax, (%rdx)
cli
```

Code VMM : `main`

```
setup_handlers();
branch_to_guest();
```

- Après sa phase d'initialisation, l'hyperviseur se branche directement sur le code du système invité en mode utilisateur

Interception d'instructions privilégiées

Code système invité

```
mov     %rax, (%rdx)
cli
```

Code VMM : `main`

```
setup_handlers();
branch_to_guest();
```

- Après sa phase d'initialisation, l'hyperviseur se branche directement sur le code du système invité en mode utilisateur
- Les instructions non privilégiées sont alors exécutées normalement
  - L'hyperviseur assure l'isolation mémoire par pagination

Interception d'instructions privilégiées

Code système invité

```
mov     %rax, (%rdx)
cli
```

Code VMM : `main`

```
setup_handlers();
branch_to_guest();
```

- Après sa phase d'initialisation, l'hyperviseur se branche directement sur le code du système invité en mode utilisateur
- Les instructions non privilégiées sont alors exécutées normalement
  - L'hyperviseur assure l'isolation mémoire par pagination
- Le processeur n'exécute pas les instructions privilégiées en mode utilisateur

Interception d'instructions privilégiées

Code système invité

```
mov     %rax, (%rdx)
cli
```

Code VMM : `fault_handler(GPF)`

```
inst = decode_faulty_inst();
emulate_instruction(inst);
```

- Après sa phase d'initialisation, l'hyperviseur se branche directement sur le code du système invité en mode utilisateur
- Les instructions non privilégiées sont alors exécutées normalement
  - L'hyperviseur assure l'isolation mémoire par pagination
- Le processeur n'exécute pas les instructions privilégiées en mode utilisateur
  - À la place, le processeur déclenche une faute de protection
  - La faute est traitée par l'hyperviseur et l'instruction invitée est émulée

Interception d'instructions privilégiées

Code système invité

```
mov     %rax, (%rdx)
cli
```

Code VMM : `fault_handler(GPF)`

```
inst = decode_faulty_inst();
emulate_instruction(inst);
```

- Après sa phase d'initialisation, l'hyperviseur se branche directement sur le code du système invité en mode utilisateur
- Les instructions non privilégiées sont alors exécutées normalement
  - L'hyperviseur assure l'isolation mémoire par pagination
- Le processeur n'exécute pas les instructions privilégiées en mode utilisateur
  - À la place, le processeur déclenche une faute de protection
  - La faute est traitée par l'hyperviseur et l'instruction invitée est émulée

Interception d'instructions privilégiées

Code système invité

```
mov     %rax, (%rdx)
cli
```

Code VMM : `signal_handler(SIGILL)`

```
inst = decode_faulty_inst();
emulate_instruction(inst);
```

- Après sa phase d'initialisation, l'hyperviseur se branche directement sur le code du système invité en mode utilisateur
- Les instructions non privilégiées sont alors exécutées normalement
  - L'hyperviseur assure l'isolation mémoire par pagination
- Le processeur n'exécute pas les instructions privilégiées en mode utilisateur
  - À la place, le processeur déclenche une faute de protection
  - La faute est traitée par l'hyperviseur et l'instruction invitée est émulée
  - Dans le cas d'un hyperviseur de type 2, l'OS hôte redirige la faute de protection vers l'hyperviseur

Exécution directe et instructions *silent fail*

- Certaines instructions ont un comportement différent en mode utilisateur et en mode noyau
  - En x86, l'instruction `popf` a la sémantique suivante

```
POPF:
    Charger cpu.rflags depuis la pile
        sauf les flags VIF, VIP, VM, IOPL et IF
    Si mode noyau:
        charger aussi IOPL et IF depuis la pile
```

Exécution directe et instructions *silent fail*

- Certaines instructions ont un comportement différent en mode utilisateur et en mode noyau
  - En x86, l'instruction `popf` a la sémantique suivante

```
POPF:
    Charger cpu.rflags depuis la pile
        sauf les flags VIF, VIP, VM, IOPL et IF
    Si mode noyau:
        charger aussi IOPL et IF depuis la pile
```

- Si le système invité exécute l'instruction `popf` en mode noyau, il s'attend à voir les flags `IOPL` et `IF` modifiés
  - L'exécution de `popf` en mode utilisateur ne cause pas de faute
  - Pas d'interception par l'hyperviseur $\rightarrow$ pas d'émulation

- La présence d'instruction *silent fail* rend l'exécution directe impossible sur l'architecture concernée

Exécution directe et instructions *silent fail*

- Certaines instructions ont un comportement différent en mode utilisateur et en mode noyau
  - En x86, l'instruction `popf` a la sémantique suivante

```
POPF:
    Charger cpu.rflags depuis la pile
        sauf les flags VIF, VIP, VM, IOPL et IF
    Si mode noyau:
        charger aussi IOPL et IF depuis la pile
```

- Si le système invité exécute l'instruction `popf` en mode noyau, il s'attend à voir les flags `IOPL` et `IF` modifiés
  - L'exécution de `popf` en mode utilisateur ne cause pas de faute
  - Pas d'interception par l'hyperviseur $\rightarrow$ pas d'émulation

- La présence d'instruction *silent fail* rend l'exécution directe impossible sur l'architecture concernée
  - Possible avec la paravirtualisation

Exécution directe et interception : résumé

- Un cas particulier de moniteur de machine virtuelle est l'hyperviseur
  - La machine virtuelle exposée au système invité a la même architecture que la machine physique
  - La traduction binaire dynamique y est plus rapide $\rightarrow$ copie des instructions non privilégiées

- L'exécution directe est la méthode qui consiste à exécuter directement le code invité en mode utilisateur
  - Isolation mémoire identique à celle des processus hôtes
  - Instructions privilégiées interceptées et émulées par l'hyperviseur

- Certains jeux d'instructions contiennent des instructions *silent fail*
  - Comportement différent en mode noyau et mode utilisateur
  - Interdisent l'exécution directe pour l'architecture en question

- Plus efficace que la *Dynamic Binary Translation* (facteur x1.3)
- Exemple d'hyperviseur qui utilise l'exécution directe
  - Xen

Plan du cours

1. Virtualisation d'instructions
   - Simulation *cycle accurate* et émulation
   - Émulation d'instructions et état virtuel
   - *Dynamic Binary Translation* et *basic blocks*
2. Virtualisation de ressources
   - Virtualisation mémoire et *shadowing*
   - Paravirtualisation et interface matérielle
3. Virtualisation comme isolation
   - Exécution directe et interception
   - Support materiel à la virtualisation

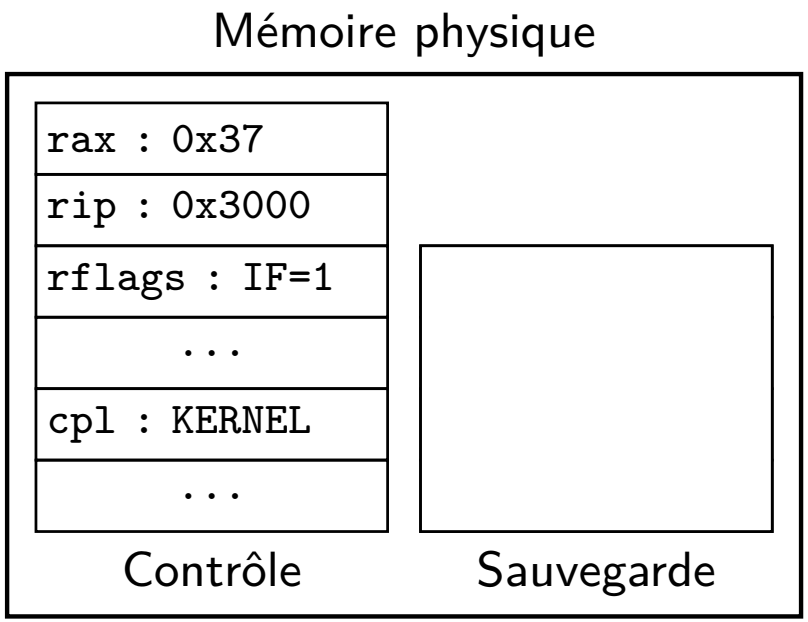
Coûts de la virtualisation

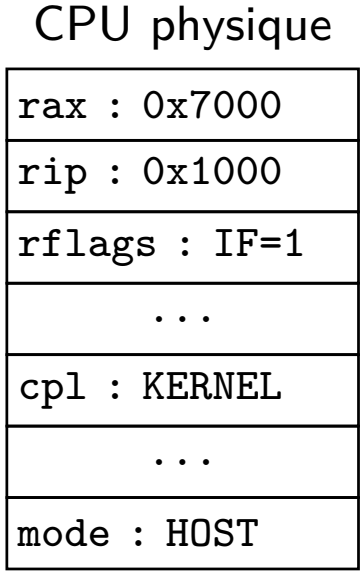
- Une DBT efficace ou une exécution directe supprime le sucoût d'exécution des instructions non privilégiées
- Les opérations privilégiées paravirtualisées ne sont pas sensiblement dégradées

- Les opérations privilégiées qui réduisent les performances de l'invité sont notamment (sans paravirtualisation) :
  - Le *shadowing* de table de pages (coût du *context switch*)
  - L'émulation de matériel d'entrées/sorties (disque et réseau)
- L'exécution directe est impossible pour l'architecture x86 du fait des instructions *silent fail*

- Les proceseurs x86 récents (Intel et AMD) fournissent une assistance matérielle à la virtualisation

Assistance matérielle à la virtualisation : principe



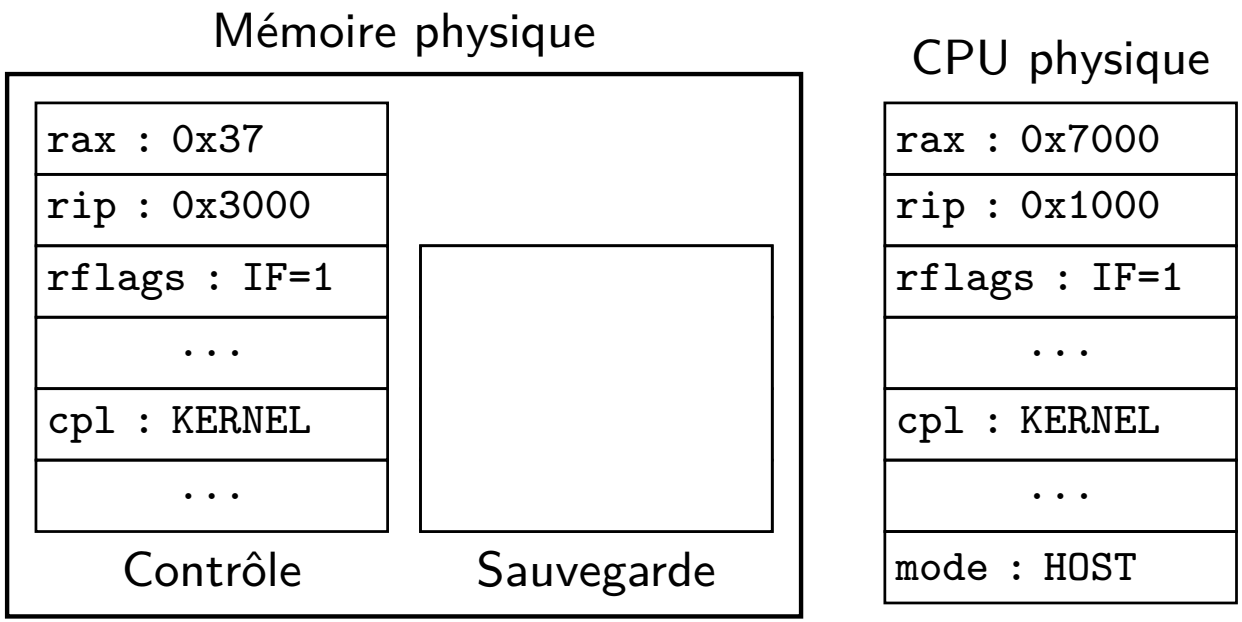


Code hyperviseur

```
...
vmrun
call    <on_vmexit>
...
```

- L'hyperviseur définit une zone de contrôle par vCPU et une zone de sauvegarde par CPU.

Assistance matérielle à la virtualisation : principe

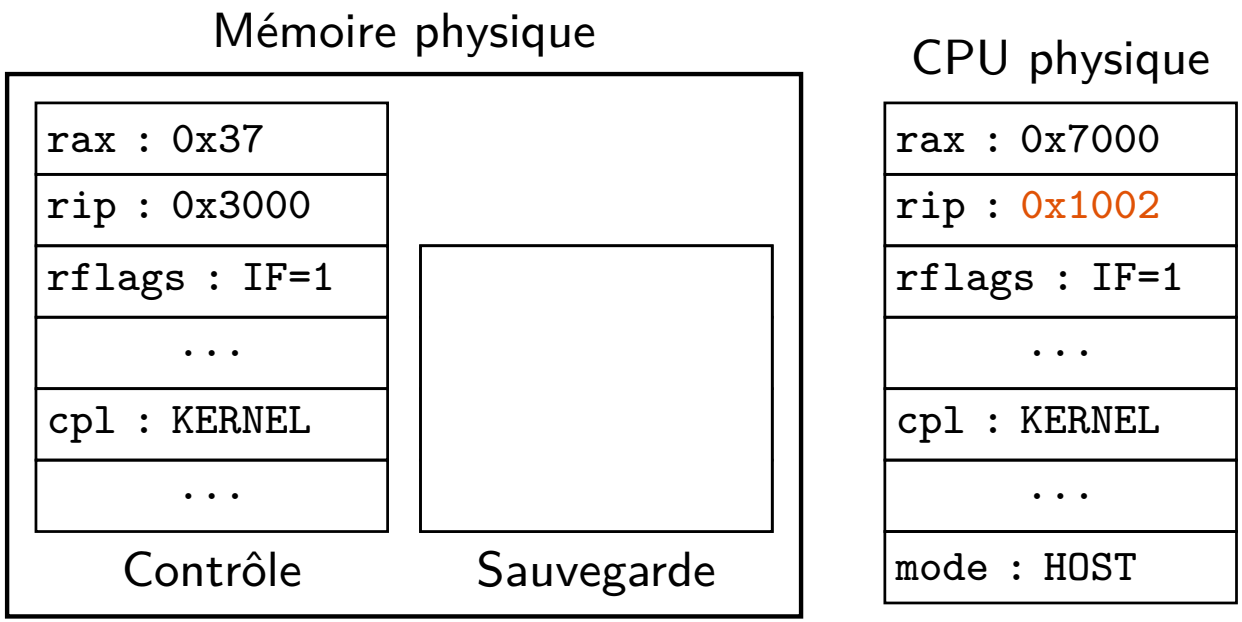


Code hyperviseur

```
...
vmrun
call    <on_vmexit>
...
```

- L'hyperviseur définit une zone de contrôle par vCPU et une zone de sauvegarde par CPU.
- Une instruction dédiée permet au processeur de passer en mode invité

Assistance matérielle à la virtualisation : principe

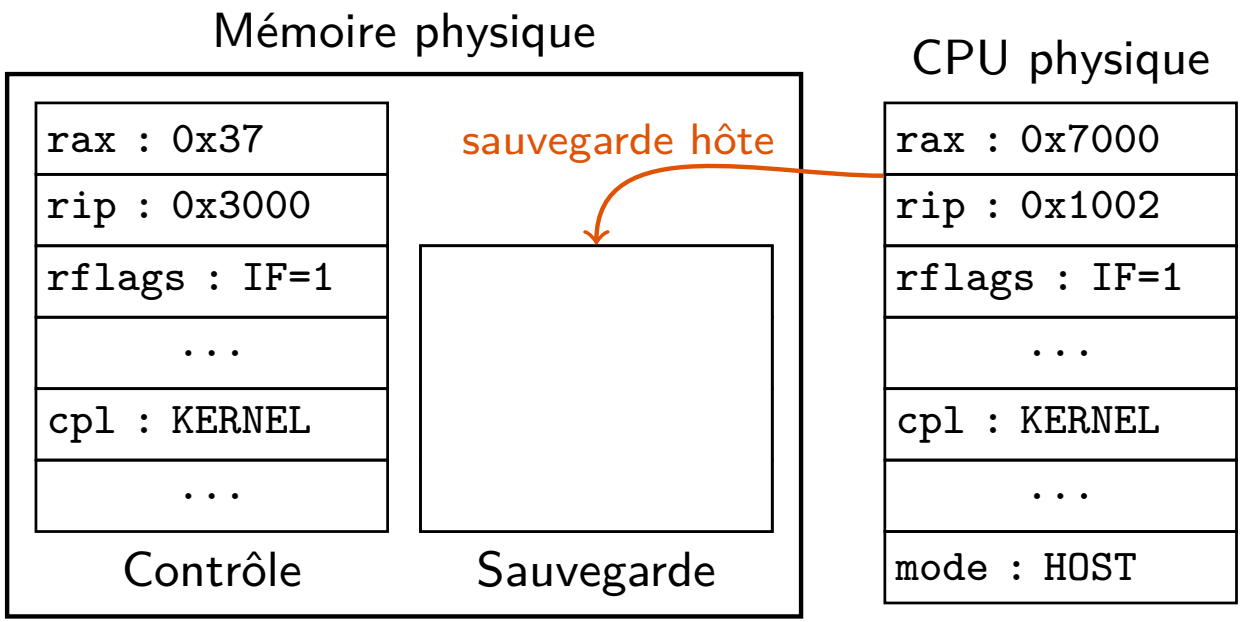


Code hyperviseur

```
...
vmrun
call    <on_vmexit>
...
```

- L'hyperviseur définit une zone de contrôle par vCPU et une zone de sauvegarde par CPU.
- Une instruction dédiée permet au processeur de passer en mode invité

Assistance matérielle à la virtualisation : principe

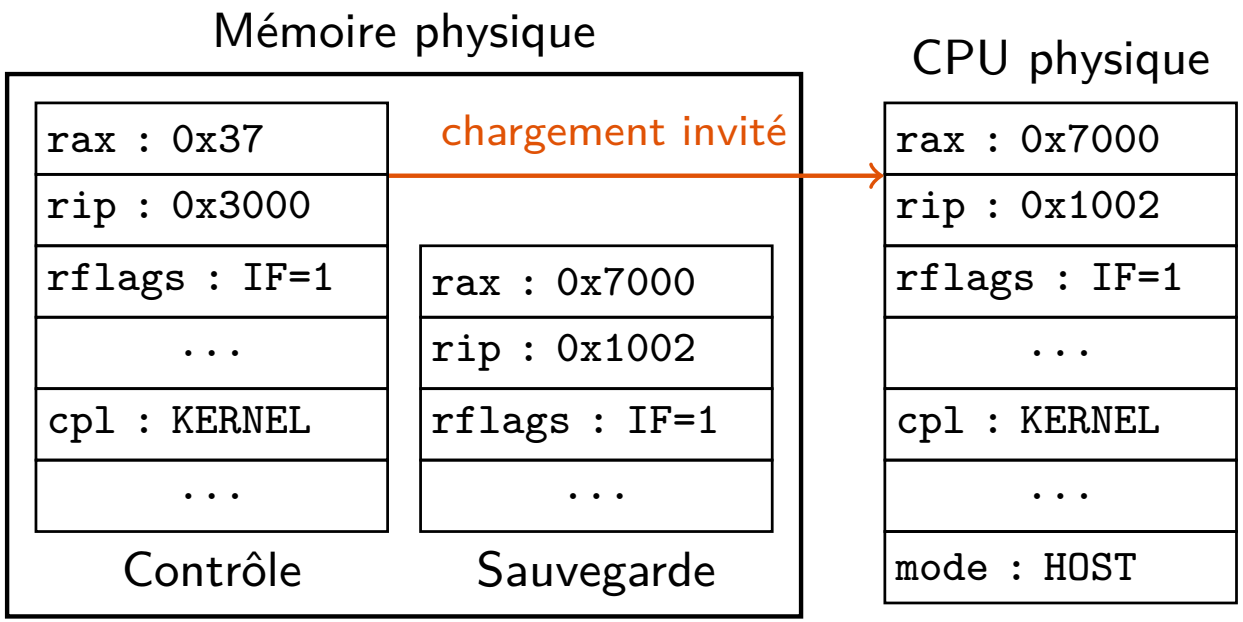


Code hyperviseur

```
...
vmrun
call    <on_vmexit>
...
```

- L'hyperviseur définit une zone de contrôle par vCPU et une zone de sauvegarde par CPU.
- Une instruction dédiée permet au processeur de passer en mode invité

Assistance matérielle à la virtualisation : principe

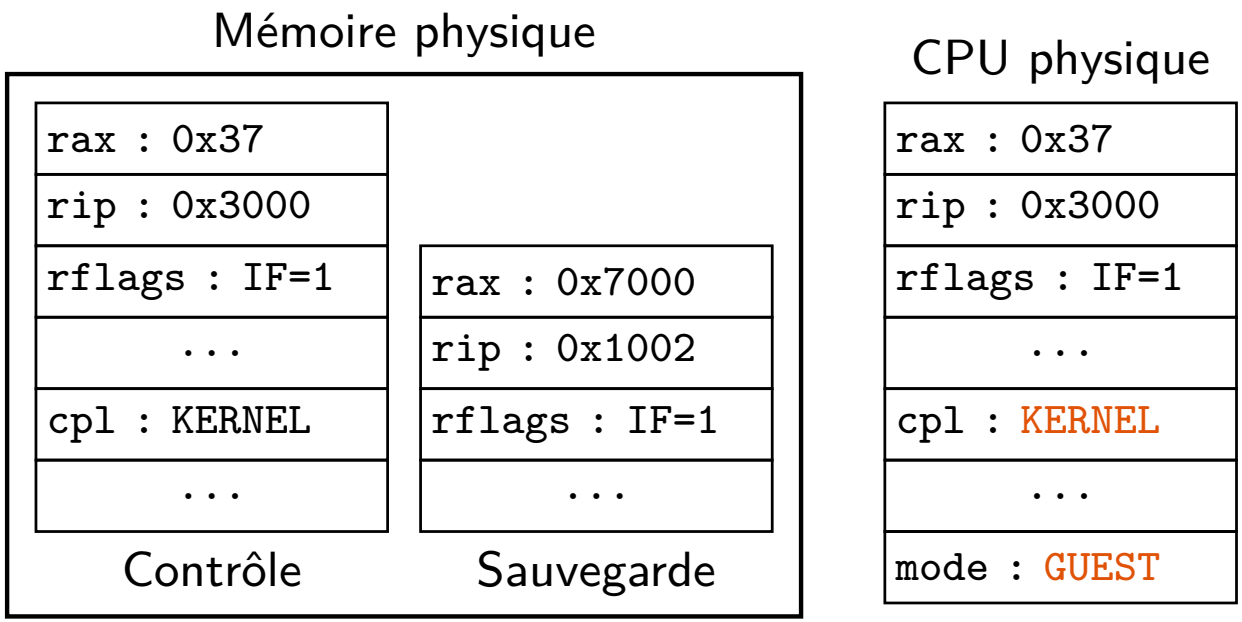


Code hyperviseur

```
...
vmrun
call    <on_vmexit>
...
```

- L'hyperviseur définit une zone de contrôle par vCPU et une zone de sauvegarde par CPU.
- Une instruction dédiée permet au processeur de passer en mode invité

Assistance matérielle à la virtualisation : principe

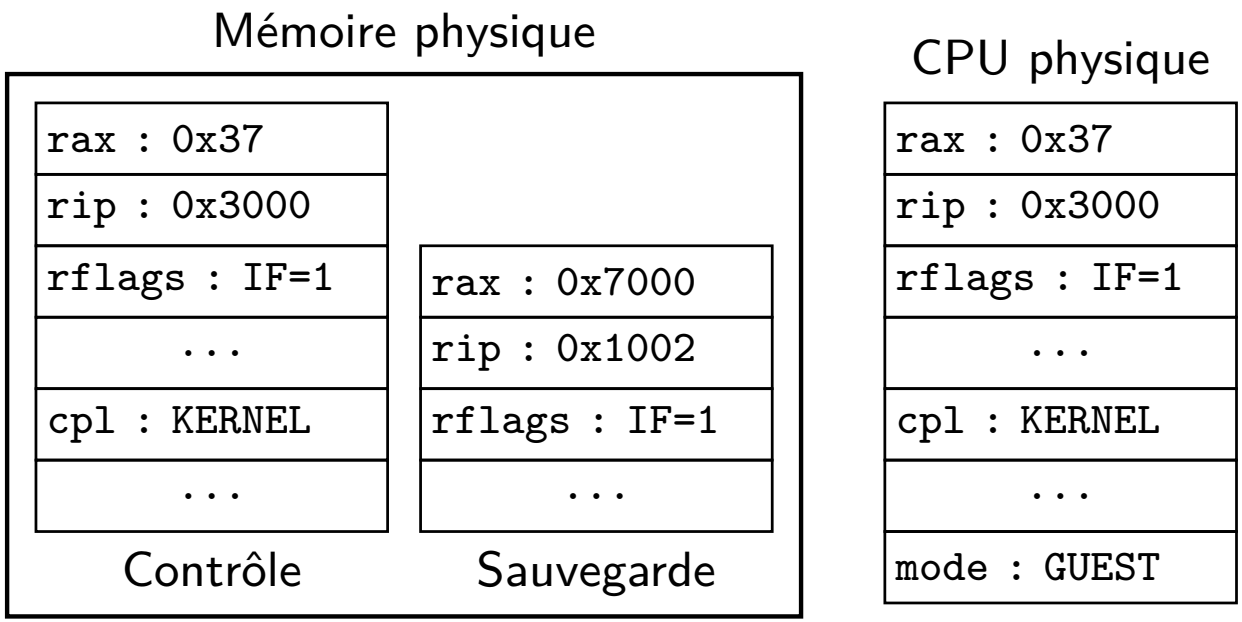


Code système invité

```
...
add     $4, %rax
cli
hlt
...
```

- L'hyperviseur définit une zone de contrôle par vCPU et une zone de sauvegarde par CPU.
- Une instruction dédiée permet au processeur de passer en mode invité
- Le système invité peut s'exécuter en privilège noyau / mode invité

Assistance matérielle à la virtualisation : principe

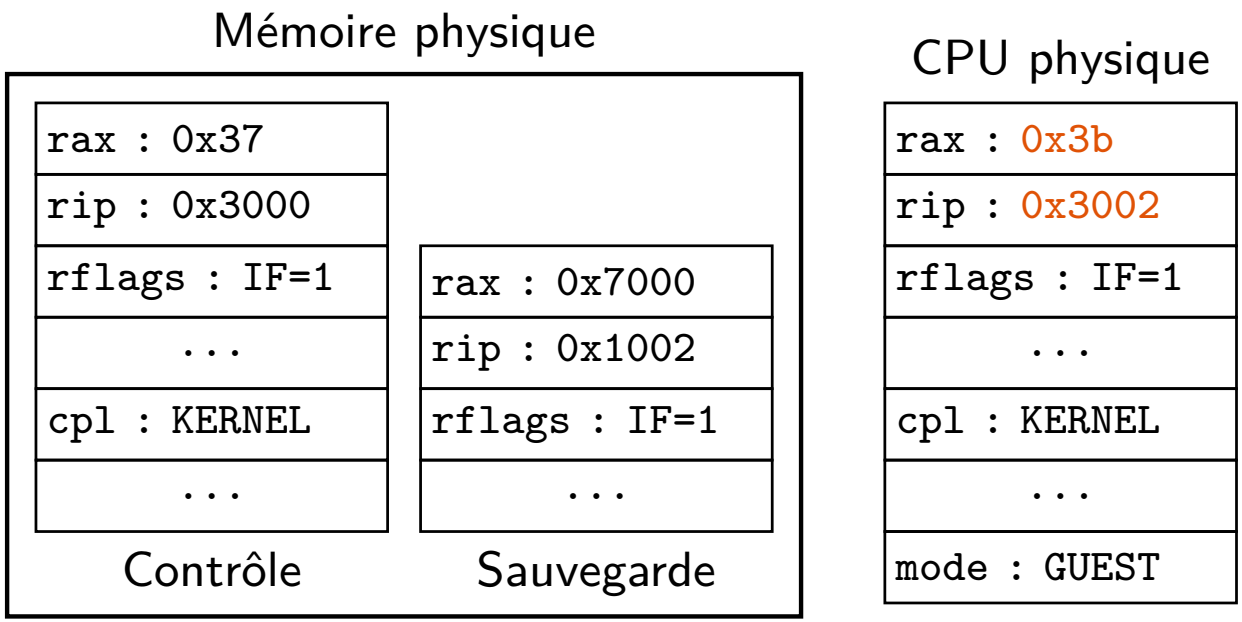


Code système invité

```
...
add     $4, %rax
cli
hlt
...
```

- L'hyperviseur définit une zone de contrôle par vCPU et une zone de sauvegarde par CPU.
- Une instruction dédiée permet au processeur de passer en mode invité
- Le système invité peut s'exécuter en privilège noyau / mode invité

Assistance matérielle à la virtualisation : principe

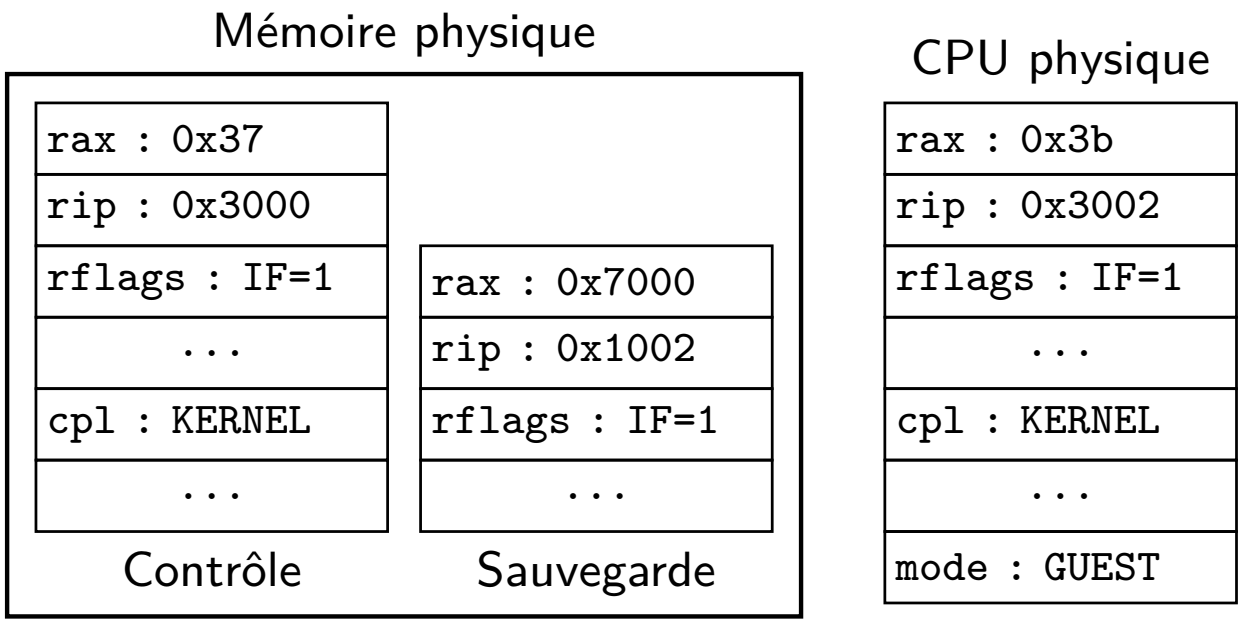


Code système invité

```
...
add     $4, %rax
cli
hlt
...
```

- L'hyperviseur définit une zone de contrôle par vCPU et une zone de sauvegarde par CPU.
- Une instruction dédiée permet au processeur de passer en mode invité
- Le système invité peut s'exécuter en privilège noyau / mode invité

Assistance matérielle à la virtualisation : principe

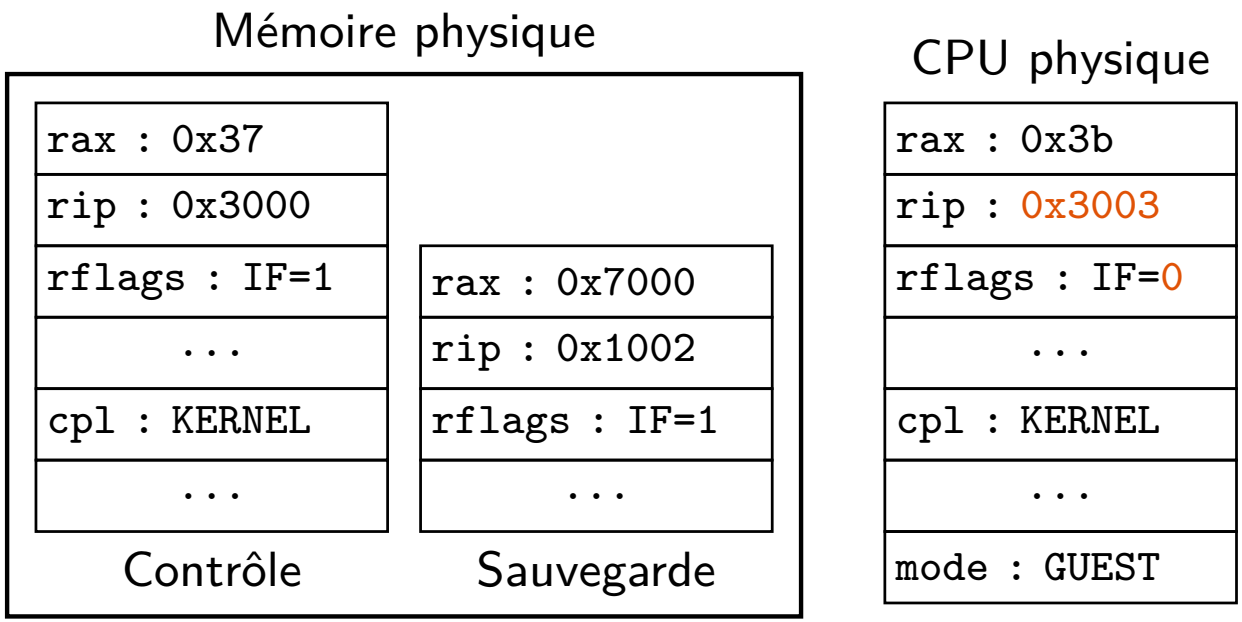


Code système invité

```
...
add     $4, %rax
cli
hlt
...
```

- L'hyperviseur définit une zone de contrôle par vCPU et une zone de sauvegarde par CPU.
- Une instruction dédiée permet au processeur de passer en mode invité
- Le système invité peut s'exécuter en privilège noyau / mode invité
  - Les instructions privilégiées sont exécutées directement

Assistance matérielle à la virtualisation : principe

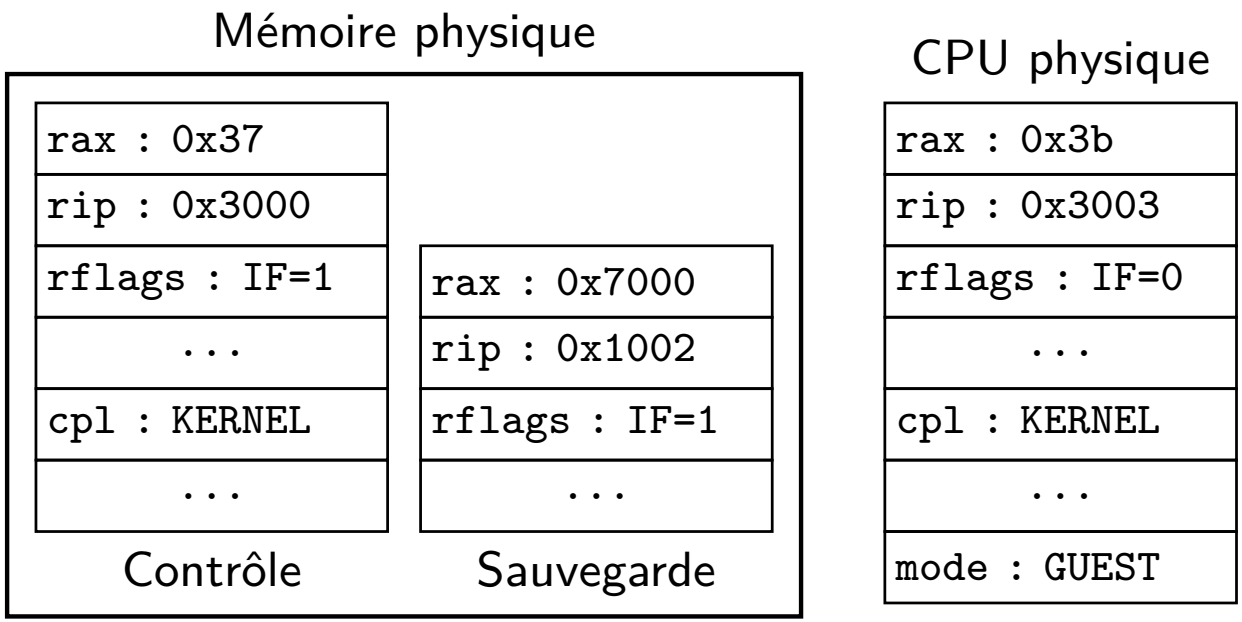


Code système invité

```
...
add     $4, %rax
cli
hlt
...
```

- L'hyperviseur définit une zone de contrôle par vCPU et une zone de sauvegarde par CPU.
- Une instruction dédiée permet au processeur de passer en mode invité
- Le système invité peut s'exécuter en privilège noyau / mode invité
  - Les instructions privilégiées sont exécutées directement

Assistance matérielle à la virtualisation : principe

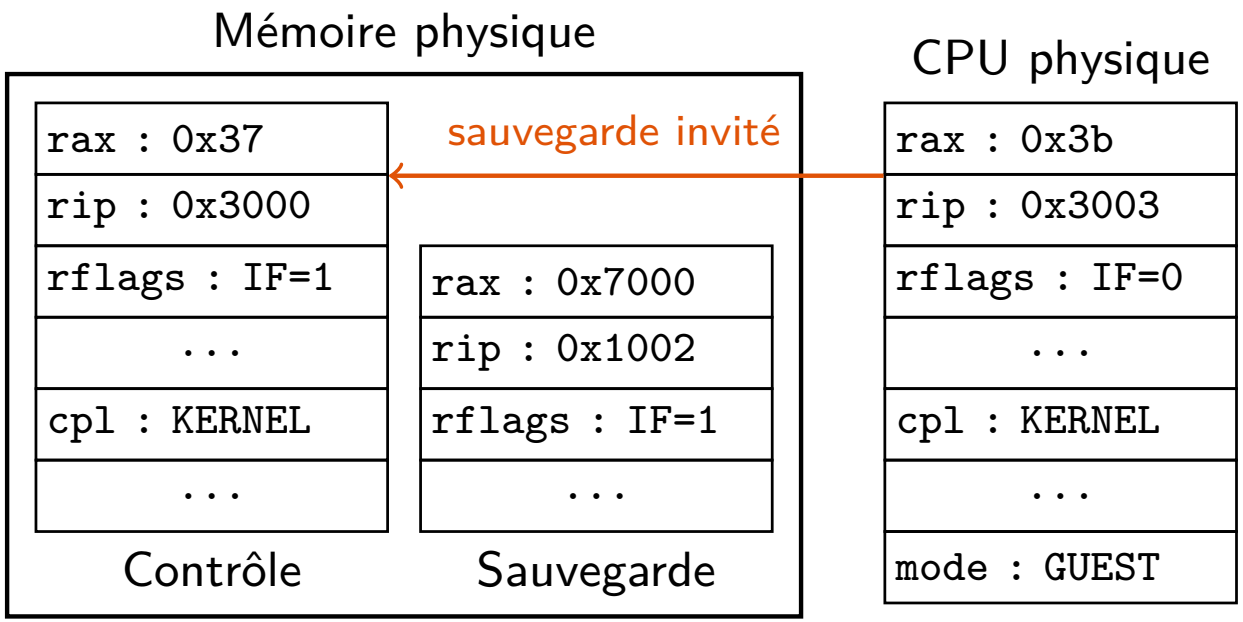


Code système invité

```
...
add     $4, %rax
cli
hlt
...
```

- L'hyperviseur définit une zone de contrôle par vCPU et une zone de sauvegarde par CPU.
- Une instruction dédiée permet au processeur de passer en mode invité
- Le système invité peut s'exécuter en privilège noyau / mode invité
  - Les instructions privilégiées sont exécutées directement
- Le processeur peut néanmoins intercepter certaines instructions
  - Il y a alors une `VMEXIT`

Assistance matérielle à la virtualisation : principe

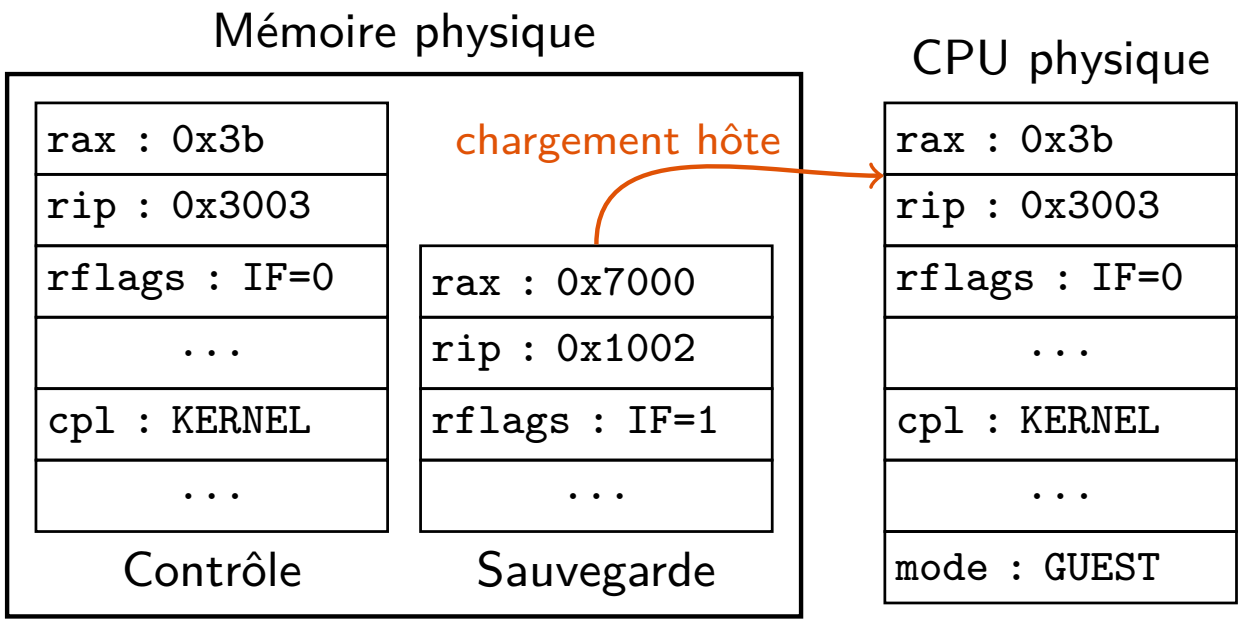


Code système invité

```
...
add     $4, %rax
cli
hlt
...
```

- L'hyperviseur définit une zone de contrôle par vCPU et une zone de sauvegarde par CPU.
- Une instruction dédiée permet au processeur de passer en mode invité
- Le système invité peut s'exécuter en privilège noyau / mode invité
  - Les instructions privilégiées sont exécutées directement
- Le processeur peut néanmoins intercepter certaines instructions
  - Il y a alors une `VMEXIT`

Assistance matérielle à la virtualisation : principe

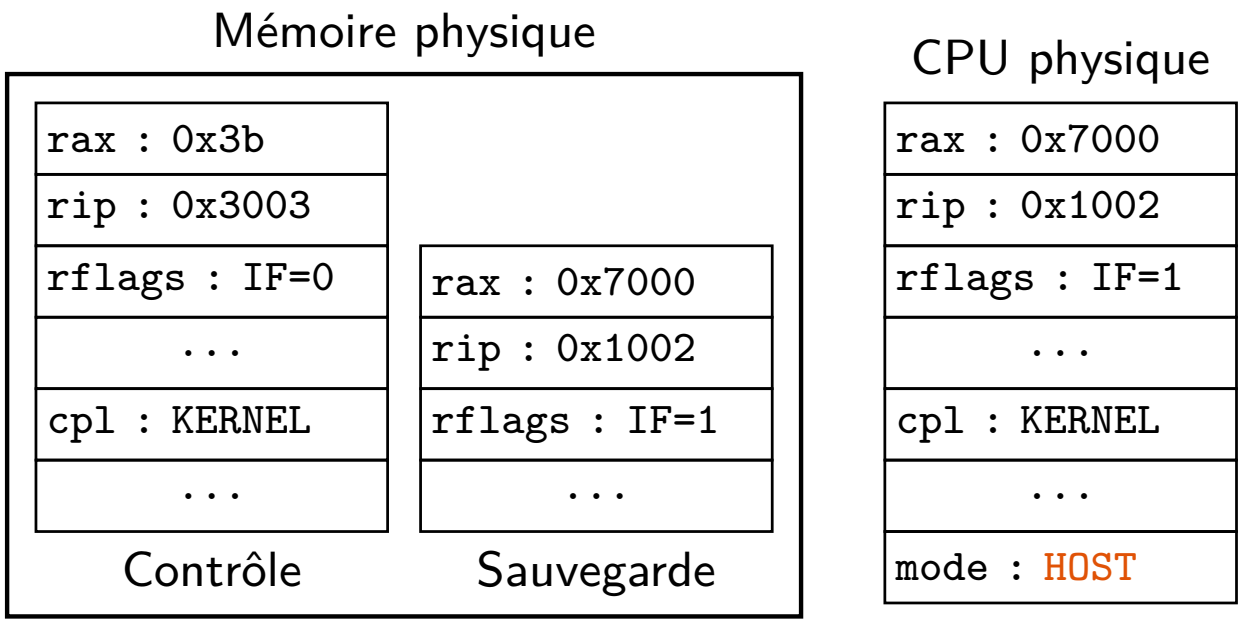


Code système invité

```
...
add     $4, %rax
cli
hlt
...
```

- L'hyperviseur définit une zone de contrôle par vCPU et une zone de sauvegarde par CPU.
- Une instruction dédiée permet au processeur de passer en mode invité
- Le système invité peut s'exécuter en privilège noyau / mode invité
  - Les instructions privilégiées sont exécutées directement
- Le processeur peut néanmoins intercepter certaines instructions
  - Il y a alors une `VMEXIT`

Assistance matérielle à la virtualisation : principe



Code hyperviseur

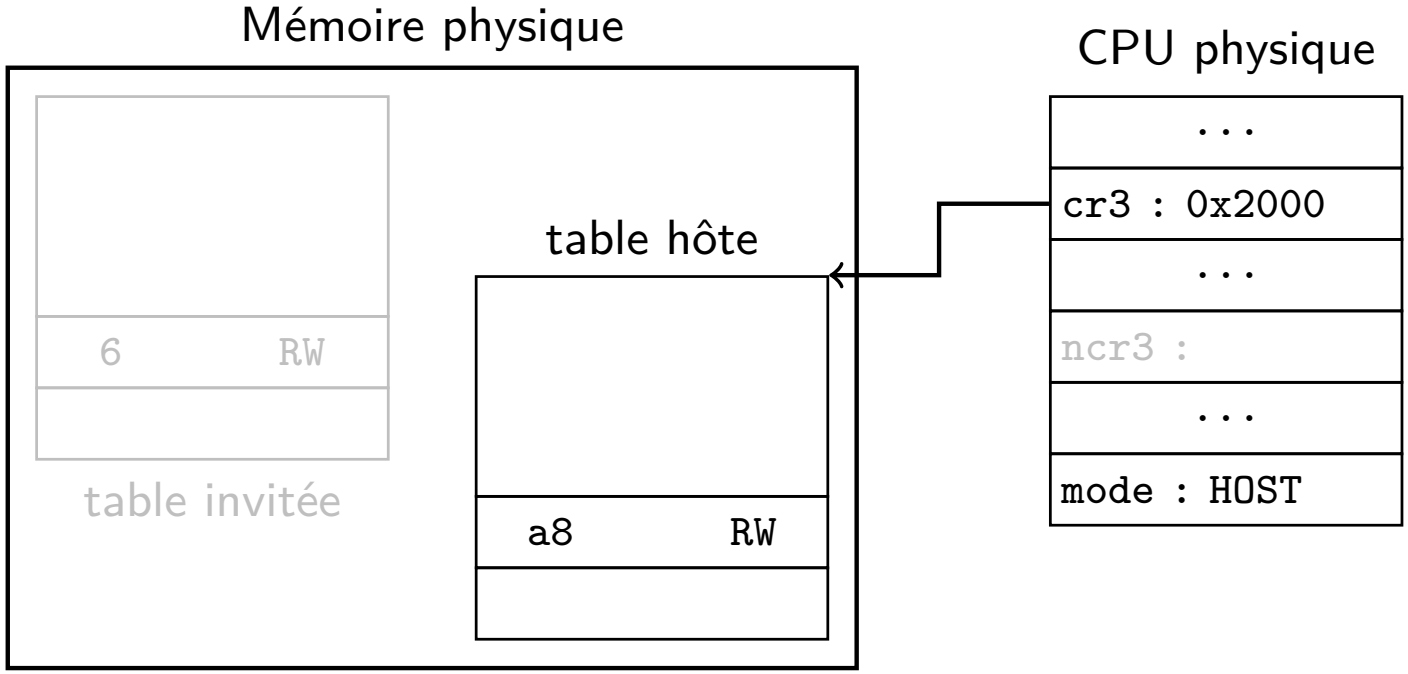
```
...
vmrun
call    <on_vmexit>
...
```

- L'hyperviseur définit une zone de contrôle par vCPU et une zone de sauvegarde par CPU.
- Une instruction dédiée permet au processeur de passer en mode invité
- Le système invité peut s'exécuter en privilège noyau / mode invité
  - Les instructions privilégiées sont exécutées directement
- Le processeur peut néanmoins intercepter certaines instructions
  - Il y a alors une VMEXIT

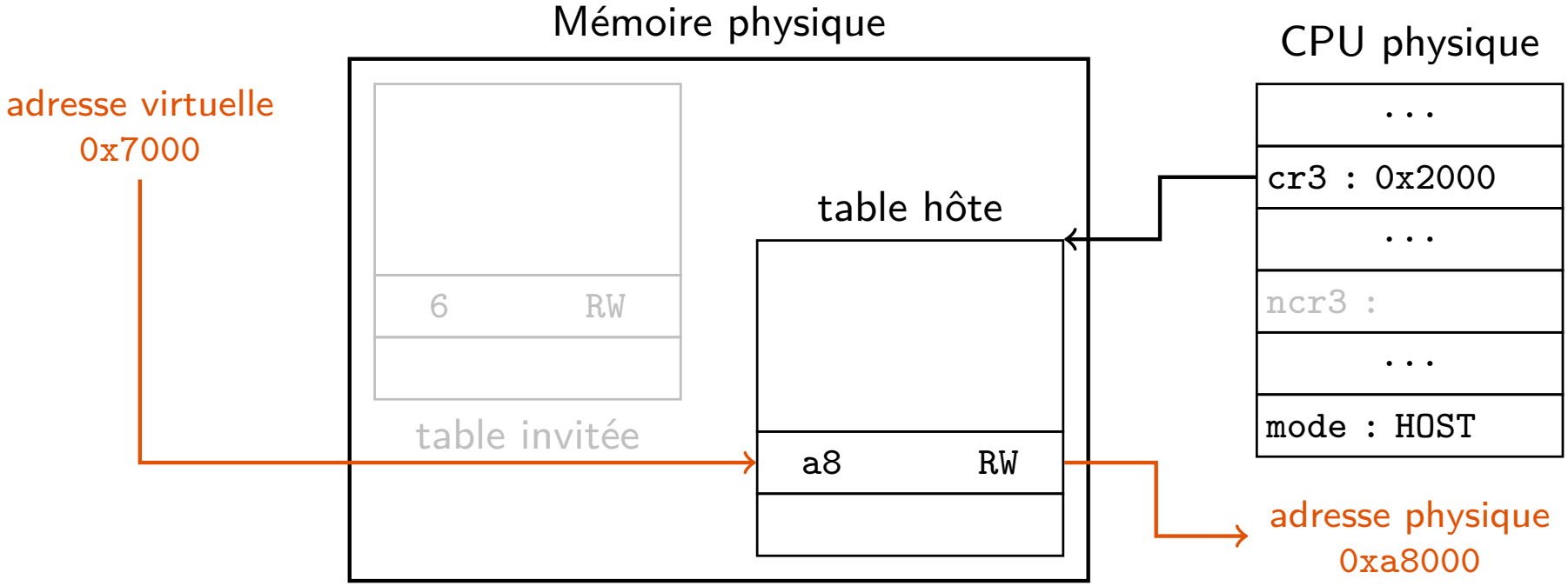
Interception matérielle d'instructions

- La zone de contrôle du vCPU ne stocke pas uniquement l'état des registres en mode invité
- Elle contient également divers champs de contrôle
  - Avant d'exécuter `vmrun`, l'hyperviseur configure ces champs
  - Indiquent au processeur comment réagir à certains types d'évènements
- Un des champs indique quelles instructions provoquent une `VMEXIT`
  - Instructions privilégiées → `mov %cr3`, ...
  - Instructions *silent fail* → `popf`, ...
  - Instructions non privilégiées → `cpuid`, ...
- Après un `VMEXIT`, l'hyperviseur peut décoder l'instruction au `rip` invité et l'émuler

Assistance matérielle à la pagination

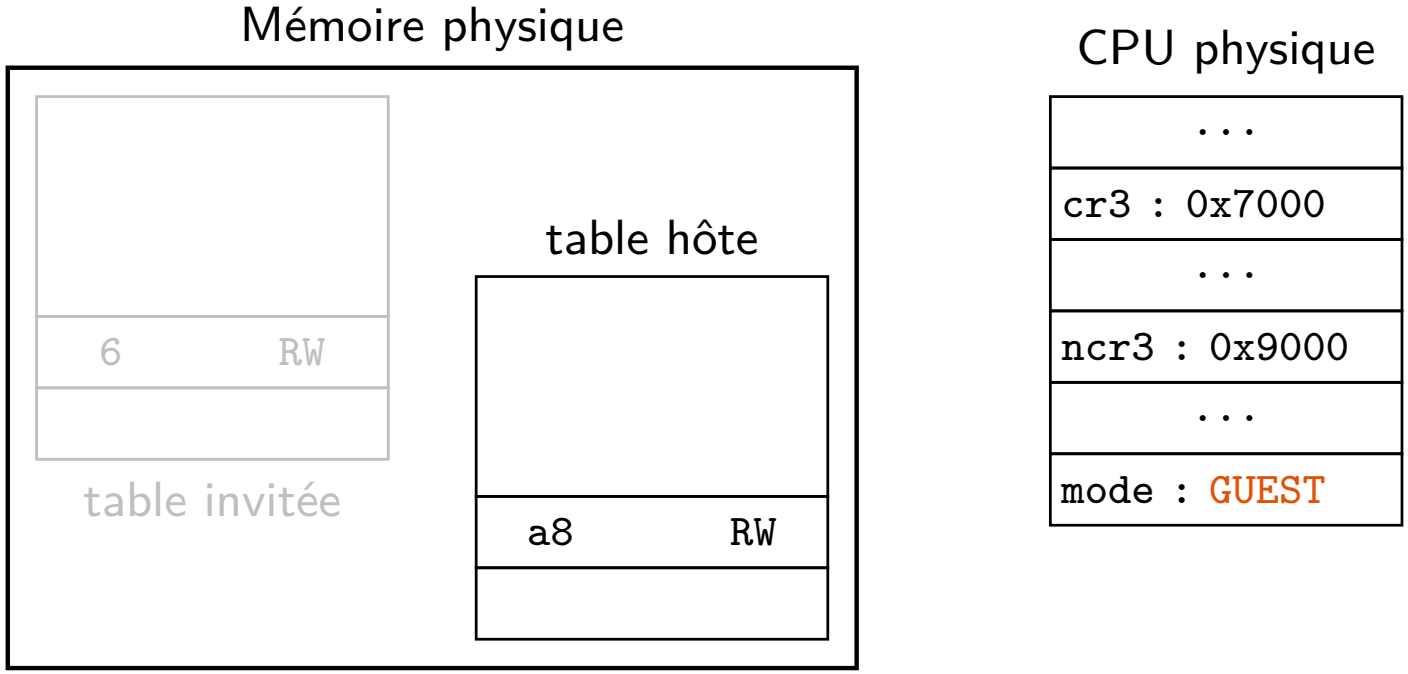


Assistance matérielle à la pagination



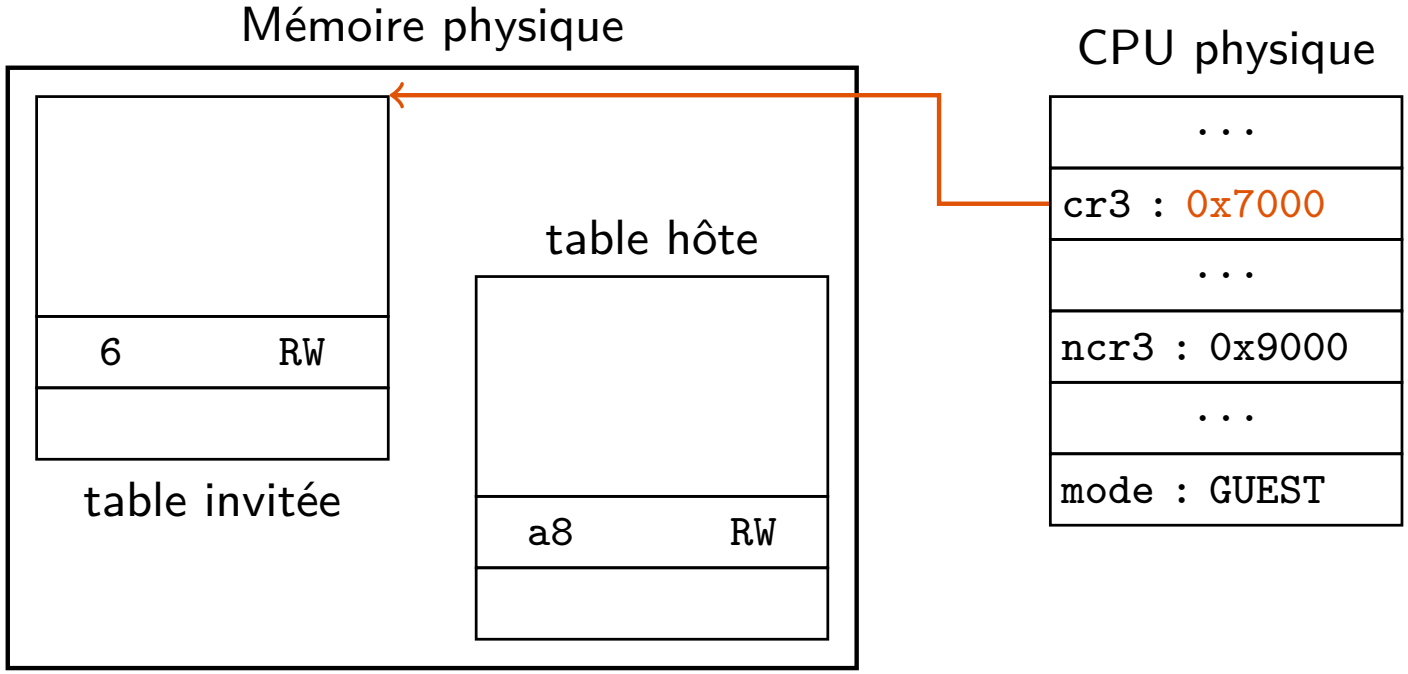
- En mode hôte, la MMU traduit les adresses avec la table des pages de l'hyperviseur pointée physiquement par le `cr3`

Assistance matérielle à la pagination



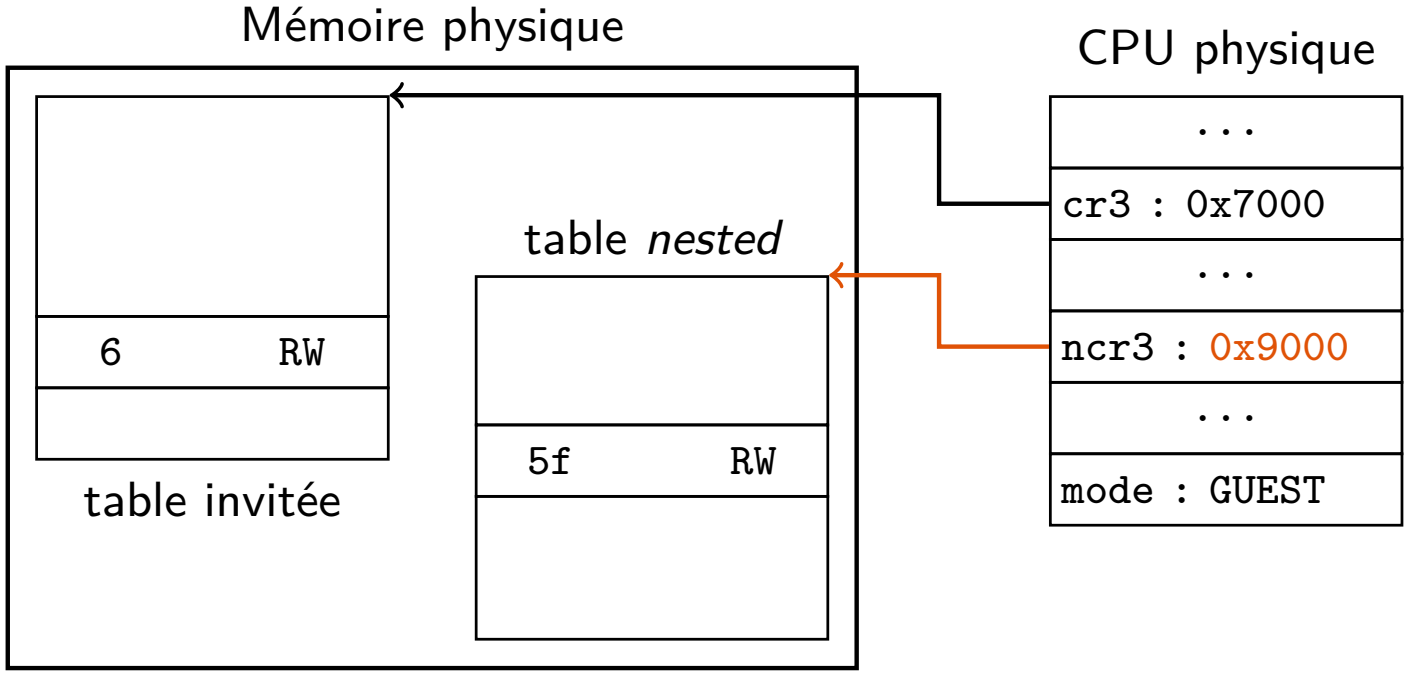
- En mode hôte, la MMU traduit les adresses avec la table des pages de l'hyperviseur pointée physiquement par le `cr3`

Assistance matérielle à la pagination



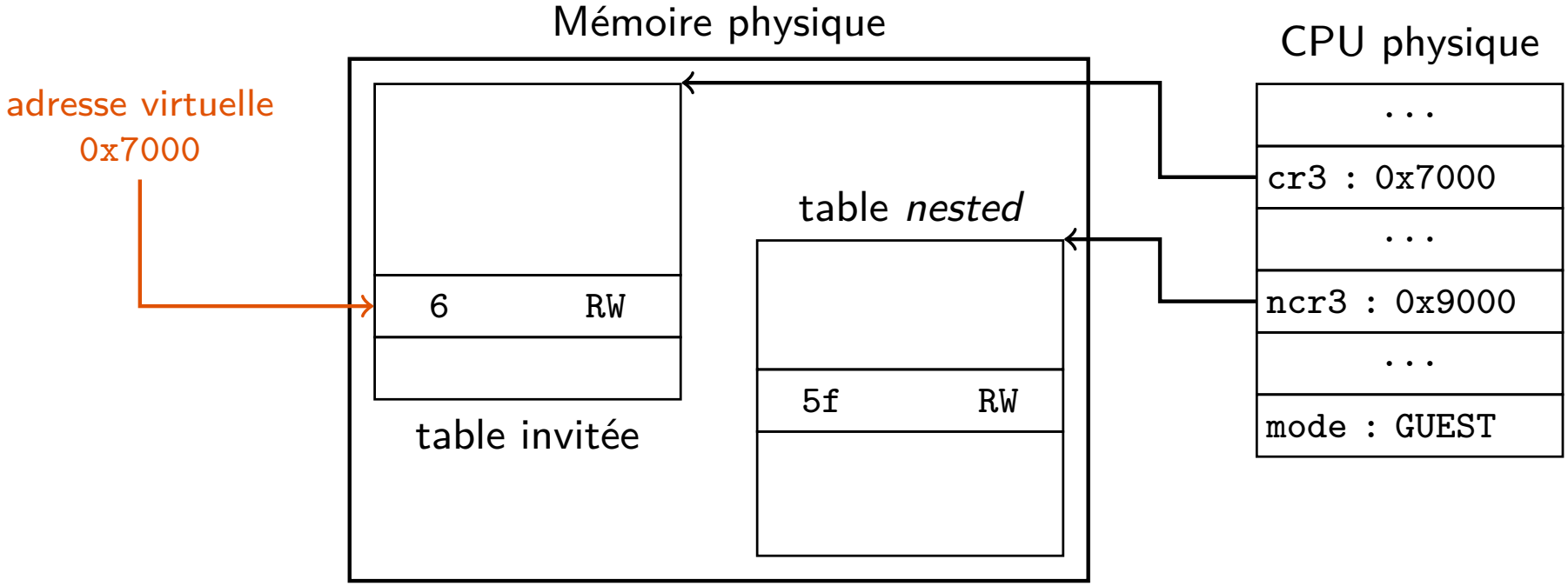
- En mode invité, le `cr3` pointe physiquement sur la table des pages du système invitée

Assistance matérielle à la pagination



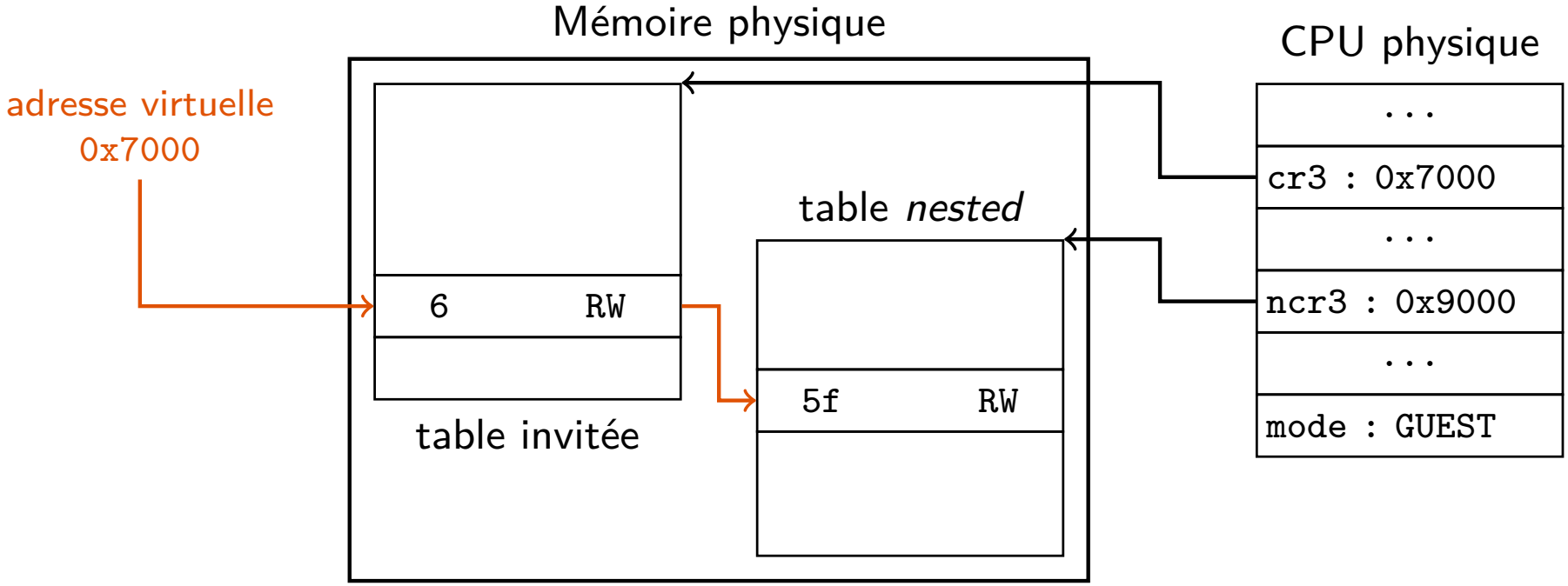
- En mode invité, le `cr3` pointe physiquement sur la table des pages du système invitée
- Un autre registre, le `ncr3` contient l'adresse machine d'une seconde table des pages, accessible uniquement par l'hyperviseur

Assistance matérielle à la pagination



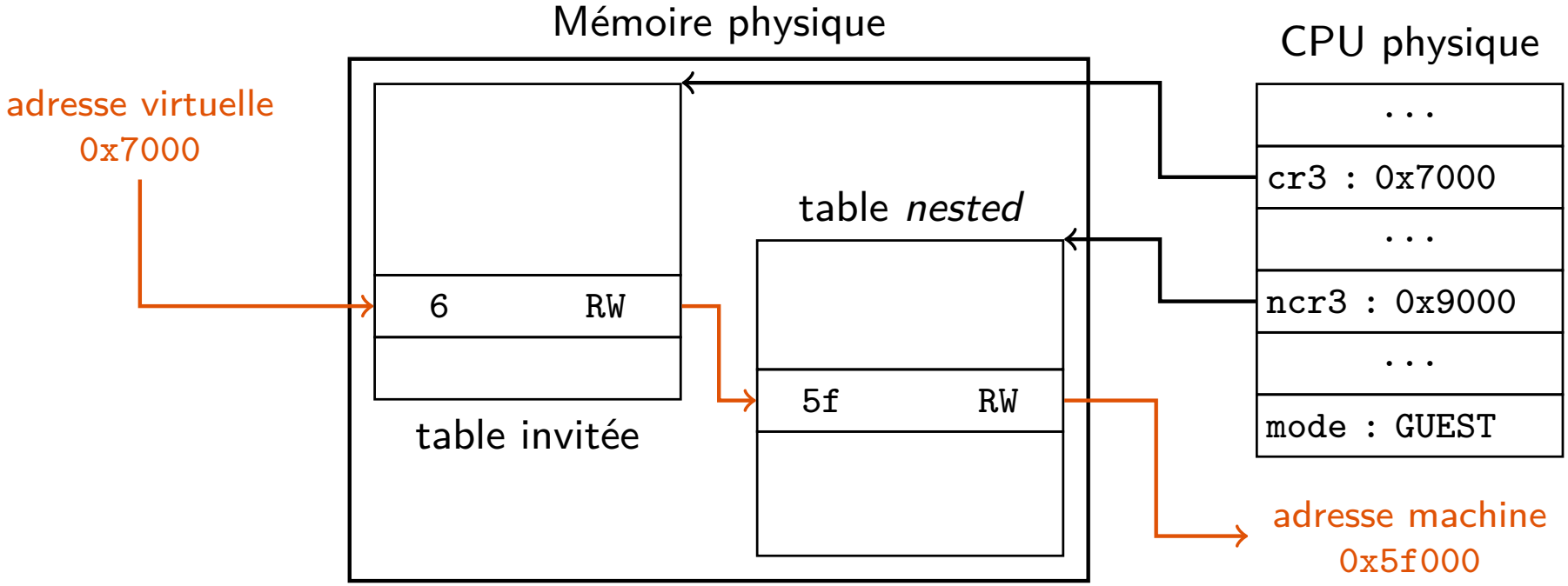
- En mode invité, le `cr3` pointe physiquement sur la table des pages du système invitée
- Un autre registre, le `ncr3` contient l'adresse machine d'une seconde table des pages, accessible uniquement par l'hyperviseur
- Lors d'un accès, la MMU traduit l'adresse virtuelle en adresse physique avec la table des pages invité

Assistance matérielle à la pagination



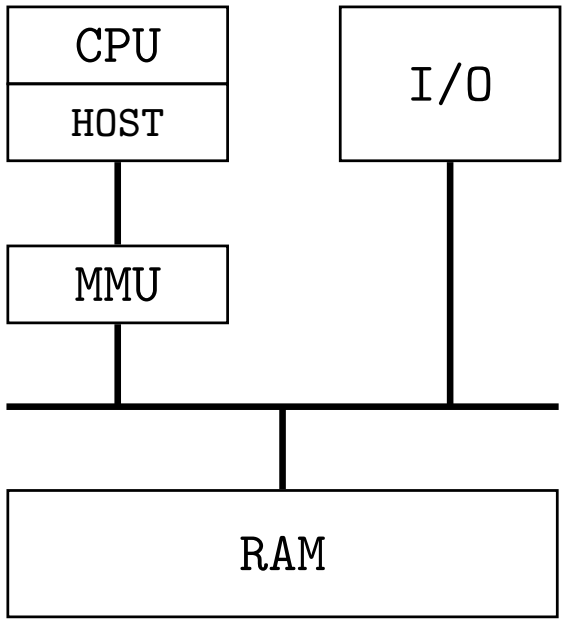
- En mode invité, le `cr3` pointe physiquement sur la table des pages du système invitée
- Un autre registre, le `ncr3` contient l'adresse machine d'une seconde table des pages, accessible uniquement par l'hyperviseur
- Lors d'un accès, la MMU traduit l'adresse virtuelle en adresse physique avec la table des pages invité, puis traduit cette adresse physique en adresse machine avec la table des pages *nested*

Assistance matérielle à la pagination



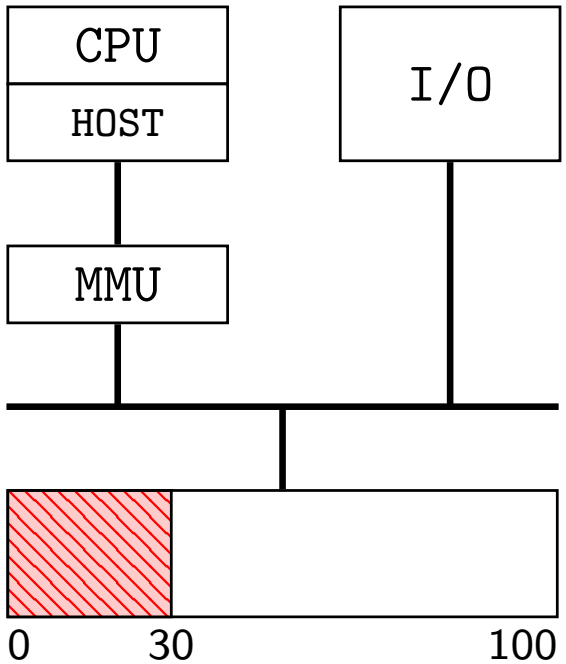
- En mode invité, le `cr3` pointe physiquement sur la table des pages du système invitée
- Un autre registre, le `ncr3` contient l'adresse machine d'une seconde table des pages, accessible uniquement par l'hyperviseur
- Lors d'un accès, la MMU traduit l'adresse virtuelle en adresse physique avec la table des pages invité, puis traduit cette adresse physique en adresse machine avec la table des pages *nested*

IOMMU et transferts DMA



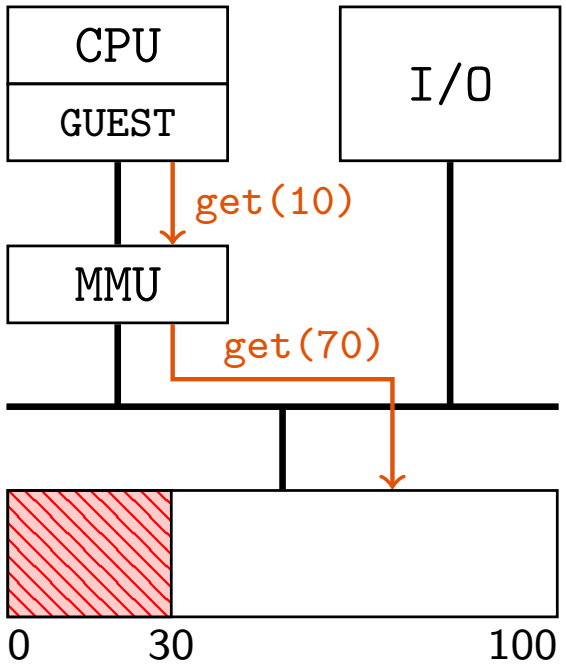
- La MMU traduit toute adresse sortante du CPU

IOMMU et transferts DMA



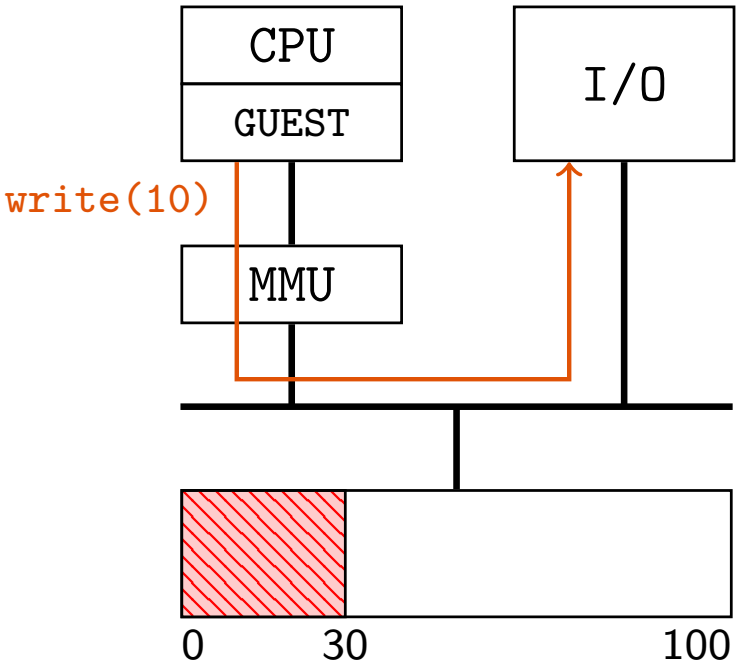
- La MMU traduit toute adresse sortante du CPU
- L'hyperviseur configure la MMU pour protéger sa mémoire de l'invité

IOMMU et transferts DMA



- La MMU traduit toute adresse sortante du CPU
- L'hyperviseur configure la MMU pour protéger sa mémoire de l'invité
- Les adresses virtuelles de l'invité sont traduite en adresses machines dédiées à l'invité

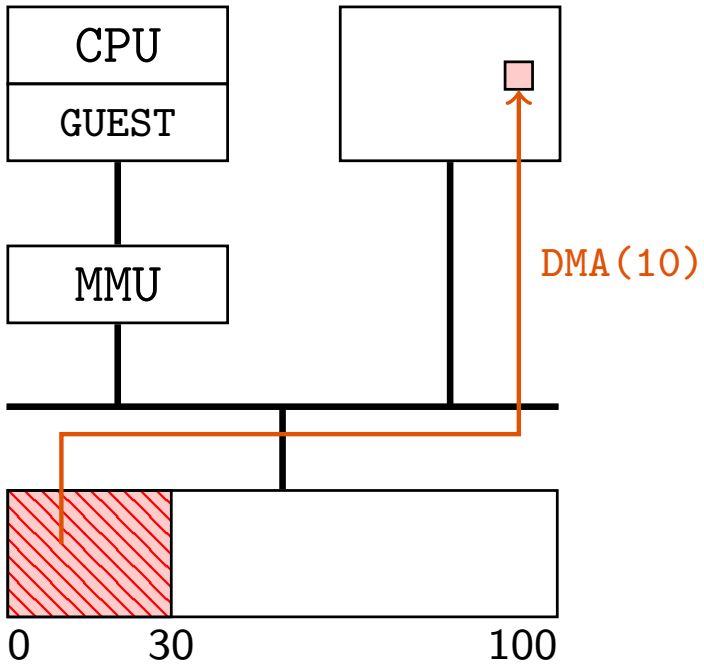
IOMMU et transferts DMA



- La MMU traduit toute adresse sortante du CPU
- L'hyperviseur configure la MMU pour protéger sa mémoire de l'invité
- Les adresses virtuelles de l'invité sont traduite en adresses machines dédiées à l'invité

- Pour accélerer les entrées / sorties, l'hyperviseur pourrait dédier un disque complet au système invité
  - Les requêtes aux contrôleurs DMA ne sont pas des adresses

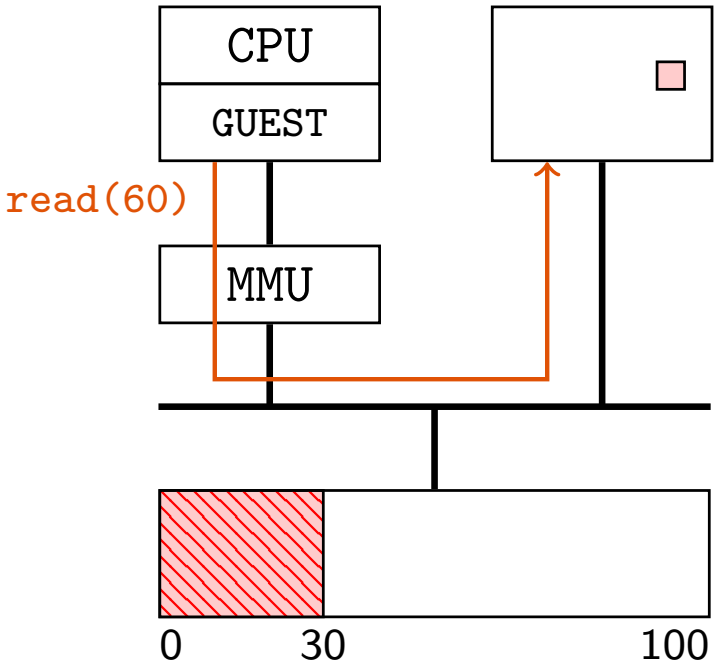
IOMMU et transferts DMA



- La MMU traduit toute adresse sortante du CPU
- L'hyperviseur configure la MMU pour protéger sa mémoire de l'invité
- Les adresses virtuelles de l'invité sont traduite en adresses machines dédiées à l'invité

- Pour accélerer les entrées / sorties, l'hyperviseur pourrait dédier un disque complet au système invité
  - Les requêtes aux contrôleurs DMA ne sont pas des adresses
  - Les adresses sortantes des contrôleurs DMA ne sont pas traduites

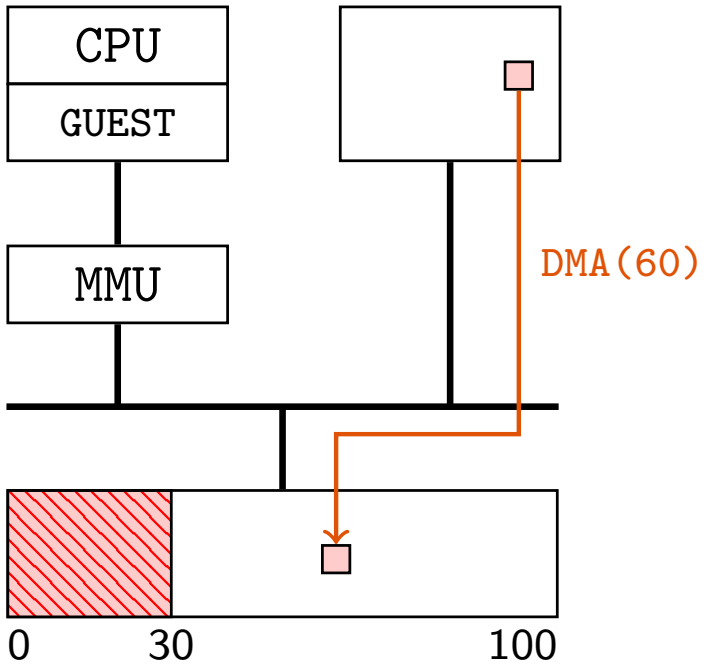
IOMMU et transferts DMA



- La MMU traduit toute adresse sortante du CPU
- L'hyperviseur configure la MMU pour protéger sa mémoire de l'invité
- Les adresses virtuelles de l'invité sont traduite en adresses machines dédiées à l'invité

- Pour accélerer les entrées / sorties, l'hyperviseur pourrait dédier un disque complet au système invité
  - Les requêtes aux contrôleurs DMA ne sont pas des adresses
  - Les adresses sortantes des contrôleurs DMA ne sont pas traduites

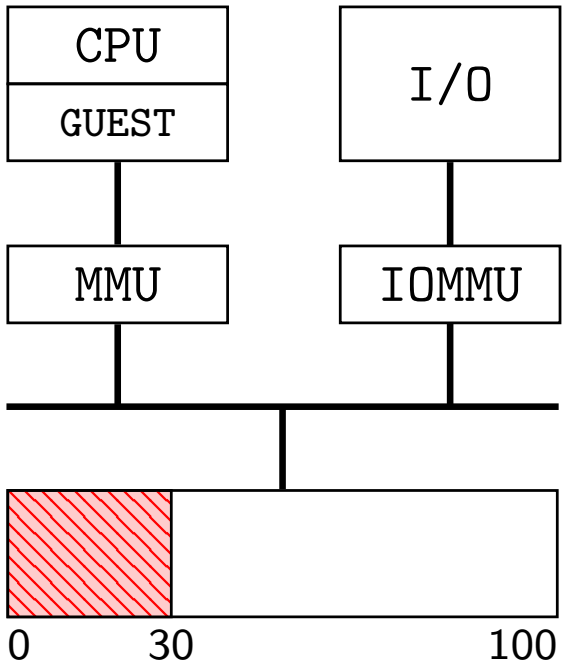
IOMMU et transferts DMA



- La MMU traduit toute adresse sortante du CPU
- L'hyperviseur configure la MMU pour protéger sa mémoire de l'invité
- Les adresses virtuelles de l'invité sont traduite en adresses machines dédiées à l'invité

- Pour accélerer les entrées / sorties, l'hyperviseur ne peut pas dédier un disque complet au système invité sans aide du matériel
  - Les requêtes aux contrôleurs DMA ne sont pas des adresses
  - Les adresses sortantes des contrôleurs DMA ne sont pas traduites
  - Le système invité peut contourner l'isolation mémoire

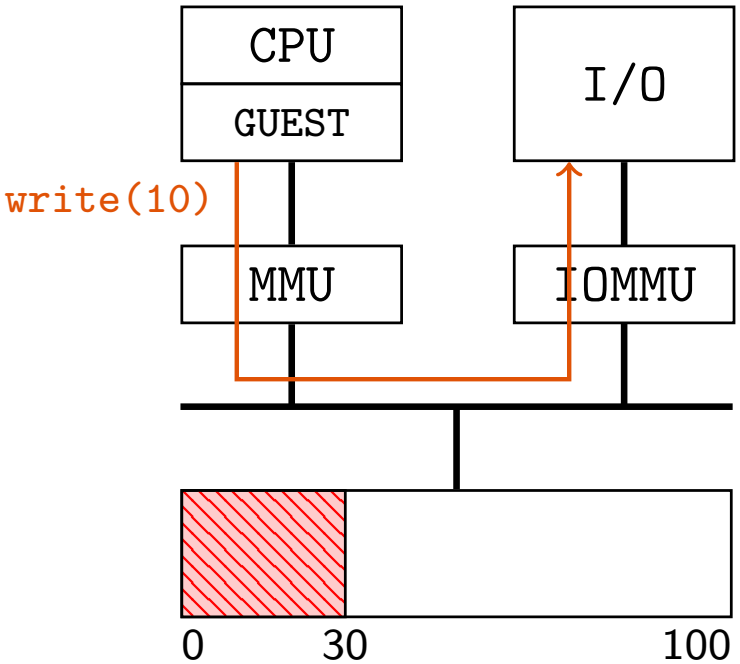
IOMMU et transferts DMA



- La MMU traduit toute adresse sortante du CPU
- L'hyperviseur configure la MMU pour protéger sa mémoire de l'invité
- Les adresses virtuelles de l'invité sont traduite en adresses machines dédiées à l'invité

- Les machines modernes sont dotées d'IOMMU, placées entre la mémoire et les contrôleurs DMA

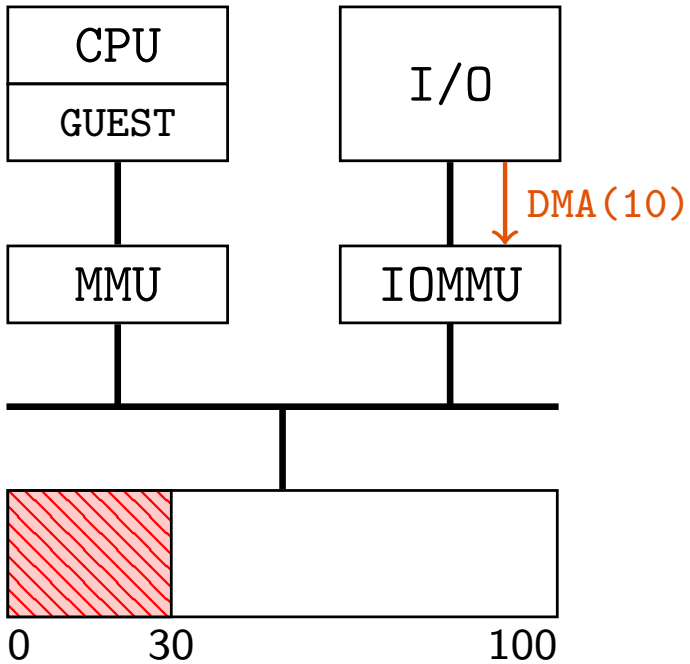
IOMMU et transferts DMA



- La MMU traduit toute adresse sortante du CPU
- L'hyperviseur configure la MMU pour protéger sa mémoire de l'invité
- Les adresses virtuelles de l'invité sont traduite en adresses machines dédiées à l'invité

- Les machines modernes sont dotées d'IOMMU, placées entre la mémoire et les contrôleurs DMA
  - Les requêtes aux contrôleurs DMA ne sont pas des adresses

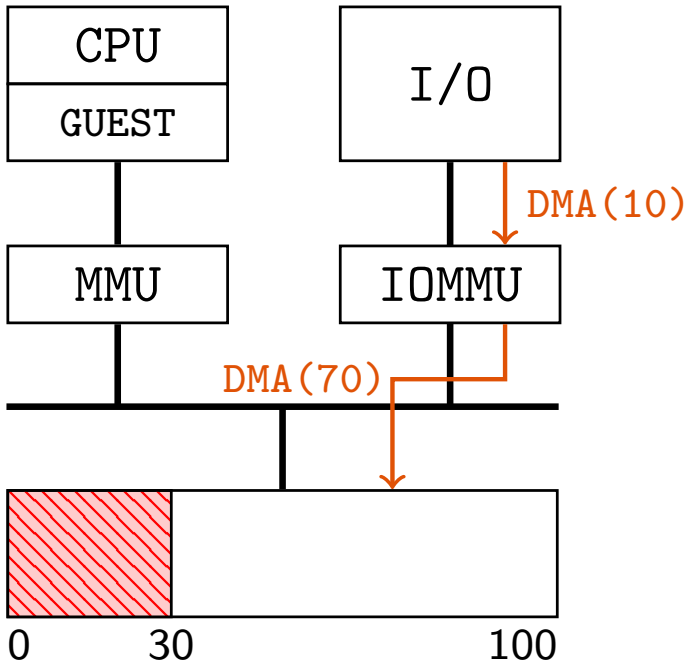
IOMMU et transferts DMA



- La MMU traduit toute adresse sortante du CPU
- L'hyperviseur configure la MMU pour protéger sa mémoire de l'invité
- Les adresses virtuelles de l'invité sont traduite en adresses machines dédiées à l'invité

- Les machines modernes sont dotées d'IOMMU, placées entre la mémoire et les contrôleurs DMA
  - Les requêtes aux contrôleurs DMA ne sont pas des adresses
  - Les adresses sortantes du contrôleur DMA sont traduites par l'IOMMU

IOMMU et transferts DMA



- La MMU traduit toute adresse sortante du CPU
- L'hyperviseur configure la MMU pour protéger sa mémoire de l'invité
- Les adresses virtuelles de l'invité sont traduite en adresses machines dédiées à l'invité

- Les machines modernes sont dotées d'IOMMU, placées entre la mémoire et les contrôleurs DMA
  - Les requêtes aux contrôleurs DMA ne sont pas des adresses
  - Les adresses sortantes du contrôleur DMA sont traduites par l'IOMMU
  - L'hyperviseur configure les IOMMU pour protéger sa mémoire de l'invité

Support matériel à la virtualisation : résumé

- Les architectures x86 récentes fournissent un support matériel à l'hyperviseur pour augmenter les performances des systèmes invités

- Les processeurs récents ont un mode (hôte ou invité) en plus du niveau de privilège (noyau ou utilisateur)

- En mode invité, le processeur assure automatiquement la plupart des fonctions d'émulation auparavant à la charge de l'hyperviseur
  - Interception et émulation d'instructions privilégiées
  - Isolation mémoire
  - Gestion des entrées / sorties

- Avec ce support matériel, la virtualisation a un surcoût négligeable
- Exemple de d'hyperviseur qui utilise l'assistance matérielle
  - KVM
  - Xen

Conclusion : virtualisation d'instructions

- La simulation *cycle accurate* est la méthode la plus précise pour simuler une machine virtuelle $\rightarrow$ c'est aussi la plus lente
  - Simulation de l'implémentation du processeur (registres, câbles, ...)
  - Utile pour tester du nouveau matériel

- L'émulation d'instructions consiste à accomplir logiciellement les actions du processeur sans en reproduire l'implémentation
  - Chaque instruction est décodée puis interprètée
  - Le système invité modifie ainsi une machine virtuelle dont l'état est stocké en mémoire

- La traduction binaire dynamique consiste à traduire le code du système invité en un code exécutable sans risque par le système hôte
  - Chaque *basic block* est traduit vers le jeu d'instruction de l'hôte
  - Les instructions privilégiées sont remplacées par des fonctions d'émulation

Conclusion : virtualisation de ressources

- Pour isoler le système invité du système hôte, les adresses virtuelles de l'invité (GVA) doivent être traduites en adresses physiques de l'hôte (HPA) qui ne sont pas déjà utilisées
- De plus, le système invité doit avoir l'illusion qu'il fait correspondre ses adresses virtuelles (GVA) vers ses adresses physiques (GPA)
- La traduction logicielle d'adresses consiste à ajouter à chaque instruction invitée d'accès mémoire un code qui traduit la GVA accédée en HPA choisie par l'hôte
- Le *shadowing* mémoire consiste à configurer la table des pages hôte pour refléter les modifications faites par l'invité à sa propre table des pages $\rightarrow$ la MMU traduit ensuite les GVA en HPA
- Un système invité paravirtualisé communique explicitement avec l'hyperviseur au moyen d'*hypercalls* plutôt que de passer par une interface de machine virtuelle

Conclusion : virtualisation comme isolation

- L'exécution directe consiste à exécuter directement le code du système invité dans un processus hôte
  - L'isolation est assurée de la même manière que pour un processus
  - Les instructions privilégiées sont interceptées par le système hôte puis émulées par l'hyperviseur
  - Les instructions *silent fail* rendent cette technique inutilisable sur certaines architectures

- Les processeurs modernes proposent une assistance matérielle à la virtualisation qui assure la plupart des fonctions d'émulation
  - Le coût de l'émulation devient négligeable
  - Le rôle de l'hyperviseur est d'assurer les fonctions de haut niveau (ordonnancement, consolidation, placement mémoire, ...)