

Virtual Machine Monitors (Hypervisors)

Pr. Olivier Gruber

Université Grenoble-Alpes

Laboratoire d'Informatique de Grenoble

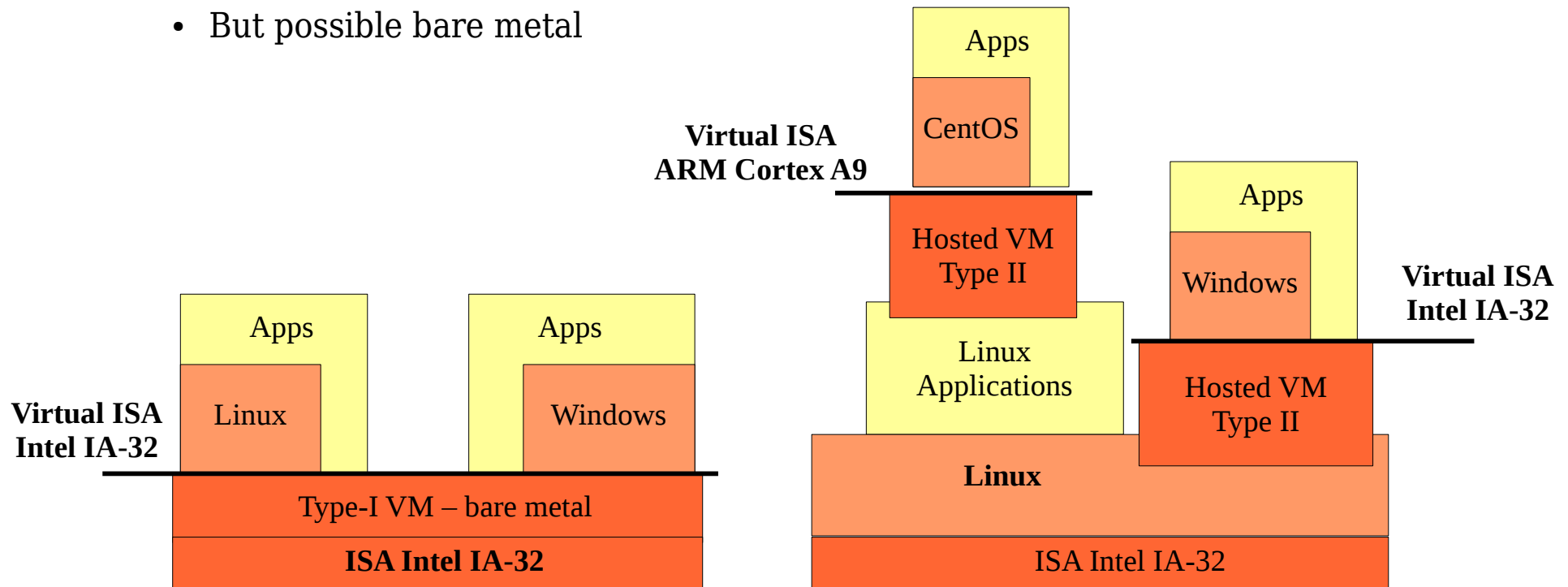
Olivier.Gruber@imag.fr

- Introduction
 - Introduce Virtual Machine Monitors (hypervisors)
 - Discuss how useful they are
 - Design and implementation study
- Real-Machine Virtualization
 - Discussing efficiency
 - State management and processor virtualization
 - Memory virtualization
 - I/O virtualization
 - Dynamic Binary Translation (DBT)

Virtual Machine Monitors

3

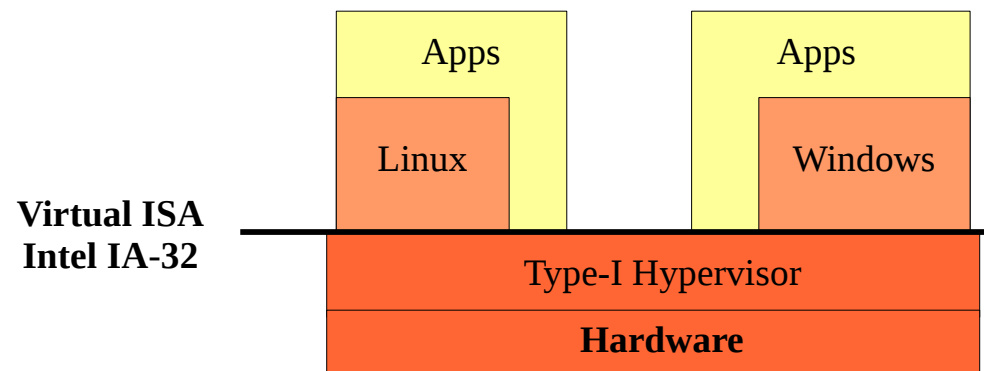
- Two main types of same-ISA System Vms
 - **Same ISA**
 - **Type-I** VMs, also known as **bare-metal**
 - **Type-II** VMs, also known as hosted VMs
 - **Different ISA**
 - Usually hosted VMs (type-II)
 - But possible bare metal



Type-I Virtual Machines

4

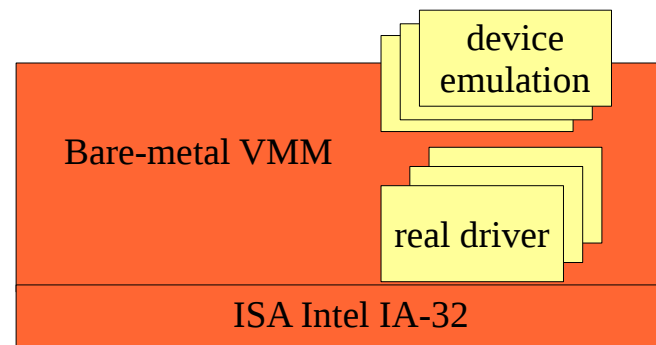
- Run directly on bare hardware
 - Run in kernel mode
 - All traps and interrupts go to the VMM
 - **Guest software runs in user mode**
 - Guest can be unmodified or para-virtualized



Type-I Virtual Machines

5

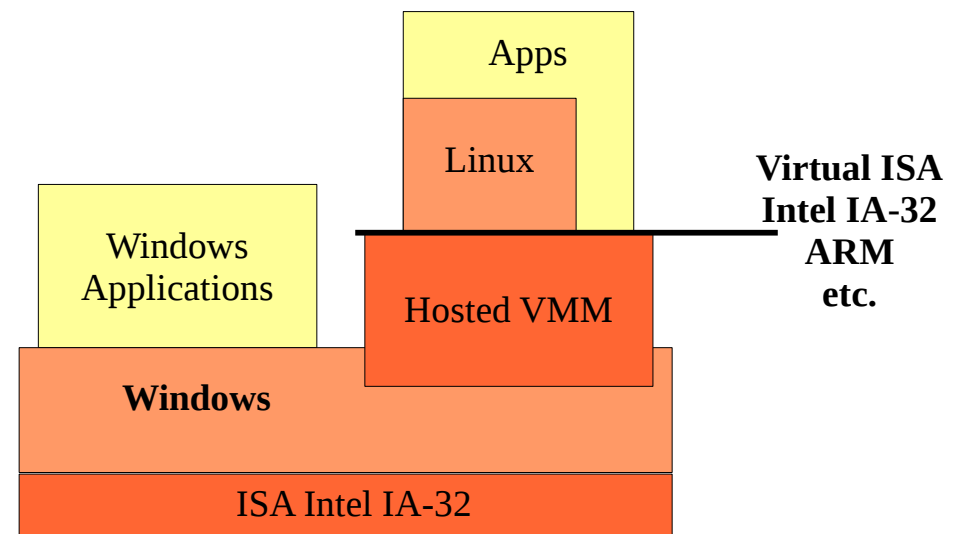
- Virtual ISA may differ from real ISA
 - Generic I/O devices from existing hardware
 - New I/O devices emulated on others (serial line on Ethernet for e.g.)
 - Less or more cores, less or more memory
 - Often the same instruction sets, but not always



Type-II Virtual Machines

6

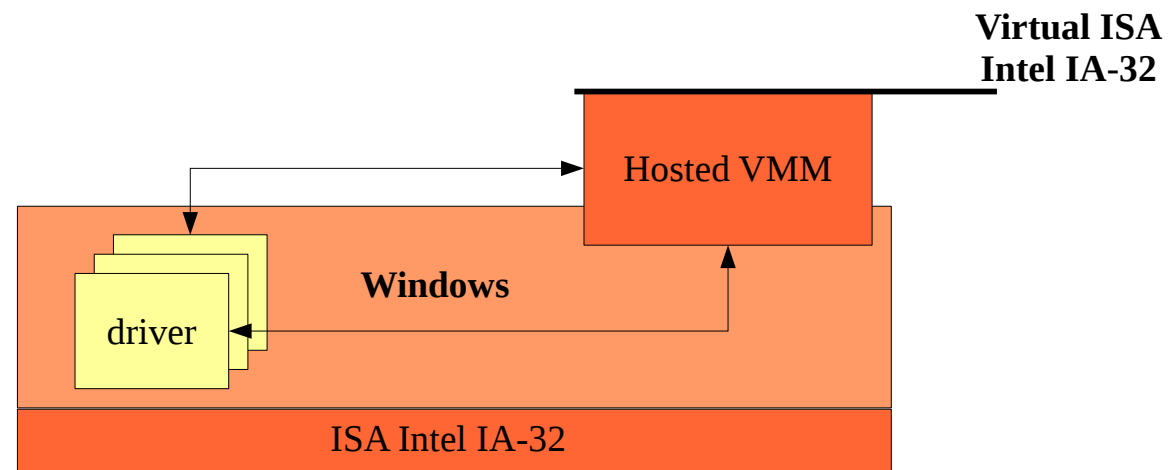
- A regular application in a process
 - Runs in user-mode, usually with a kernel module
 - Usually host a single guest operating system
 - Guest runs in user mode
- Virtual ISA may differ from real ISA
 - May provide the same ISA or a different one
 - With identical or different virtual devices



Type-II Virtual Machines

7

- Can leverage the drivers from the host operating system
 - Can still virtualize new devices or more generic devices
- Can integrate with the host environment
 - Can appear as a window on the host desktop
 - Can provide cut&paste abilities
 - Can provide a shared file system
 - Can provide debugging (like gdb stub in qemu)



Back to the Future...

- A short trip along memory lane

- System/360 was the first hypervisor-based technology (in the 70's)
- The Model 67 introduced the idea of a self-virtualizing instruction set
- The instruction set could even be recursively virtualized
- Meaning a guest could host multiple guests

} This is hard to do efficiently
with today's technology.

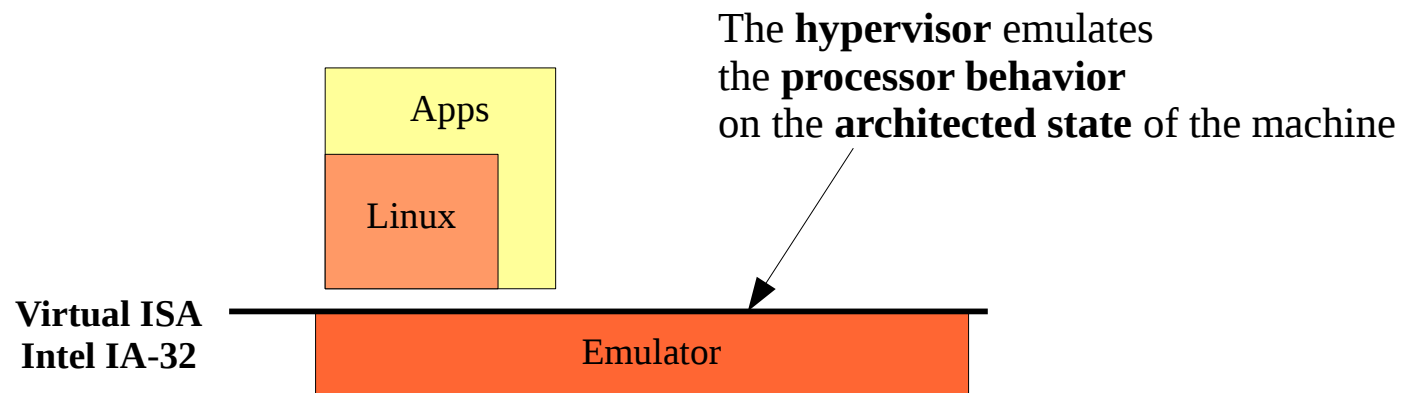
- A Cloud perspective

- The Internet technology + Hypervisor technology
- No longer renting physical machines, but virtual ones
- Can collapse/spread guests onto real machines
- Drastically reduces the energy costs (power and cooling)

Real Machine Emulation

9

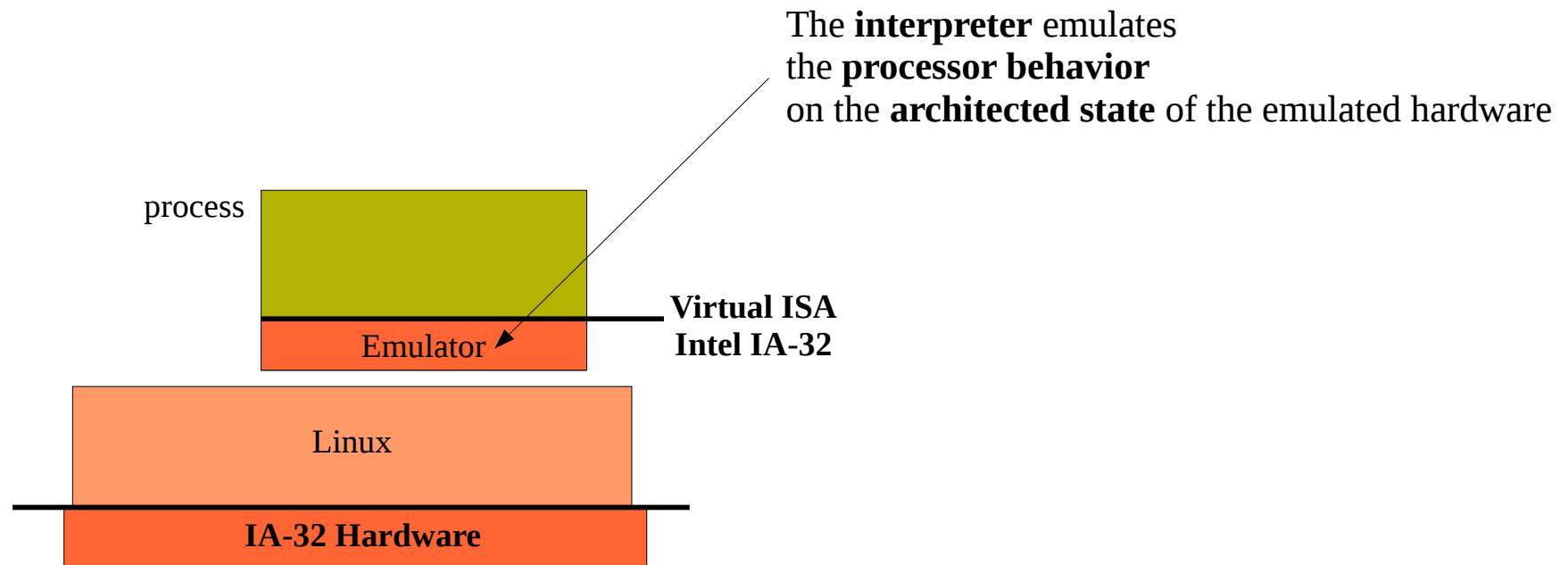
- Through emulation, we can always implement an hypervisor
 - Emulation means either **interpretation** or **binary translation**
 - **Provides the illusion of a real machine**
 - With the same ISA or not



Real Machine Emulation

10

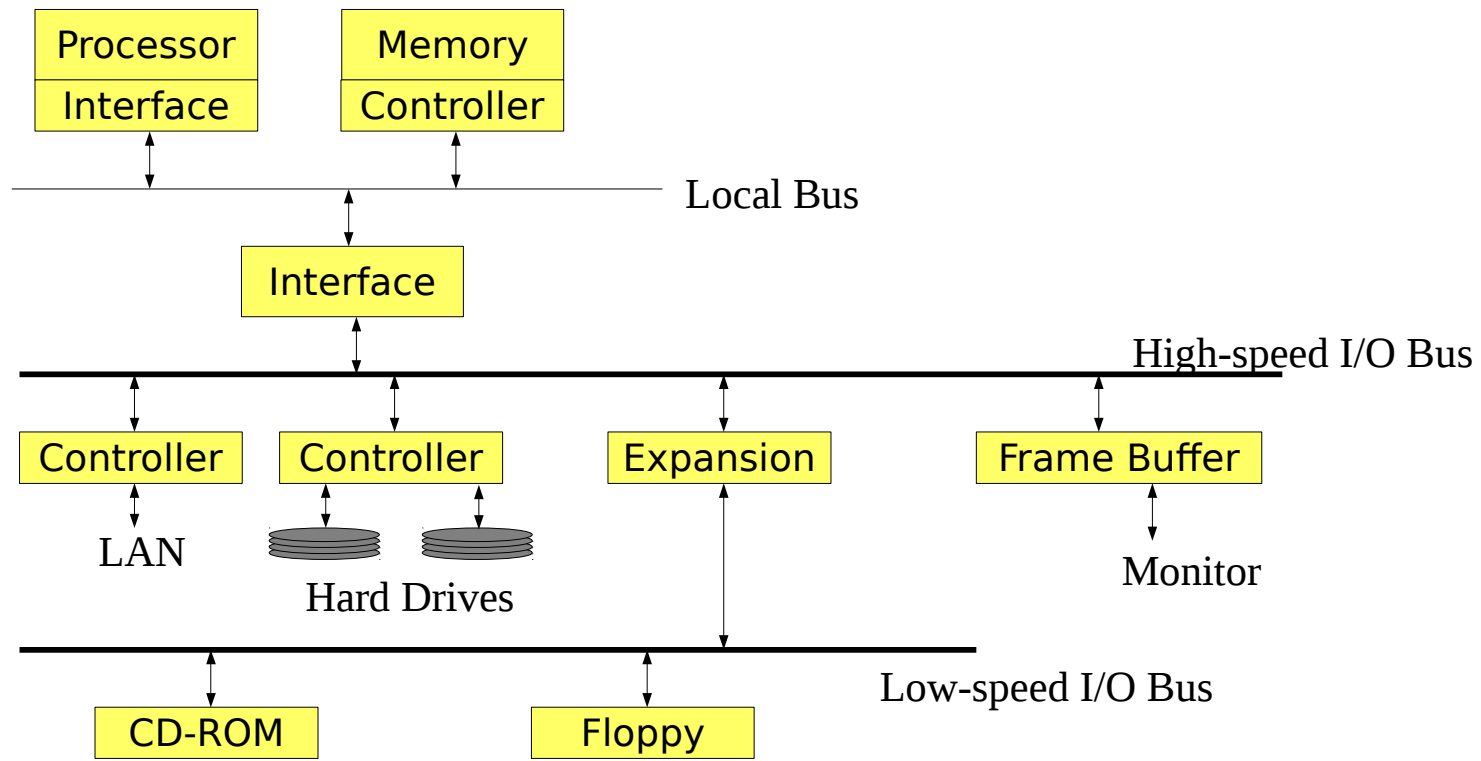
- Emulation by interpreter
 - Emulator is just a C program
 - Like any interpreter (JavaScript or Java)
 - Interpreter:
 - Fetch-decode-issue loop
 - Interpreted instructions manipulate the architected state



Architected State

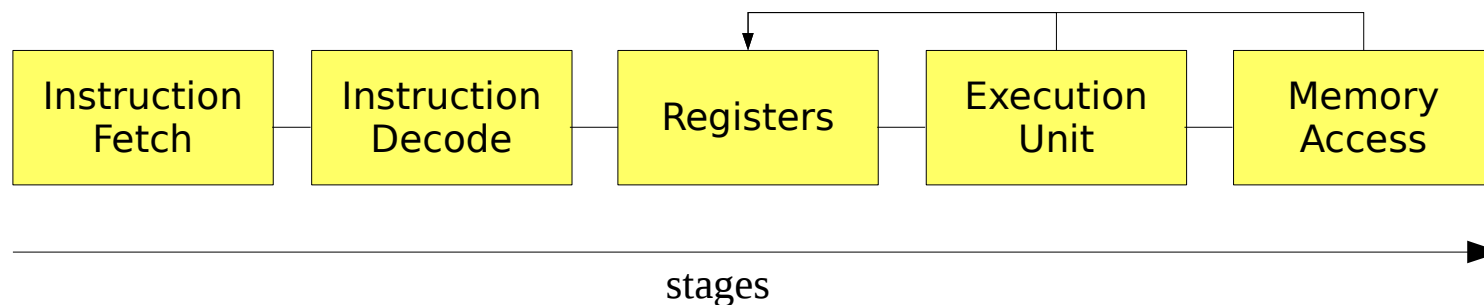
11

- What is the architected state of the emulated hardware?
 - The description of the hardware
 - The description of the current state of that hardware



- Emulates the processor on the architected state
 - Processor state
 - General purpose registers, floatint-point stacks or registers
 - Special registers such as status flags or timer value
 - Memory state
 - The content for the physical memory
 - The MMU state
 - Device states
 - The state for each device in use
 - Interrupt controller, pending/active interrupts

The interpreter emulates the processor on the architected state



- What about performance?
 - System C simulation
 - Very precise hardware simulation, but very very slow
 - Interpreters (e.g. Bosch)
 - Emulate different processors
 - Good faithful simulation of hardware behavior, but quite slow
 - Dynamic binary translation (QEMU)
 - Could also emulate different hardware
 - Faster if virtualizing the same ISA
 - Good top speed
 - Average is 5 to 20 times slower than native speed
 - Hardware-assisted virtualization (KVM, Xen, or Oracle VirtualBox)
 - All sensitive instructions trap, close to native speed
 - Often associated with para-virtualization for even greater speed
 - Often less than 30% overhead

Discussing Efficiency

14

- Quest for speed...
 - From interpretation to native execution...
- Interpreters (e.g. Bosch)
 - Fetch-decode-issue instructions in software
- Dynamic binary translation (QEMU)
 - Guest instructions are translated into host instructions
 - Same ISA: only a few instructions need translation
 - Different ISA: all instructions are translated (e.g. ARM → IA-32)
- Hardware-assisted virtualization (KVM, Xen, or Oracle VirtualBox)
 - Only support the same ISA (real and virtual)
 - All instructions execute natively
 - All sensitive instructions trap for enabling emulation
 - Novel hardware support for faster virtualization

- The challenge...
 - Execute native instructions for speed
 - Keep in control in order to preserve the illusion
 - In particular, isolate and multiplex guest VMs
- A difficult illusion to preserve
 - Multiple guest VMs sharing a single hardware
 - Guest VMs execute code that contains privilege instructions
 - Guest VMs observe and reconfigure the hardware
- Virtualization challenges
 - How do we virtualize the processor?
 - How do we virtualize memory?
 - How do we virtualizing devices?

Time-Sharing Guest VMs

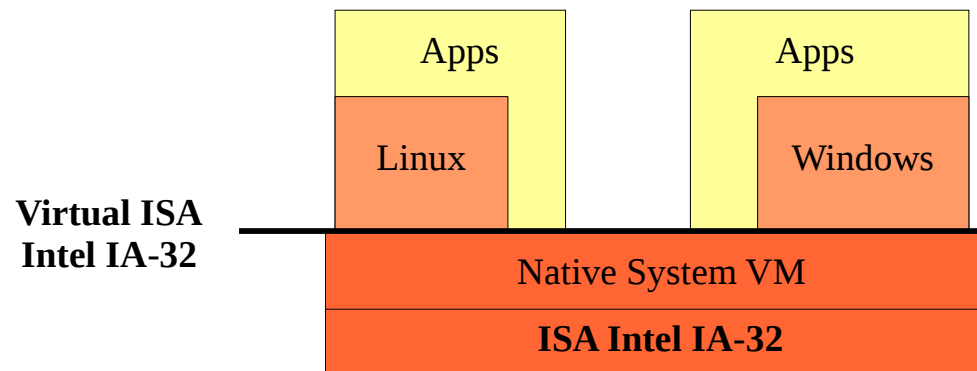
16

- **Processor Virtualization**

- Very similar issues to time-sharing applications
- Regular operating systems context-switch between applications
- Hypervisors context-switch between guest operating systems

- First, how is it done in a traditional operating system?

- Second, how would you do it for an hypervisor?



Traditional Multi-tasking

17

- Bare-metal event-oriented systems
 - No illusion → No virtualization
 - Developers know about tasks and events
- Cooperative multi-threading
 - No illusion → No virtualization
 - Threads yield the processor voluntarily
- Transparent scheduling
 - Threads appear to run uninterrupted
 - But they are not, they are multiplexed over cores
 - Illusion → virtualization

Traditional Multi-tasking

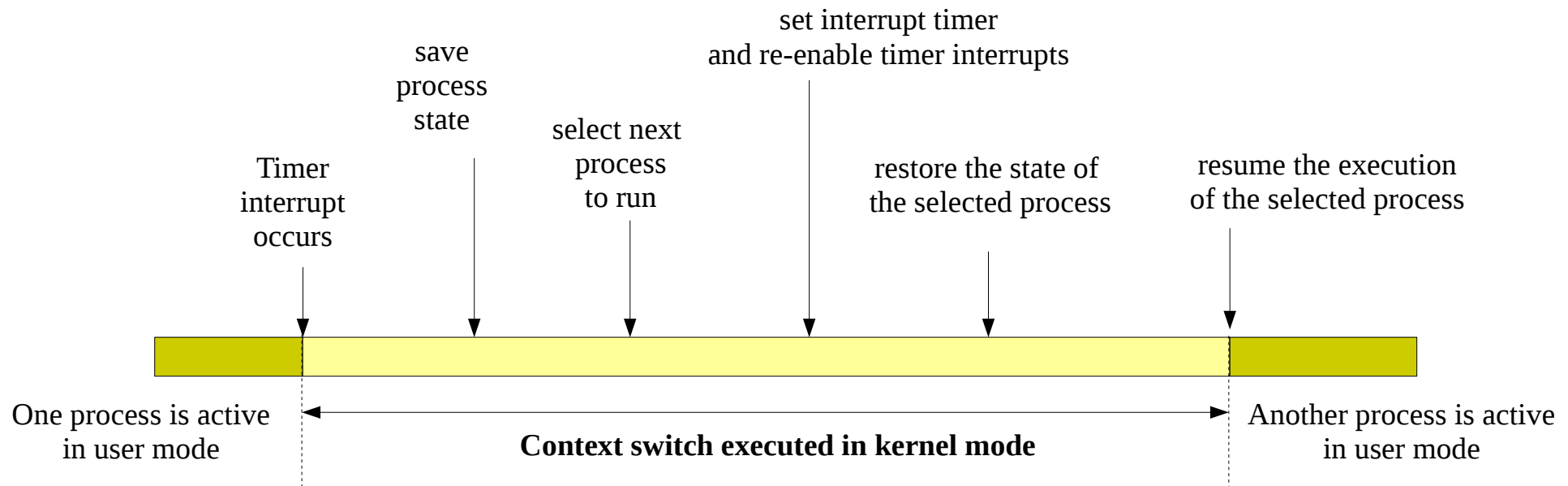
18

- Traditional design in an operating system
 - Use a timer to keep the control over the processor
 - Use an MMU to virtualize the shared memory
 - Use system calls to protect resource management
- Design relies on two modes of operation
 - Kernel and user mode
 - Privileged instructions only in kernel mode
- A small "architected state" per process
 - Its page table
 - Saved registers
 - Pending signals
 - Current syscall, if any

Traditional Multi-tasking

19

- Process scheduling overview
 - We have an architected state per process
 - We need to multi-task several processes

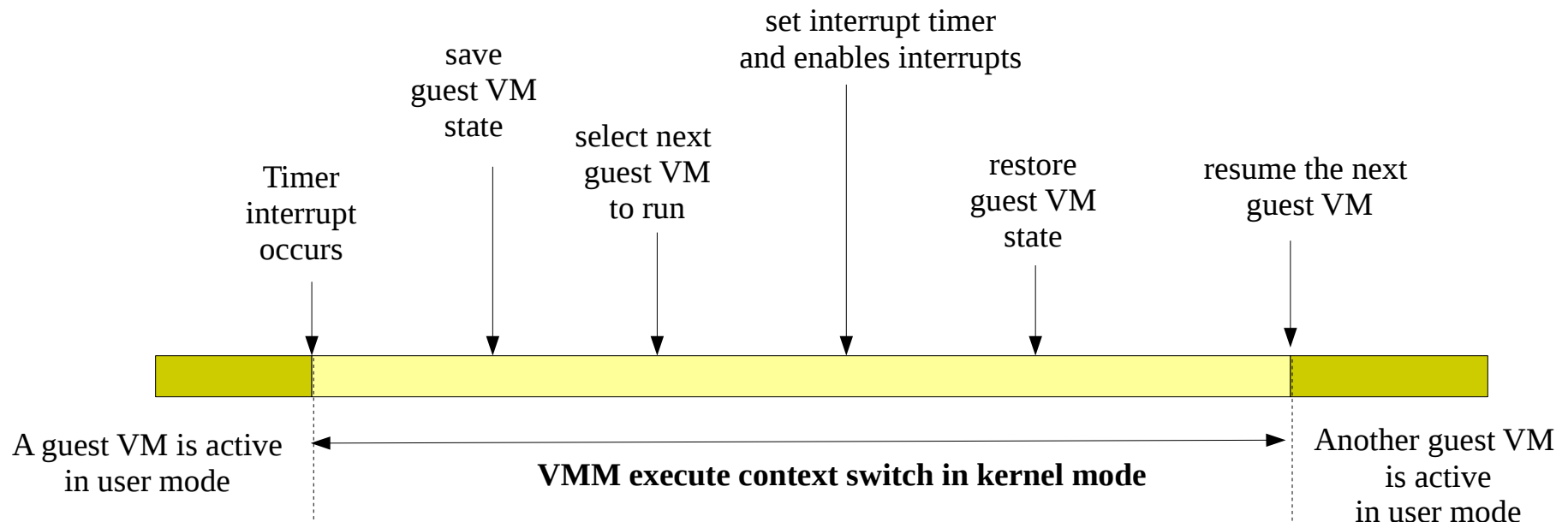


What is the difference with multiplexing guest operating systems then?

Multi-tasking Guest VMs

20

- Guest VM scheduling overview
 - We have one architected state per guest
 - We need to multiplex guests above



Multi-tasking Guest VMs

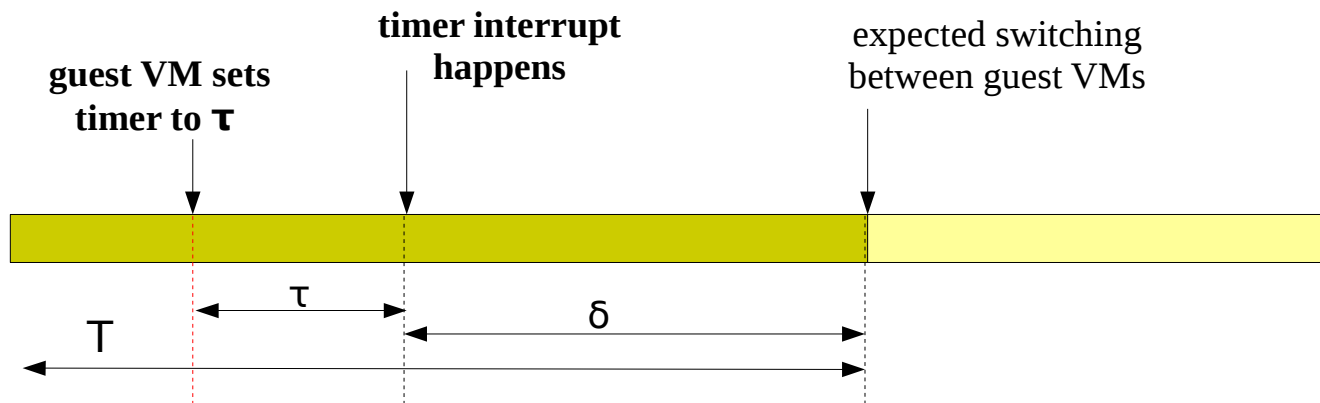
21

- Main difference
 - We are multiplexing operating systems
 - They expect to run in kernel mode
 - They actually manipulate the timer
- How do we schedule them and stay in control?
 - Only the VMM runs in kernel mode
 - Guest VMs run in user mode
 - Required emulation
 - The instructions manipulating the timer
 - The instructions manipulating the interrupt vector

Multi-tasking Guest VMs

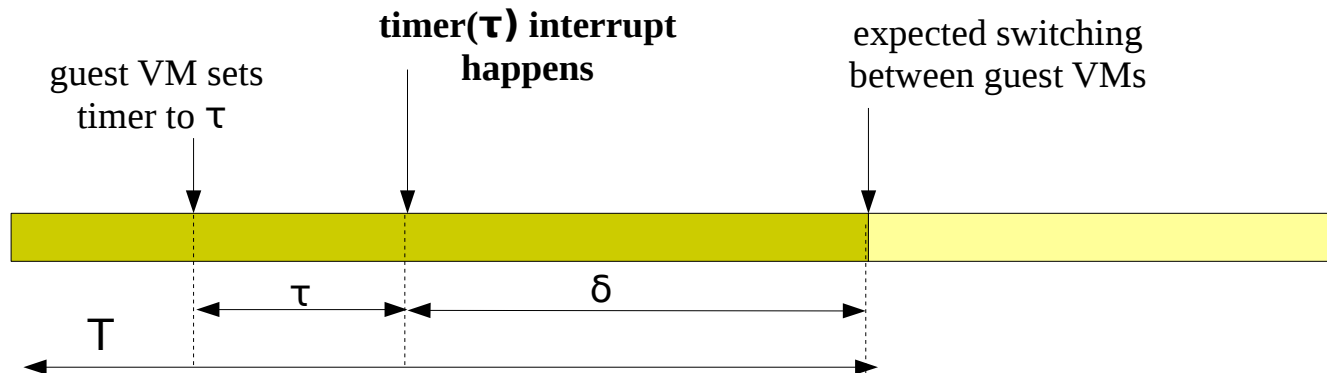
22

- Require emulation (cont...)
 - We want that each guest VM be scheduled for a time slice T
 - We must preserve the control of the timer
 - We must decide what to do upon timer interrupts

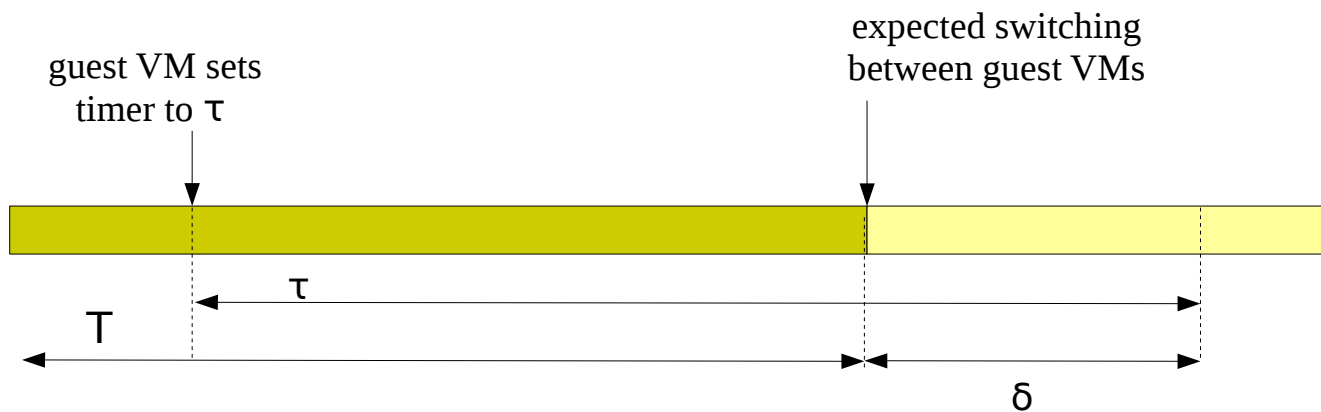


System Virtual Machines

23

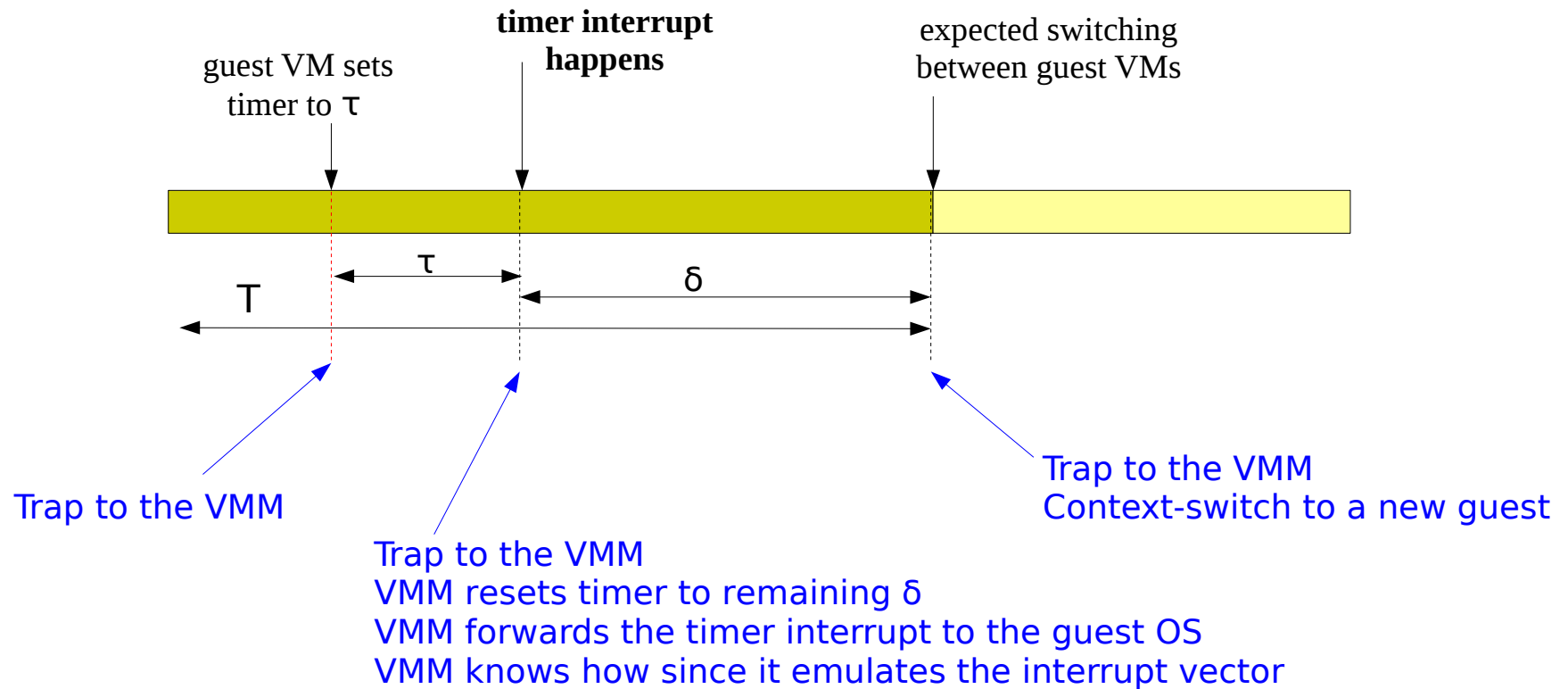


In reality, two cases to consider... Either τ is shorter than the remaining time in T or not



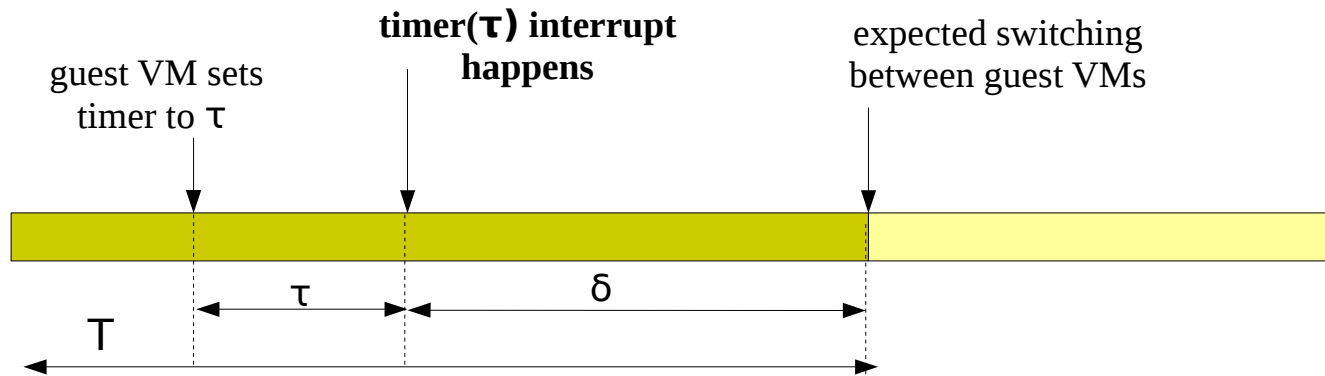
Multi-tasking Guest VMs

24

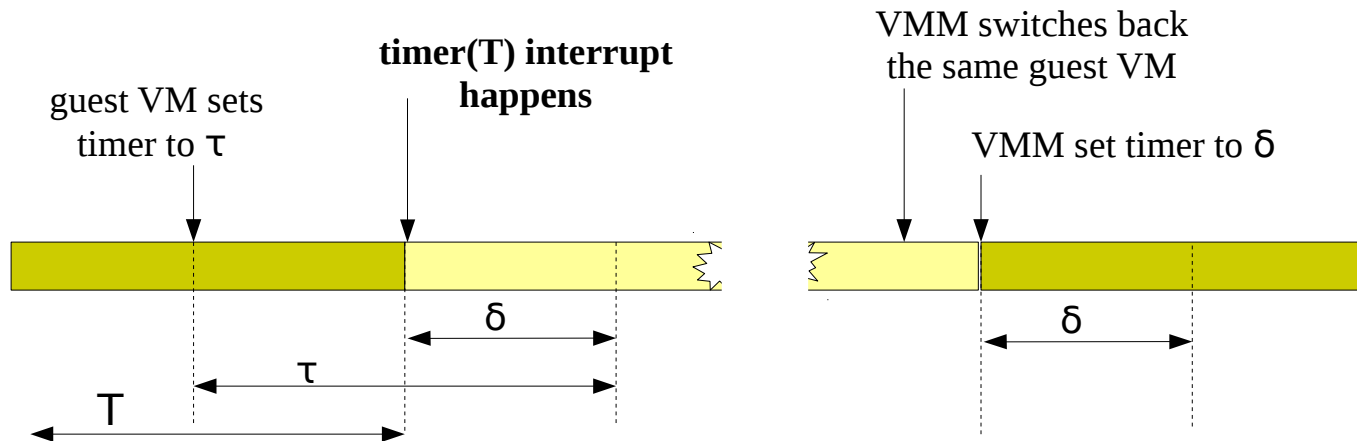


Multi-tasking Guest VMs

25



Overall summary:



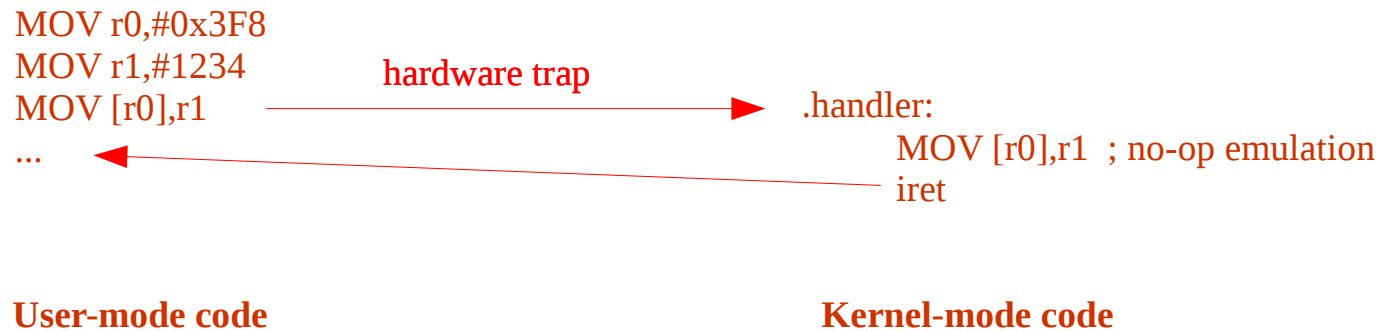
- What are the tricks we used?
 - Emulated the interrupt vector
 - So that we know where to jump in the guest OS when forwarding an interrupt
 - Emulated the timer register
 - So that we know the timer value desired by guest Vms
 - Provide the expected value if the OS reads the timer
- How did we emulate?
 - We did not interpret all instructions
 - Guest code runs natively, so how do we emulate?

Multi-tasking Guest VMs

27

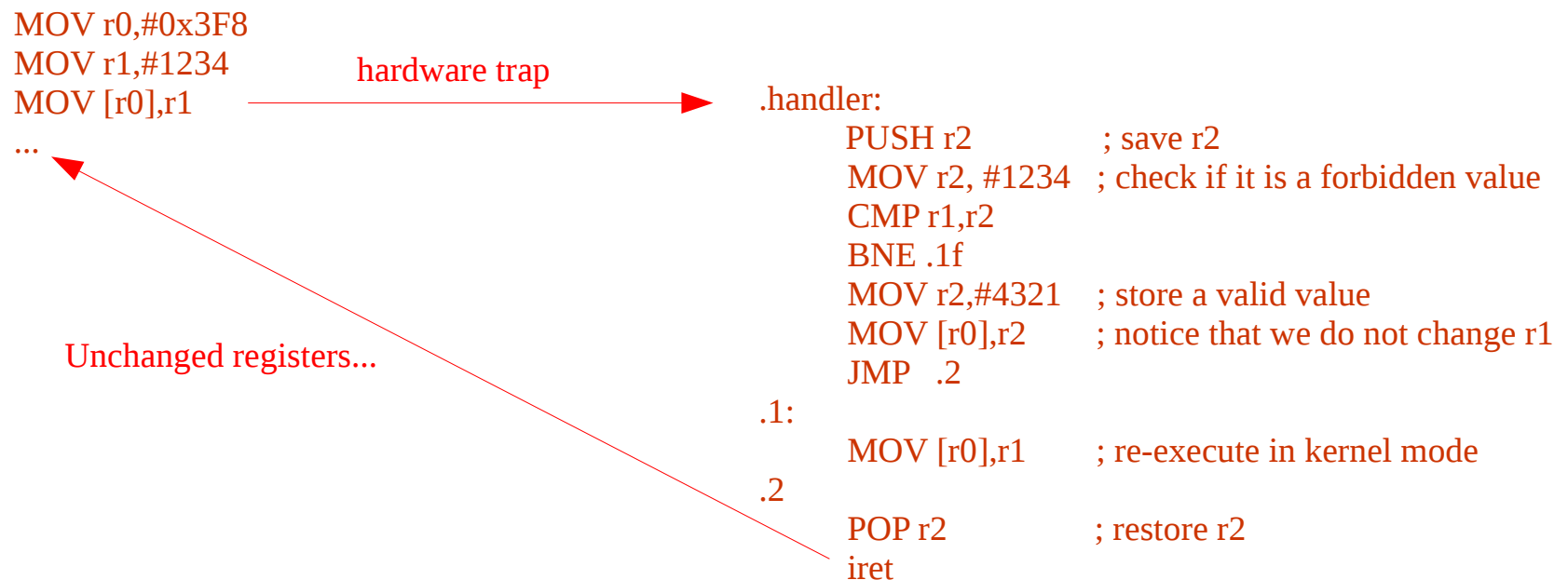
- Using traps
 - A trap happens on an instruction
 - After the trap, the instruction can be re-executed or not
 - This means the instruction can be entirely emulated

Emulated store operation on kernel-protected page



- Using traps
 - A trap happens on an instruction
 - After the trap, the instruction can be re-executed or not
 - This means the instruction can be entirely emulated

Emulated store operation on kernel-protected page

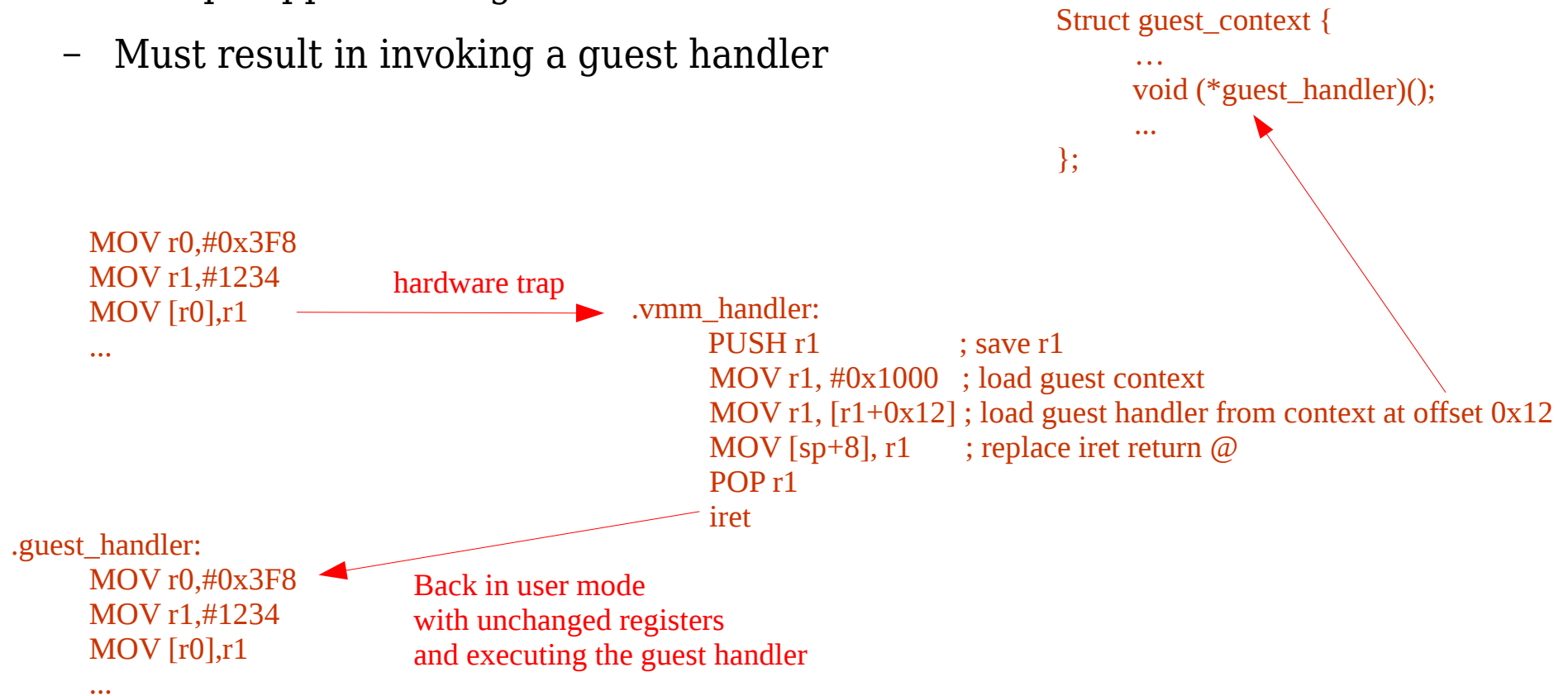


- So how did we emulate?
 - We can protect pages (kernel protected)
 - So read/write instructions will trap in user mode
 - We can assume that sensitive instructions are privileged
 - Since the guest runs in user mode, these instructions will trap
 - In the trap handlers, the VMM is back in control
 - Fully interpret the instruction and resume after the instruction
 - May change the hardware context and re-executes the instruction
 - May invoke guest handler code

Multi-tasking Guest VMs

30

- Forwarding traps
 - A trap happens on a guest instruction
 - Must result in invoking a guest handler



Emulation Challenge

31

- Guest VM run entirely in user mode
 - But it believes that it runs in kernel mode
 - It expects to have full control of the machine
- Guest will use **many sensitive instructions**
 - Setting interrupt and trap vectors
 - Changing page table pointers
 - Changing between user and kernel modes
 - Etc.
- **Sensitive instructions**
 - Precisely those that must be emulated...

- **Control-sensitive instructions**

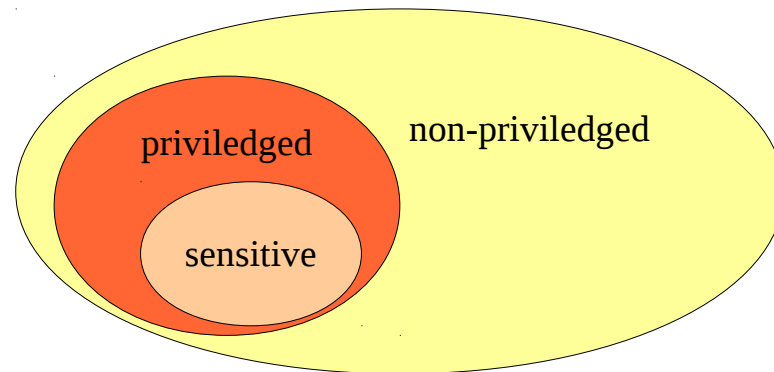
- *Attempt to change the configuration of resources in the system*
- Examples
 - Changing the system mode (user to kernel for e.g.)
 - Changing a page table or switching page tables

- **Behavior-sensitive instructions**

- *Depend on the configuration of resources*
- Examples
 - Reading the timer or the system mode
 - Reading and writing memory (virtual memory)
 - Setting processor flags whose behavior depends on the system mode
 - E.g. the interrupt enable/disable flag can only be changed in system mode

- Virtualization theorem

- *An efficient system virtual machine may be constructed if the set of sensitive instructions for the real ISA is a subset of the privileged instructions.*



- **Privileged instructions**

- Instructions that trap in user mode

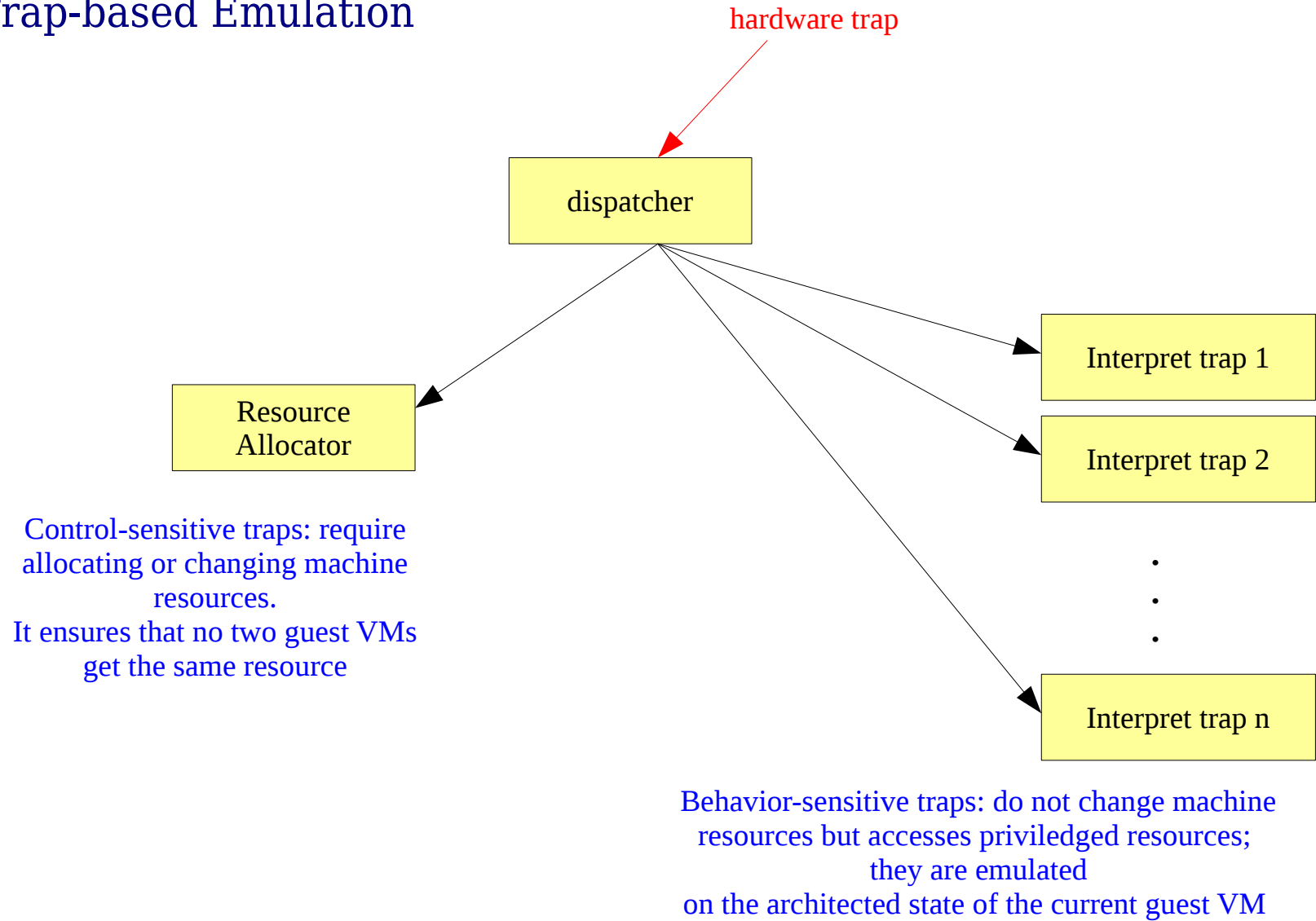
- **Important**

- It is not sufficient that the behavior be different in user mode
 - E.g. such as loading the IA-32 flag registers that leaves the interrupt mask unchanged in user mode but not in kernel mode

Emulation Challenge

34

- Trap-based Emulation



Emulation Challenge

35

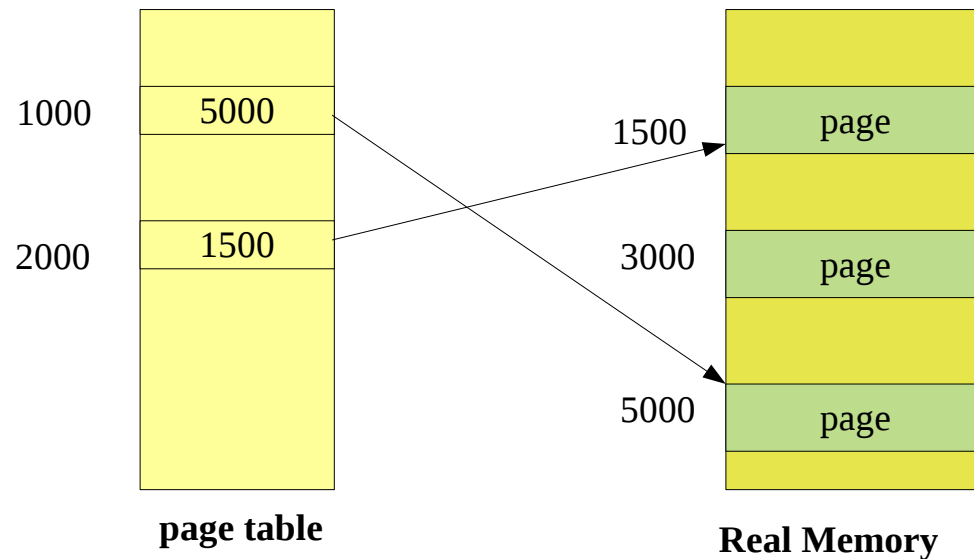
- We just discussed trap-based emulation
 - Easy to apply to the timer emulation
 - As well as the interrupt and trap vector
 - So we emulated the processor of a real machine ✓
- We still need to discuss how to emulate
 - Memory, including the MMU
 - Devices, including the interrupt controller

- Traditional Operating System
 - Physical memory is virtualized through MMUs
 - Providing applications with the illusion of private memory
- With a VMM
 - How do we virtualize physical memory to guest operating systems?
 - Guest operating systems will still need to virtualize memory
 - We need two levels of virtualization
 - But we have only one MMU...

Emulating Memory

37

- Virtualize an architected page table
 - A guest VM view
 - **Real memory**
 - **Virtual memory per process**



Virtual memory for a process P

Real page 3000 is not mapped to any process, but it exists.

The real memory depends on the actual DDR banks plugged in your mother board...

And some architected regions such as the memory-mapped I/O ports

Architected Page Table Emulation

38

- Virtualize an architected page table
 - Again a story of make believe through emulation
- Each guest VM
 - Manages several page tables
 - Changes the page table pointer upon context-switching
- Emulation through traps
 - Load and store instructions in the page table register are privileged
 - They will trap when used by the guest VM in user mode

Architected Page Table Emulation

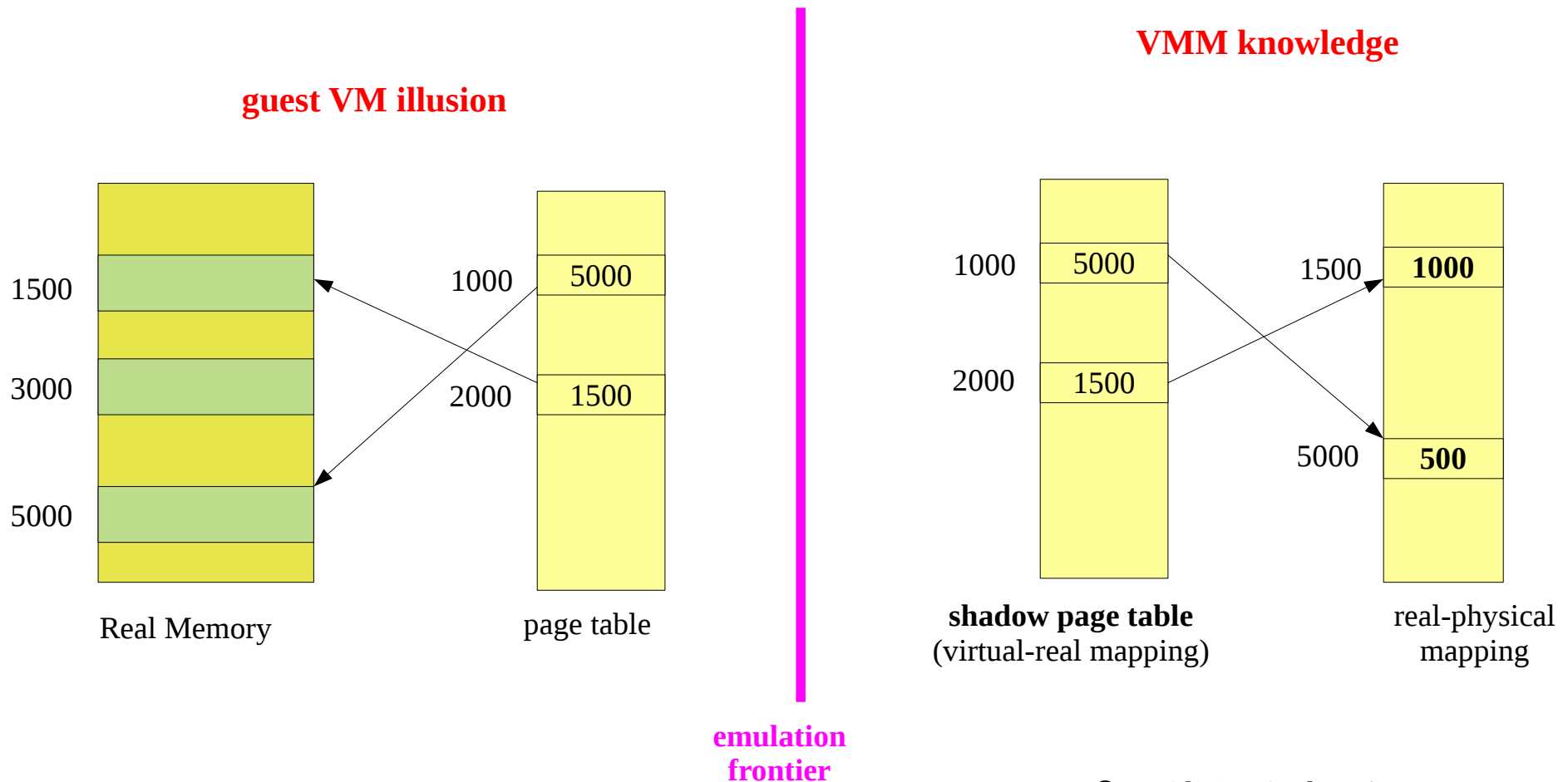
39

- The VMM emulates the virtual-to-real-to-physical mapping
 - Per page table
 - Keep a ***shadow page table*** in the architected state of the guest VM
 - Track the mapping **virtual-to-real** mapping
 - Per guest VM
 - Keep the **real-to-physical** mapping
 - Across guest VMs
 - Keeps track of which physical pages are allocated and to which guest VM

Architected Page Table Emulation

40

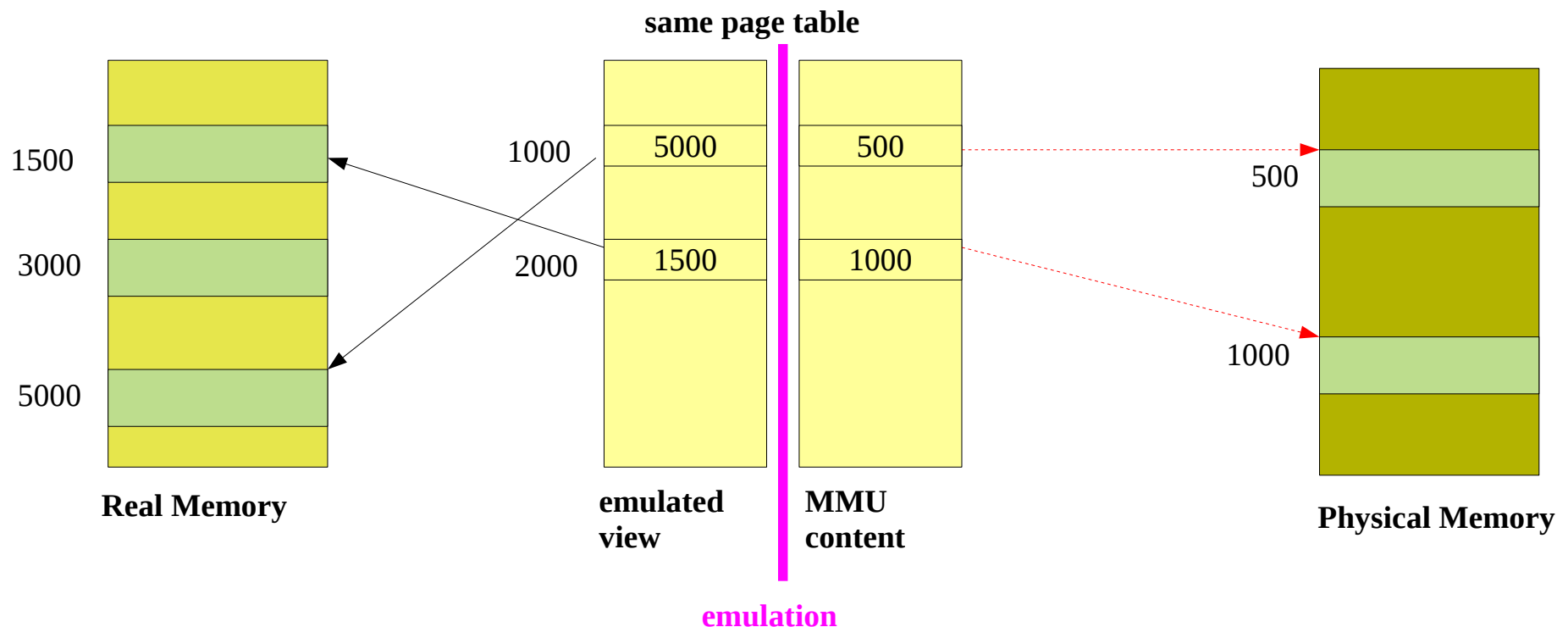
- Architected state per guest VM
 - Different page tables (virtual-to-real mappings)
 - One real-to-physical mapping



Architected Page Table Emulation

41

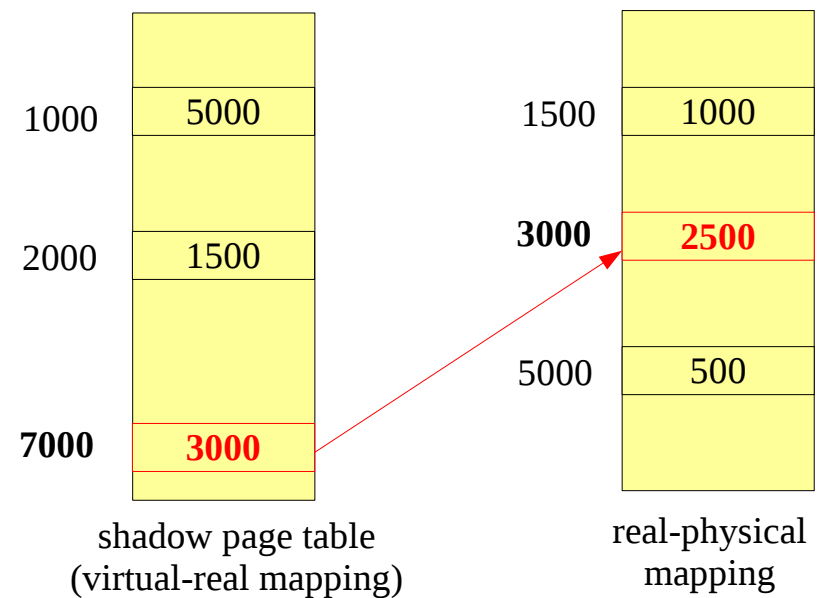
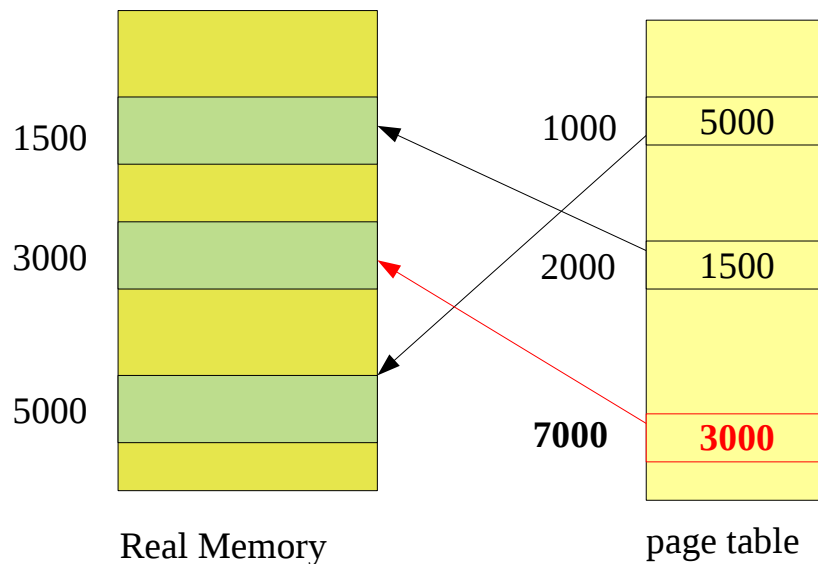
- The VMM emulates the virtual-to-real-to-physical mapping
 - Load and store instructions in the page table are privileged
 - They will trap when used by the guest VM in user mode
 - Translate real-to-physical before storing addresses in hardware MMU
 - Translated physical-to-real before returning addresses to guest VM



Architected Page Table Emulation

42

- Allocating a real page in the current page table
 - Virtual @=7000, real @=3000
- Need to see if the real page has a physical page
 - If not, need to allocated one (@=2500)

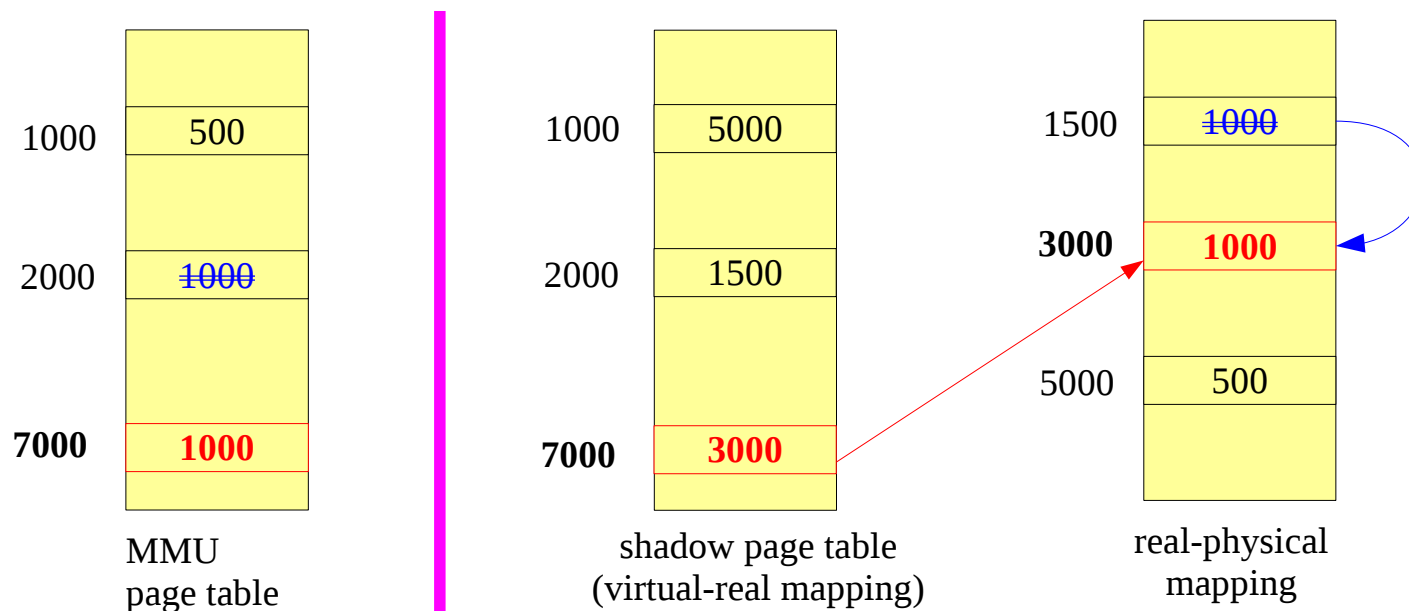


emulation
frontier

Architected Page Table Emulation

43

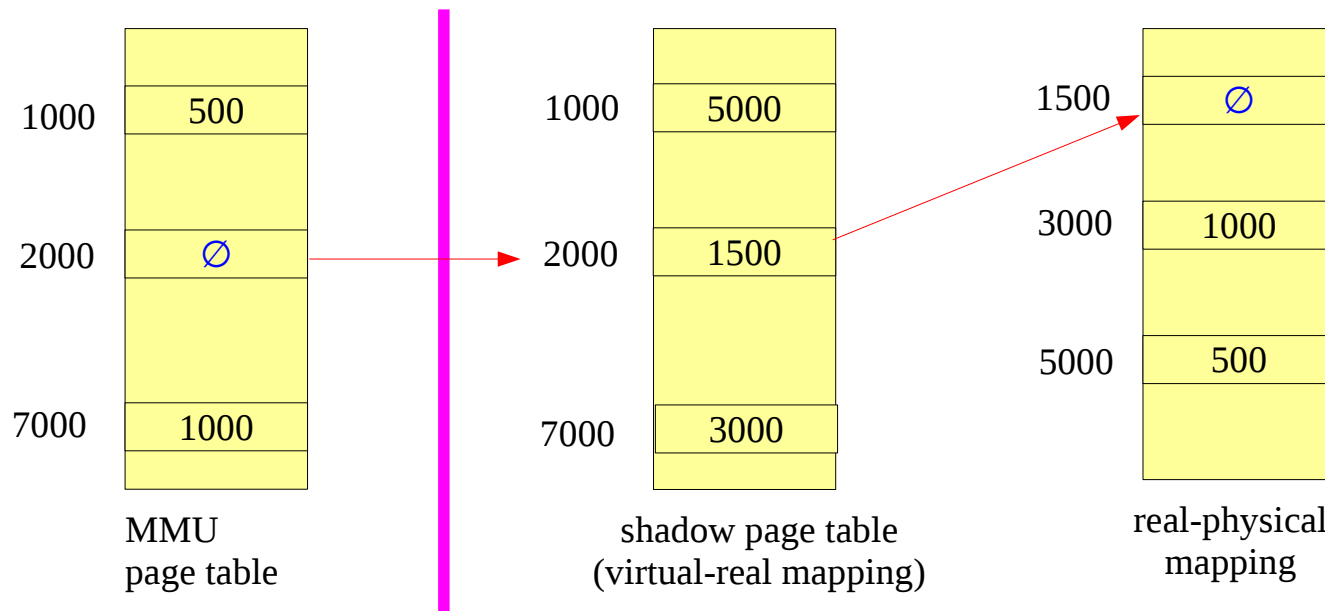
- Potential physical page replacement
 - Need to invalidate the corresponding entry in its real-physical mapping
 - May not always be in the mapping of the current guest VM
- If in the current architected state
 - Need to invalidate entry in the MMU page table
 - Need to flush TLB (or at least the corresponding TLB entry)



Architected Page Table Emulation

44

- Emulate Page faults
 - Page faults may be on real or physical memory pages
 - Real memory page faults need to be forwarded to the guest VM
 - Physical memory page faults must be handled by the VMM
 - Example virtual address 2000 triggers a page fault
 - Guest VM expects it to be in memory (real memory)
 - But it is not in physical memory, VMM needs to page it in

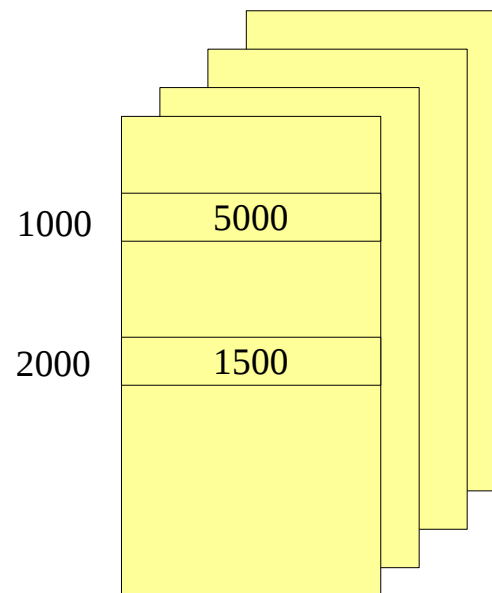


Architected Page Table Emulation

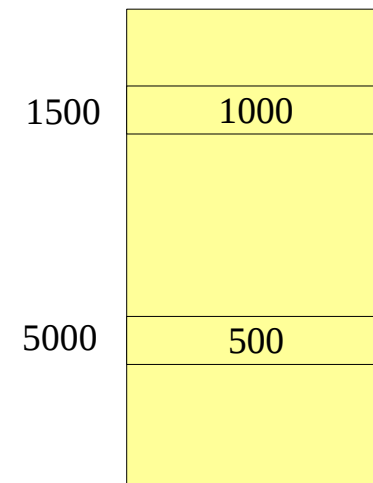
45

- Emulate address-space switching within a guest VM
 - Changing the page table pointer traps in VMM
 - Need to select the new virtual-real mapping
 - So to keep it consistent with the guest VM expected state

**VMM knowledge
per guest VM**



shadow page tables

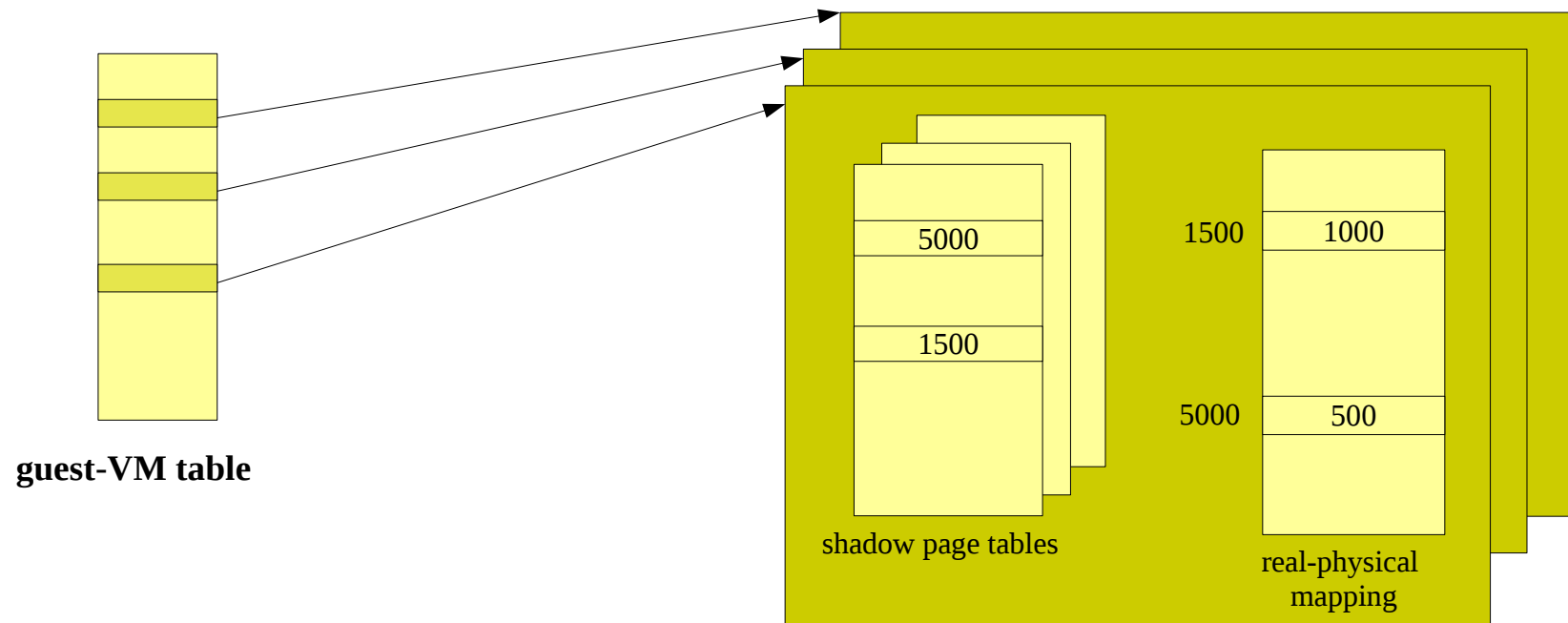


**real-physical
mapping**

Architected Page Table Emulation

46

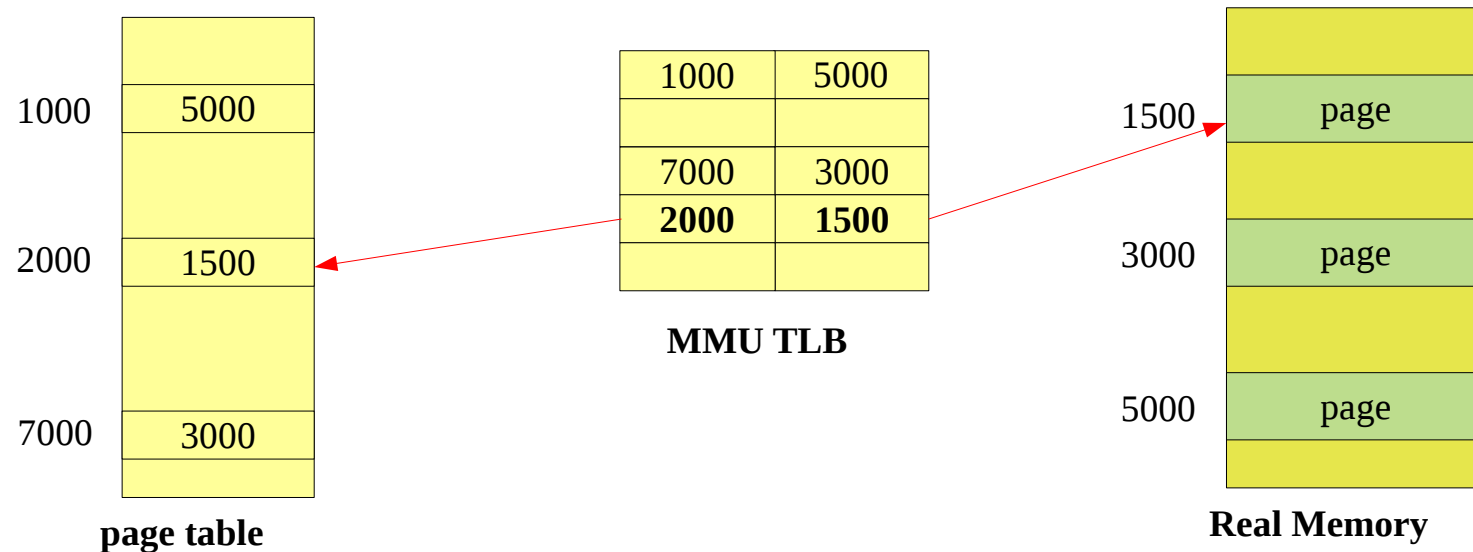
- Switching between guest VMs
 - Switching the architected state for the new guest VM



Architected TLB Emulation

47

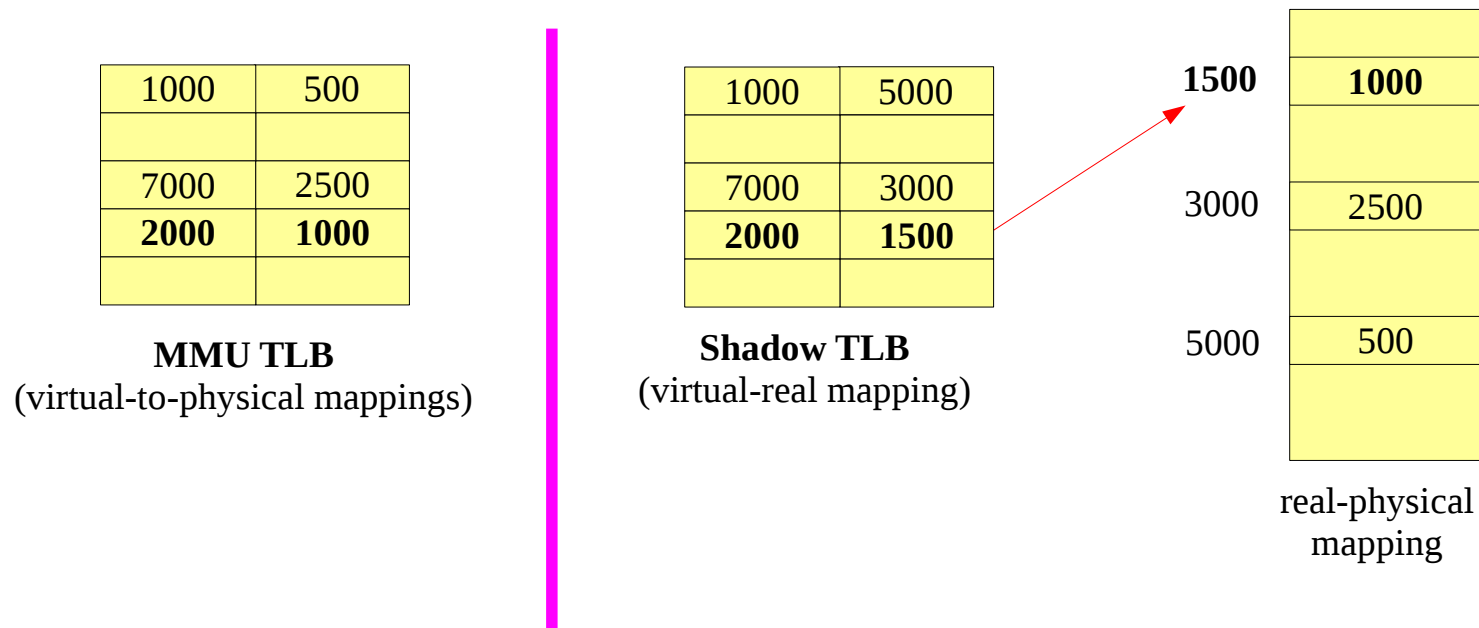
- Fairly similar to virtualizing an architected page table
 - Operations that manipulate the TLB are privileged
 - VMM emulates the TLB manipulation
- Guest VM view
 - A software-managed page table
 - An architected TLB



Architected TLB Emulation

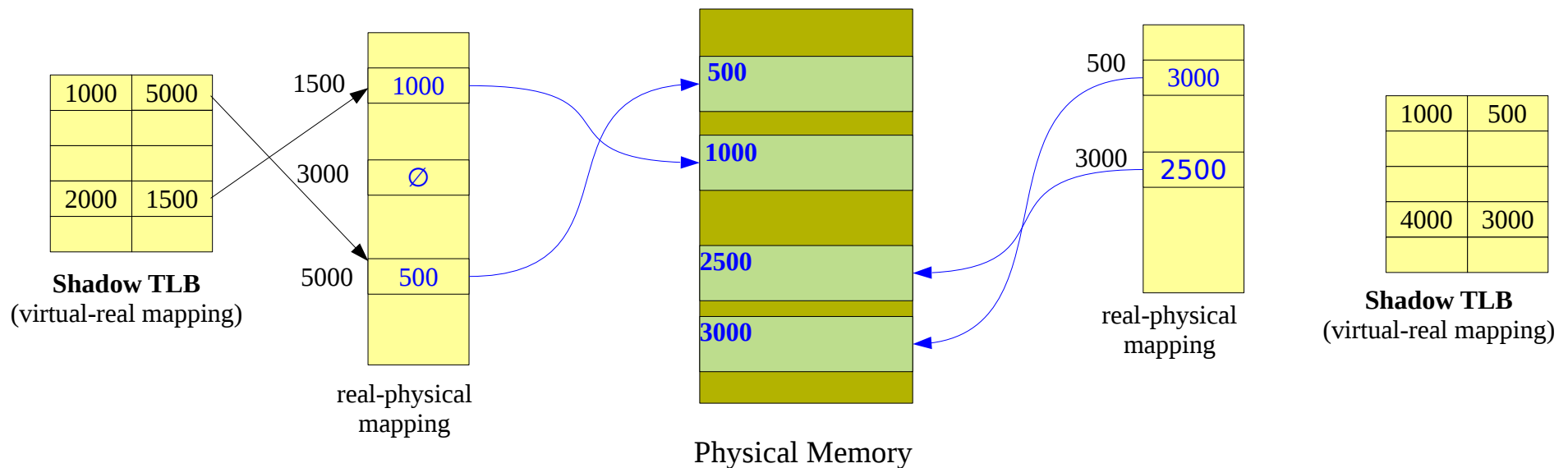
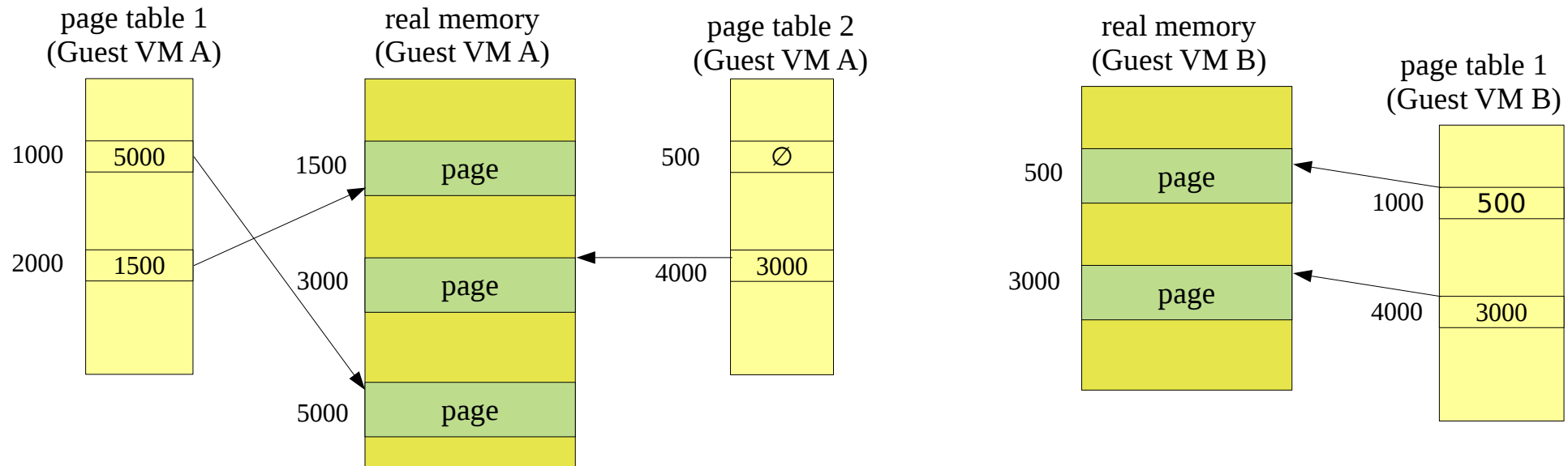
48

- VMM emulates the TLB manipulation
 - VMM manages a shadow TLB, **but only one per guest VM**
 - VMM still manages a real-to-physical mapping



Architected TLB Emulation

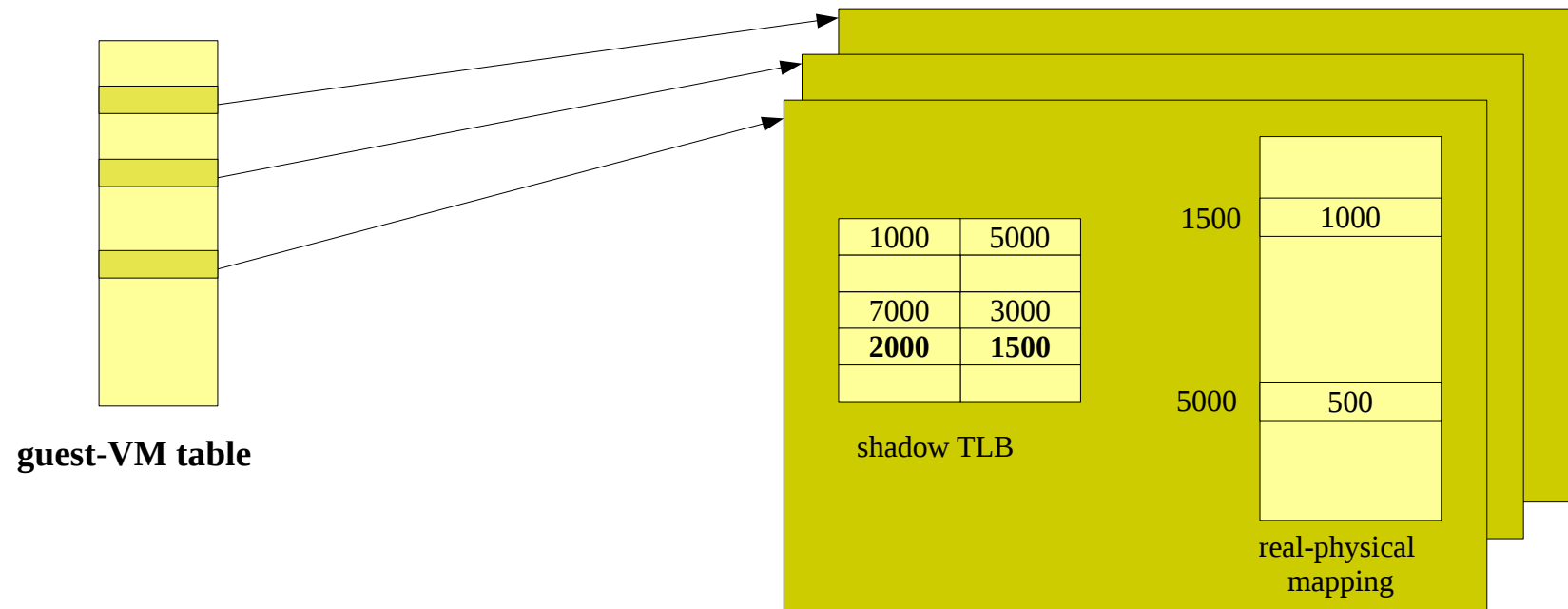
49



Architected TLB Emulation

50

- Switching between guest VMs
 - Switching the architected state for the new guest VM
 - Flush the entire TLB



Architected TLB Emulation

51

- Address Space Identifier (**ASID**)
 - Included support in architected software-managed TLBs
 - An architected ASID register contains the current ASID
 - ASID register is assigned on address space switch
 - Enables to mix address translations for different address spaces
 - ASIDs are checked upon every TLB translation
 - A translation is accepted only if the ASIDs match

ASID	virtual page	real page
2	1000	5000
3	1000	500
2	2000	1500
3	4000	3000

MMU TLB
(virtual-to-real mappings)

Architected TLB Emulation

52

- Emulating ASID management
 - Each guest VM may manage its own ASIDs
 - We may have conflicts between ASIDs across guest VMs
 - We need a mapping between virtual to real ASIDs

1	1000	5000
1	2000	1500
2	4000	3000

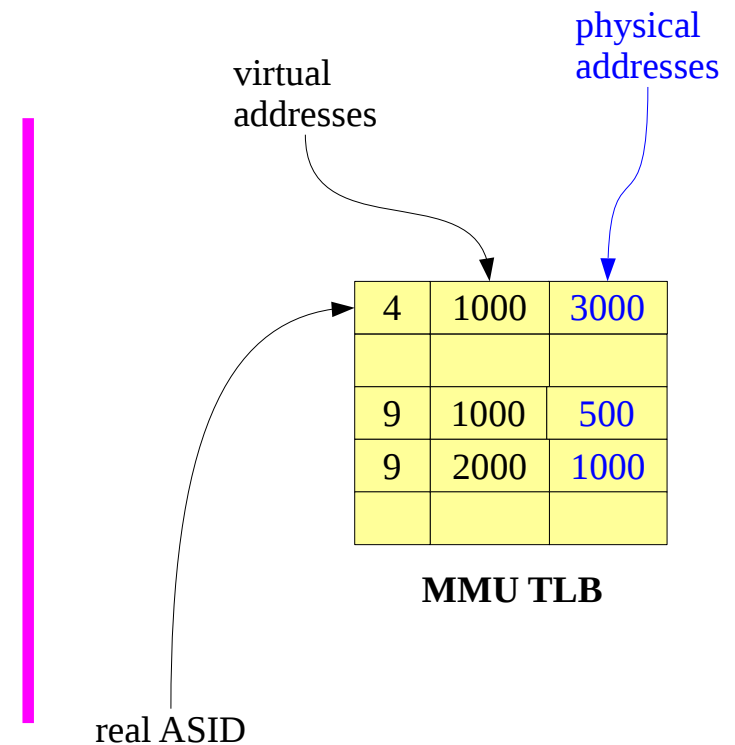
Shadow TLB
Guest VM A

1	1000	500
1	4000	3000

Shadow TLB
Guest VM B

VM-A:1	9
VM-A:2	-
VM-B:1	4

ASID
Mapping



Emulation Challenge

53

- Emulate processors ✓
- Emulate memory, including the MMU ✓
- Emulate devices, including the Programmable Interrupt Controller (PIC)

Device Emulation Challenge

54

- Virtualizing I/O devices is one of the more difficult aspects of VMM
 - Each I/O device has its own characteristics
 - Each I/O device needs to be controlled in its own special way
- The number of device types is growing constantly
 - In Linux, device drivers represent 70% of the code
 - In Linux, device drivers represent 70% of the bugs
 - Some devices only have proprietary drivers (typical of GPUs)

Device Emulation Challenge

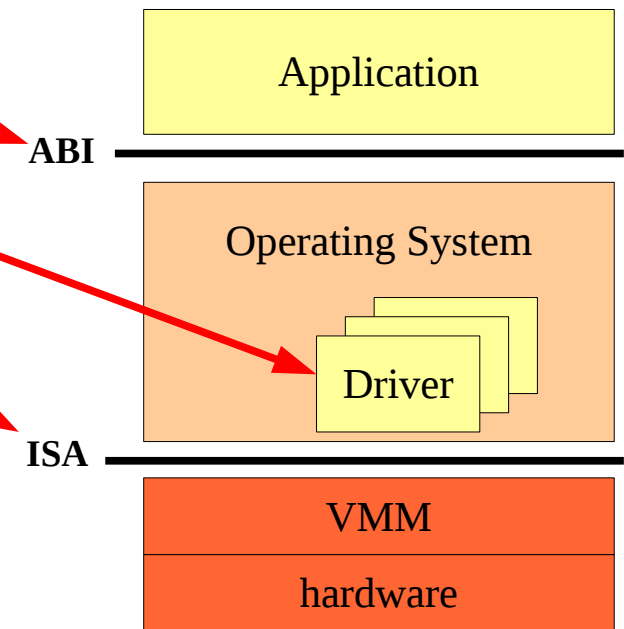
55

- Different device families
 - Each with different challenges
- Dedicated devices
 - E.g. keyboards or mouse or screen
 - Dedicated at least for some long period of time
- Partitioned devices
 - E.g. disks that can be partitioned across guest VMs
 - Each partition is virtualized as an independent disk
- Shared devices
 - E.g. network adapters, multiplexing packet transfers
 - Need to be actively shared between guest Vms

Emulating Devices

56

- Three possible emulation levels
 - At the system call interface (ABI)
 - At the device driver interface
 - At the hardware interface (ISA)
- Emulating at the hardware level
 - Unavoidable if unmodified guests
 - Guest drivers
 - Read/write hardware registers
 - Rely on interrupts
 - Often rely on ring-buffers and DMA



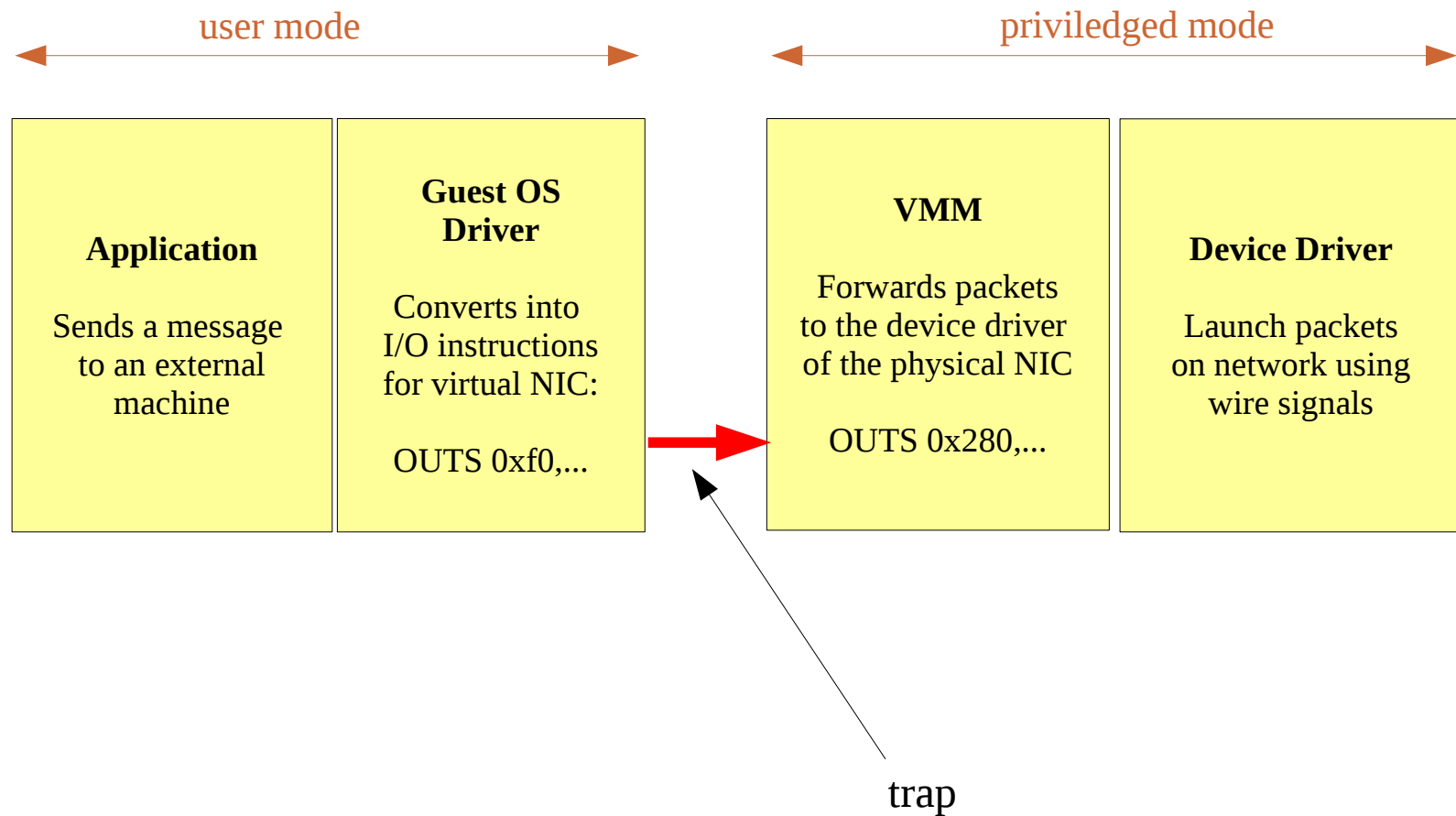
- Example: Network Interface Card (NIC)
 - Like an NE2K on IA-32
 - With IN/OUT or INS/OUTS instructions
 - On I/O ports (memory-mapped hardware registers)
 - Write commands and parameters
 - Read status
 - Ring buffers – the usual approach
 - Fixed-size circular buffer (hence the name ring)
 - Allocated in memory, as a shared data structure
 - One side pushes and the other consumes
 - One ring buffer for sending and another one for receiving

- Emulation: Network Interface Card (NIC)
 - Protect mmio regions via MMU
 - So I/O instructions on ports will trap
 - Let's assume that the virtual device is the same as the real one
- Emulation on the trap
 - Change the I/O port
 - From the virtual to real port number
 - Translate packet buffer address
 - The packet buffer address is a real address (not a virtual address)
 - Use the real-physical mapping to find the correct physical page
 - Reissue the I/O with correct I/O port and buffer address
 - Watch for **non-contiguous buffers** in physical memory (contiguous in real)
 - This means that the VMM needs **a device driver for the real hardware**

Emulating Devices

59

- Example: network virtualization

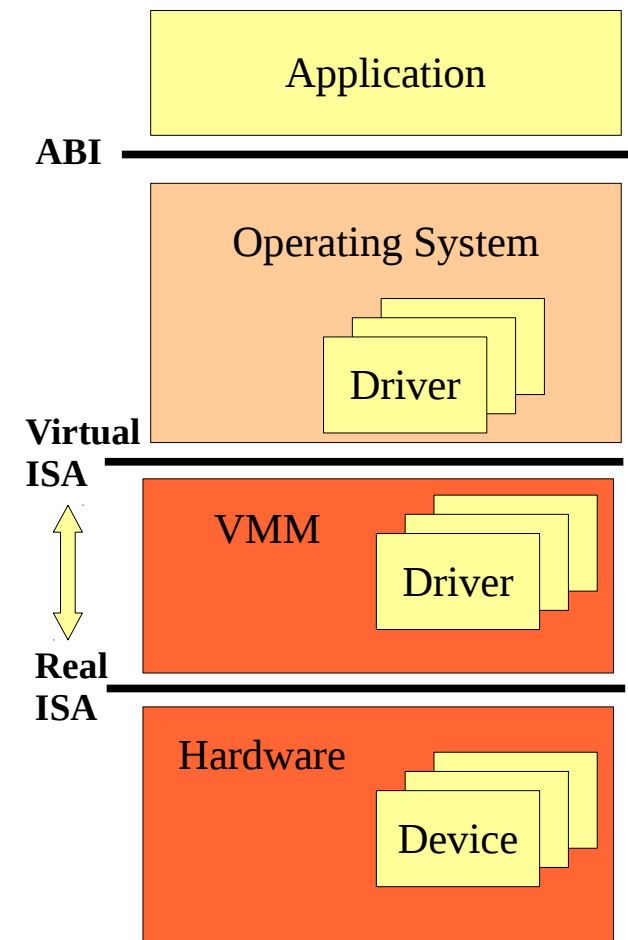


- Emulation: Network Interface Card (NIC)
 - Let's assume that the virtual device is **different** than the real one
- Emulation
 - Need to emulate the entire finite state machine of the device
 - Accumulate values written in I/O ports
 - Emulate correctly all values read from I/O ports
 - Emulate correctly all the status responses (including errors)
 - Must have a complete specification of the device behavior
 - Often fuzzy, this is why writing device drivers is black art...
 - Word sizes might be different...
 - Endianness might be different...
 - Sometimes must emulate precise timing...
 - ...

Emulating Devices

61

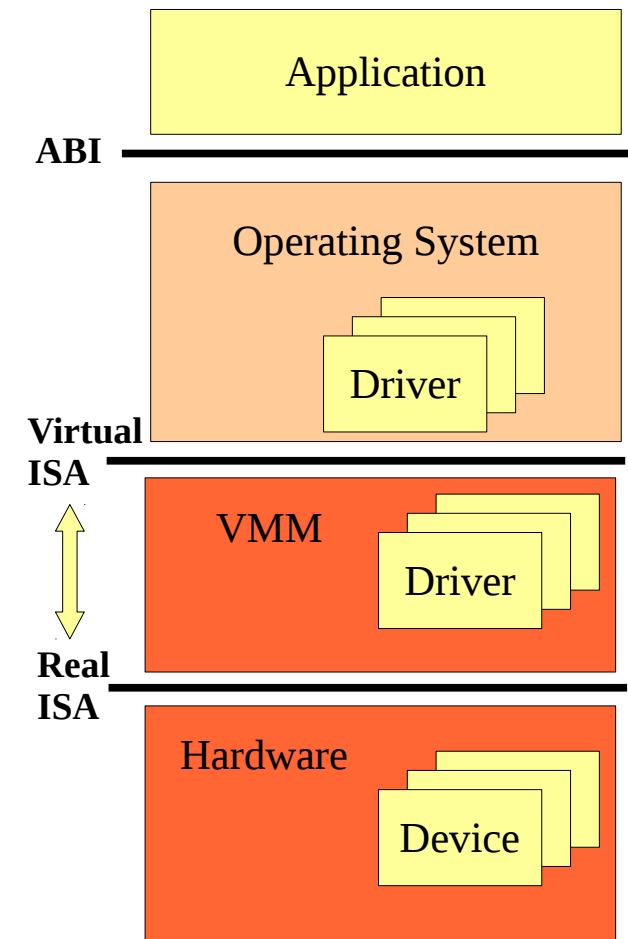
- Type-I Hypervisor Challenge
 - VMM drivers are “translators”
 - Upper interface: emulate a virtual device
 - Lower interface: drive a physical device
- **Requires many drivers in VMM**
 - Even if only a few devices are emulated
 - Need one driver per real device
- **Example:**
 - For an emulated NIC e1000
 - Still need to drive **all** existing network cards



Emulating Devices

62

- Type-I Hypervisor Summary
 - Easy to intercept
 - Trap on operations on mmio registers
 - Hard to emulate
 - One high-level I/O may requires several low-level I/O loads or stores
 - **Need a very precise emulation**, including the idiosyncracies of the real hardware...
 - So it is **even worse** than developing drivers for a regular OS



Emulating Devices

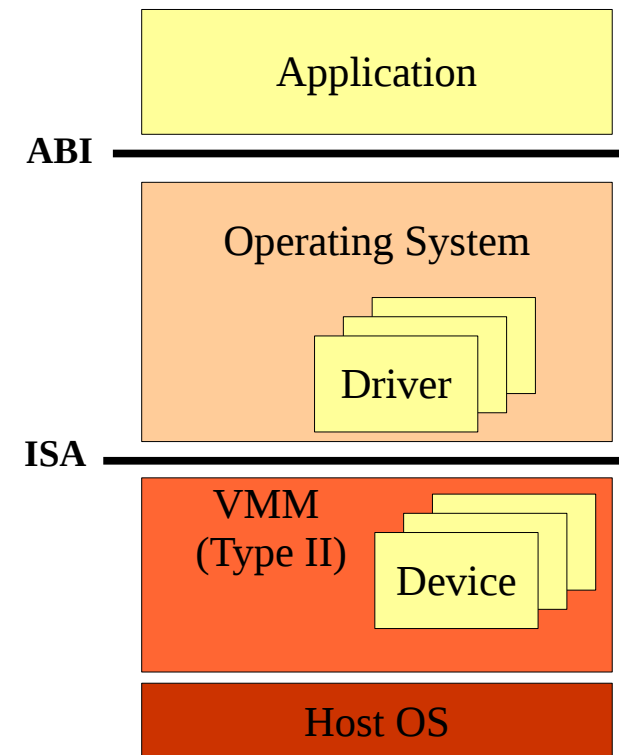
63

- Type-II Hypervisors

- A simpler situation
- Usually emulate only a few devices
- Still require a precise emulation (difficult)
- Implemented using the host OS (easy)

- Examples:

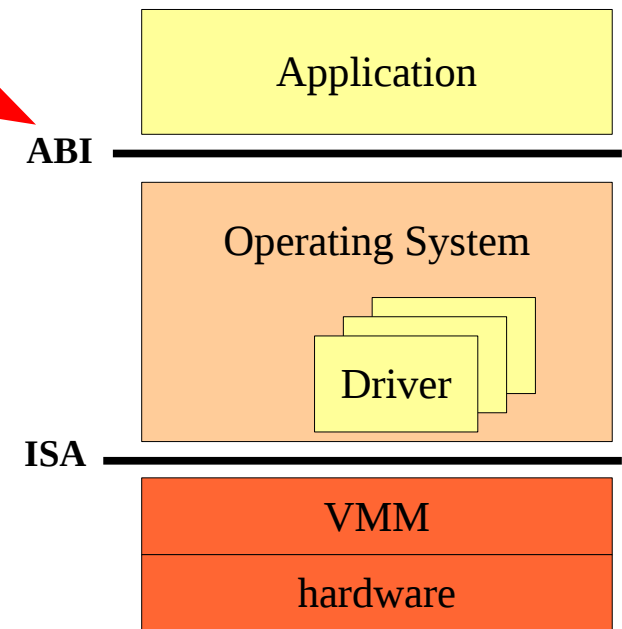
- Disk devices
 - Usually implemented on top of regular files
 - Could be on partitions on real disks
- Network devices
 - Implemented over a tunnel (dedicated network)
 - Or over a TAP (direct access to network frames)



Emulating Devices

64

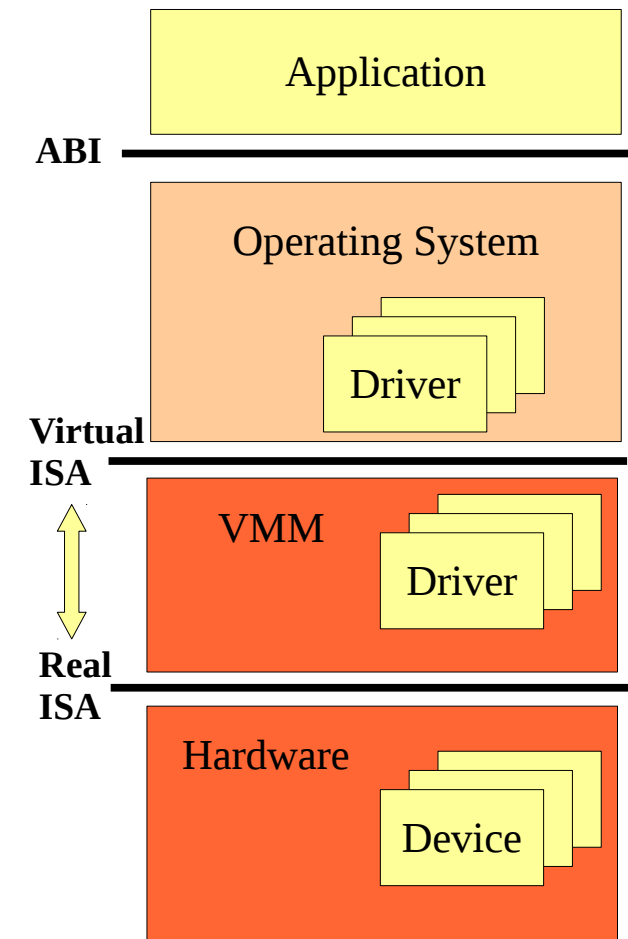
- Emulate at the system call interface
 - Could be more efficient in theory
 - Capture the original I/O at the ABI level
 - Emulate it entirely in one trap
- Daunting task in general
 - The VMM needs an ABI mirroring the guest OS ABIs with many system calls
 - Can only be done if the VMM team has intimate knowledge of the guest operating systems
 - No longer a general transparent solution



Emulating Devices

65

- Device Drivers - The plague
 - Micro-kernels faced a similar challenge...
 - They needed user-level device drivers
 - Which meant to port/rewrite all device drivers
 - It was one of the main show-stoppers
- Hypervisor Solution
 - **Introduce para-virtualization**
 - Dropped the transparency goal
 - Guest OS are **modified** to use **hypercalls**
 - Guest drivers are modified too

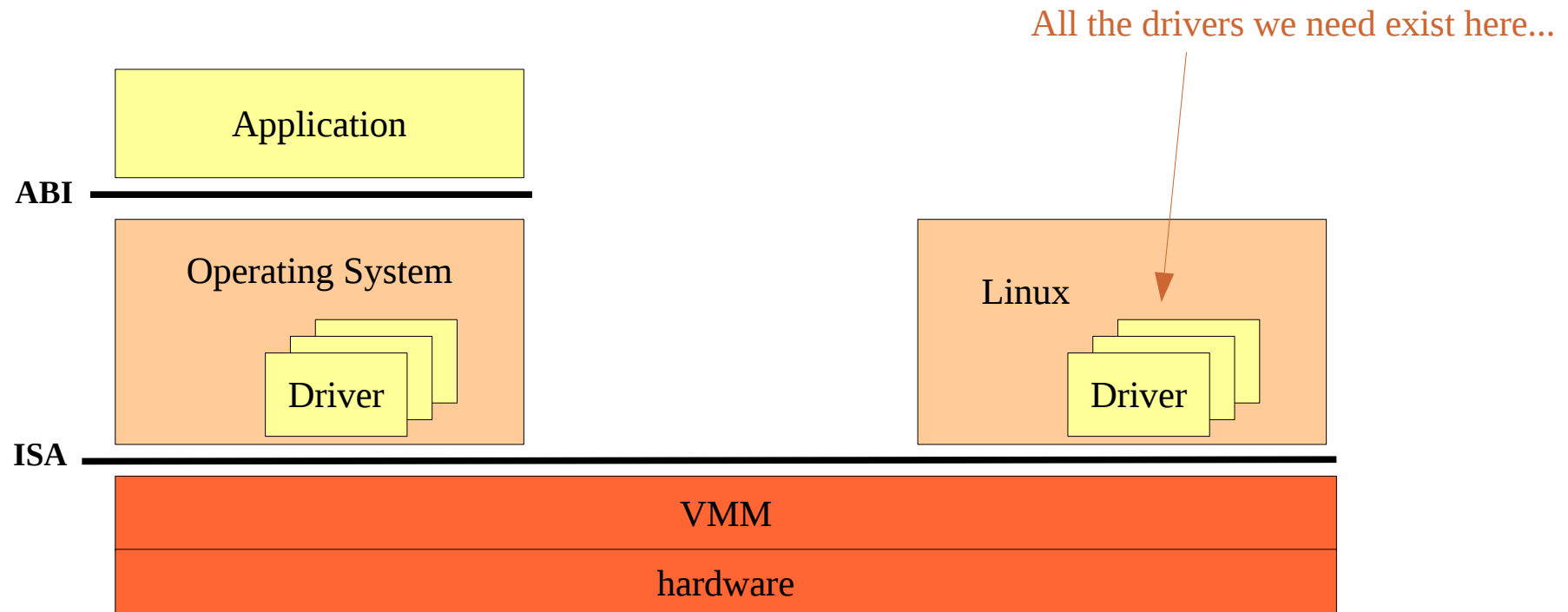


Your Mind at work...

66

- Type-I Hypervisors

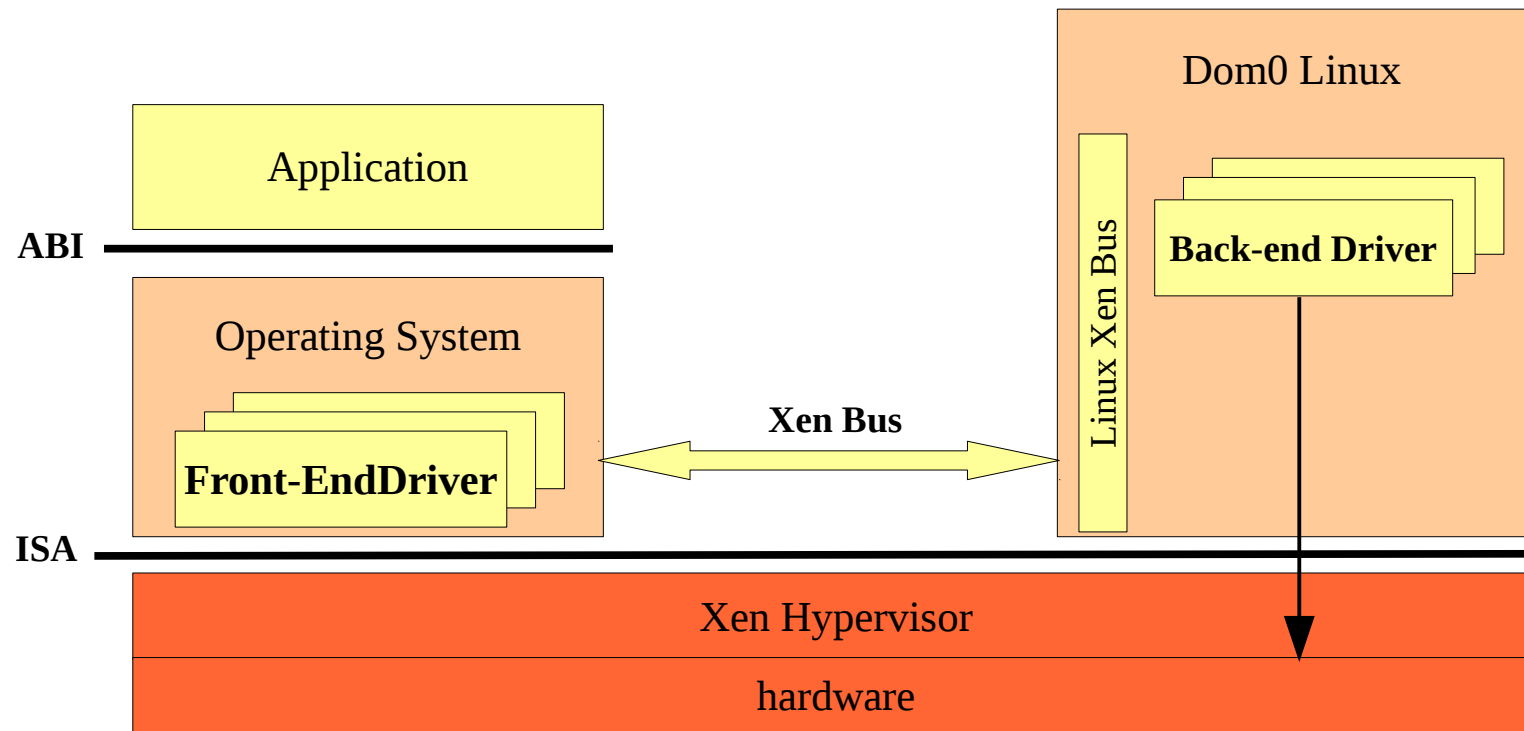
- The problem - Avoid having drivers inside the VMM
- The idea - How could existing linux drivers be leveraged?



Xen Bus – Overview

67

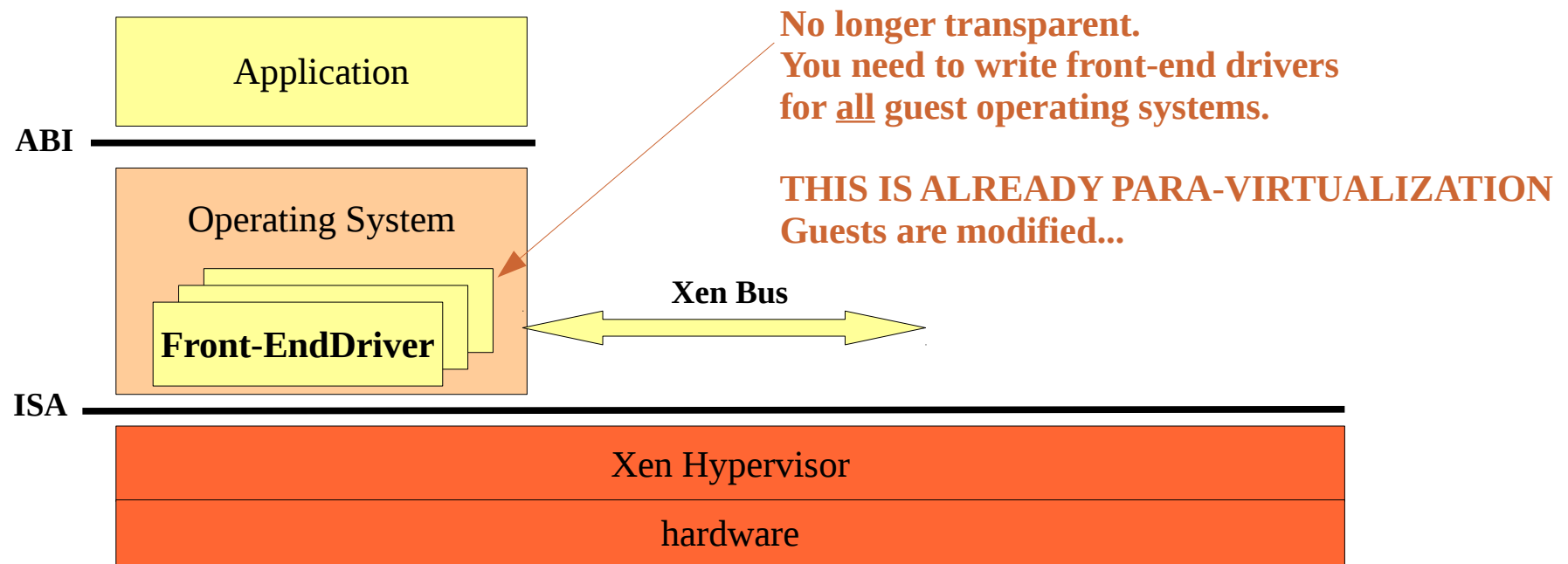
- Xen Bus – A split-driver model
 - Front-end drivers in the guest
 - Back-end drivers in other guests, **usually Dom0 linux**
 - **Back-end drivers are kernel modules**



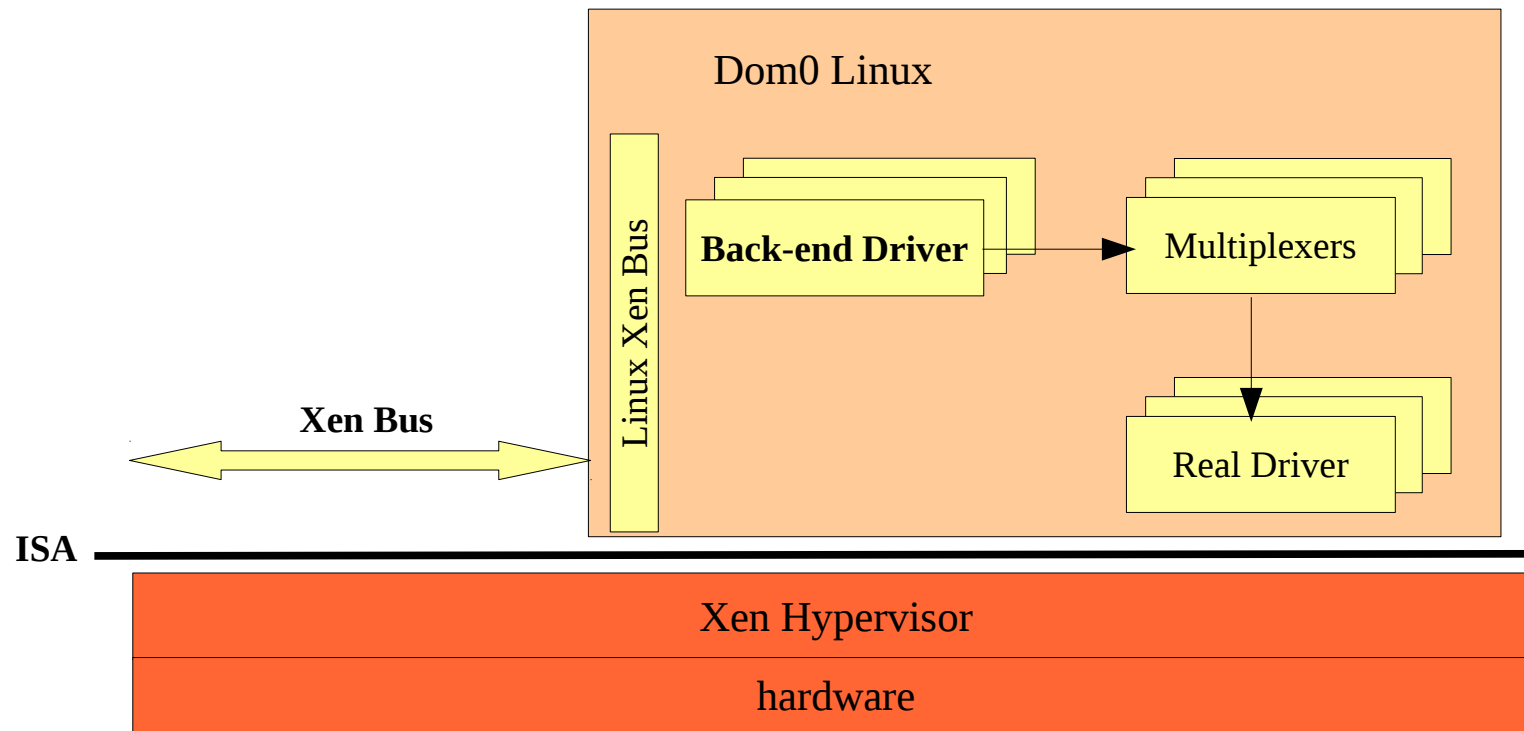
Xen Bus – DomU

68

- Xen Domain Unprivileged (DomU)
 - For the guest OS, front-end drivers are regular drivers
 - Front-end drivers work over the Xen Bus
 - Sending/receiving messages via shared memory

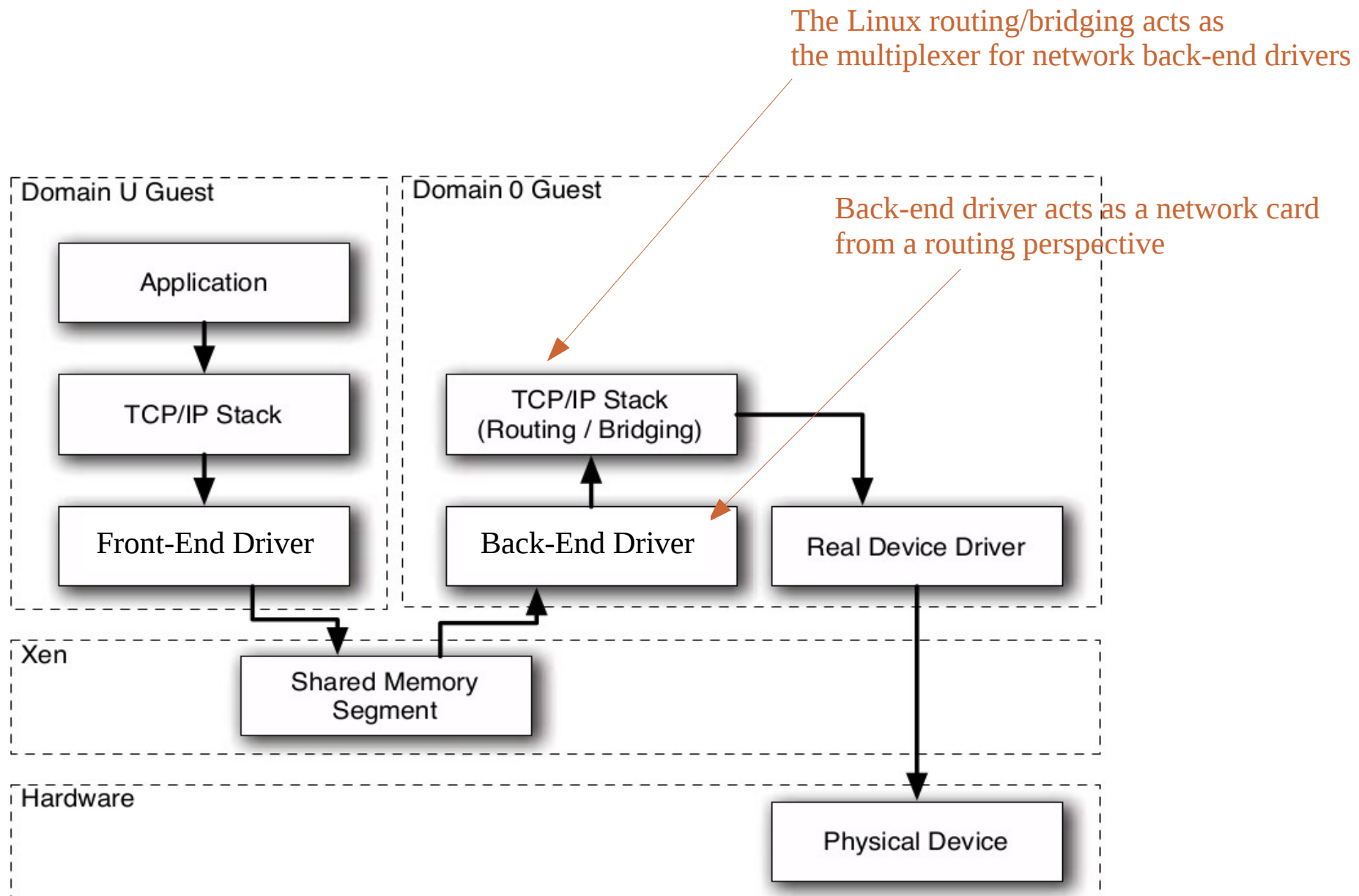


- Xen Dom0 framework
 - Back-end drivers attached to the Xen bus (a linux bus)
 - Multiplexer components (multiplexing virtual devices on real devices)
 - Linux device drivers drive real devices (no emulation)



Xen Bus - Dom0

70



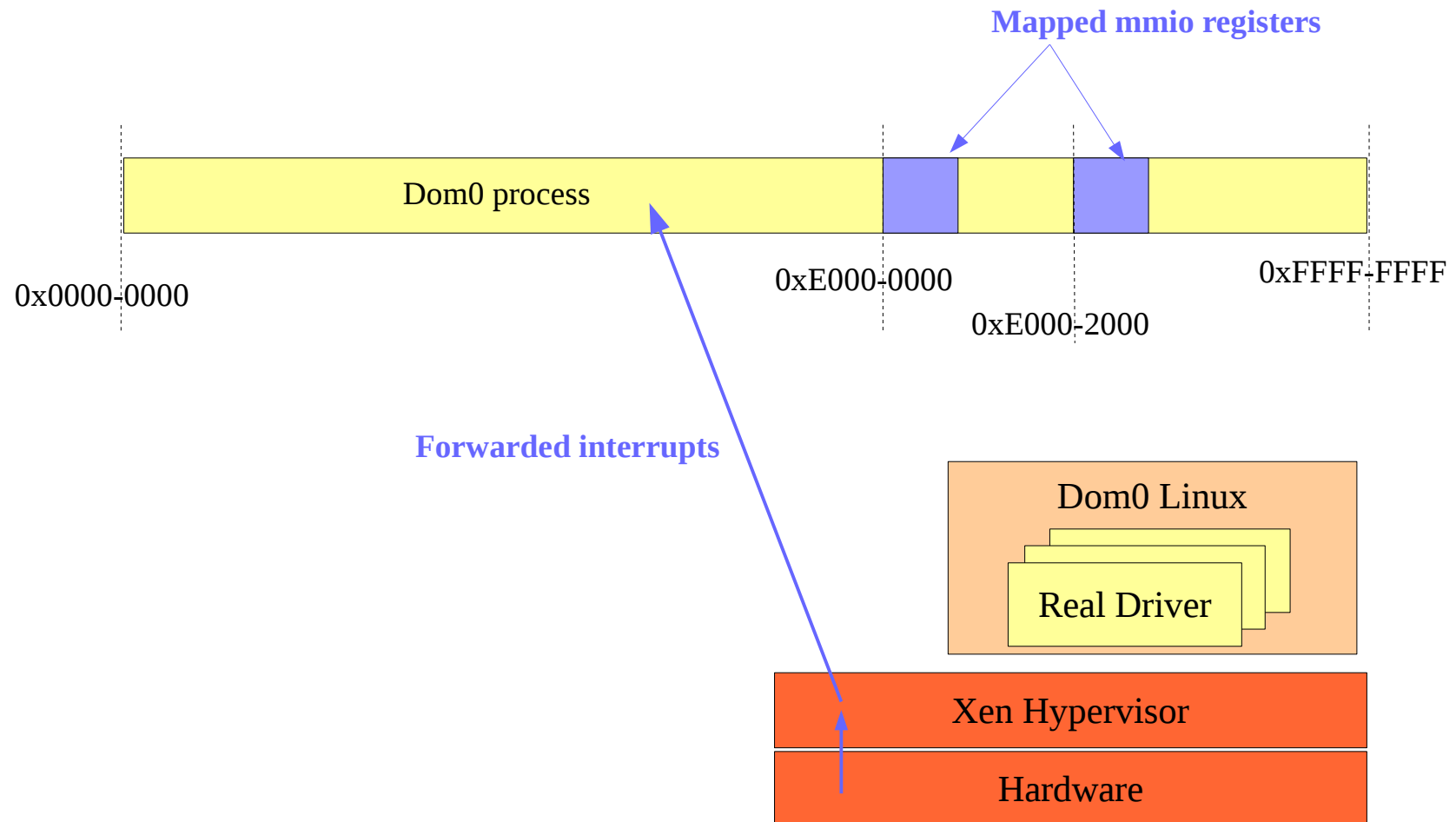
- Xen Bus – Generic Driver Classes
 - **Generic block devices**
 - Disk: read/write blocks at an offset
 - Multiplex on the file system (one file per disk)
 - NIC: send/receive blocks
 - Multiplex through network routing/bridging
 - **Generic character devices**
 - Serial lines
 - Used for the boot console
 - Multiplexed in software (no real hardware serial lines)
 - Multiplexed on sockets for remote consoles to domains

- Xen Bus – Generic Driver Classes
 - **Generic block devices**
 - **Generic character devices**
- Is that enough?
 - Xen is mostly used in the Cloud
 - Limited number of needed devices, mostly disks and network cards
 - Xen defines vanilla devices
 - No need for complex ioctls, supporting proprietary extensions
 - Desktop/Laptop challenge
 - Many more devices, very different
 - Example: video cards
 - Dom0 trick still works

Xen Bus – Dom0

73

- Dom0 linux sees the real hardware
 - Interrupts, mmio registers, and DMA



Xen Bus – Key Concepts

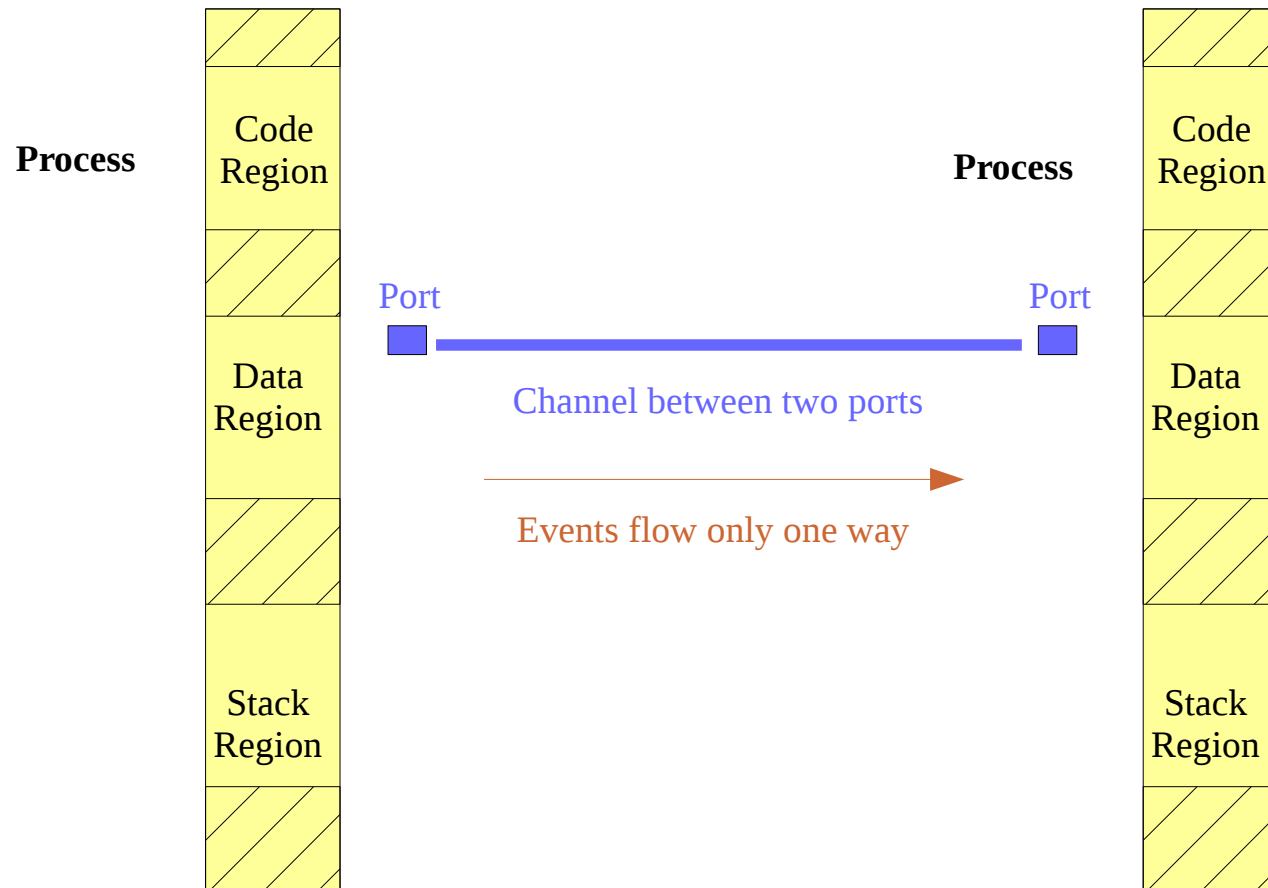
74

- Xen Bus – Events
 - Interrupt-like events for cross-domain notifications
 - Events can only be delivered through channels
- Xen Bus – Shared memory
 - Grant/Map mechanism for sharing pages across domains
 - Based on grant references (integer values) per domain
- Xen Bus – Xen Store
 - A broker between domains

Xen Bus - Events

75

- Xen Bus - Events
 - Xen events are like interrupts (one bit notification)
 - Xen events are delivered via channels



- Xen Bus - Events
 - Xen events are like interrupts (one bit notification)
 - Xen events are delivered via channels
- Steps to create a channel
 - Receiving guest creates a new, unbound port
 - Receiving guest advertises the existence of the port (typically via the Xen-Store)
 - The sending guest binds a local port to the remote one

Xen Bus – Events

77

Receiver Domain

```
int alloc_port(domid_t remote_domain, evtchn_port_t *port_pt) {
    evtchn_alloc_unbound_t op ;
    /* Setup the request */
    op . dom = DOMID_SELF ;
    op . remote_dom = remote_domain ;
    /* make the hypercall */
    if (HYPERVISOR_event_channel_op (EVTCHNOP_alloc_unbound, &op ) != 0)
        return -1;
    *port_pt = op . port;
    return 0;
}
```

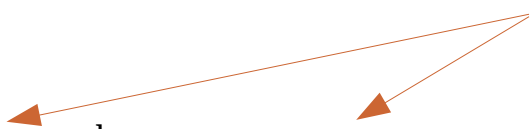
The sender domain that will be allowed to bind



Sender Domain

```
int bind_port(domid_t remote_domain, evtchn_port_t remote_port, evtchn_port_t *local_port_pt) {
    evtchn_bind_interdomain_t op ;
    /* Setup the request */
    op . remote_dom = remote_domain ;
    op . remote_port = remote_port;
    /* Make the hypercall */
    if (HYPERVISOR_event_channel_op ( EVTCHNOP_bind_interdomain, &op ) != 0 )
        return -1;
    *local_port_pt = op . local_port;
    return 0;
}
```

Obtained somehow, usually via the XenStore
The remote domain is the receiver domain



- Sending event through a bound local port

```
void signal_port(evtchn_port_t local_port) {  
    struct evtchn_send event ;  
    /* Setup the request */  
    event . port = local_port ;  
    /* Make the hypercall */  
    HYPervisor_event_channel_op ( EVTCHNOP_send , &event ) ;  
}
```

Just the local port of the channel
is needed to signal the other side.



- Event delivery to a handler
 - One handler for all events, the handler acts as a dispatcher
- Events are delivered asynchronously, like interrupts
 - So the handler can be called at any time
 - In between any two assembly instructions
- Event delivery can be masked, like for real interrupts
 - Events can be masked at a channel level or at VCPU level or at domain level
 - Undelivered events are remembered, like for real interrupts
 - Very similar to the hardware Programmable Interrupt Controller (PIC)

Xen Bus – Events

80

This code in the guest...

```
.event_handler:
    PUSH_REGS      ; save registers
    push %esp      ; push argument
    call event_dispatch
    add $4,%esp     ; pop argument
    POP_REGS       ; restore registers
    reti
```

Not really possible...
Not a hardware interrupt...

This code in the guest...

```
void event_dispatch(struct regs *regs) {
    unsigned int pending_selector;
    vcpu_info_t * vcpu = &shared_info.vcpu_info[0] ;
    /* Grab pending events and reset to receive next ones */
    vcpu->evtchnup_call_pending = 0 ;
    pending_selector = atomic_xchg (&vcpu->evtchn_pending_sel , 0 ) ;
    /* Dispatch events */
    while ( pending_selector != 0 ) {
        /* Each bit set to one is one pending event */
        evt_chn_port_t port = TRANSLATE(pending_selector);
        handlers[port](port,regs);
    }
    return;
}
```

**Guest table of event handlers,
indexed by local ports;
local ports that were exported through
the Xen Store.**

Xen Bus – Events

81

```
.event_handler:  
    PUSH_REGS  
    push %esp  
    call event_dispatch  
    add $4,% esp  
    POP_REGS  
    reti
```

```
.event_handler:  
    PUSH_REGS  
    push %esp  
    call event_dispatch  
    add $4,% esp  
    POP_REGS  
    HYPervisor_ret
```

Why use **reti** with hardware interrupts?

- return to the interrupted execution
- re-enable interrupts
- restore the processor mode (user mode in many cases)

Problem:

- not an hardware interrupt

First solution:

- Emulate if **reti** instruction traps in user mode

Second solution:

- Provide an **hypercall** to replace the **reti** instruction
 - Re-enable events
 - Return to the interrupted execution


Note:

But both are expensive in terms of performance
Many Xen bus events → Many **reti**

Xen promotes a faster and more complex approach based on re-entrant handlers. It is based on highly platform-specific assembly code.

Xen Bus – Key Concepts

82

- Xen Bus – Events ✓
 - Interrupt-like events for cross-domain notifications
 - Events can only be delivered through channels
- Xen Bus – Shared memory ←
 - Grant/Map mechanism for sharing pages across domains
 - Based on grant references (integer values) per domain
- Xen Bus – Xen Store
 - A broker between domains

- Grant/Map mechanism for sharing pages across domains
 - **Mostly used as circular buffers**
 - To send/receive requests and completion status
 - To send/receive data such as blocks for block devices
- A notion of grant table per domain
 - A domain grants pages for other domains to map
- A granted page is identified by a domid and reference
 - The domid is the identifier of the domain that granted the page
 - The reference is the index of the page in the grant table

Xen Bus – Shared Memory

84

- Granting a page

```
#include <public/xen.h>
extern grant_entry_t * grant_table; ← One such table per guest...
```

```
void grant_page (void* shared_page) {
```

```
    uint16_t flags ;
    gnttab_setup_table_t setup_op ;
```

```
    /* Setup the grant table request */
    setup_op . dom = DOMID_SELF ;
    setup_op . nr_frames = 1;
    setup_op . frame_list = grant_table ;
```

```
    /* Make the hypercall */
    HYPERVISOR_grant_table_op ( GNTTABOP_setup_table , &setup_op ,1) ;
```

```
    /* Fill up the grant table */
    grant_table[0] . domid = DOMID_FRIEND ;
    grant_table[0] . frame = shared_page >> 12 ;
```

```
    memory_write_barrier();
    grant_table[0] . flags = GTF_permit_access & GTF_reading & GTF_writing ;
}
```

Note: x86 has ordered writes
so only a compiler WB
is necessary

Filled the flags last
→ validate the grant table
→ requires a memory Write Barrier

Xen Bus – Shared Memory

85

- Mapping a page

```
#include <public/xen.h>
```

```
int map_page(domid_t domid, grant_ref_t gref, void* shared_page,  
             grant_handle_t * handle_pt) {
```

```
    gnttab_map_grant_ref_t map_op;
```

```
    /* Setup the map request */
```

```
    map_op . host_addr = shared_page;
```

```
    map_op . flags = GNTMAP_host_map ;
```

```
    map_op . dom = domid ;
```

```
    map_op . ref = gref ;
```

← The local address where to map the shared page

← Identifies the granted page (domid_t, grant_ref_t)

```
    /* Make the hypercall */
```

```
    HYPERVISOR_grant_table_op ( GNTTABOP_map_grant_ref , &op , 1 ) ;
```

```
    if (map_op . status != GNTST_okay )
```

```
        return -1; /* something went wrong */
```

```
    *handle_pt = map_op . handle;
```

```
    return 0;
```

```
}
```

Xen Bus – Key Concepts

86

- Xen Bus – Events ✓
 - Interrupt-like events for cross-domain notifications
 - Events can only be delivered through channels
- Xen Bus – Shared memory ✓
 - Grant/Map mechanism for sharing pages across domains
 - Based on grant references (integer values) per domain
- Xen Bus – Xen Store ←
 - A broker between domains

- Xen Store is a simple hierarchical storage system
 - Hierarchy of directories
 - Directories contain <key,value> pairs
 - Key and values are textual strings
- Two main usages
 - Export administrative information about running guests
 - Broker <domain,port> and <domain,grant> pairs

- The Xen Store – A regular device
 - A regular device, shared amongst all domains
 - A regular back-end device, provided by dom0
 - So each DomU must have at least the front-end driver for the store
 - All domains have one shared page with the store on dom0
 - The page contains two rings to carry the communication with the store
 - Bootstrap
 - Each domain has a default info page mapped by the Xen hypervisor
 - The info page contains all sorts of bootstrap information
 - A replacement for the kernel args provided by boot loaders
 - The info page contains the necessary info setting up the font driver
 - A <domain,port> pair for events
 - A <domain,grant> for the shared page

Xen Bus – Key Concepts

89

- Xen Bus – Events ✓
 - Interrupt-like events for cross-domain notifications
 - Events can only be delivered through channels
- Xen Bus – Shared memory ✓
 - Grant/Map mechanism for sharing pages across domains
 - Based on grant references (integer values) per domain
- Xen Bus – Xen Store ✓
 - A broker between domains

Emulation Challenge

90

- Emulate processors ✓
- Emulate memory, including the MMU ✓
- Emulate devices, including the interrupt controller ✓

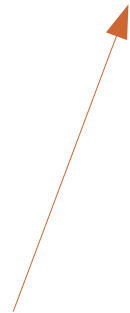
So we are done 😊

But let's step back for a moment...

Stepping Back

91

- Transparent virtualization
 - Unmodified guests - the original goal
 - Achieved, but plagued by the need for VMM drivers
- Para-virtualization was introduced
 - To leverage Linux drivers, through split drivers
 - To reduce the overhead of emulation traps



With unmodified guests, all sensitive instructions trap.

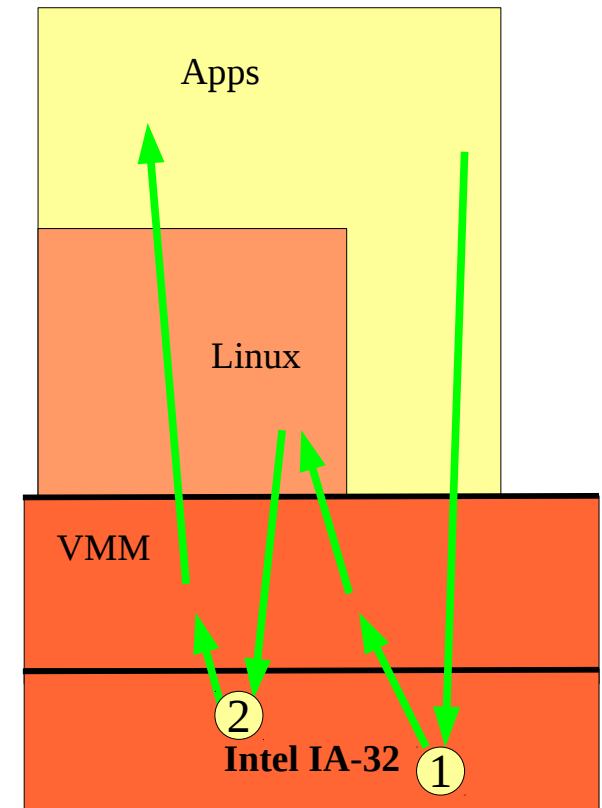
→ this means a lot of traps and traps are expensive on modern processors.

Stepping Back

92

- Page Fault Trap - Type-I hypervisors
 - 2 mode switch (kernel-user)
 - 2 hardware traps
 - The actual hardware trap ①
 - The emulation of the RETI ②

Trap-based emulation is not free!

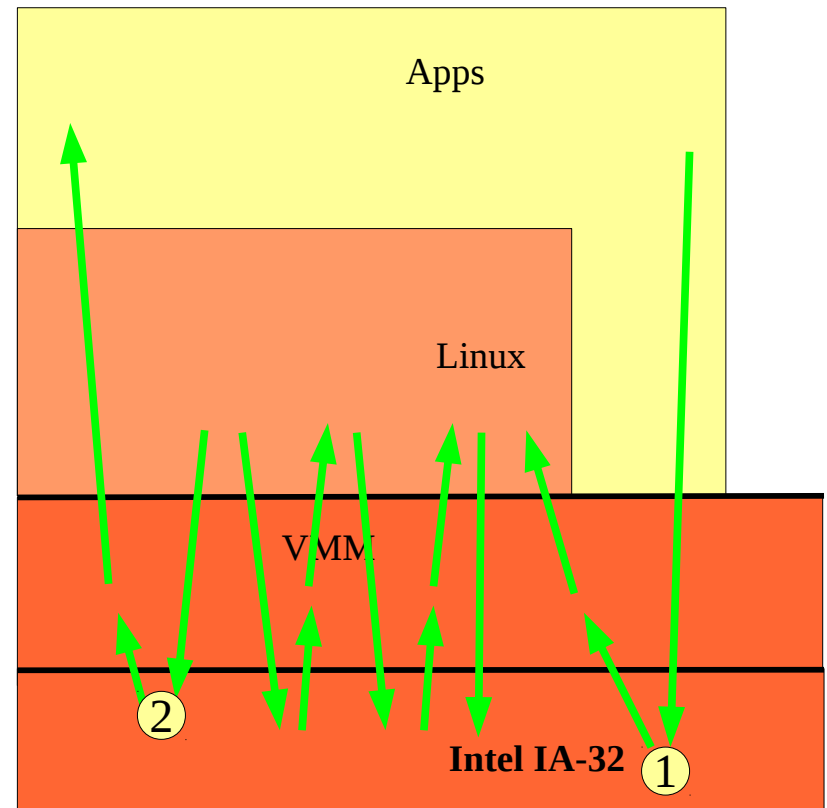


Stepping Back

93

- Page Fault Trap - Type-I hypervisors
 - 2 mode switch (kernel-user)
 - 2 hardware traps
 - The actual hardware trap ①
 - The emulation of the RETI ②
 - Plus the emulation of page entries
 - More traps...
 - Manipulating the page table

Trap-based emulation is not free!

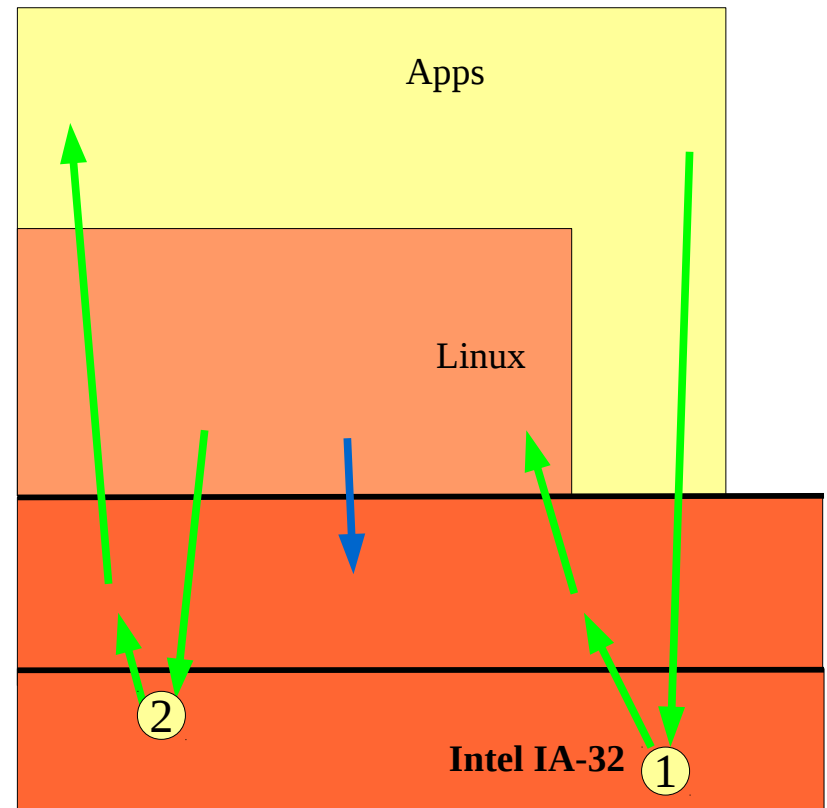


Stepping Back

94

- Page Fault Trap - Type-I hypervisors
 - 2 mode switch (kernel-user)
 - 2 hardware traps
 - The actual hardware trap ①
 - The emulation of the RETI ②
 - Use **hypercalls** in the handler
 - Works like regular syscalls
 - But to the underlying VMM
 - **Grouping sensitive operations**
 - Example
 - Manipulating the page table

Hypercalls reduce the number of traps
Applied to larger operations

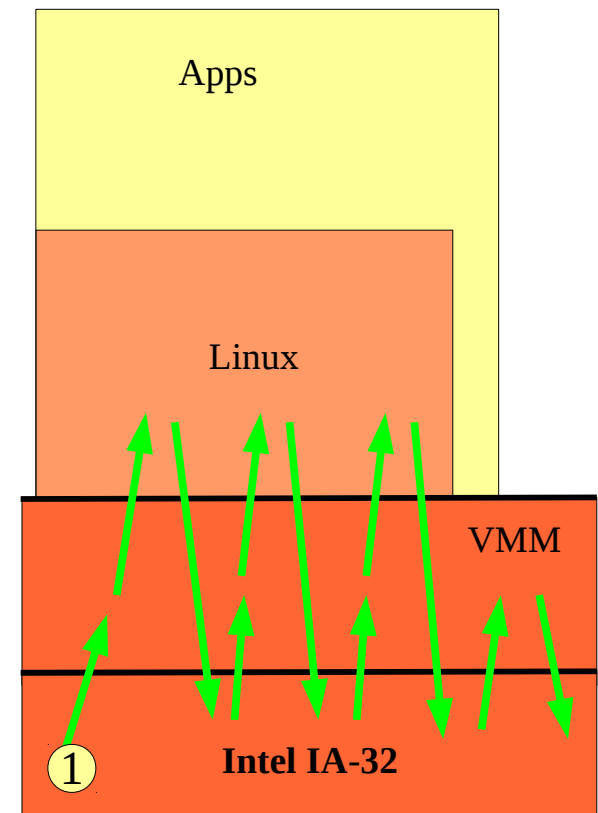


Stepping Back

95

- Context Switch - Type-I hypervisors
 - Starts with a timer interrupt ①
 - But the handler uses many sensitive instructions...
 - Change page table
 - Flush TLB
 - Set timer
 - Re-enable interrupts

Trap-based emulation is definitely not free!

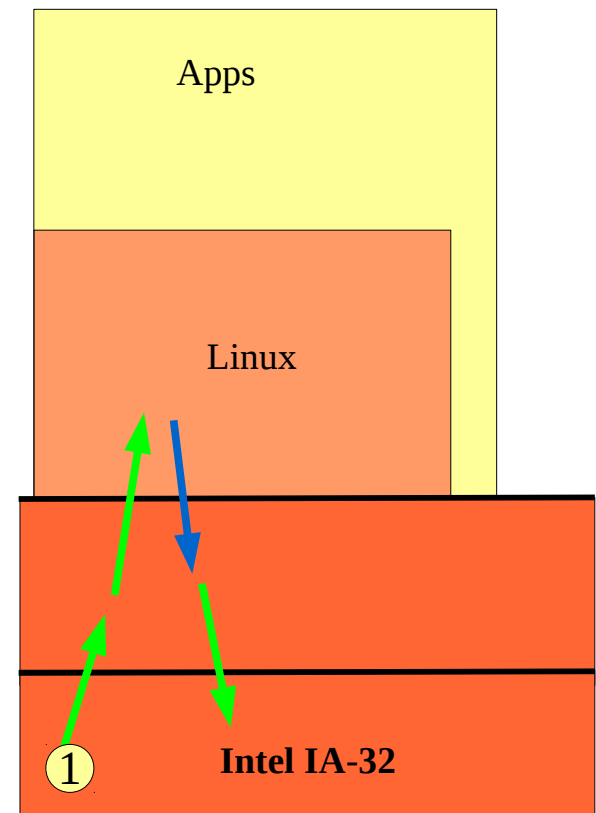


Stepping Back

96

- Context Switch - Type-I hypervisors
 - Starts with a timer interrupt ①
 - But the handler uses many sensitive instructions...
 - Change page table
 - Flush TLB
 - Set timer
 - Re-enable interrupts
 - Efficiently replaced by a **hypercall**
 - Only one hypercall is needed
 - Asking the VMM to switch to a process
 - Essentially, a shadow page table and registers

Using hypercalls is an efficient optimization...



- Towards better hardware support
 - Introduce new hardware mechanisms
 - As usual, helping with costly software mechanisms
- Example - extended page tables
 - Two page tables in the MMU
 - One for mapping virtual to real
 - One for mapping real to virtual
 - Upon a TLB miss
 - The hardware walks both page tables
 - Translating virtual to real
 - Then translating real to physical
 - No need for page table emulation
 - The guest OS manipulates the virtual-to-real mapping
 - The VMM manipulates the real-to-physical mapping

- Let's discuss para-virtualization
 - Overall, provides visible system calls (hypercalls) to guest VMs
 - Goal is to improve performance through cooperation
 - Today, most virtualized systems are para-virtualized

But then, what is the difference between a para-virtualized guest using the Xen bus for using remote services on the one hand...

and on the other hand, a micro-kernel where personalities use user-level services through IPC messaging...

What do you think?

Stepping Back

99

- Let's discuss hardware-assisted virtualization
 - A major hardware evolution
 - Like it was the case for the MMU and cache lines
 - Or hardware threads
 - All sensitive operations are designed to trap in user mode
 - Introducing new hardware mechanisms
 - But requires to virtualize the same ISA (virtual and real)

So what should we do if we need to emulate an IA-32 on a ARM?

Or you are running on an older hardware, without virtualization support?

- Handle problematic ISA
 - Not all ISA support efficient virtualization
 - Condition: if all sensitive instructions trap in user-mode
 - Rationale: so that we can emulate the stand-alone behavior
 - Efficient virtualization requirements
 - **ReqA:** Instructions that attempt to change or reference the mode of the VM or the state of the real machine
 - **ReqB:** Instructions that read or change sensitive registers and/or memory locations such as a clock register and interrupt registers
 - **ReqC:** Instructions that reference the storage protection system, memory system, or address relocation system.
 - Intel IA-32
 - 17 instructions are sensitive and not privileged

- Intel IA-32
 - Segment memory
 - Model
 - Two tables
 - Global Descriptor Table (GDT)
 - Local Descriptor Table (LDT)
 - Both contains segment descriptors that provide base address, access rights, type, length, and usage information
 - All memory accesses pass through these tables when the processor is in protected mode
 - GDTR and LDTR are two registers that contains the physical addresses and sizes for their respective table
 - Problematic instructions
 - SGDT and SLDT instructions store either the GDTR or LDTR registers in memory
 - LAR loads access rights from a segment descriptor
 - LSL loads the segment limit
 - VERR and VERW checks if a segment is readable or writable

- Handle problematic ISA
 - Interrupts and traps
 - Model
 - Interrupt Descriptor Table (IDT)
 - Holds gate descriptors that provide access to interrupt and trap handlers
 - IDTR register holds the physical address and size for the IDT
 - Problematic instructions
 - Unprivileged SIDT instruction stores the IDTR in memory
 - Privileged write instruction to the SIDT registers

- Handle problematic ISA
 - Machine Status Word
 - Bit 0 to 5 holds system flags controlling the operation mode and state of the processor
 - 0: Protection Enabled
 - 1: Monitor Coprocessor
 - 2: Emulation or not for floating-points
 - 3: Task Switched allows delayed saving of the floating point unit context on a task switch until the unit is accessed by the new task
 - 4: Extension Type signals the presence of a special Intel co-processor
 - 5 Numeric Error: controls the FPU error reporting
 - Instruction SMSW
 - Store Machine Status Word (SMSW) into a register or memory
 - It is sensitive and unprivileged
 - Example the **P**rotection**E**nabled flag that can be observed by a guest OS
 - Only provided for backward compatibility with Intel 286
 - From Intel 386, supposed to use a MOV privileged instruction to load and store control registers

- Handle problematic ISA
 - EFLAGS register
 - Holds flags that control the operation mode and state of the processor
 - Interrupt masking
 - I/O privilege level
 - Interrupt pending flag
 - Representative of the processor mode
 - Guest VM will expect to be able to change these and read them
 - Problematic instructions (POPF and PUSHF)
 - Pushes and pops from the stack the EFLAGS register
 - PUSH instruction
 - Pushes on the stack any register, including CS and SS
 - Both hold the Current Processor Level (CPL)
 - Would allow a guest VM to examine and realize that the CPL is not 0 but 3

- Handle problematic ISA
 - CALL, JMP, INT n, and RET instructions
 - Discussing CALL
 - Far and near calls to the same privilege level
 - Far call to a different level or task switch
 - Behavior thus depends on CPL
 - A task uses a different stack for every privilege level
 - So a guest OS will expect a stack switch
 - Although in reality caller and callee on in CPL 3
 - Discussing RET
 - RET can be used for near, far and inter-privilege returns
 - Clears certain segment registers (DS,ES,FS and GS) on inter-privilege returns towards lower-levels
 - Similar issues with other instructions

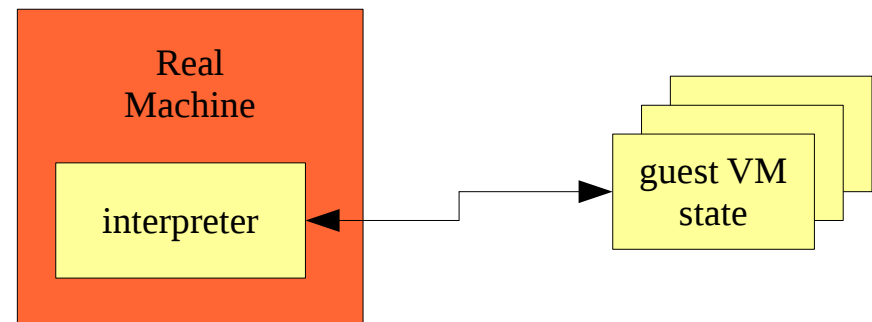
- Handle problematic ISA
 - MOV instruction
 - Load and store registers
 - Problems
 - On CS and SS registers, allows to read the CPL
 - Loading CS will trap
 - Loading SS is problematic for guest OSes running in CPL 3

Is all hope lost?

Emulating Instructions

107

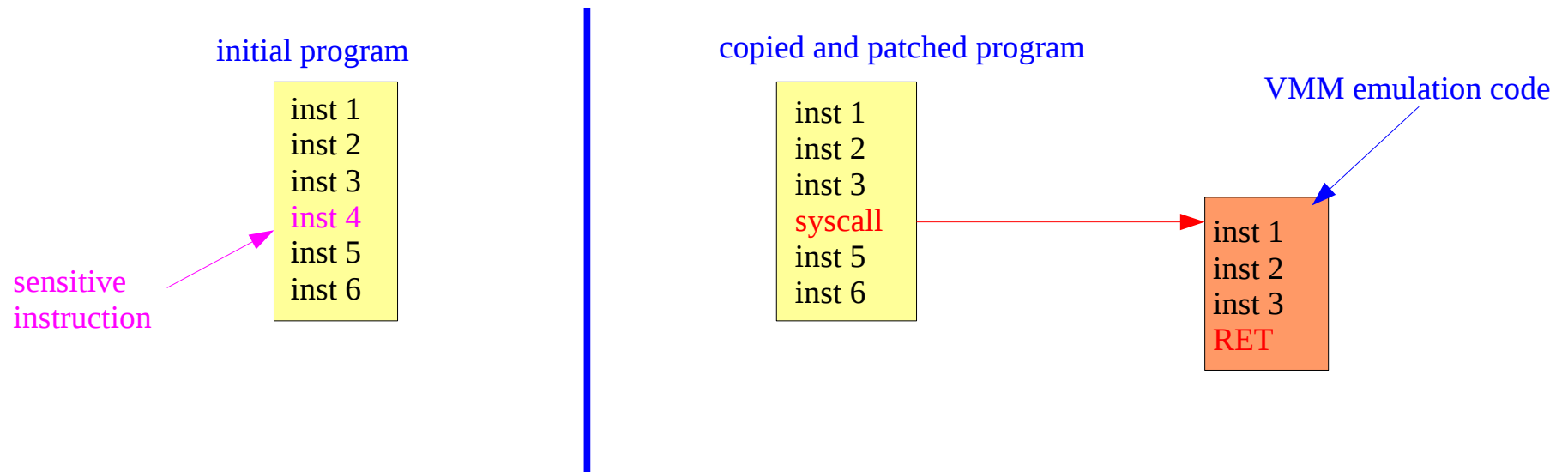
- Interpretation is always possible
 - Fetch, decode and interpret assembly instructions
 - Essentially an interpreter for assembly codes
 - State management
 - Use a state block per guest VM
 - Use an indirection pointer
 - Switch pointers when switching guest VMs
- Regarded as inefficient
 - Interpreting instructions is slow
 - For example, register moves are now in fact memory moves



Dynamic Binary Translation

108

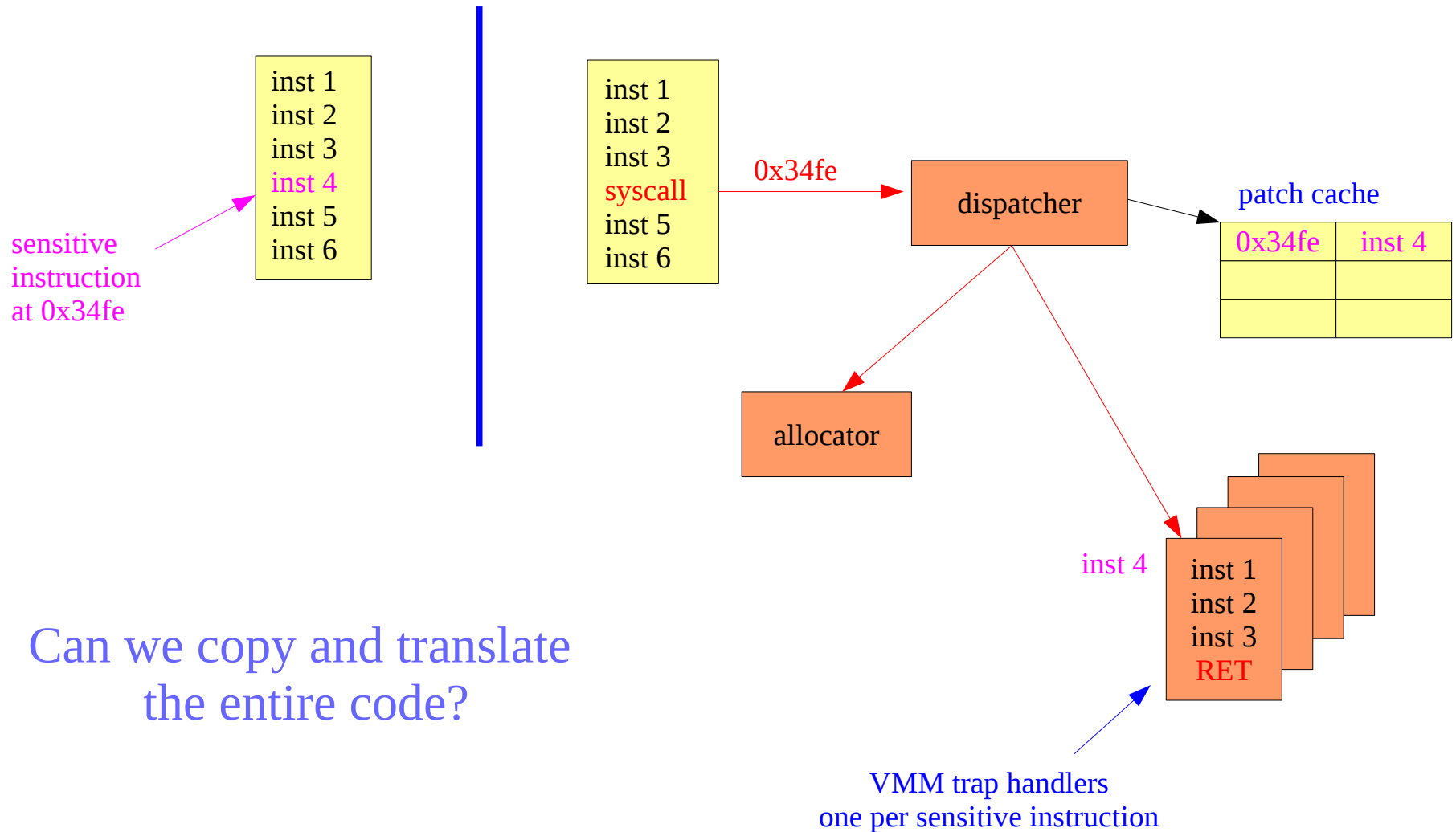
- Binary translation is faster
 - Same instruction set
 - We copy the code, patching sensitive non-privileged instructions
 - Different instruction set
 - We copy and translate the code
 - Still patching sensitive non-privileged instructions



Dynamic Binary Translation

109

- More complete picture of binary translation



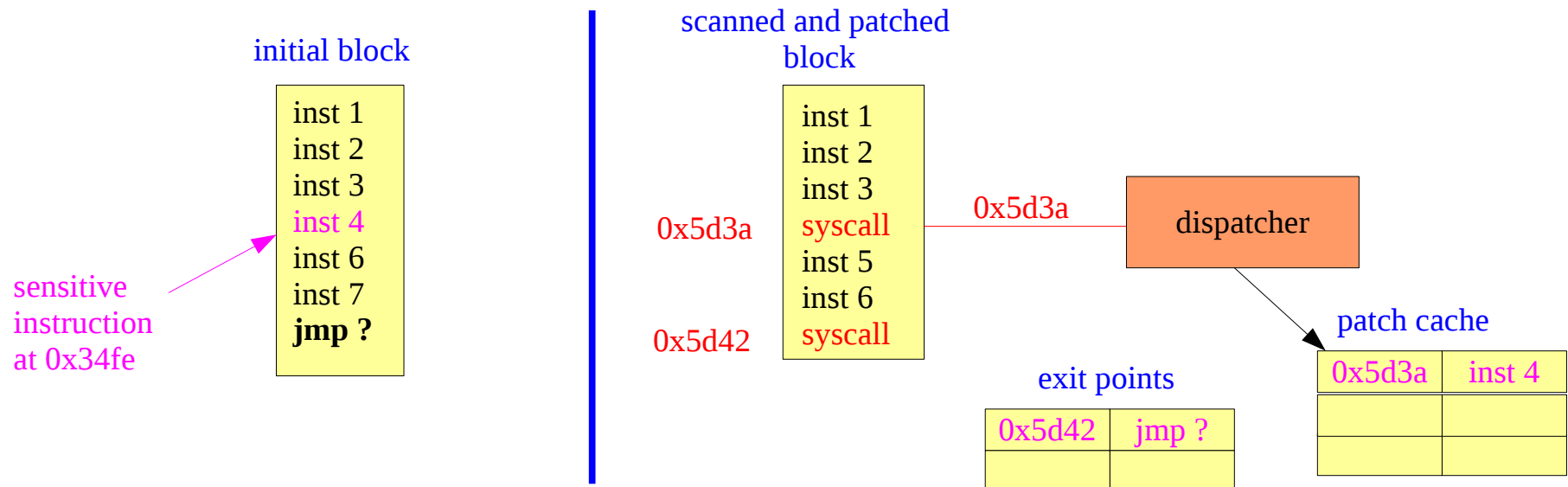
- Can we copy and patch/translate the entire code?
 - The code may be fairly large
 - We may only know a few entry points (sometimes only one)
 - Dynamically loaded code
 - Like shared libraries or kernel modules
 - Code is not known statically
- Problems
 - Not all branch instructions are known statically
 - Addresses can be computed
 - Example: a jump through a function pointer table
 - Self-modifying code
 - This happens more than you think
 - High-level VMs typically do that all the time as optimizations

- Why do we copy code?
 - Replaced instructions may be smaller than the syscall instruction
 - Especially true in CISC
 - So we can't do it in place
 - The emulated and real ISA might be different
 - ARM-32 on IA-32
 - This is also necessary for self-inspecting and self-modifying code
 - We need to protect code pages (that have been translated)
 - We need a translation map between source and target block addresses
 - Upon self-modification, we need to invalidate the corresponding code blocks

Dynamic Binary Translation

112

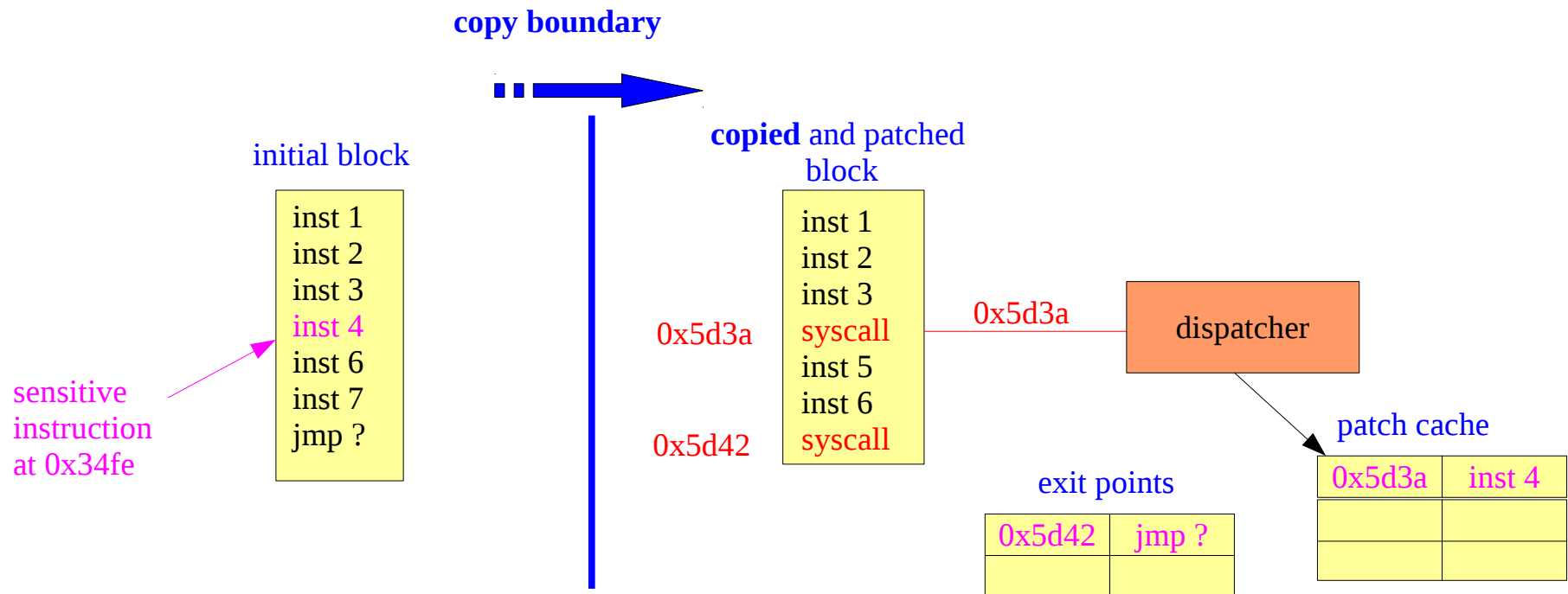
- Introducing a dynamic code cache
 - We know the entry point
 - We can copy and patch/translate basic code blocks
- Key points
 - Each exit point in a code block becomes a trap to the VMM
 - **Execution only happens within the cache**



Dynamic Binary Translation

113

- Dynamic code cache
 - Only retains the most recently used patched code blocks
 - Implements a replacement policy
 - On cache miss, we translate the code block at the target address
 - Self-modifying code → invalidate corresponding code blocks



- Static versus dynamic code blocks
 - Static code blocks represent the static control flow
 - Each block is a sequence with a single entry point and a single exit point
 - A block begins and ends at all branch or jump instructions
 - A block begins and ends at all branch or jump targets

add	
load	block 1
store	

loop: load	
add	
store	block 2
brcond skip	

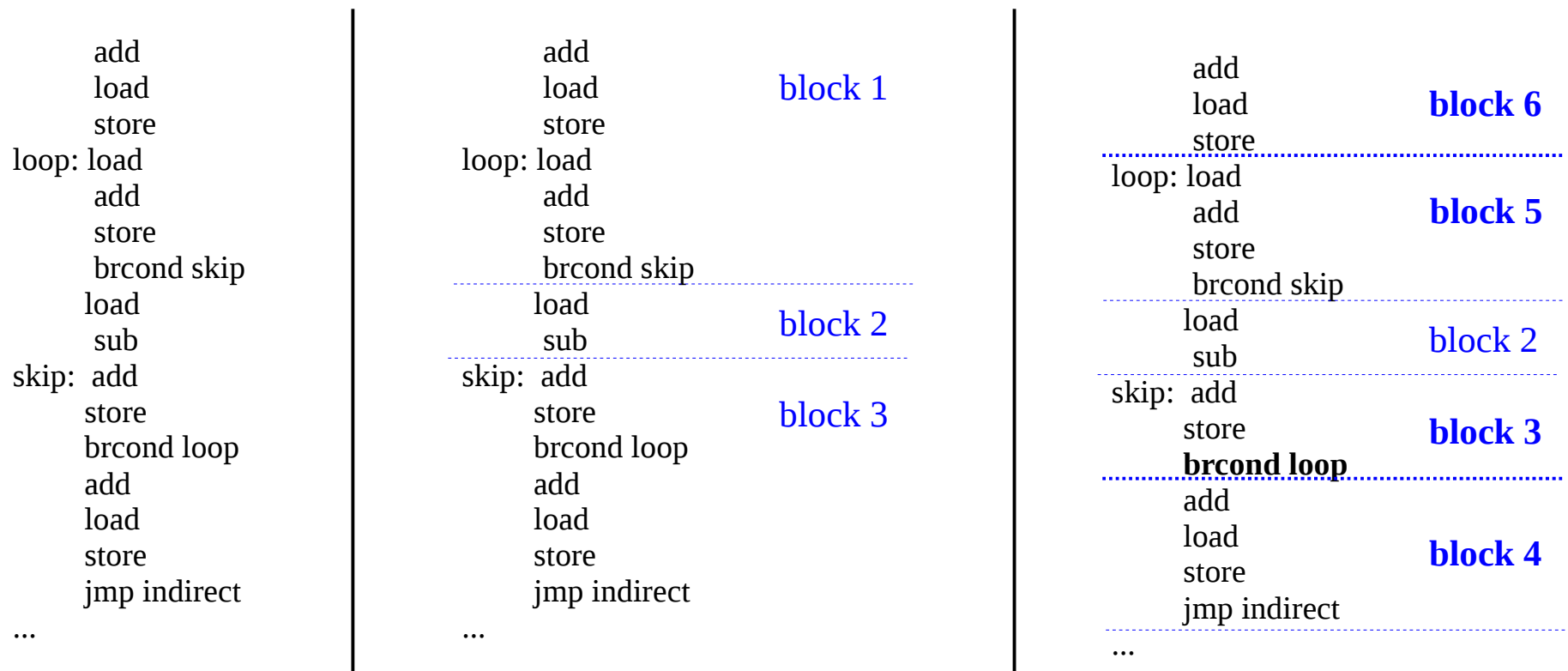
load	
sub	block 3

skip: add	
store	block 4
brcond loop	

add	
load	
store	block 5
jmp indirect	

...	

- Producing static code blocks require stronger code flow analysis
 - Must handle backward branch instructions
 - Requires splitting blocks
 - Requires updating source to target address maps
 - Which is more costly if syscalls introduce address shifts



- Since we have a code cache
 - We seek fast translation
 - We prefer dynamic code blocks
- Dynamic code blocks
 - Begins at the instruction executed after immediately after a branch or a jump
 - Follows the execution stream
 - Ends with the next branch or jump

Dynamic Binary Translation

117

- Dynamic code blocks

```
    add
    load                                block 1
    store
-----
loop: load
    add                                block 2
    store
    brcond skip
-----
    load                                block 3
    sub
-----
skip: add
    store                                block 4
    brcond loop
-----
    add
    load                                block 5
    store
    jmp indirect
-----
...
```

```
    add
    load
    store                                block 1
loop: load
    add
    store
    brcond skip
```

```
    load                                block 2
    sub
skip: add
    store
    brcond loop
```

```
loop: load                                block 3
    add
    store
    brcond skip
```

```
skip: add                                block 4
    store
    brcond loop
```

- Dynamic code blocks
 - Slightly larger than static code blocks
 - Hence a footprint overhead (possibly redundant instructions in the cache)
 - But present more opportunities for optimization
 - Faster to generate
 - Just parse instruction streams to next branch
 - Never split existing dynamic blocks on backward branches
 - Important to be faster
 - Produce a dynamic block on a branch miss in the code cache
- Remember
 - A translation can occur
 - For example from ARM to IA-32 assembly

- Book chapters/sections on main concepts of VMM
 - J. Smith, R. Nair, Virtual Machines: Versatile Platforms for systems and processes. Morgan Kaufmannn, 2005. Chapter 8.
 - A. Tanenbaum. Modern operating systems (3rd edition). Pearson education, 2007. Section 8.3.
- VMM optimizations for memory management
 - Carl A. Waldspurger. Memory resource management in VMware ESX server. In *Proc. 5th Symposium on Operating Systems Design and Implementations*, Boston, MA, USA, 2002.
- X86 virtualization issues
 - J. S. Robin and C. E. Irvine, Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor, In *Proc. 9th Usenix Security Symposium*, 2000.

References (2/2)

- I/O virtualization
 - Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the 2001 Usenix Annual Technical Conference*, Boston, MA, June 2001.
- Paravirtualization
 - Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the Fifth Symposium on Operating System Design and Implementation (OSDI 2002)*, Boston, MA, December 2002.
 - Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proc. 19th Symp. on Operating Systems Principles*, pages 164–177, Bolton Landing, NY, USA, October 2003
- Binary translation versus hardware-assisted virtualization
 - Keith Adams and Ole Agesen. A comparison of the software and hardware techniques for x86 virtualization. In *Proc. 12th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, USA, October 2006.