Virtual Machine Monitors

Olivier Gruber, Ph.D.

Full-time Professor – Université Joseph Fourier

Laboratoire d'Informatique de Grenoble

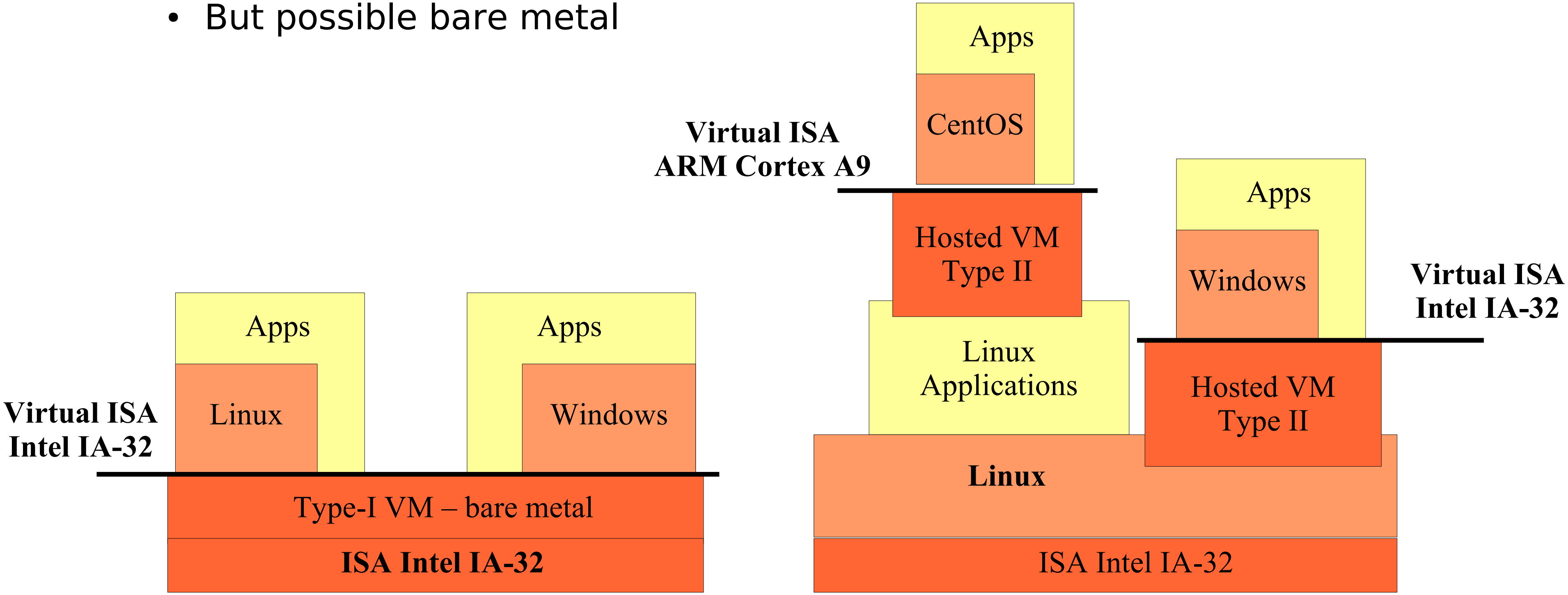
Olivier.Gruber@imag.fr

Outline

- 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)

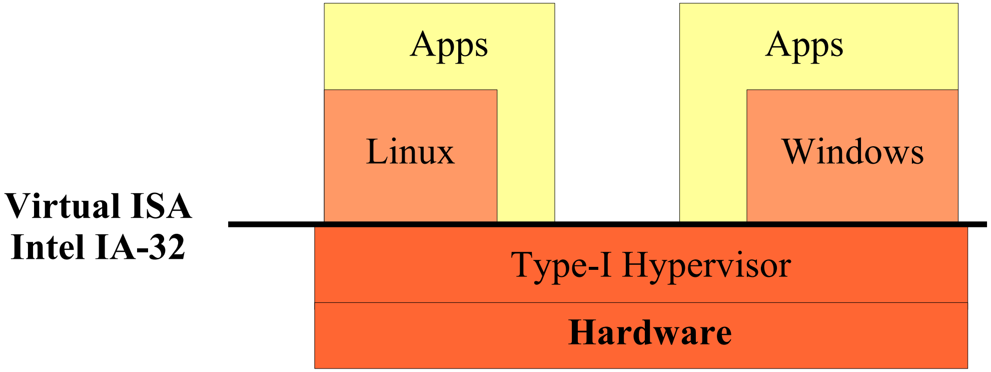
Virtual Machine Monitors

- 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



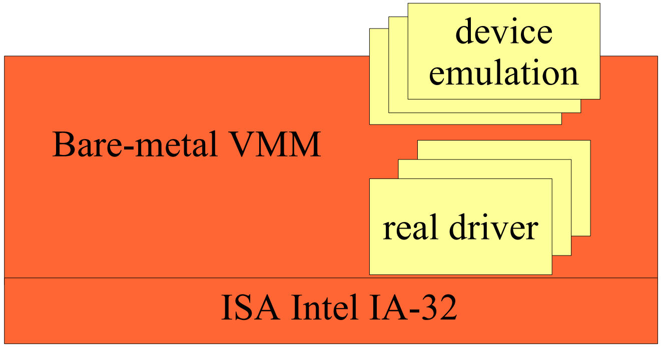
Type-I Virtual Machines

- 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



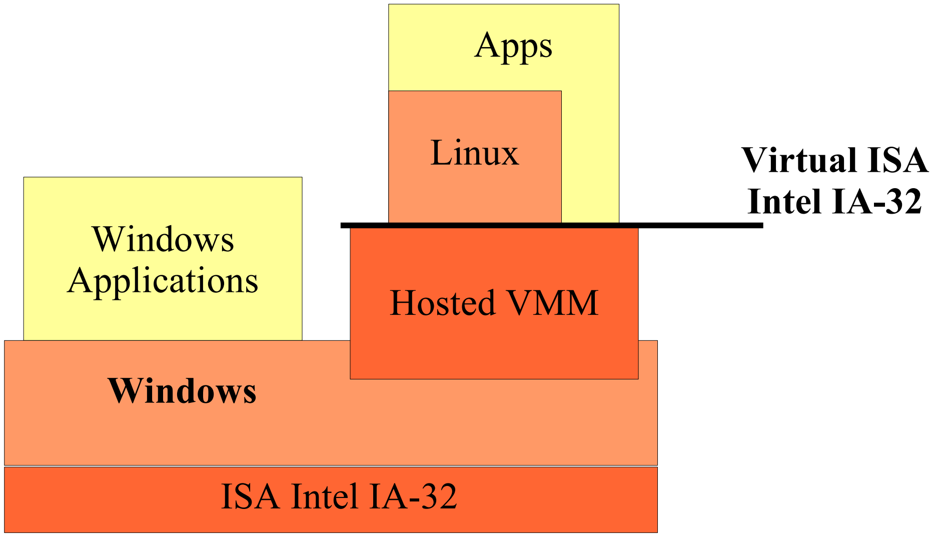
Type-I Virtual Machines

- 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



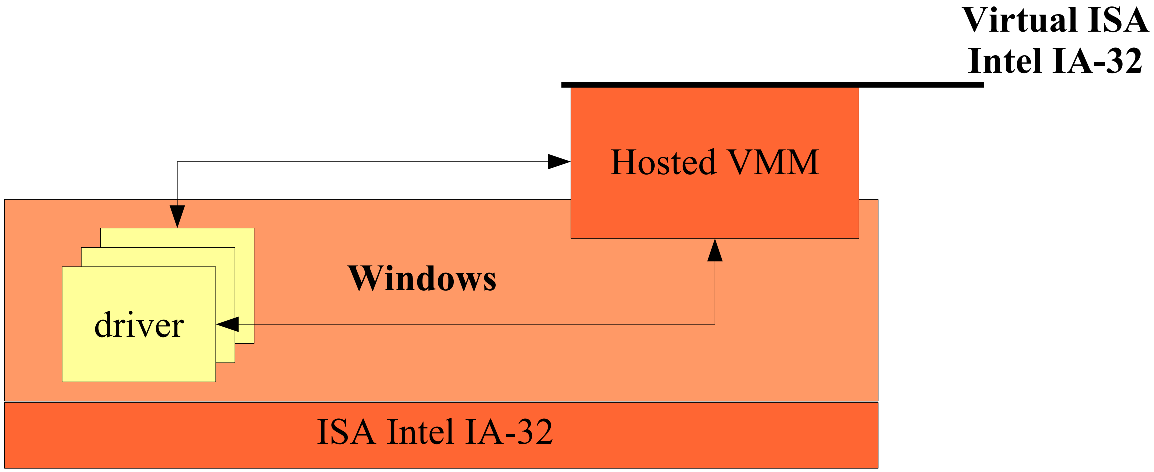
Type-II Virtual Machines

- A regular application
  - 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



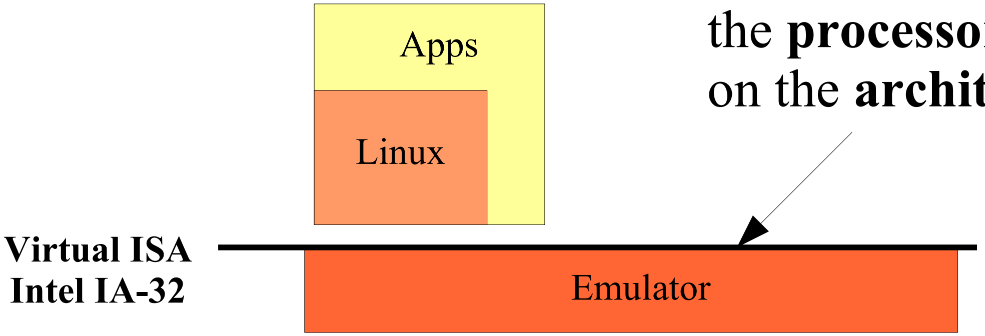
Type-II Virtual Machines

- 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)



Real Machine Emulation

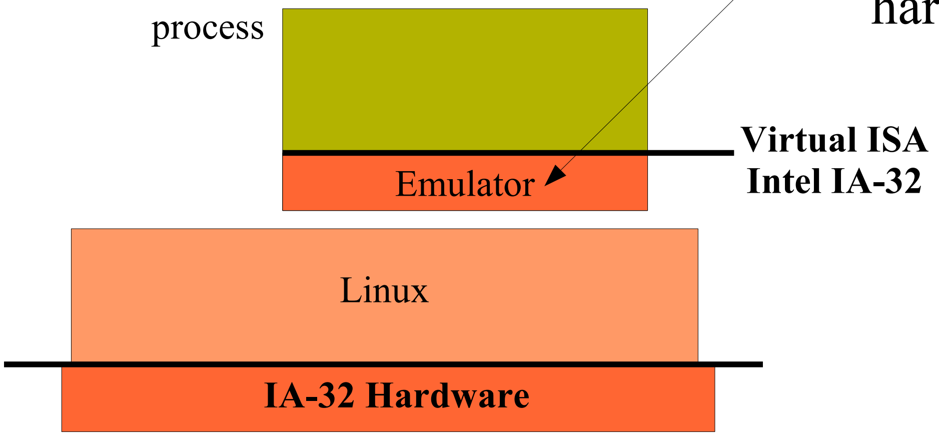
- Through emulation, we can always implement an hypervisor
  - Emulation means either **interpretation** or **binary translation**
  - **Provides the illusion of a real machine**
  - With the same ISA or not



The **hypervisor** emulates
the **processor behavior**
on the **architected state** of the machine

Real Machine Emulation

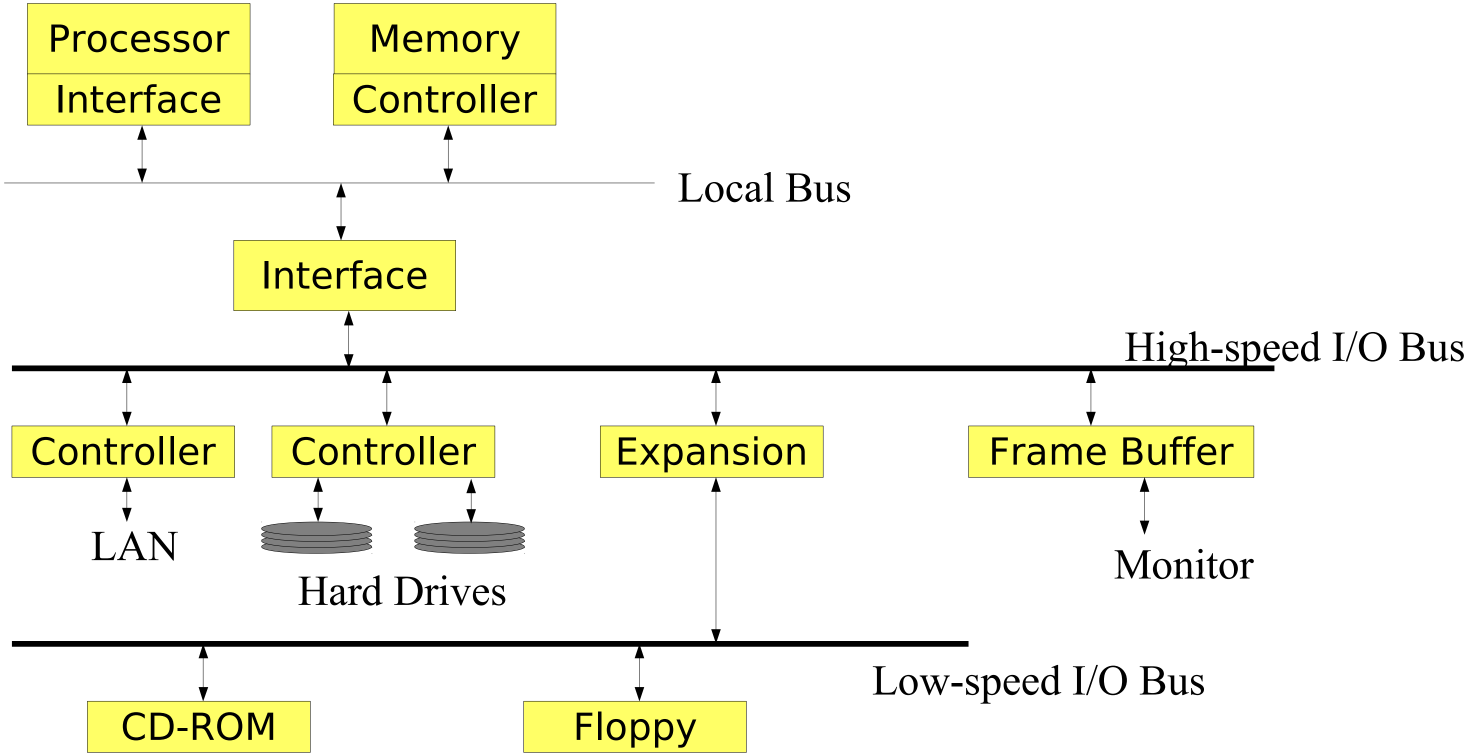
- Emulation by interpreter
  - Emulator is just a C program
    - Like any interpreter (JavaScript or Java)
  - Interpreter:
    - Fetch-decode-issue loop
    - Interpreted instructions manipulate the architected state



The **interpreter** emulates
the **processor behavior**
on the **architected state** of the emulated
hardware

Architected State

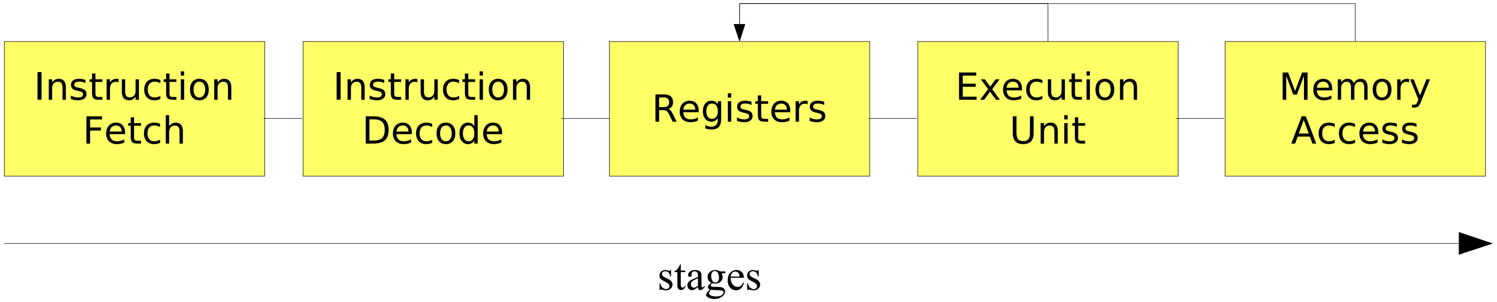
- What is the architected state of the emulated hardware?
  - The description of the hardware
  - The description of the current state of that hardware



Interpretation

- Emulates the processor on the architected state
  - Processor state
    - General puspose 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
    - Pending interrupts, including the timer interrupt

The interpreter emulates the processor on the architected state



Discussing Efficiency

- 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

- 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 support for faster virtualization

Discussing Efficiency

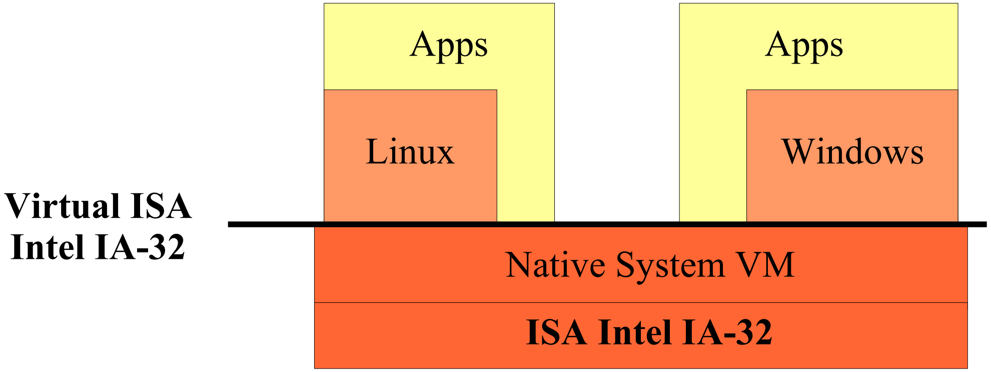
- The challenge...
  - Execute native instructions for speed
  - Keep in control in order to preserve the illusion
  - In particular, isolate and multiplex guest VMs
- A difficult illusion to preserve
  - Multiple guest VMs sharing a single hardware
  - Guest VMs execute code that contains priviledge 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

- 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?

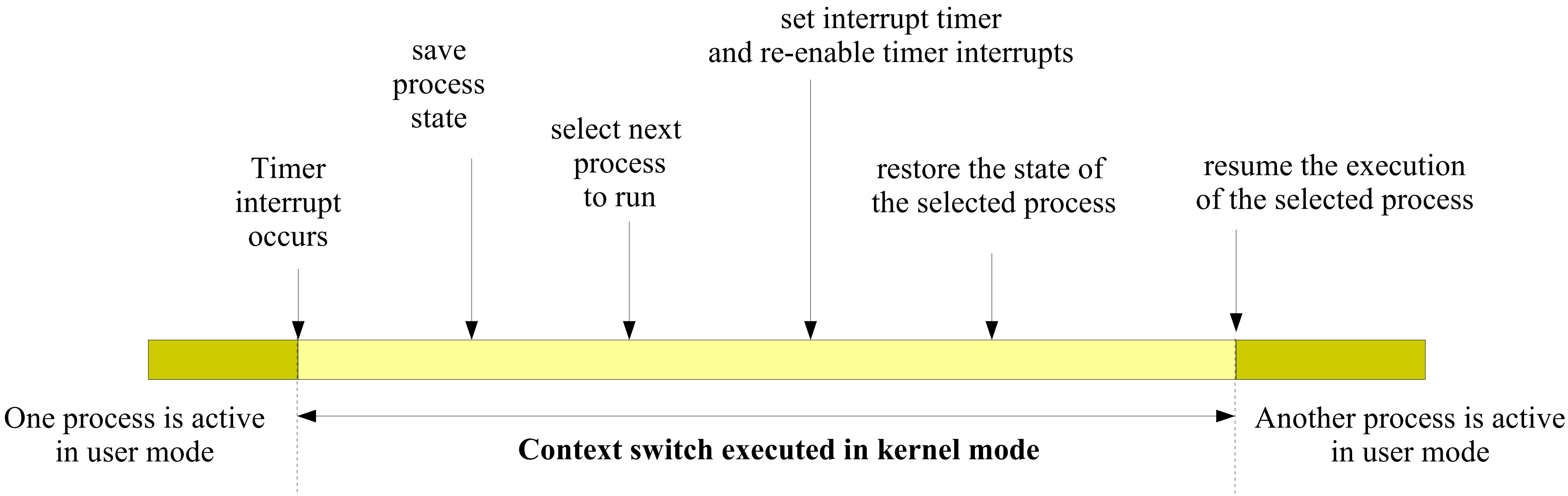


Traditional Multi-tasking

- 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
  - Priviledged instructions only in kernel mode
- A small ''architected state'' per process
  - Its page table
  - Saved registers
  - Pending signals
  - Etc.

Traditional Multi-tasking

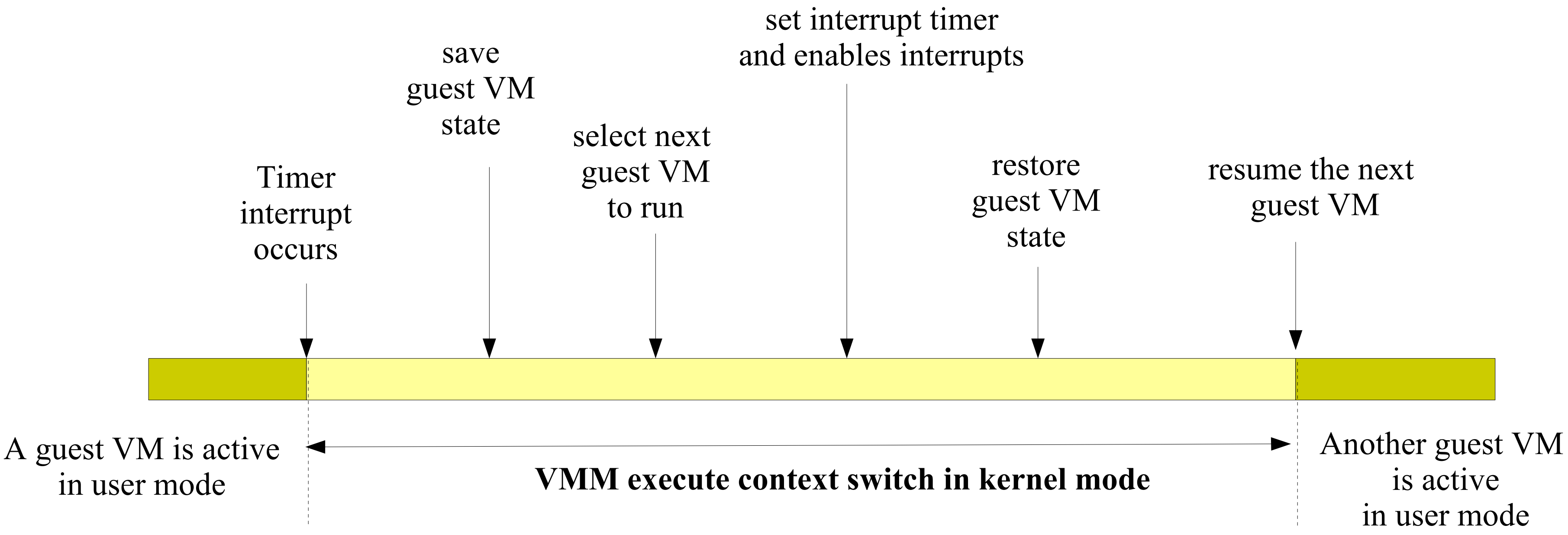
- Scheduling overview
  - We have only one real machine and its architected state
  - We need to multi-task several processes



What is the difference with multiplexing guest operating systems then?

Multi-tasking Guest VMs

- Scheduling overview
  - We have only one real machine and its architected state
  - We need to multiplex guest VMs above

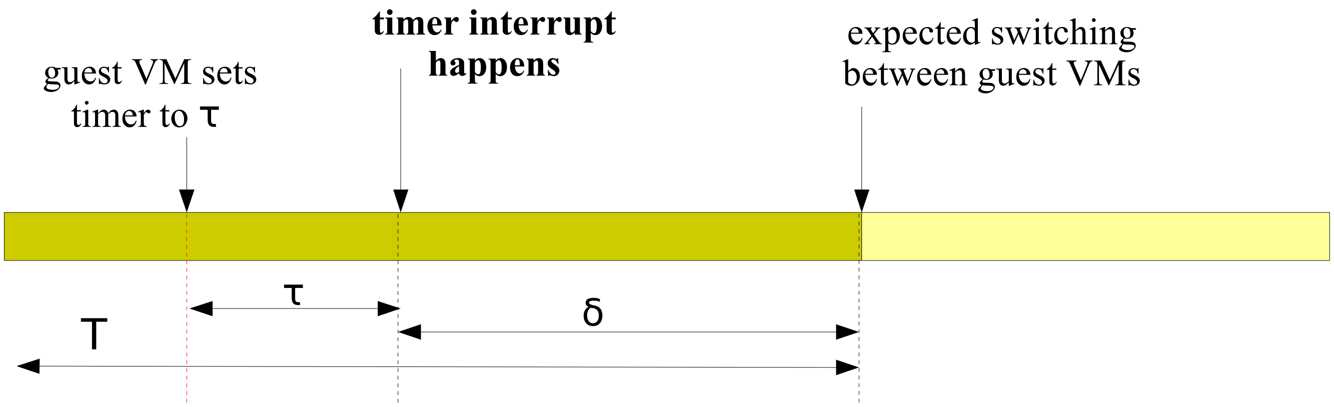


Multi-tasking Guest VMs

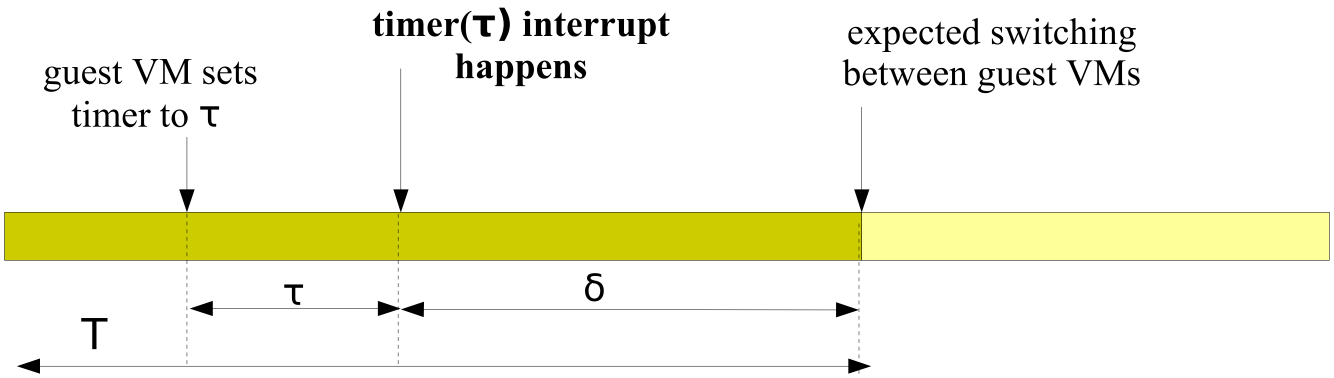
- Main difference
  - We are multiplexing operating systems
  - They expect to run in kernel mode
  - They actually manipulate the timer
- How do we schedule them and stay in control?
  - Only the VMM runs in kernel mode
  - Guest VMs run in user mode
  - Required emulation
    - The instructions manipulating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

Multi-tasking Guest VMs

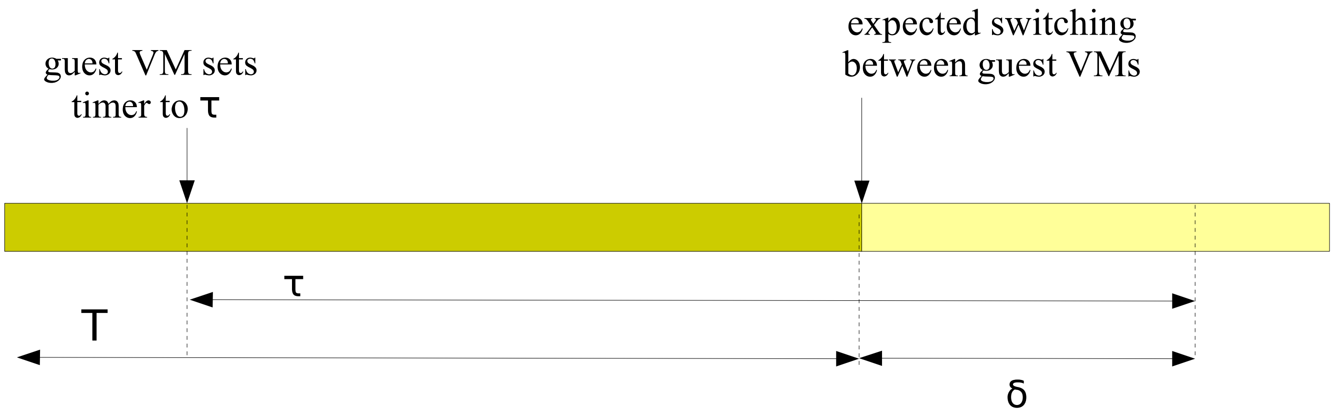
- Require emulation (cont...)
  - We want that each guest VM be scheduled for a time slice T
  - We must preserve the control of the timer
  - We must decide what to do upon timer interrupts



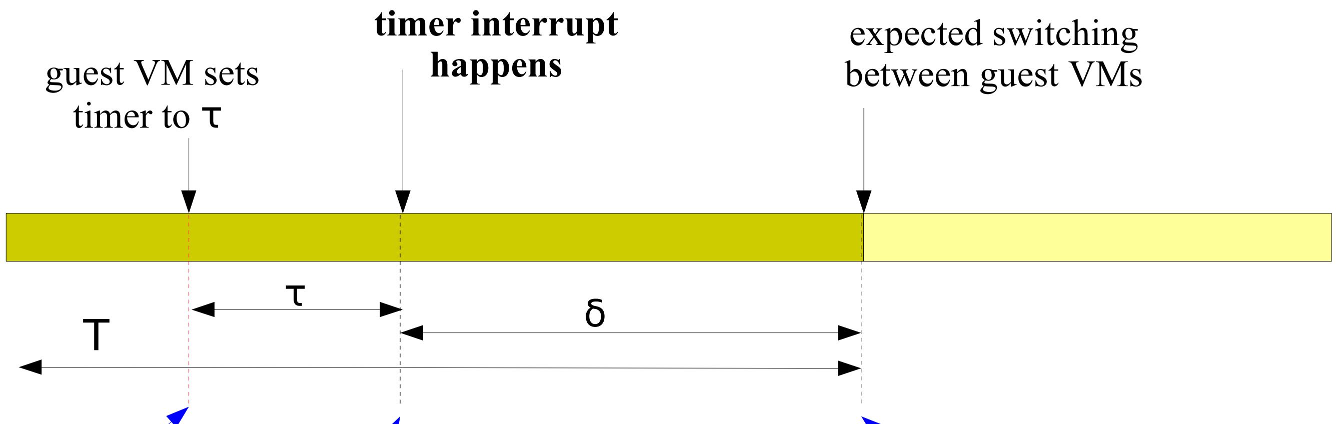
System Virtual Machines


guest VM sets timer to τ
timer(τ) interrupt happens
expected switching between guest VMs
τ
δ
T

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



Multi-tasking Guest VMs



guest VM sets timer to τ

timer interrupt happens

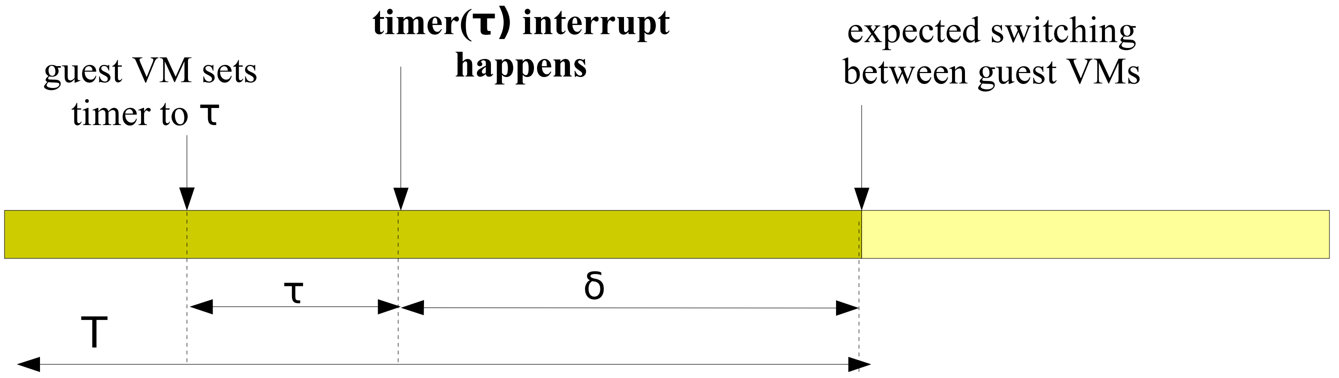
expected switching between guest VMs

Trap to the VMM

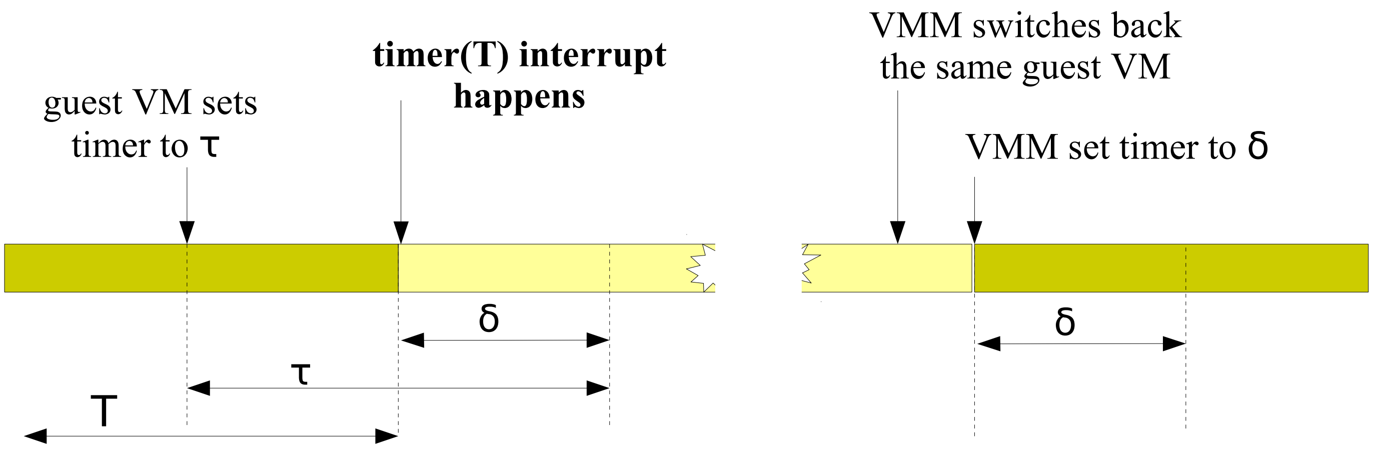
Trap to the VMM
VMM resets timer to remaining δ
VMM forwards the timer interrupt to the guest OS
VMM knows how since it emulates the interrupt vector

Trap to the VMM
Context-switch to a new guest

Multi-tasking Guest VMs



Overall summary:



Multi-tasking Guest VMs

- 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
- How did we emulate?
  - We did not interpret all instructions
  - Guest code runs natively, so how do we emulate?

Multi-tasking Guest VMs

- 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
  - Or the hardware can be reconfigured before re-executing the instruction
- So how did we emulate?
  - If we assume that *sensitive instructions* are priviledged
  - Since the guest runs in user mode, these instructions will trap
  - In the trap handlers, the VMM is back in control
  - Through traps, the VMM emulates sensitive instructions

Emulation Challenge

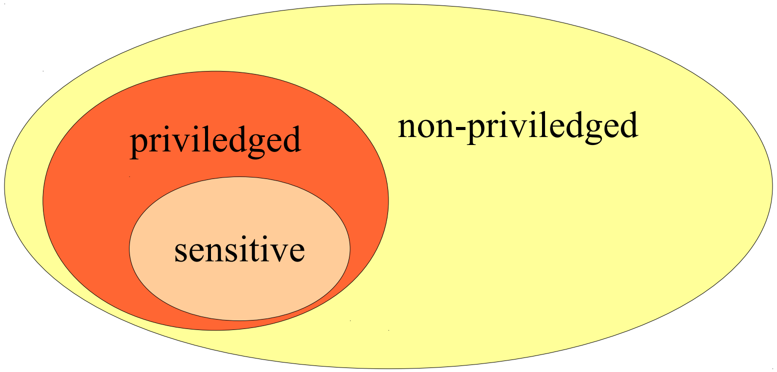
- Many more sensitive instructions
  - Guest VM run entirely in user mode
  - But it believes that it runs in kernel mode
  - It expects to have full control of the machine
  - It will use **many instructions** to manipulate hardware resources
    - Setting interrupt and trap vectors
    - Changing page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

Emulation Challenge

- **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

Emulation Challenge

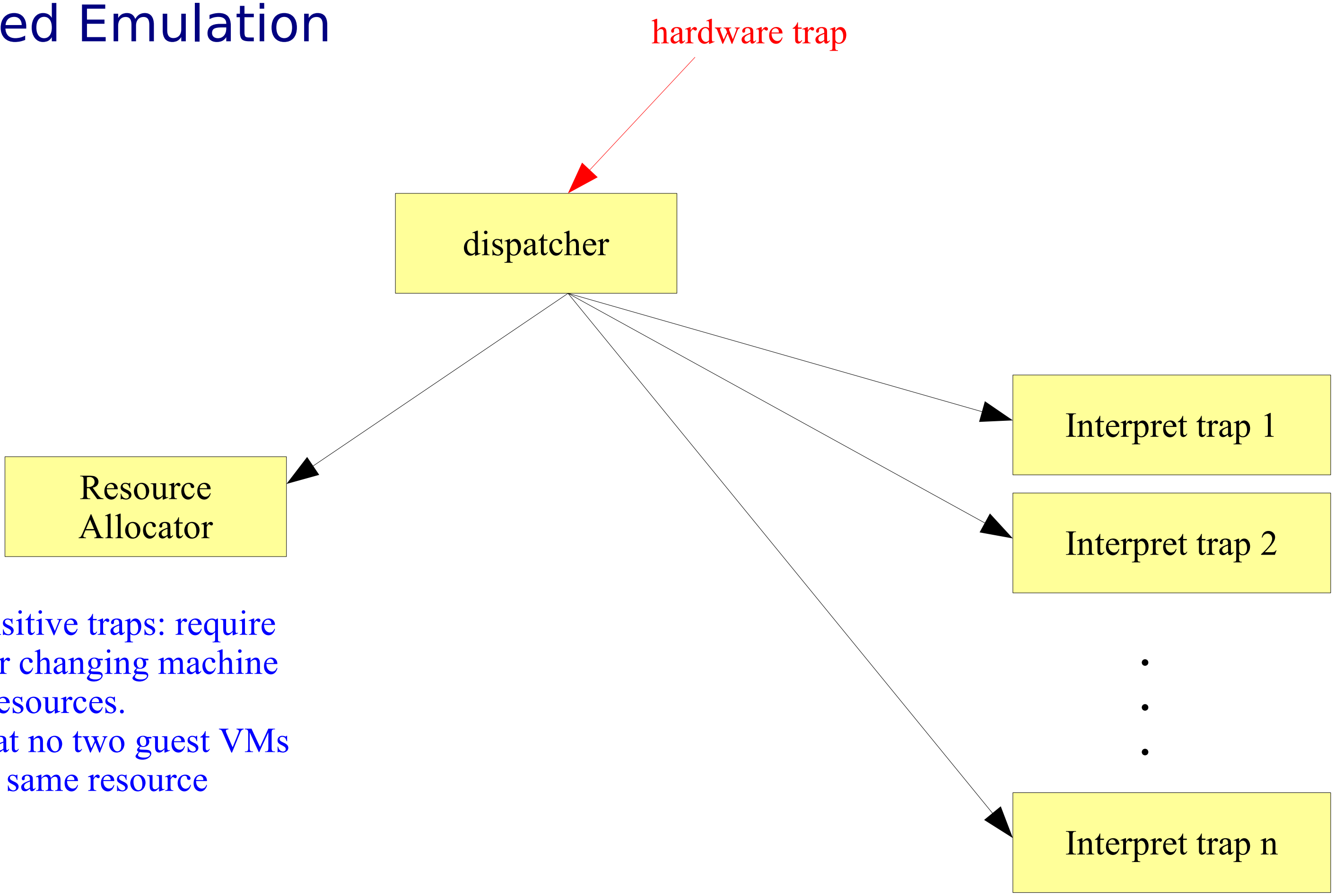
- Virtualization theorem
  - *An efficient system virtual machine may be constructed if the set of sensitive instructions for the real ISA is a subset of the priviledged instructions.*



  - **Priviledged 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

- Trap-based Emulation



Control-sensitive traps: require allocating or changing machine resources.
It ensures that no two guest VMs get the same resource

Behavior-sensitive traps: do not change machine resources but accesses priviledged resources;
they are emulated
on the architected state of the current guest VM

Emulation Challenge

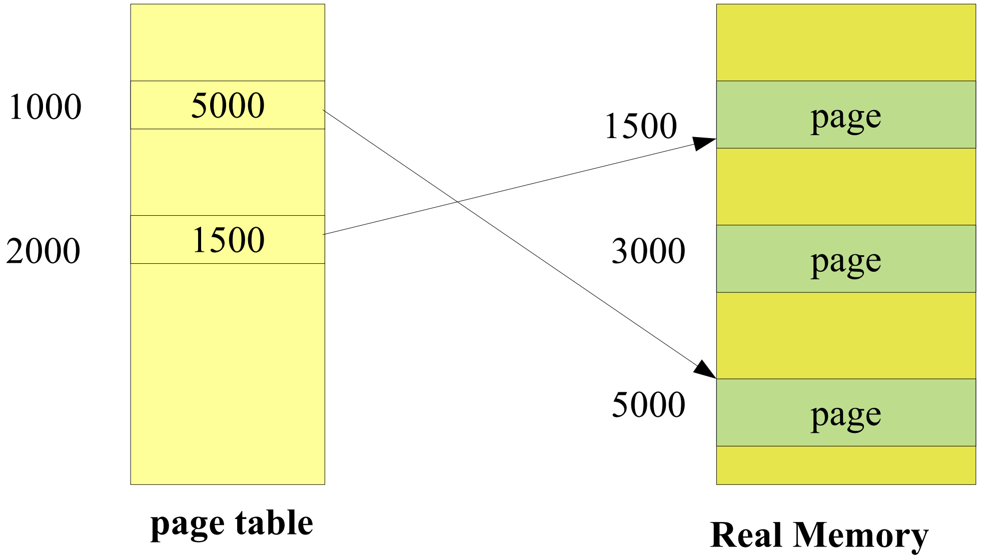
- We just discussed trap-based emulation
  - Easy to apply to the timer emulation
  - As well as the interrupt and trap vector
  - So we emulated the processor of a real machine ✔
- We still need to discuss how to emulate
  - Memory, including the MMU
  - Devices, including the interrupt controller

Emulating Memory

- 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

- 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

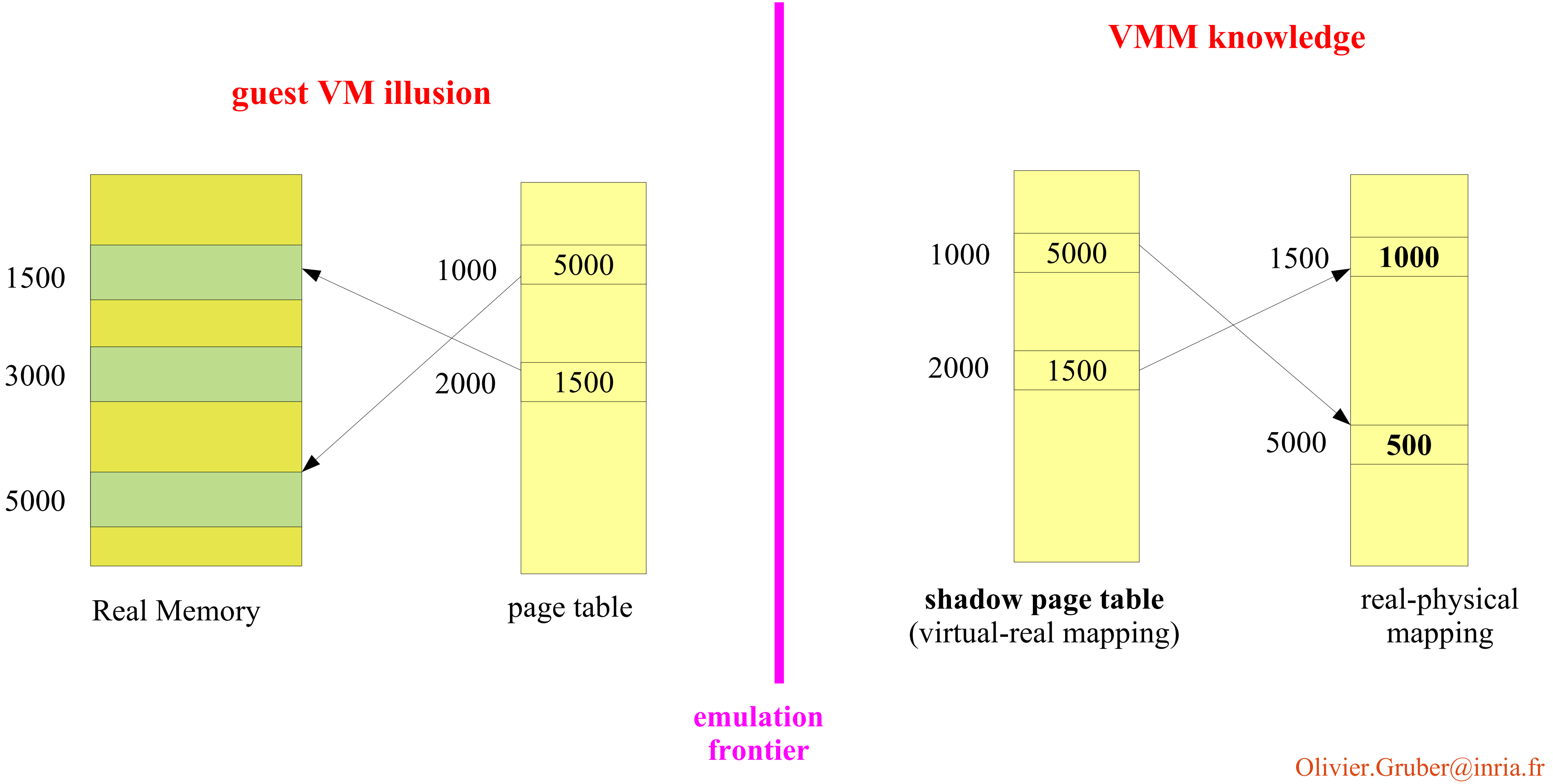
- Virtualize an architected page table
  - Again a story of make believe through emulation
- Each guest VM
  - Manages several page tables
  - Changes the page table pointer upon context-switching
- Emulation through traps
  - Load and store instructions in the page table register are priviledged
  -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

Architected Page Table Emulation

- 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

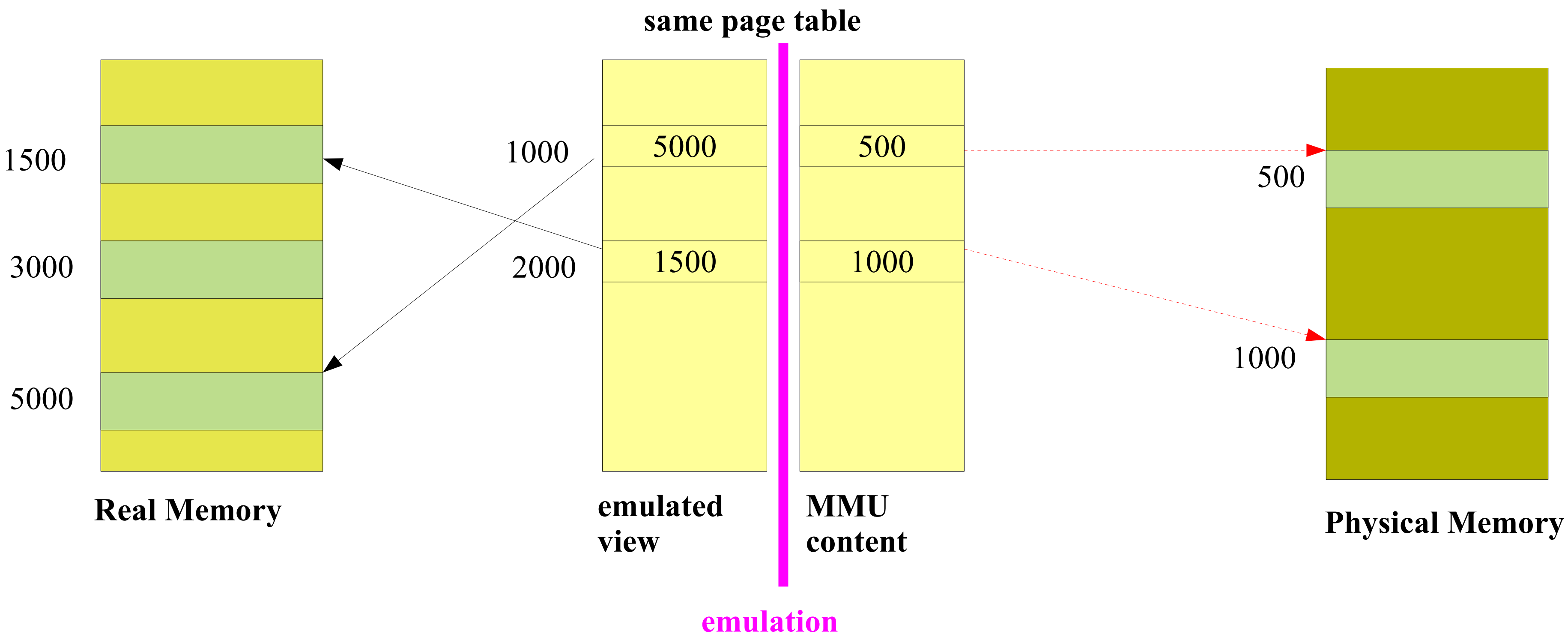
Architected Page Table Emulation

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



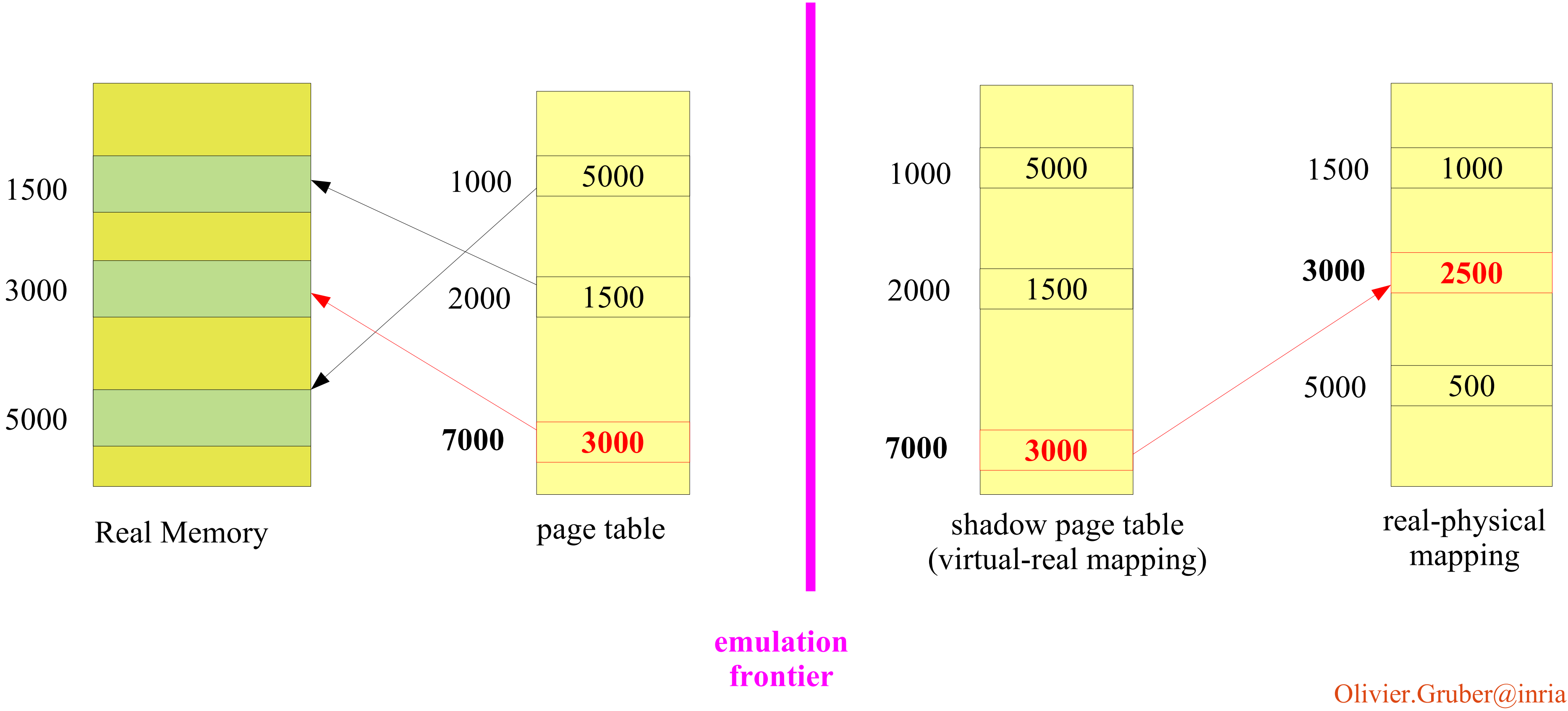
Architected Page Table Emulation

- The VMM emulates the virtual-to-real-to-physical mapping
  - Load and store instructions in the page table are priviledged
  - 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



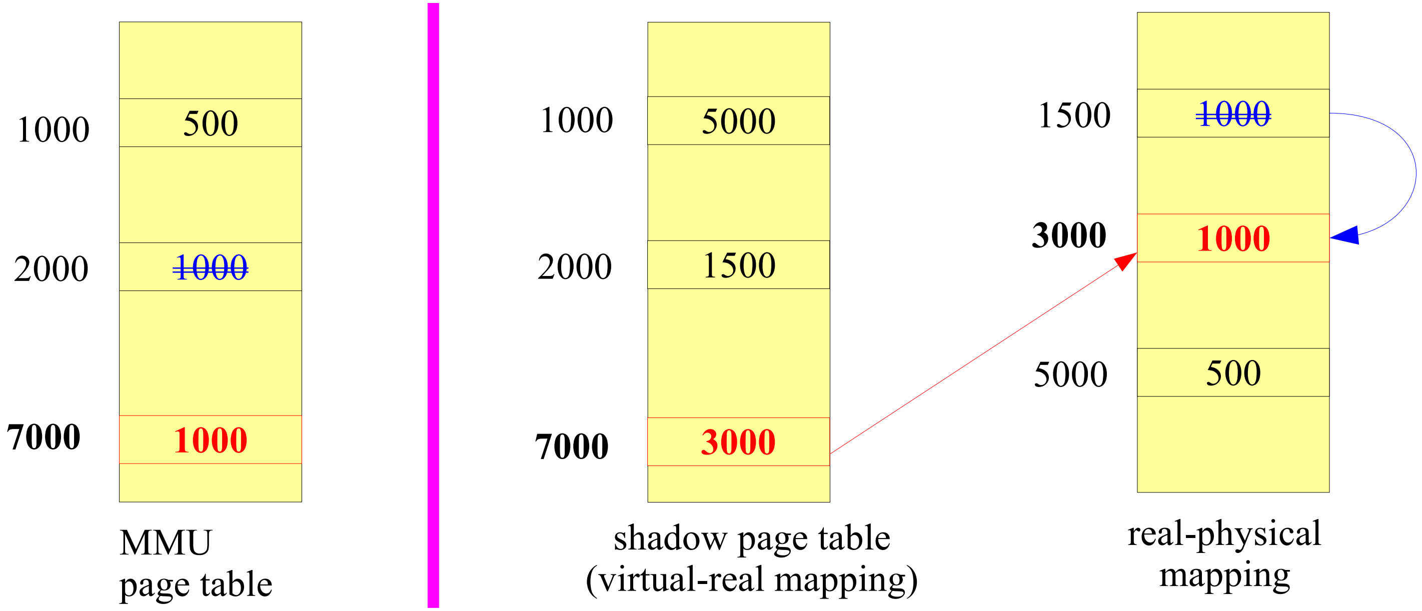
Architected Page Table Emulation

- 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)



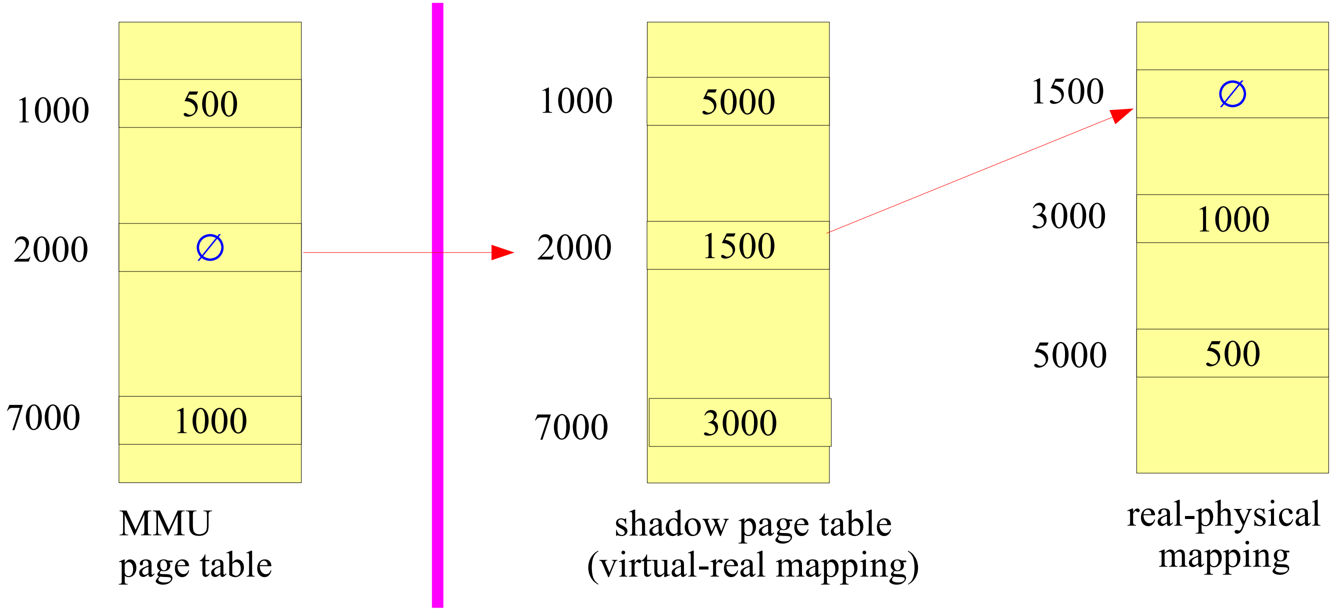
Architected Page Table Emulation

- 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

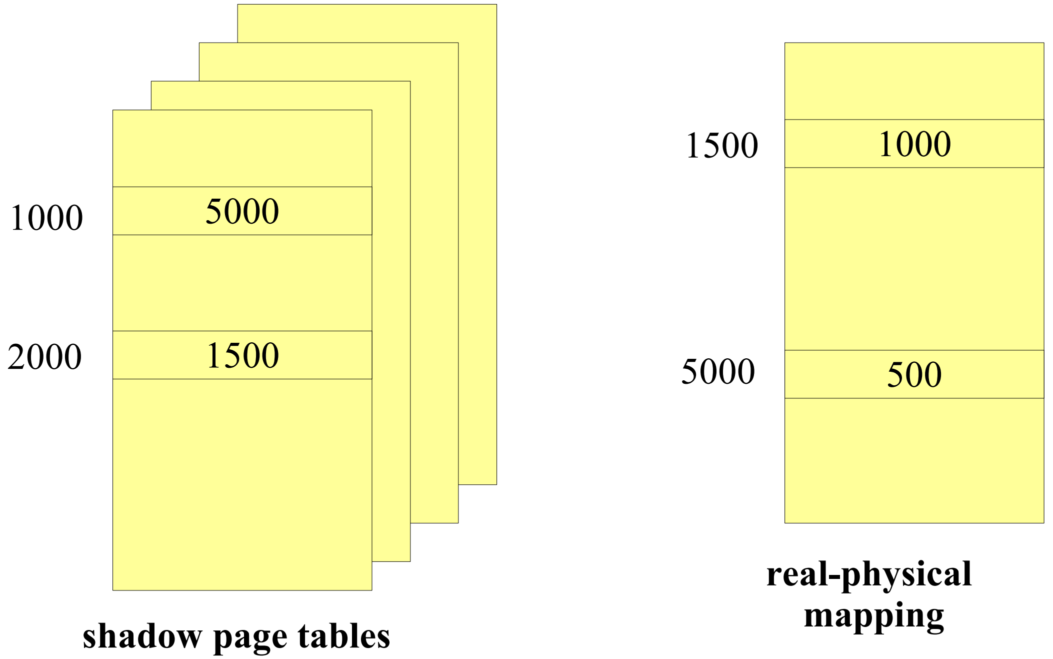
- Emulate Page faults
  - Page faults may be on real or physical memory pages
    - Real memory page faults need to be forwarded to the guest VM
  - Physical memory page faults must be handled by the VMM
    - Example virtual address 2000 triggers a page fault
    - Guest VM expects it to be in memory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



Architected Page Table Emulation

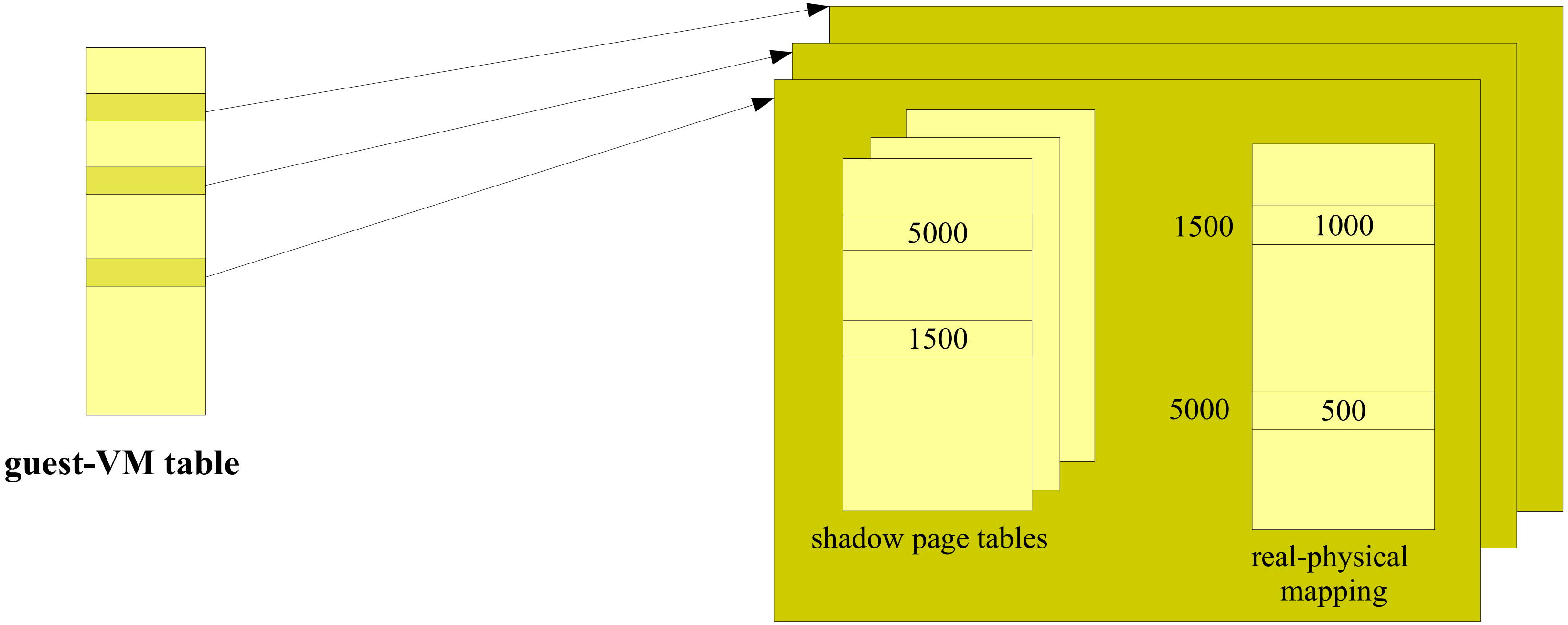
- Emulate address-space switching within a guest VM
  - Changing the page table pointer traps in VMM
  - Need to select the new virtual-real mapping
  - So to keep it consistent with the guest VM expected state

VMM knowledge per guest VM



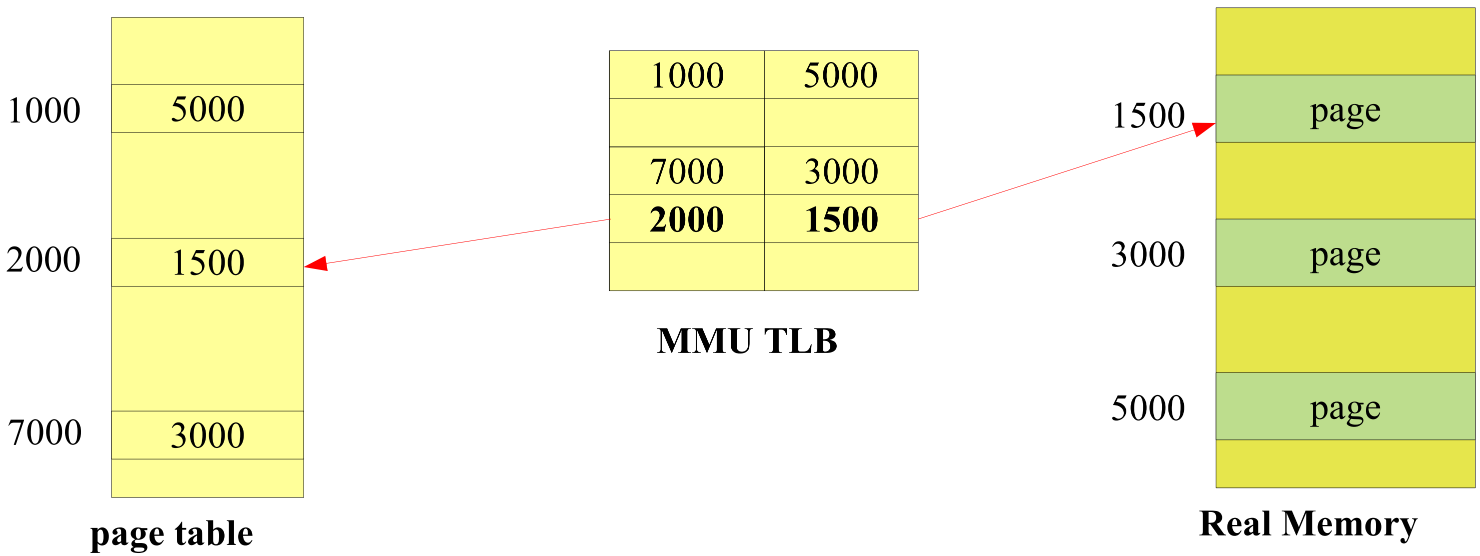
Architected Page Table Emulation

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



Architected TLB Emulation

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

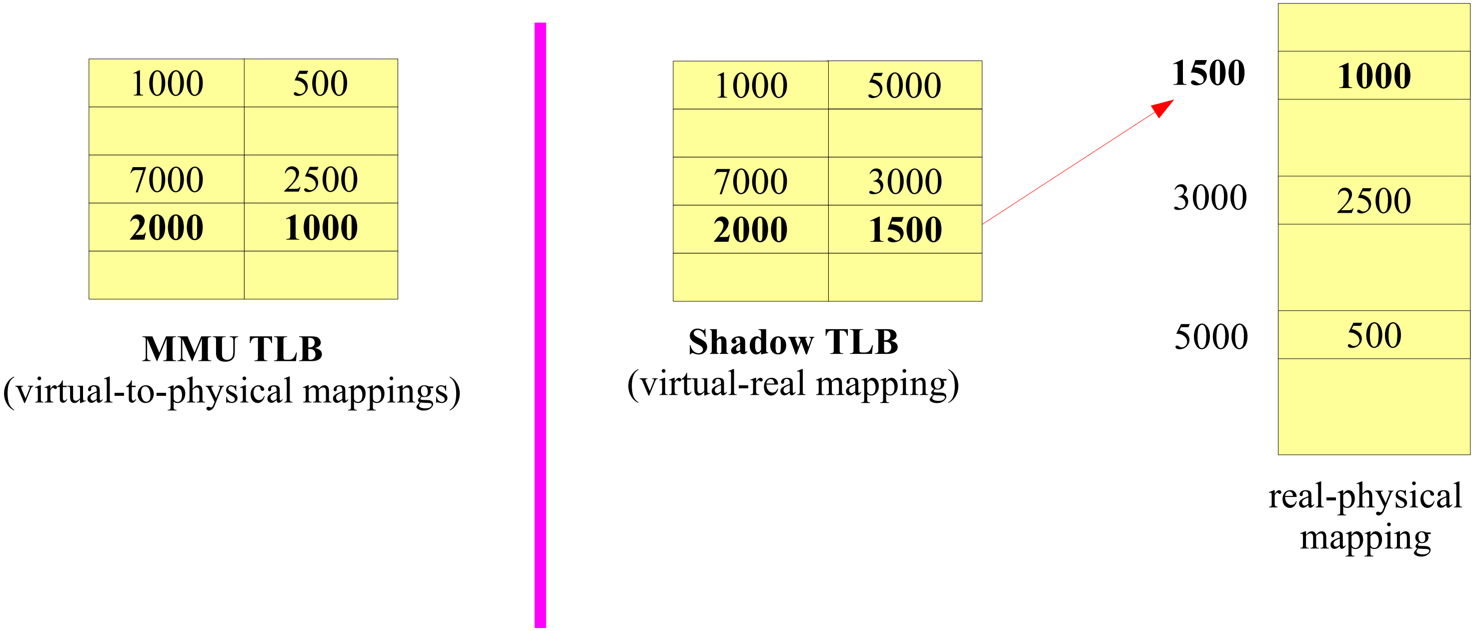


MMU TLB

1000	5000
7000	3000
2000	**1500**

Architected TLB Emulation

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



1000	500
7000	2500
2000	**1000**

MMU TLB
(virtual-to-physical mappings)

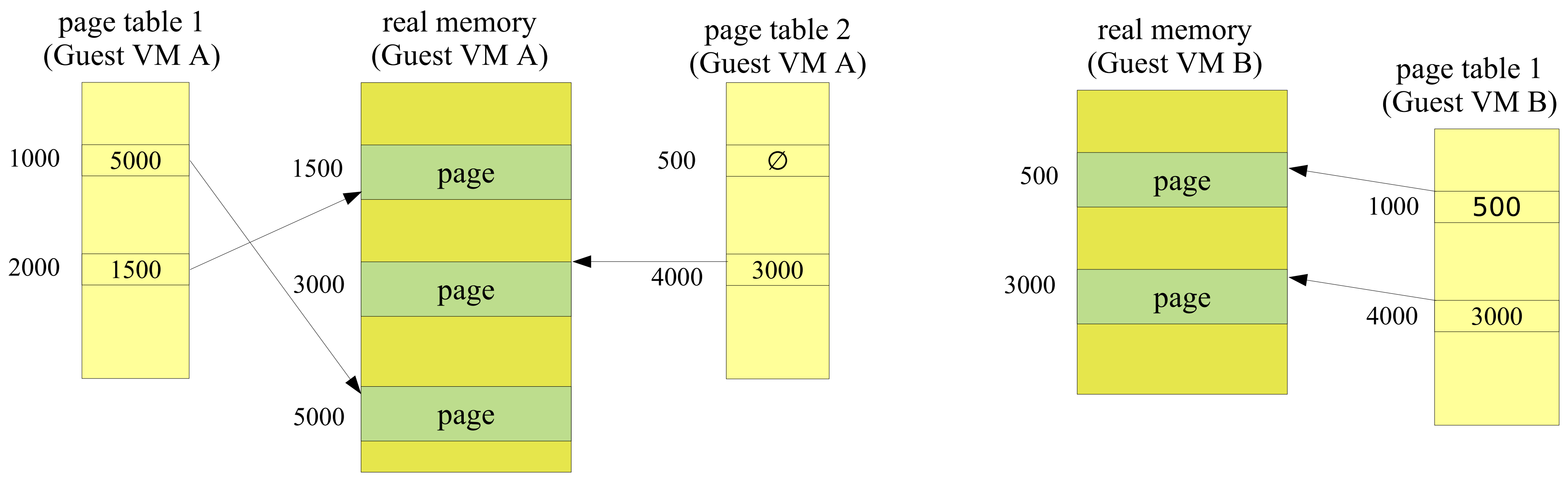
1000	5000
7000	3000
2000	**1500**

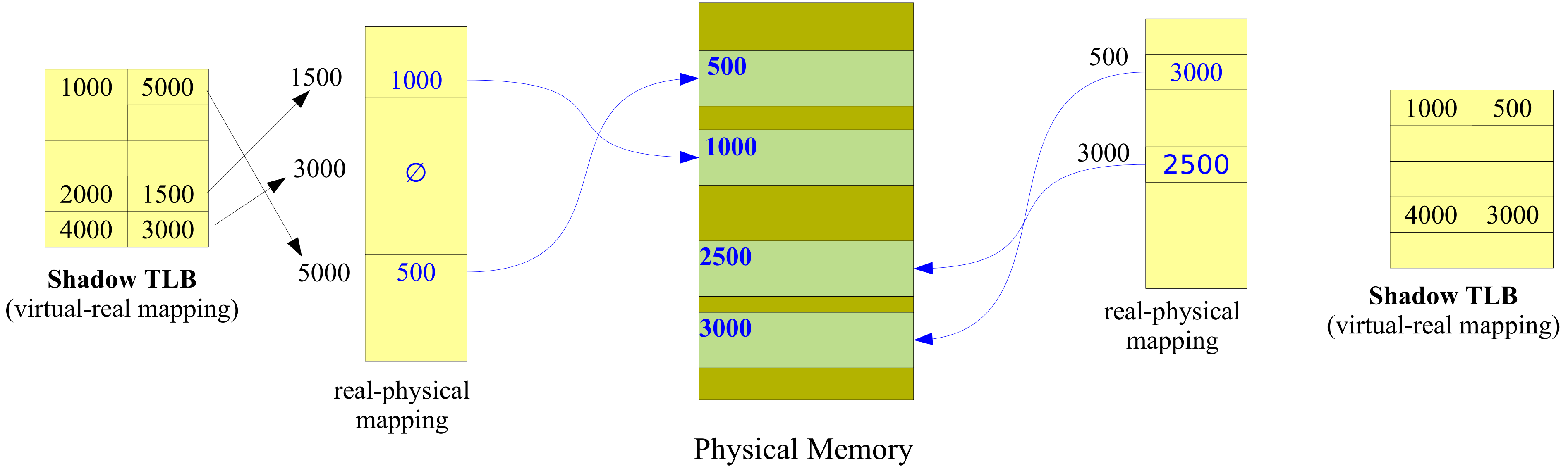
Shadow TLB
(virtual-real mapping)

1500	**1000**
3000	2500
5000	500

real-physical
mapping

Architected TLB Emulation





1000	5000
2000	1500
4000	3000

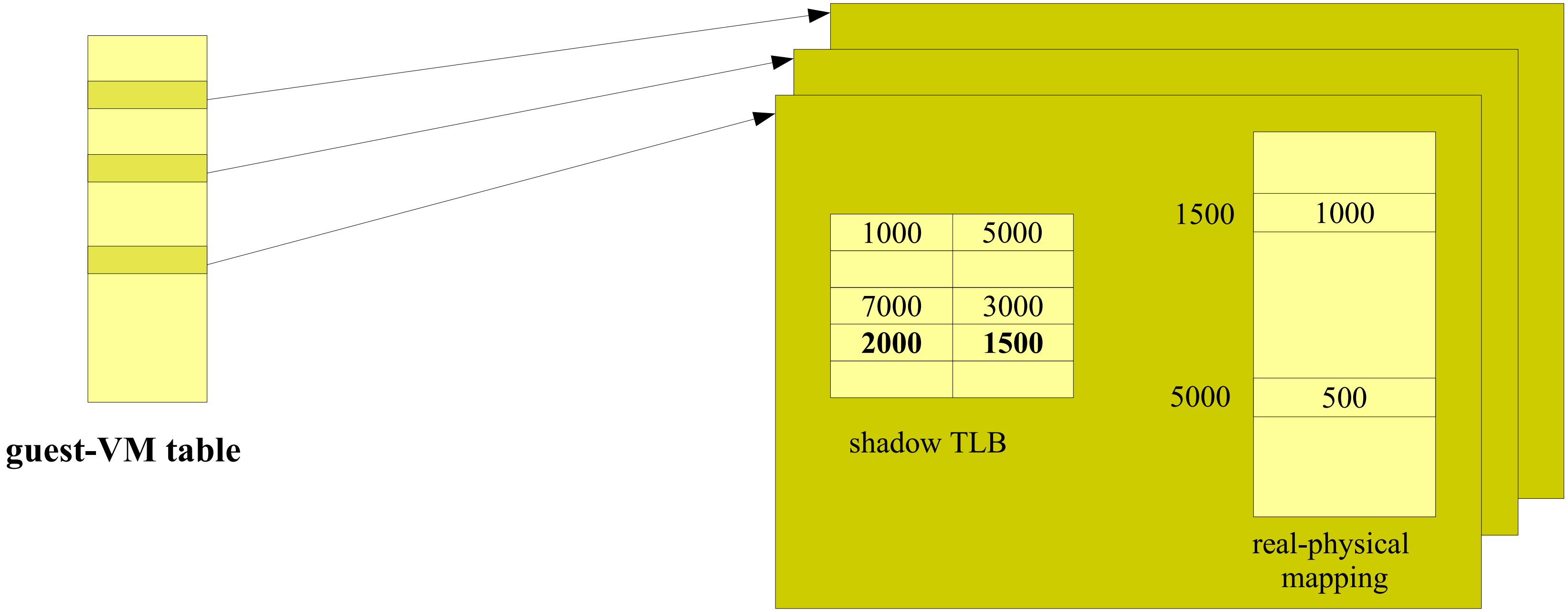
Shadow TLB
(virtual-real mapping)

1000	500
4000	3000

Shadow TLB
(virtual-real mapping)

Architected TLB Emulation

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



1000	5000
7000	3000
2000	**1500**

Architected TLB Emulation

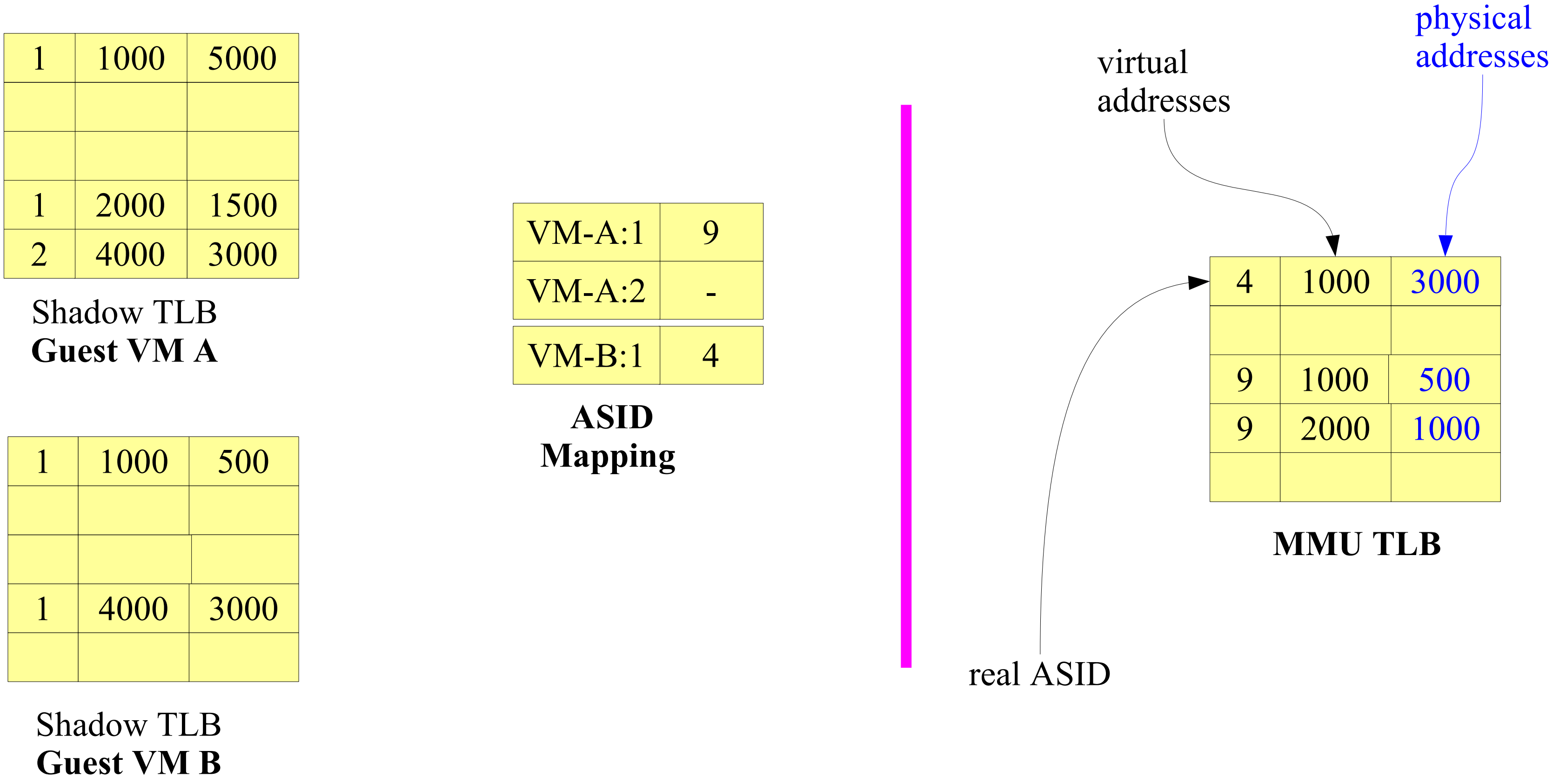
- Address Space IDentifier (**ASID**)
  - Included support in architected software-managed TLBs
    - An architected ASID register contains the current ASID
    - ASID register is assigned on address space switch
  - Enables to mix address translations for different address spaces
    - ASIDs are checked 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

- 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

4	1000	3000
9	1000	500
9	2000	1000

MMU TLB

Emulation Challenge

- Emulate processors ✔
- Emulate memory, including the MMU ✔
- Emulate devices, including the Programable Interrupt Controller (PIC)

Device Emulation

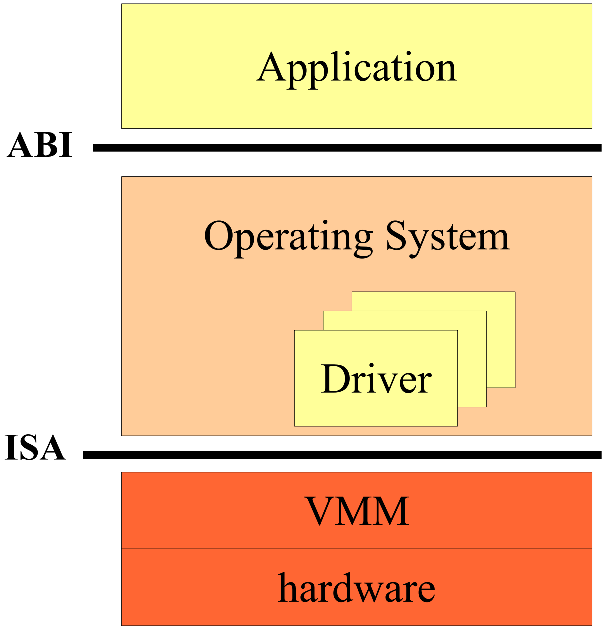
- Virtualizing I/O devices is one of the more difficult aspects of VMM
  - Each I/O device has its own characteristics
  - Each I/O device needs to be controlled in its own special way
- The number of device types is growing constantly
  - In Linux, device drivers represent 70% of the code
  - In Linux,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

Device Emulation

- 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

- 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
  - Drivers read and write hardware registers
  - Drivers rely on interrupts
  - Often rely ring-buffers and DMA



Emulating Devices

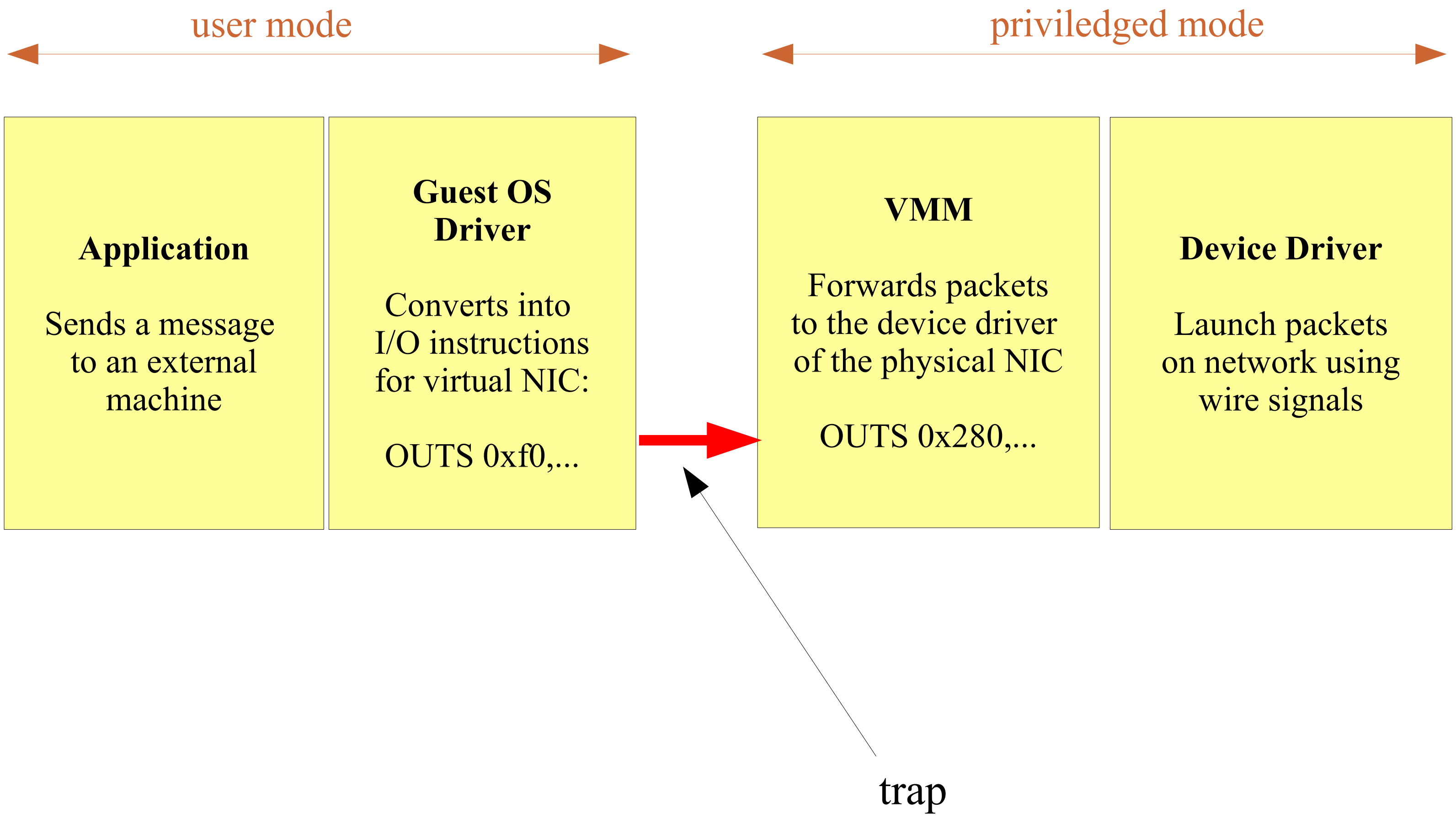
- Example: Network Interface Card (NIC)
  - Like an NE2K on IA-32
    - With IN/OUT or INS/OUTS instructions
    - On I/O ports, like 0xf0
  - Through ports (device registers really)
    - Write commands and parameters
    - Read status
  - Ring buffers – the usual approach
    - Fixed size circular 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

Emulating Devices

- Emulation: Network Interface Card (NIC)
  - Each port may be set to trap if I/O instructions are attempted
  - 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 virtual)
    - 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

- Example: network virtualization

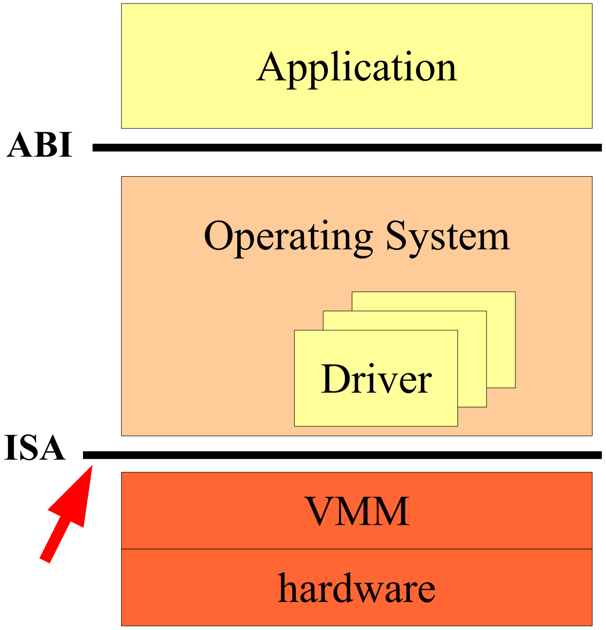

user mode
priviledged mode
Application
Sends a message to an external machine
Guest OS Driver
Converts into I/O instructions for virtual NIC:
OUTS 0xf0,...
VMM
Forwards packets to the device driver of the physical NIC
OUTS 0x280,...
Device Driver
Launch packets on network using wire signals
trap

Emulating Devices

- 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
  - Must have a complete specification of the device behavior
    - Often fuzzy, this is where writting device drivers is black art...
    - Word sizes might be different...
    - Endianness might be different...
    - Sometimes must emulate precise timing

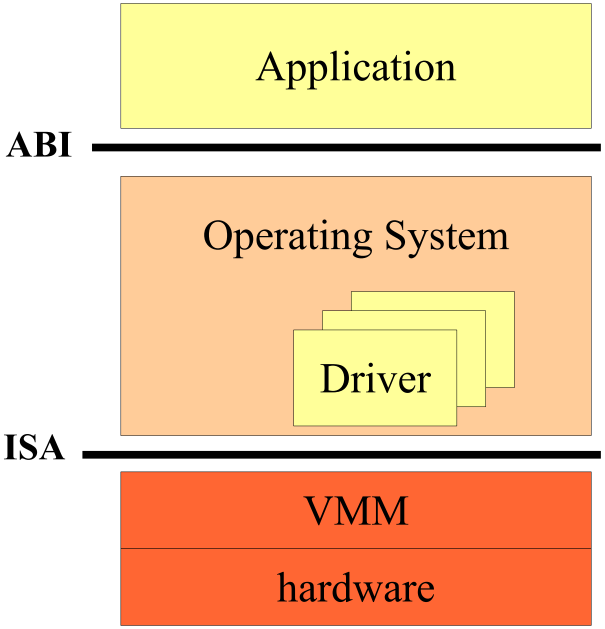
Emulating Devices

- Easy to intercept
  - Through special load/store instructions or regular load/store at special memory locations
  - Either priviledged instructions or protected memory locations
- 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



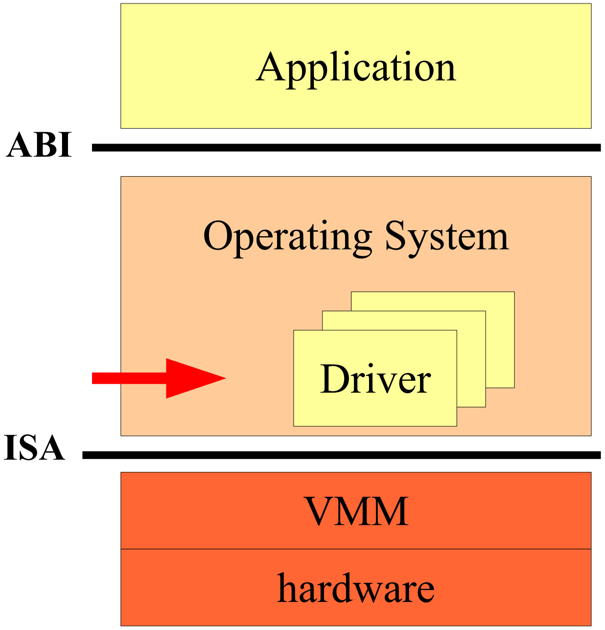
Emulating Devices

- Remember
  - Micro-kernels faced a similar challenge...
  - Having to port drivers to user-level processes
  - 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
  - Obviously no longer a show stopper...



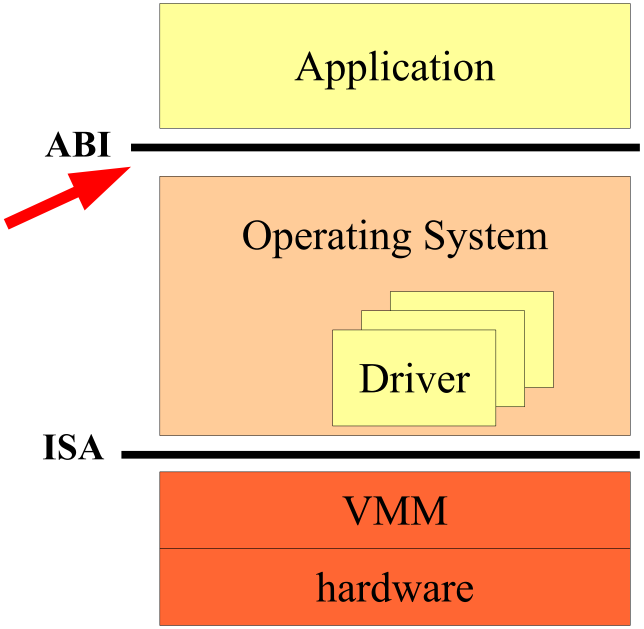
Emulating Devices

- Emulate at the device driver interface
  - A very natural interception point
  - Easy to emulate (high-level operations)
- Not general
  - Require some knowledge of the guest OS and of its internal device driver interface
  - Therefore not a general solution



Emulating Devices

- 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 shot
- Not general
  - Daunting task, 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
  - Again, not a general solution



Emulating Devices – Xen Approach

- Xen Bus – A split-driver model
  - Front-end drivers in the guest
  - Back-end drivers in other guests
  - Each front-end driver communicates with a back-end driver
  - The back-end drivers are usually provided by the Dom0
- Xen Bus – Key concepts
  - A Xen store is a naming service for front-ends to locate back-ends
  - Concept of event channels between the front-end and back-end drivers
  - Events are very much like interrupts (32bit data)
  - Use of ring-buffers of fix-sized elements, in shared virtual memory
  - Use shared virtual pages to pass out-of-band data

Emulating Devices – Xen Approach

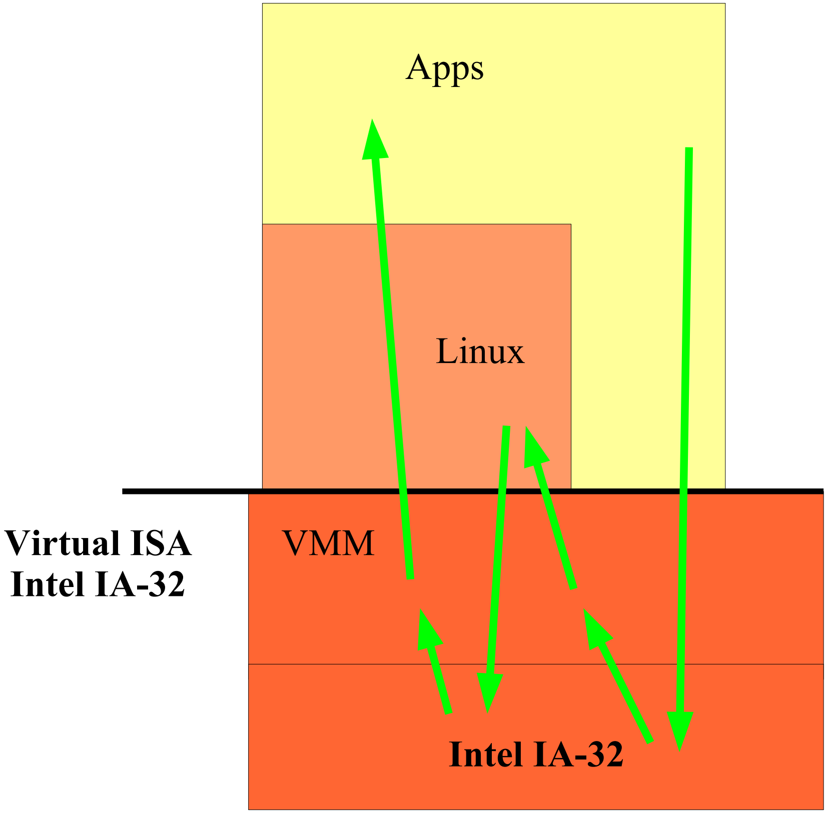
- Xen Driver Classes
  - Generic block devices
    - Disk: read/write blocks at an offset
    - NIC: send/receive blocks
  - Generic character devices
    - Serial lines
    - Input devices such as mouse or keyboards

Emulation Challenge

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

Stepping Back

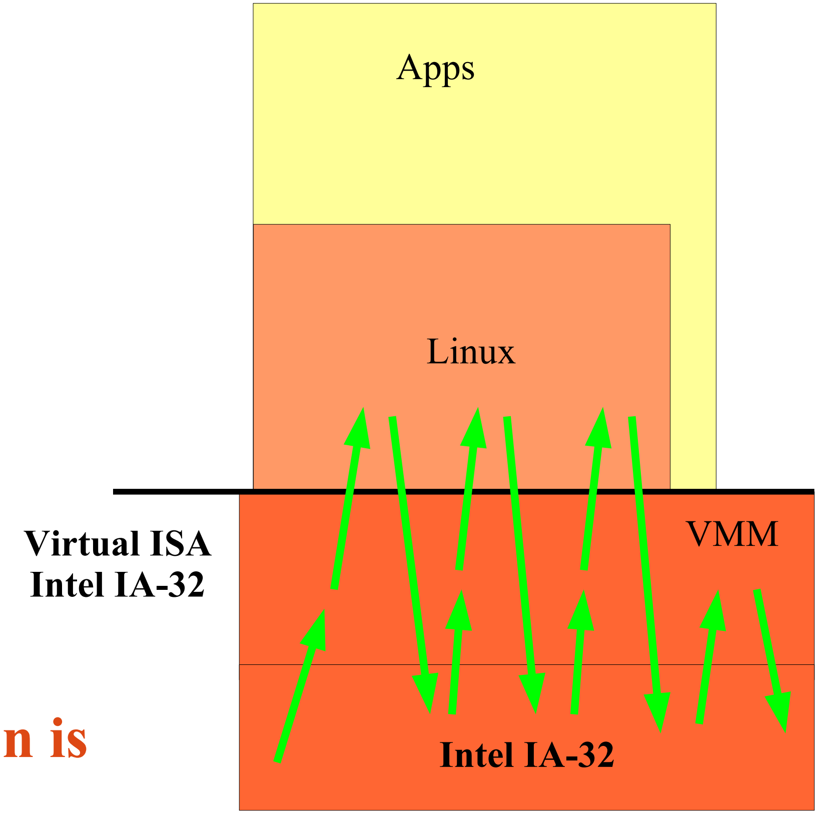
- Performance
  - If we have unmodified guests, all sensitive instructions trap
  - This means a lot of traps and traps are expensive on modern processors
- Discussing traps in Type-I hypervisors
  - Let's discuss a page fault...
  - 2 mode switch (kernel-user)
  - 2 hardware traps
    - The actual hardware trap
    - The emulation of the RETI
  - Plus the emulation of page entries
    - More traps



Trap-based emulation is not free!

Stepping Back

- Discussing traps in Type-I hypervisors
  - Let's discuss a process context switch...
  - Use many sensitive instructions...
    - Change page table
    - Flush TLB
    - Set timer
    - Re-enable interrupts
  - Solution
    - Use hypercalls
    - Such as a context-switch hypercall
    - Only one trap (hypercall)



Trap-based emulation is definitely not free!

Stepping Back

- 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

Stepping Back

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

- 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?

Emulating Instructions

- 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 priviledged

Emulating Instructions

- 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

Emulating Instructions

- 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 addresse and size for the IDT
    - Problematic instructions
      - Unpriviledged SIDT instruction stores the IDTR in memory
      - Priviledged write instruction to the SIDT registers

Emulating Instructions

- 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 unpriviledged
      - 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 priviledged instruction to load and store control registers

Emulating Instructions

- Handle problematic ISA
  - EFLAGS register
    - Holds flags that control the operation mode and state of the processor
      - Interrupt masking
      - I/O priviledge 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

Emulating Instructions

- Handle problematic ISA
  - CALL, JMP, INT n, and RET instructions
    - Discussing CALL
      - Far and near calls to the same priviledge level
      - Far call to a different level or task switch
      - Behavior thus depends on CPL
        - A task uses a different stack for every priviledge 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-priviledge returns
      - Clears certain segment registers (DS,ES,FS and GS) on inter-priviledge returns towards lower-levels
    - Similar issues with other instructions

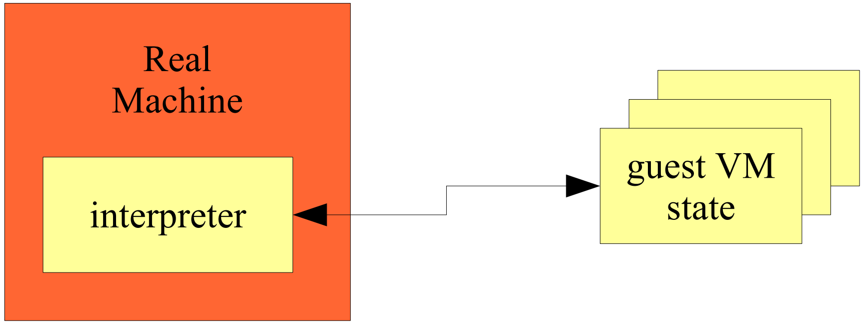
Emulating Instructions

- 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?

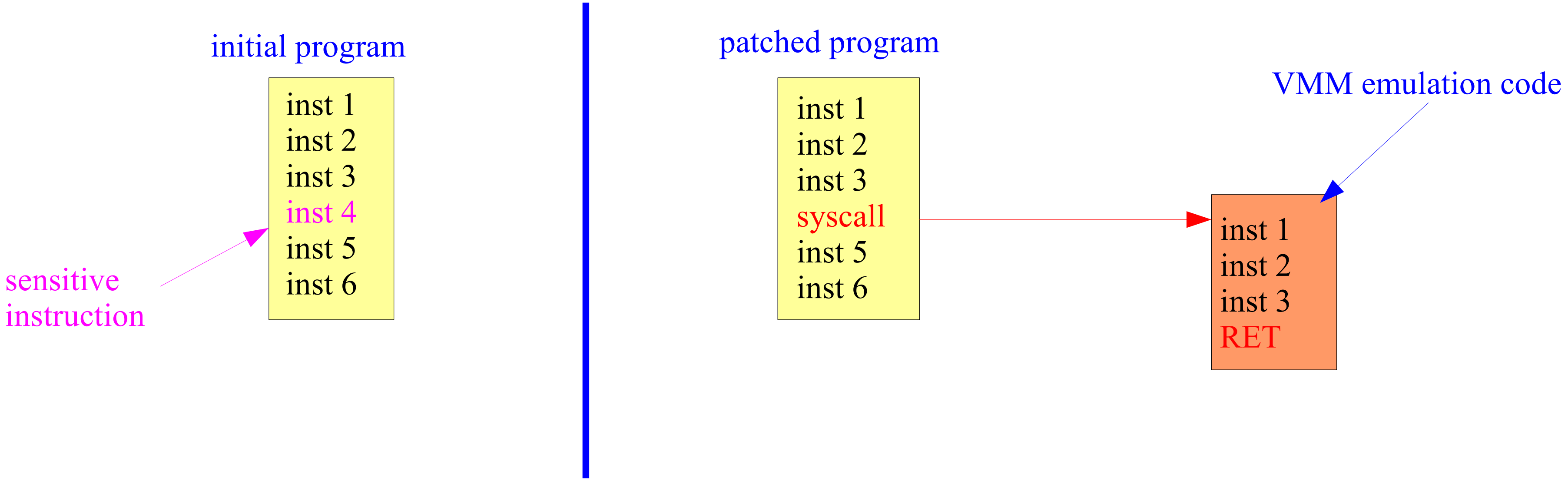
Emulating Instructions

- 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



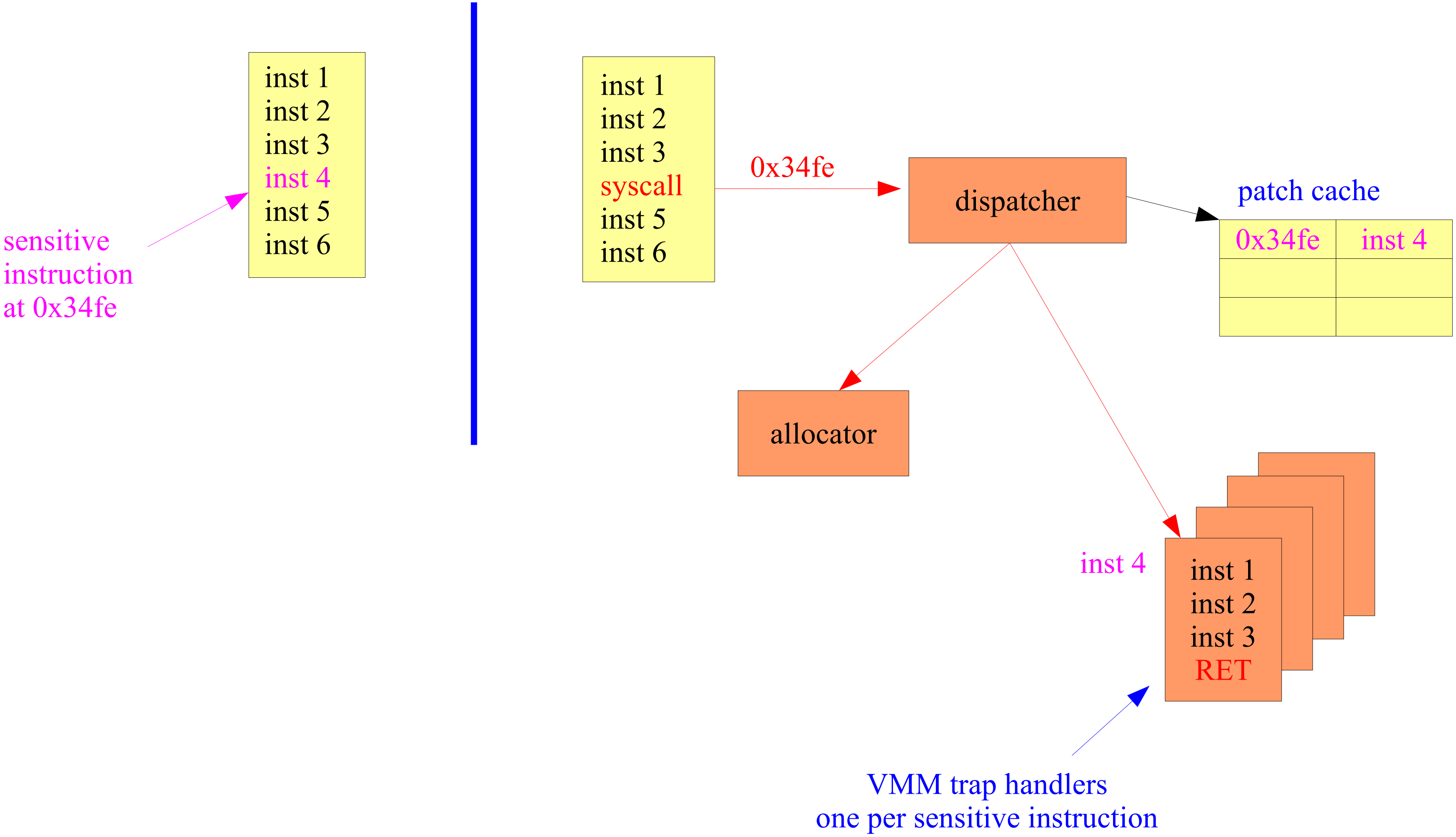
Dynamic Binary Translation

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



Dynamic Binary Translation

- Binary translation is necessary
  - More complete picture



Dynamic Binary Translation

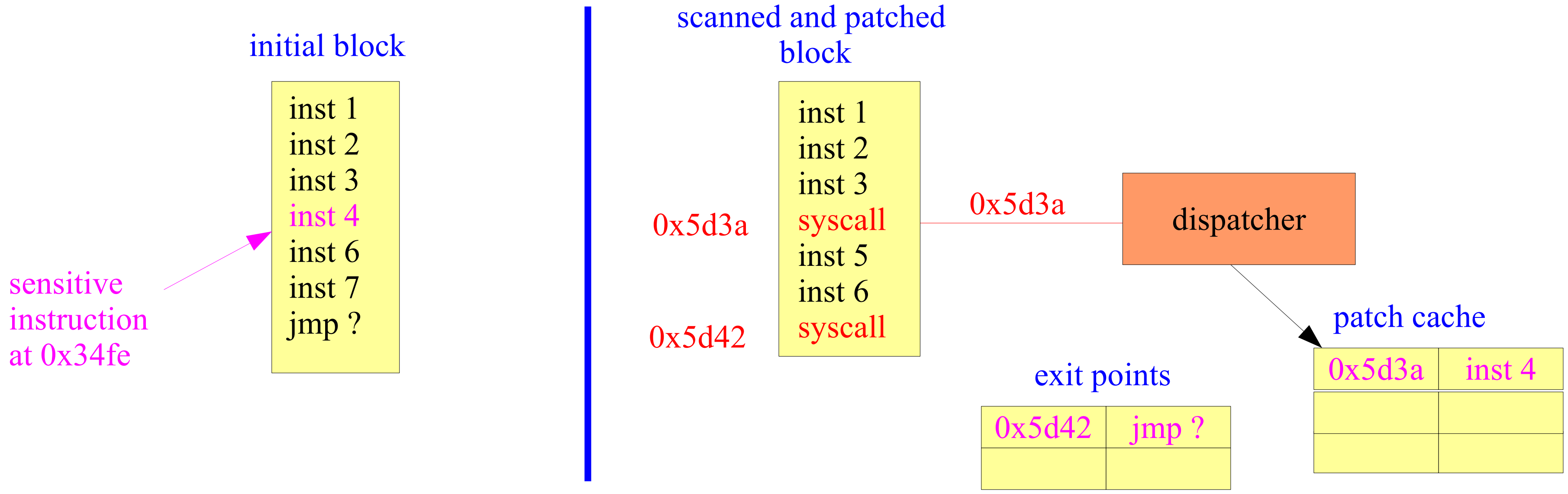
- Can we copy and patch/translate the entire code?
  - The code may be fairly large
  - We may only know a few entry points (sometimes only one)
- Problems
  - Not all branch instructions are known statically
    - Addresses can be computed
    - Example: a jump through an function pointer table
  -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

Dynamic Binary Translation

- Why do we copy code?
  - Replaced instructions may be smaller than the syscall instruction
    - So we can't do it in place
  - This is also necessary for self-inspecting and self-modifying code
    - So we need to emulate all accesses to the PC
    - 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

- 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**



initial block

inst 1
inst 2
inst 3
inst 4
inst 6
inst 7
jmp ?

sensitive instruction at 0x34fe

scanned and patched block

inst 1
inst 2
inst 3
syscall
inst 5
inst 6
syscall

0x5d3a

0x5d42

0x5d3a

dispatcher

exit points

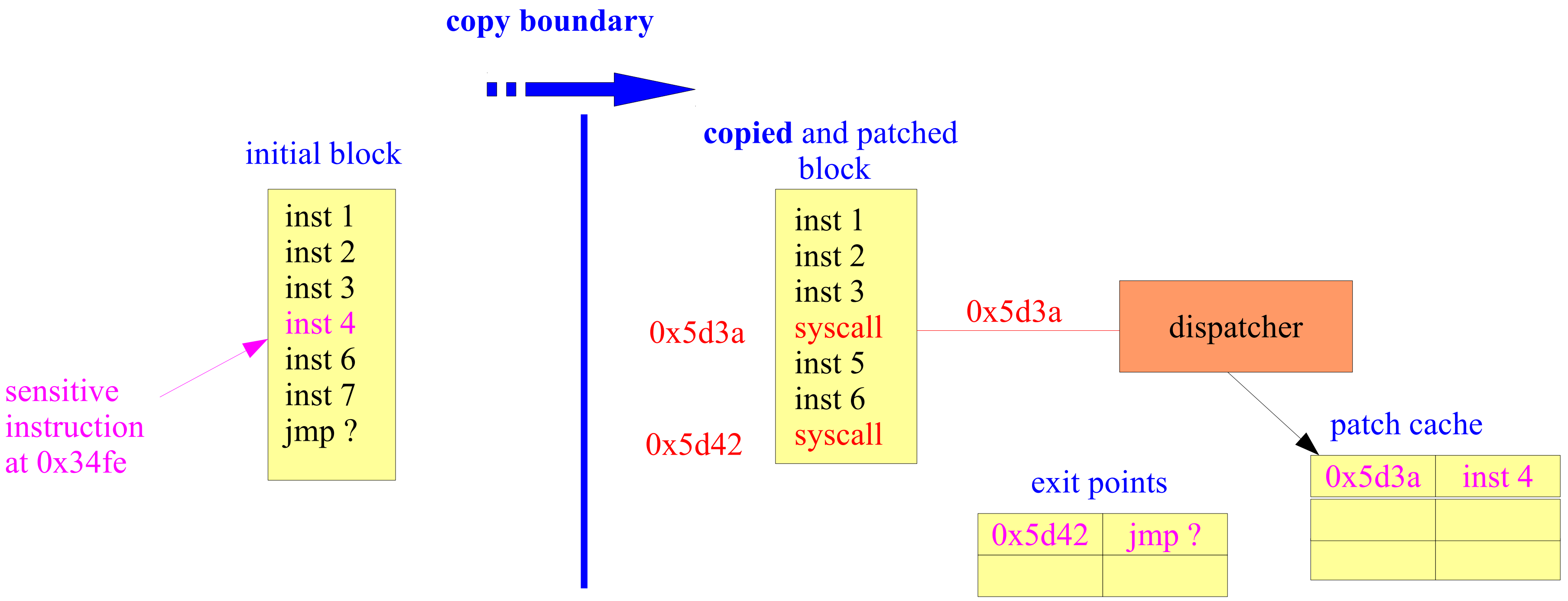
0x5d42	jmp ?

patch cache

0x5d3a	inst 4

Dynamic Binary Translation

- 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



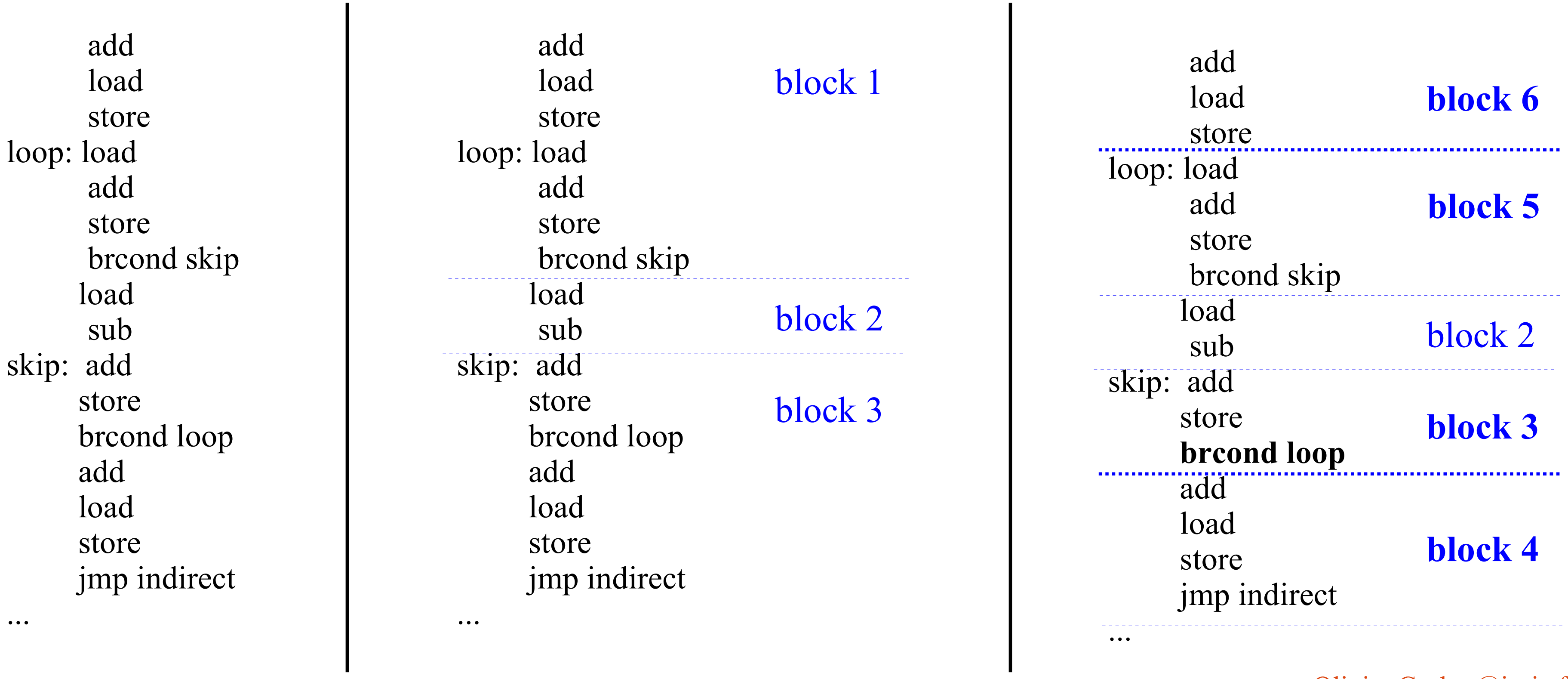
Dynamic Binary Translation

- 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
---------------------------------------
...
```

Dynamic Binary Translation

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



```
       add
       load
       store
loop:  load
       add
       store
       brcond skip
       load
       sub
skip:  add
       store
       brcond loop
       add
       load
       store
       jmp indirect
...
```

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

```
       add
       load                 block 6
       store
.......................................
loop:  load
       add                  block 5
       store
       brcond skip
---------------------------------------
       load                 block 2
       sub
---------------------------------------
skip:  add
       store                block 3
       brcond loop
.......................................
       add
       load
       store                block 4
       jmp indirect
---------------------------------------
...
```

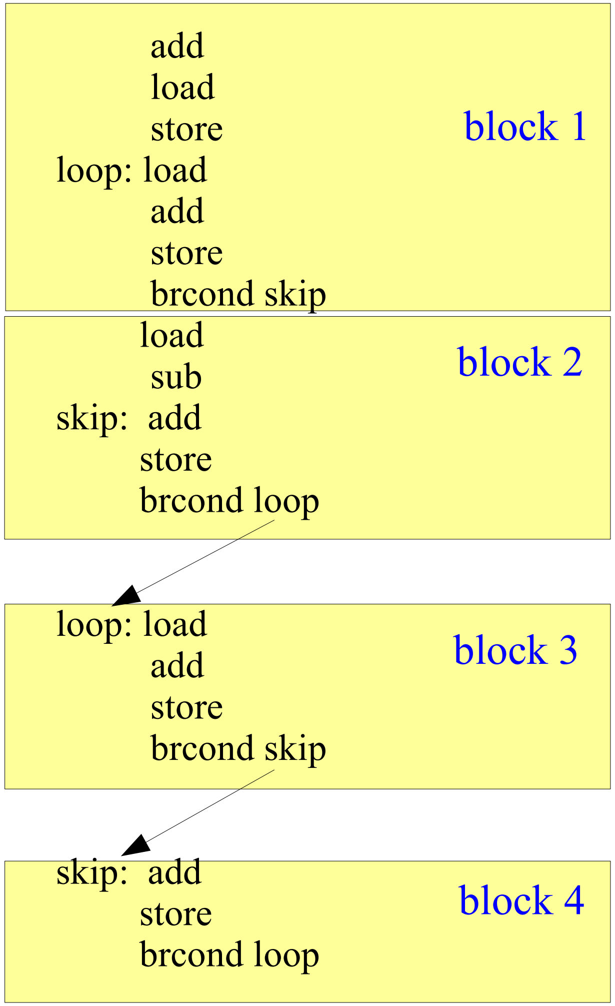
Dynamic Binary Translation

- 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

- Dynamic code blocks

```
        add
        load                  block 1
        store
loop:   load
        add                   block 2
        store
        brcond skip
        load
        sub                   block 3
skip:   add
        store                 block 4
        brcond loop
        add
        load
        store                 block 5
        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 Binary Translation

- 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

References (1/2)

- 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.

Emulating Devices

- Exampe: virtualize of a partitioned disk
  - Each guest VM sees a single disk
  - Mapped to a non-shared single partition
- Handling I/O requests
  - I/O requests are usually on contiguous addresses
  - Device drivers and hardware controller relies on this contiguity
- Challenges
  - Contiguous real address may not be contiguous in physical memory
    - VMM may have to issue multiple I/O requests on contiguous subranges
  - Some physical pages may be swapped out
    - VMM must page in the missing pages before it can request I/Os
  - Does not scale to more than a few guest VMs
    - The number of partitions is limited
    - Possible to have software-partitioning in the driver
    - Something like soft-partitions through files