Virtual Machines

Olivier Gruber, Ph.D.

Full-time Professor

Université Joseph Fourier

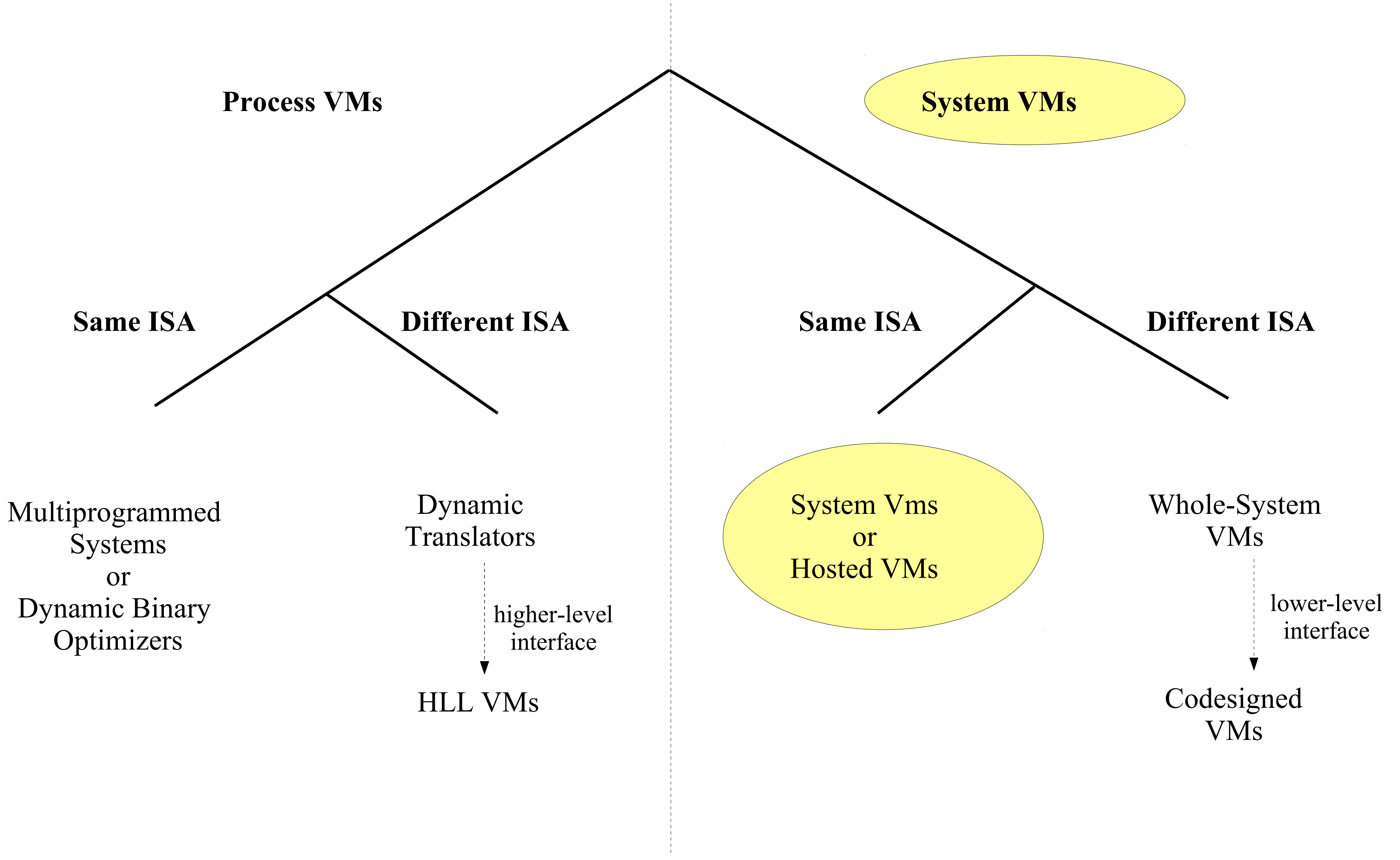
Laboratoire d'Informatique de Grenoble

Senior Resarcher

INRIA – SARDES Team

Olivier.Gruber@inria.fr

Virtual Machine Taxonomy



Outline

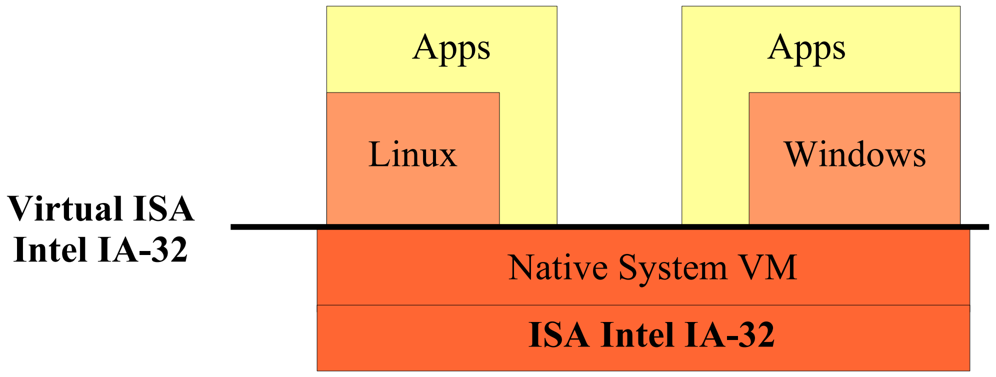
- Introduction
  - Introduce the different System Vms
  - Discuss what they are useful for
- Real-Machine Virtualization
  - Discussing efficiency
  - State management and processor virtualization
  - Memory virtualization
  - I/O virtualization

System Virtual Machines

- Two main types of same-ISA System Vms
  - **Native** VMs, also known as **bare-metal**, or **Type-I**
  - Hosted VMs, also known as **Type-II**
- Virtual Machine Monitor (**VMM**) or Hypervisor
  - Generic term for either native or hosted Vms
  - **It is a virtual machine that monitors other virtual machines**
- **Virtualize a "real machine"**
  - Run multiple guest software in complete isolation
  - Each guest software believe they are running on a real machine
  - Performance is everything... zero overhead goal...

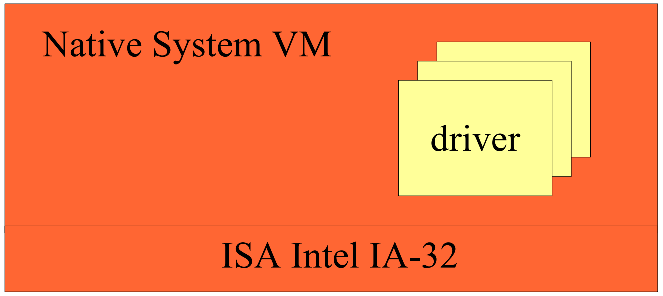
System Virtual Machines

- Native system virtual machines
  - Directly on bare hardware
    - Multi-tasking of virtual machines
    - Provides the illusion of a completely-owned real machine
    - To an entire system image, with the same ISA
  - Run in kernel mode
    - **All other software runs in user mode**
    - Including guest operating systems
    - All traps and interrupts go to the VMM



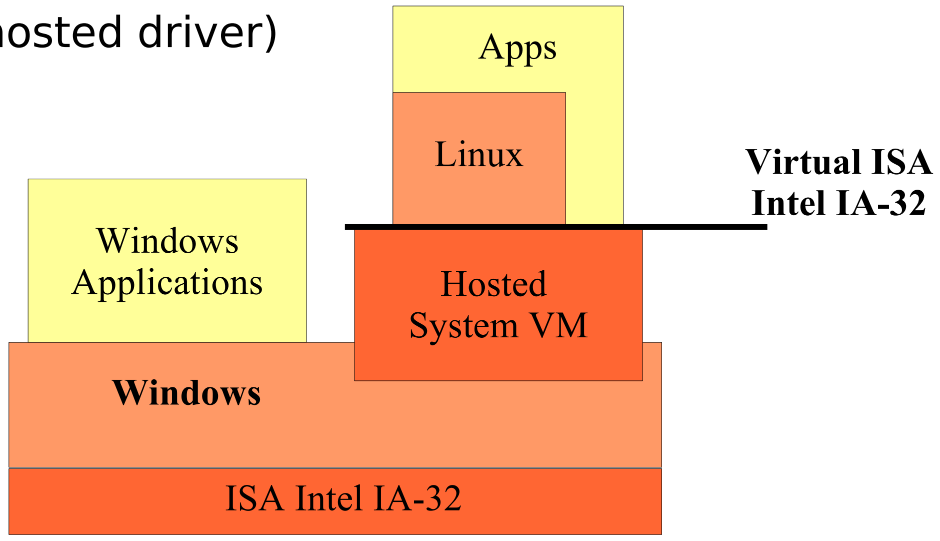
System Virtual Machines

- Discussing native system Vms
  - Must be installed on bare metal
    - Need to wipe out any pre-installed operating system
  - Must have the adequate drivers
    - The VMM virtualizes the I/O devices
      - It is therefore the VMM that interfaces directly with the I/O devices
      - So it needs drivers for the hardware
    - The VMM can virtualizes
      - 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



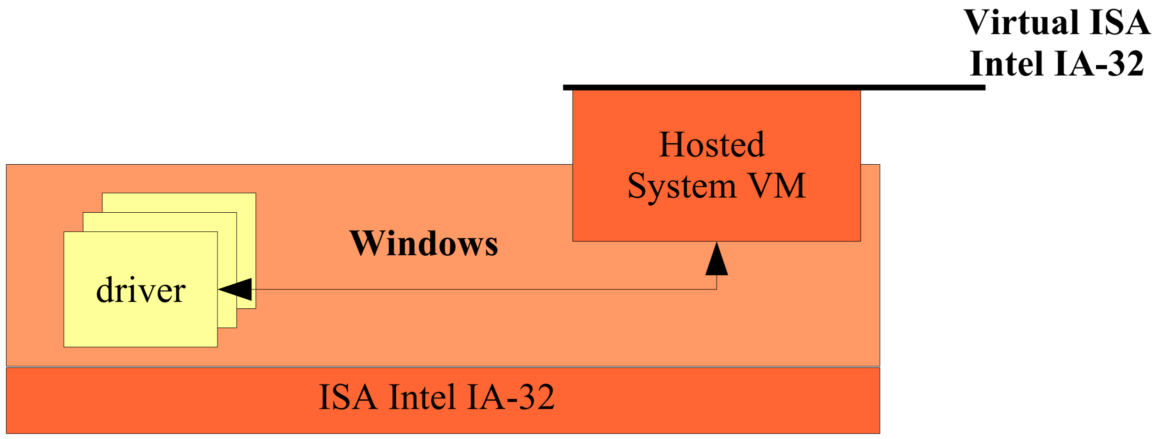
System Virtual Machines

- Hosted system virtual machines
  - Installed on a host operating system
    - As a regular application with kernel drivers
  - Host a single guest operating system
    - Expecting the same ISA as the real ISA
    - Example is VMware
  - Runs in mix mode
    - Most of the VM code runs in user mode
    - Some in kernel mode (through hosted driver)



System Virtual Machines

- Discussing hosted system VMs
  - Can be installed on an existing operating system
    - End users do not need to wipe-out their disk
    - Can leverage the drivers from the host operating system
    - Can still virtualize new devices or more generic devices
  - Can integrate in the host operating system
    - Can appear as a window on the host desktop
    - Can even provide cut&paste abilities



System Virtual Machines

- Fashionable in the 1970s
  - Hardware-software independence
  - Easier sharing of expensive mainframes
  - Potentially time-sharing single-user single-task operation systems
- When out of fashion in the 1980s
  - Not quite for everybody... still used in the IBM AS/400 or IBM 390
  - Time-sharing OSes became commonplace
  - It was the era of one hardware ran one software stack
- Back in fashion since the early 2000s
  - Freedom marches on... operating system lock down is less accepted
  - Green-IT in Cloud computing... multiplexing soft machines
  - Pragmatic approach to address the drawbacks of ''traditional operating systems''

System Virtual Machines

- System encapsulation
  - Convenient way of encapsulating the state of an entire machine
  - Facilitates checkpointing, suspend/resume
  - Portability (virtual appliances)
- System Migration
  - Ubiquitous management platform
  - Load balancing
  - High availability
- System Sandboxing
  - The system VM isolates guest VMs
  - Usefull for instance for application-hosting in server farms
  - Provides isolation guarantees to end users

System Virtual Machines

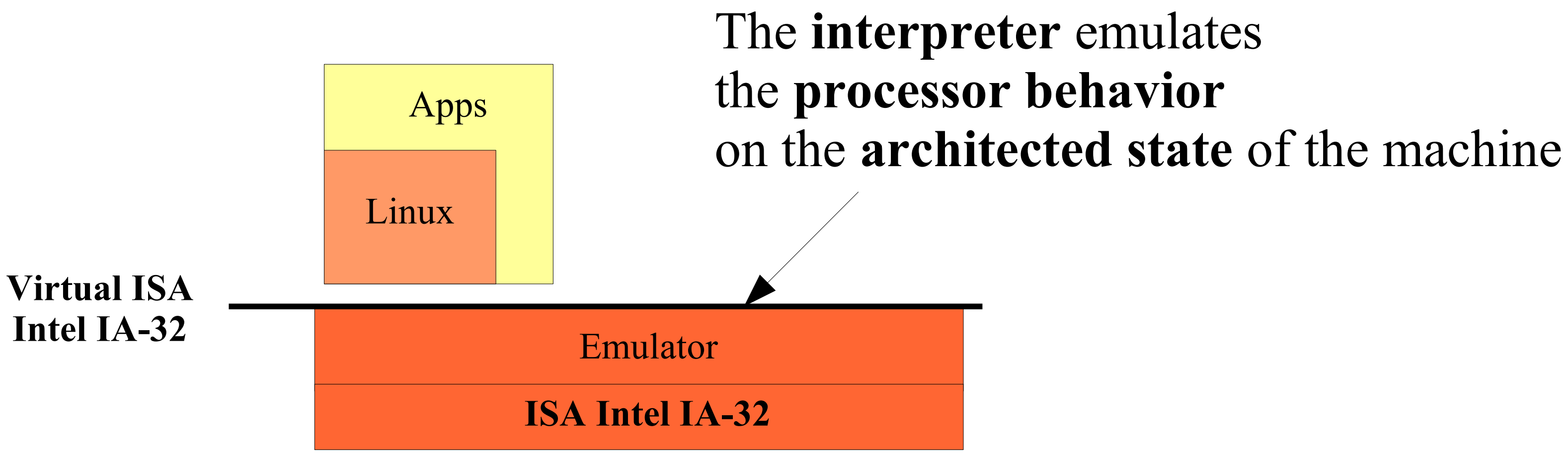
- Software management
  - Cross-development
    - Building and testing across different operating systems
  - Help-desk
    - Bring up a virtual machine that mirrors the client configuration
  - Operating system instrumentation and research
    - Just simpler when virtualized (debugging, logging, monitoring, etc.)
    - Either using native or hosted VMs

Outline

- Introduction
  - Introduce the different System Vms
  - Discuss what they are useful for
- Real-Machine Virtualization
  - Discussing design and efficiency
  - State management and processor virtualization
  - Memory virtualization
  - I/O virtualization

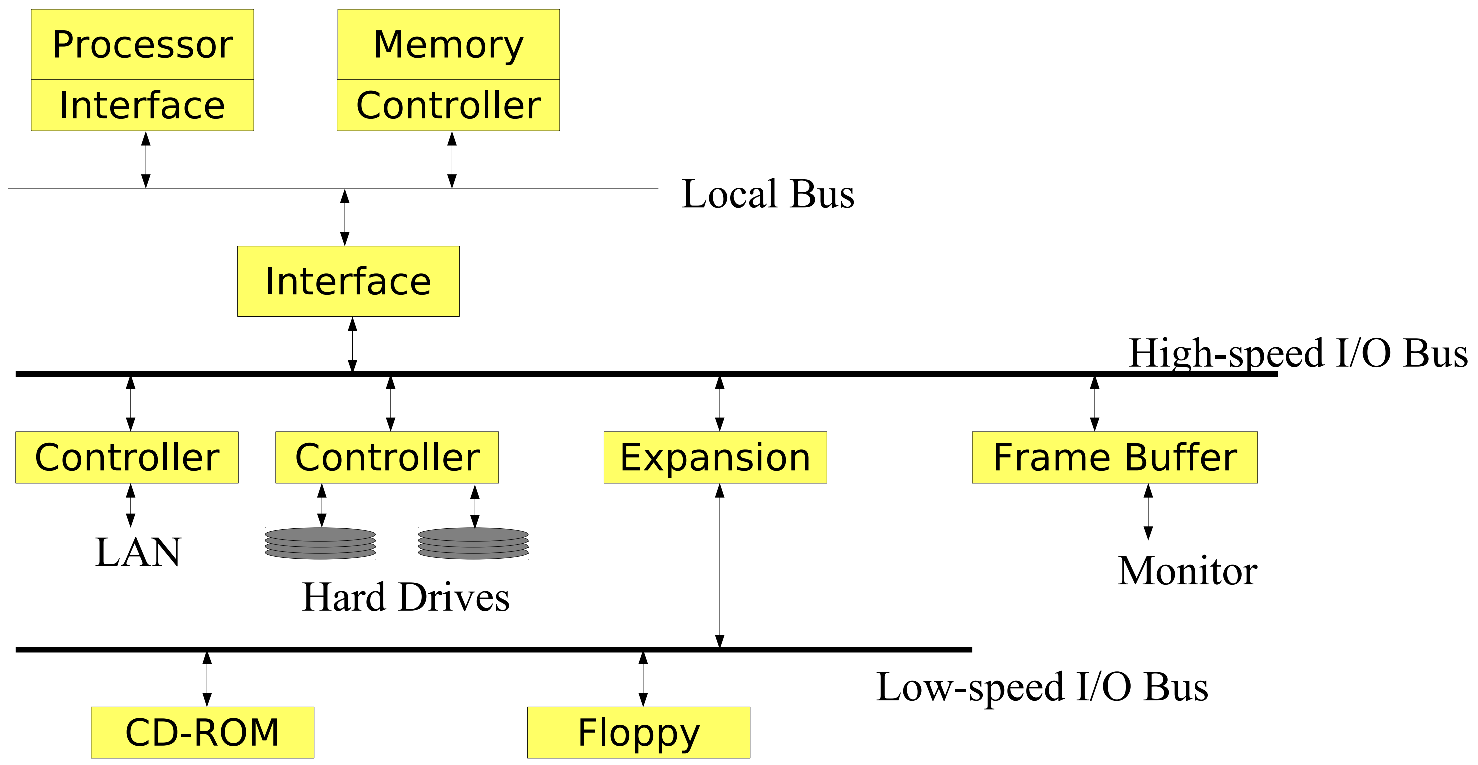
Real Machine Emulation

- Through emulation, we can always implement a system VM
  - Emulation means either **interpretation** or **binary translation**
  - **Provides the illusion of a completely-owned real machine**
  - With the same ISA or not
- The bare-metal case:


The **interpreter** emulates
the **processor behavior**
on the **architected state** of the machine
Apps
Linux
Virtual ISA
Intel IA-32
Emulator
ISA Intel IA-32

Architected State

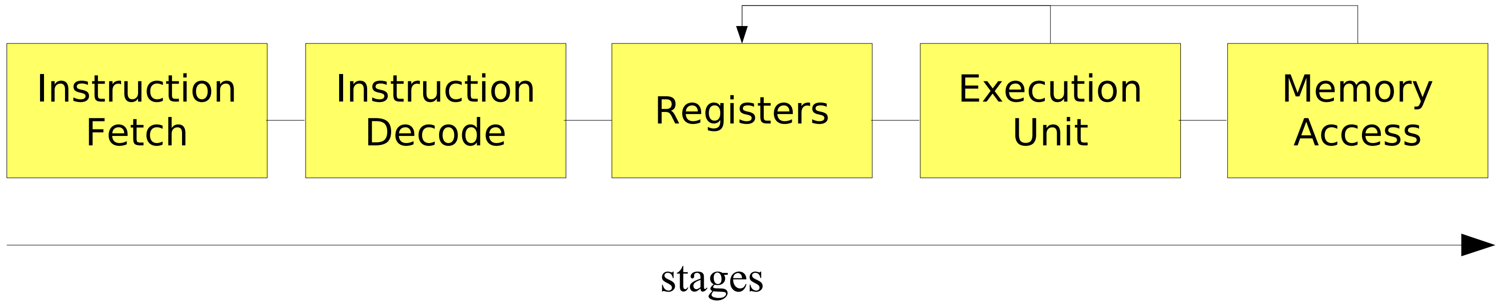
- It is the state of the machine...
  - Contained in and maintained by the hardware resources of the machine
  - We have a hierarchy of such resources



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
  - Device states
    - The state for each device in use

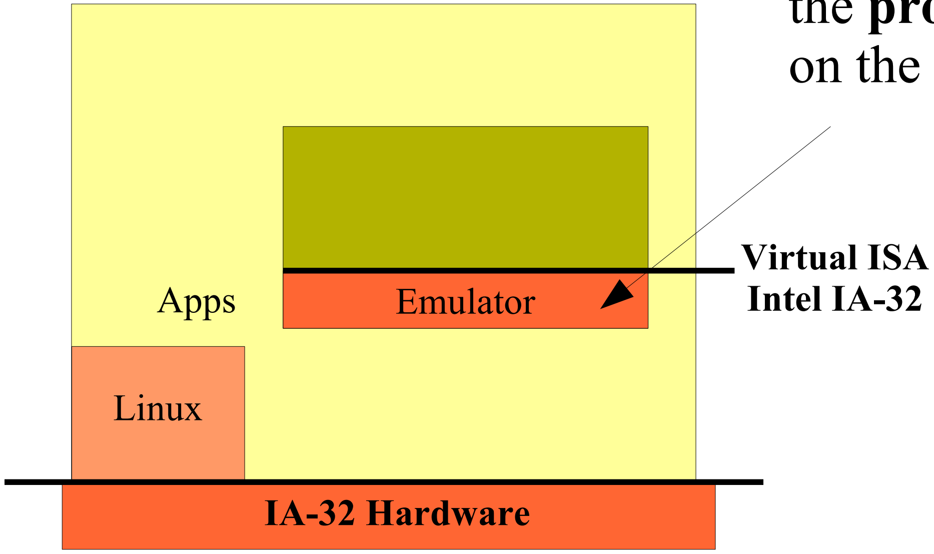
The interpreter emulates the processor on the architected state



Interpretation

- A simple design for the hosted case
  - Just a C program, like any interpreter...
  - Architected state:
    - C data structures
    - Example: a byte[] for the memory, int[] for registers, 32bits for processor flags
  - Interpreter:
    - Fetch-decode-issue loop

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

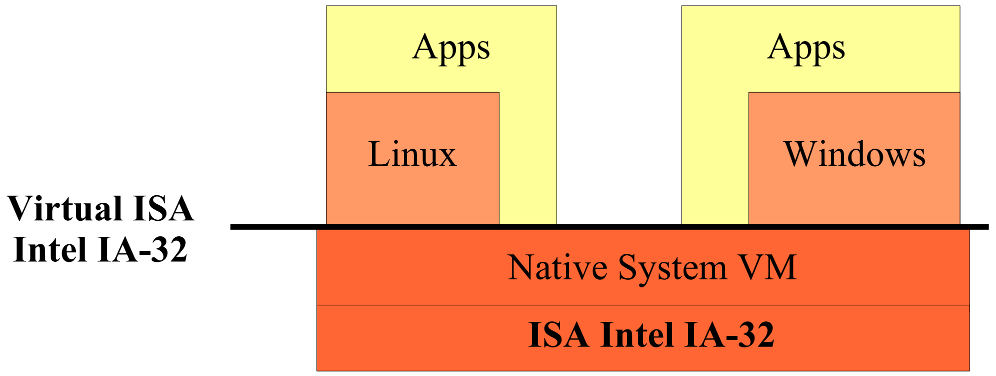


Discussing Efficiency

- What about performance?
  - It is a matter of perspective...
  - System C simulation
    - Very precise hardware simulation, very slow
  - Interpreters (e.g. Bosch)
    - Emulate different processors
    - Good simulation of hardware behavior
    - Quite slow
  - Binary translation (QEMU)
    - Could also emulate different hardware
    - Much faster on the same ISA
    - Good top speed, average is 5 to 20 times slower than native speed
  - Hardware-assisted virtualization (Xen or Oracle VirtualBox)
    - All sensitive instructions trap, close to native speed
    - Often associated with para-virtualization for even greater speed

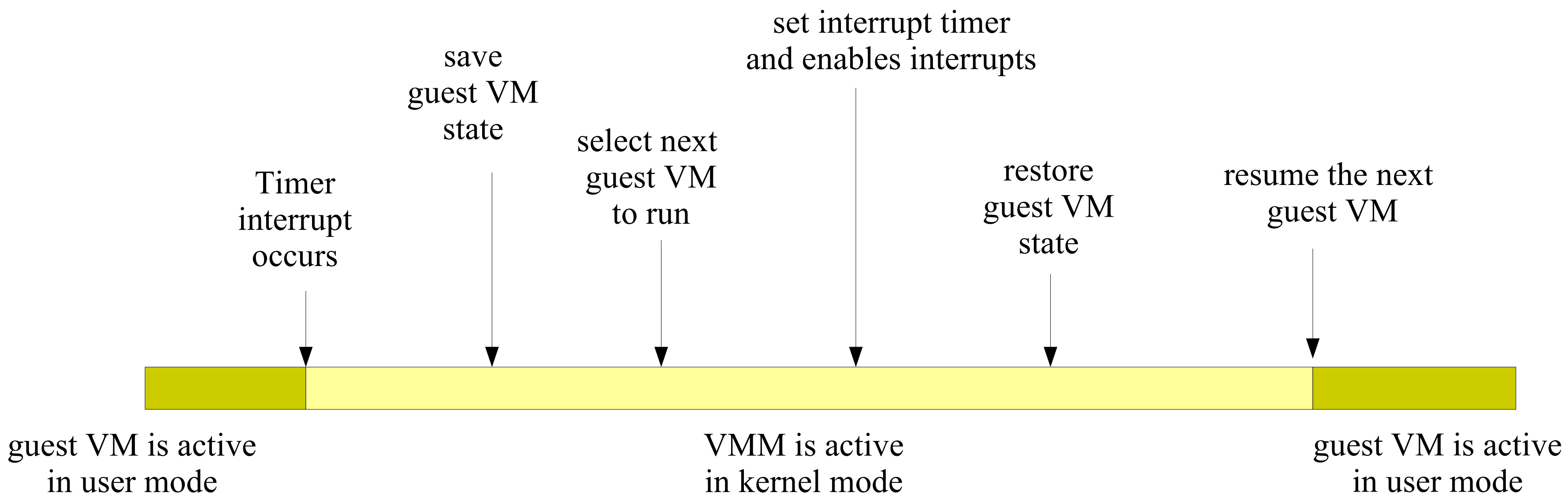
Time-Sharing Guest VMs

- Assuming a single-core machine
  - Very similar issues to time-sharing applications
  - Each guest VM needs a complete state
  - Each guest VM needs to be scheduled for execution for a time slice
  - At each scheduling, we need to switch architected states



Time-Sharing Guest VMs

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


save guest VM state
set interrupt timer and enables interrupts
select next guest VM to run
Timer interrupt occurs
restore guest VM state
resume the next guest VM
guest VM is active in user mode
VMM is active in kernel mode
guest VM is active in user mode

What is the difference with multiplexing applications then?

Traditional Multi-tasking

- Traditional design
  - Use a timer to share the processor
  - Use an MMU to virtualize the shared memory
  - Use system calls to virtualize resources
- Relies on two modes of operation
  - Kernel mode for the operating system
  - User mode for the applications
  - Priviledged instructions
    - Either not allowed at all for applications (traps)
      - Attempting to set the page table register traps
    - Or rendered innocuous (modified behavior)
      - Example loading the status register in user mode does not change all the flags, especially not the processor mode (kernel or user).

Multiplexing Guest VMs

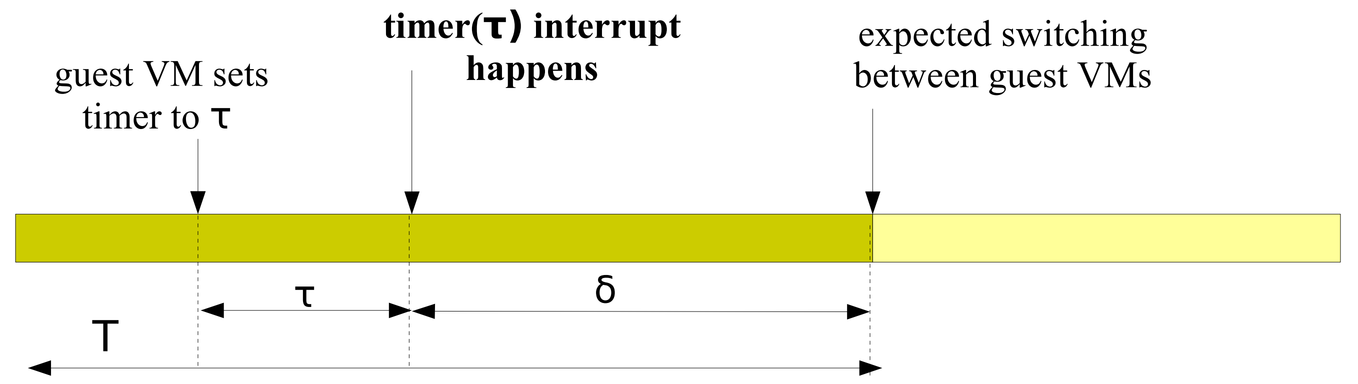
- Multiplexing guest VMs
  - We are multiplexing operating systems that are expecting to run in kernel mode and actually use priviledged instructions
- How do we virtualize then?
  - Only the VMM runs in kernel mode
  - Guest VMs run in user mode
  - VMM emulates all sensitive instructions
  - Let's assume that all sensitive instructions trap in user mode
- Let's illustrate this
  - With sharing the processor between guest VMs
  - Same approach as for multiplexing applications
  - Save and restore architected state upon switching guest Vms
  - Use a timer for getting back the processor

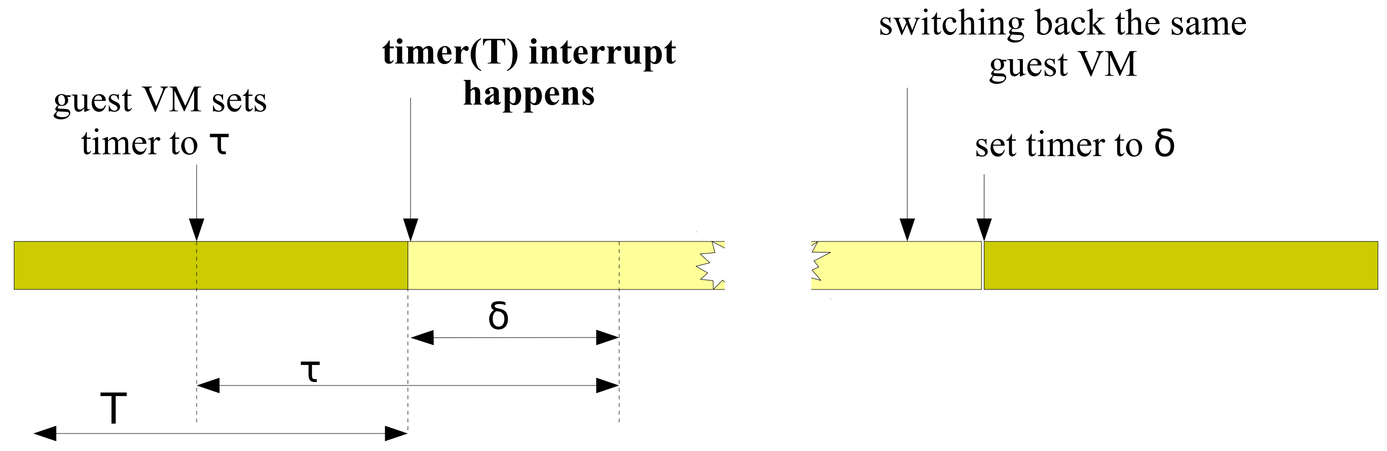
System Virtual Machines

- Protecting the timer
  - Each guest VM is an OS and it is expecting to use the timer
  - Essentially for scheduling its own applications
  - So it sets the timer interval and has its own interrupt handler
- Emulation is needed
  - Through emulation, we preserve the control of the timer
    - Each guest VM has a time slice of T
  - During T, the guest VM sets the timer to the value τ
    - If τ is smaller than the remaining time on T
      - We set the timer to τ
      - We record the remaining time δ of T
    - If τ is larger than the remaining time on T
      - We set the timer to the remaining time δ of T
      - We remember in guest VM state the value (τ-δ)

System Virtual Machines

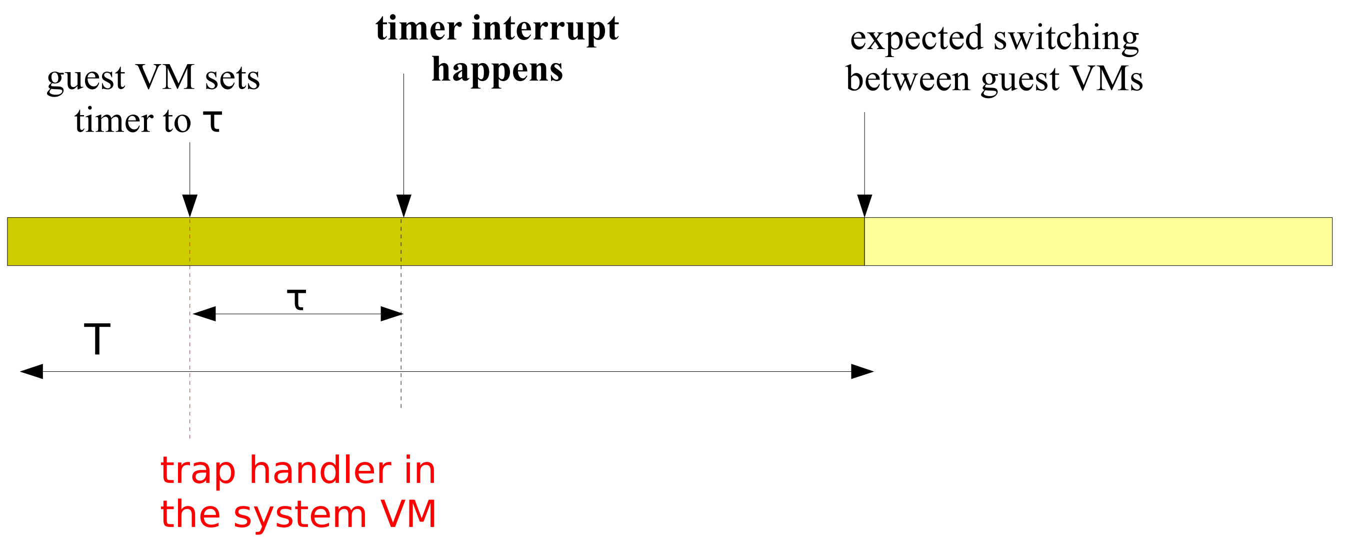
- Timer emulation


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



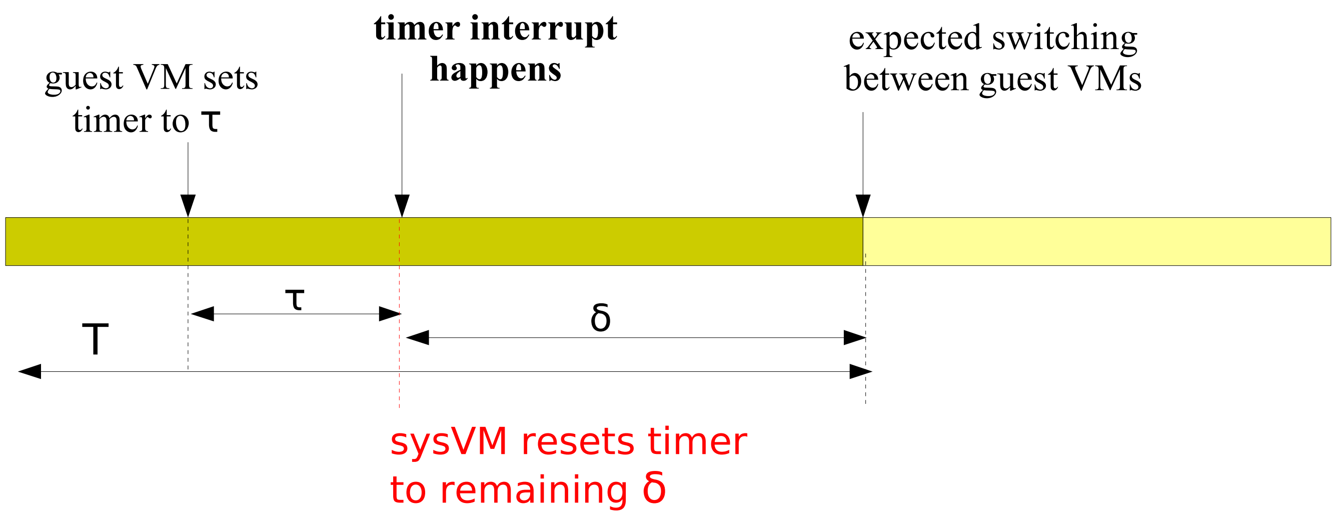
System Virtual Machines

- Timer emulation (continues...)
  - The guest VM runs in user mode
    - We need the load instruction of the timer register to trap
    - So that the system VM regains control
  - In the trap handler
    - SysVM decides what value to actually set: τ or δ (the remaining of T)


guest VM sets timer to τ
timer interrupt happens
expected switching between guest VMs
τ
T
trap handler in the system VM

System Virtual Machines

- Timer emulation (continues...)
  - In the interrupt handler
    - Interrupt back to kernel mode, to the system VM handler
    - Need to test if it is the end of the time slice T
    - If it is not the end of the time slice T
      - Resets the timer to δ (the remaining of the time slice T)
      - Pass the interrupt to the timer interrupt handler of the guest VM
      - It knows where because the VMM emulates the interrupt vector


guest VM sets timer to τ
timer interrupt happens
expected switching between guest VMs
τ
δ
T
sysVM resets timer to remaining δ

System Virtual Machines

- What are sensitive instructions?
  - Guest VM run entirely in user mode
    - But it believes that it runs in kernel mode
  - Guest VM also expects to have full control of the machine
    - It will use kernel-level instructions to manipulate resources
    - Changing page table pointers
    - Changing between user and kernel modes
    - Etc.
  - **Sensitive instructions** are precisely those that must be emulated
- Does that mean that all instructions have to be emulated?
  - Through emulation, we can always implement a system VM
    - Emulation means either interpretation or binary translation or a mix of both
  - But at a significant performance cost
    - Is there another way?

System Virtual Machines

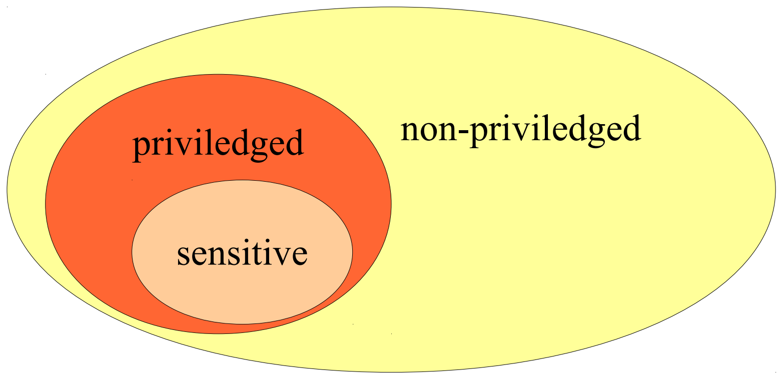
- Native execution
  - Direct execution of the guest instructions
    - Require that the virtual ISA is the same as the real ISA
  - Some instructions may still need to be emulated
    - Depends on the real ISA of the machine
    - Some are well-suited for system virtualization, others are not
- Discussing sensitive instructions
  - **Control-sensitive instructions**
    - Attempt to change the configuration of resources in the system
  - **Behavior-sensitive instructions**
    - Behavior depends on the configuration of resources

System Virtual Machines

- **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
    - Translating virtual memory addresses
    - Setting processor flags whose behavior depends on the system mode
      - E.g. the interrupt enable/disable flag can only be changed in system mode

System Virtual Machines

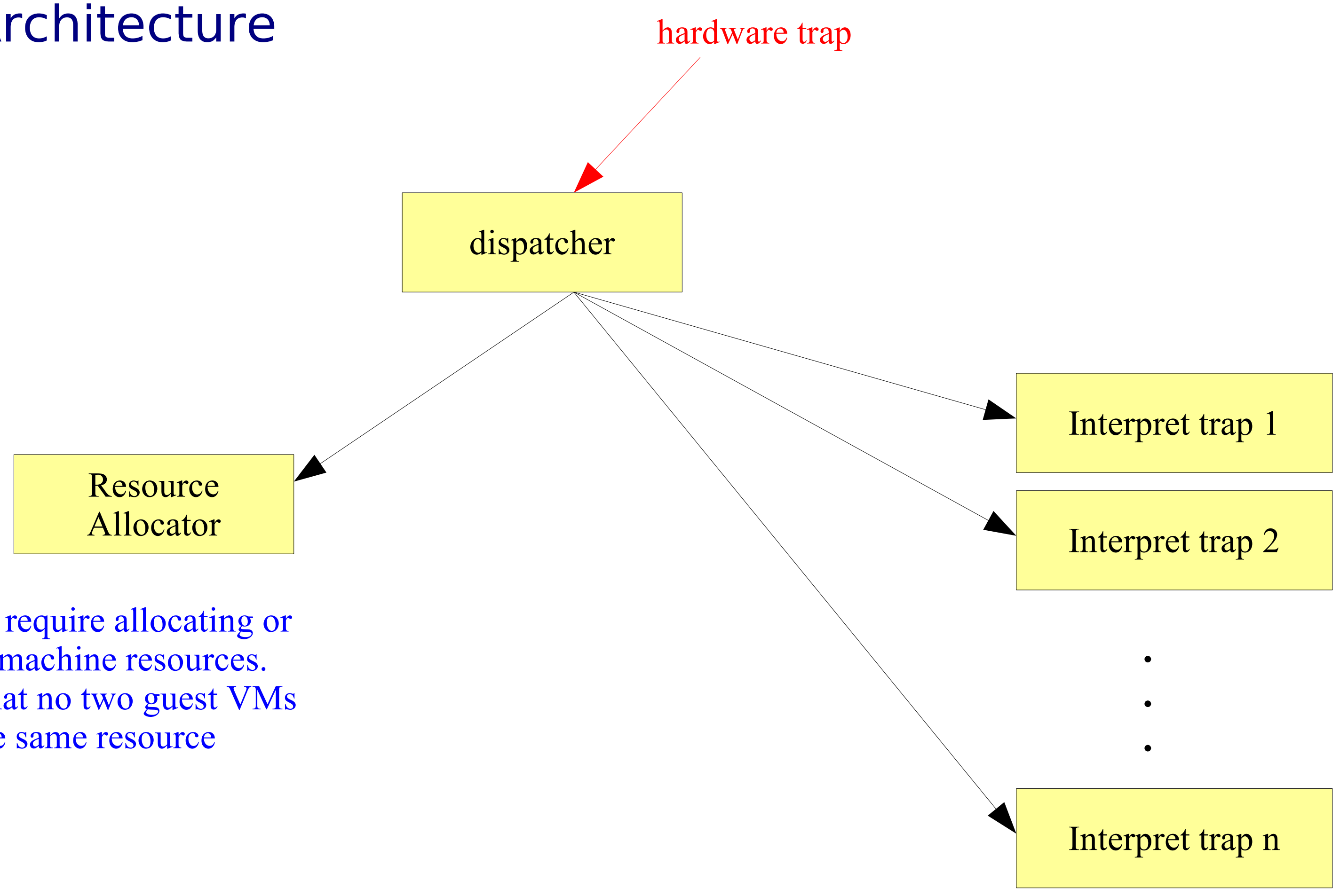
- Virtualization theorem
  - *An efficient system virtual machine may be constructed if the set of sensitive instructions for the real ISA is a subset of the priviledged instructions.*



  - **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

System Virtual Machines

- SysVM Architecture



These traps require allocating or changing machine resources.
It ensures that no two guest VMs get the same resource

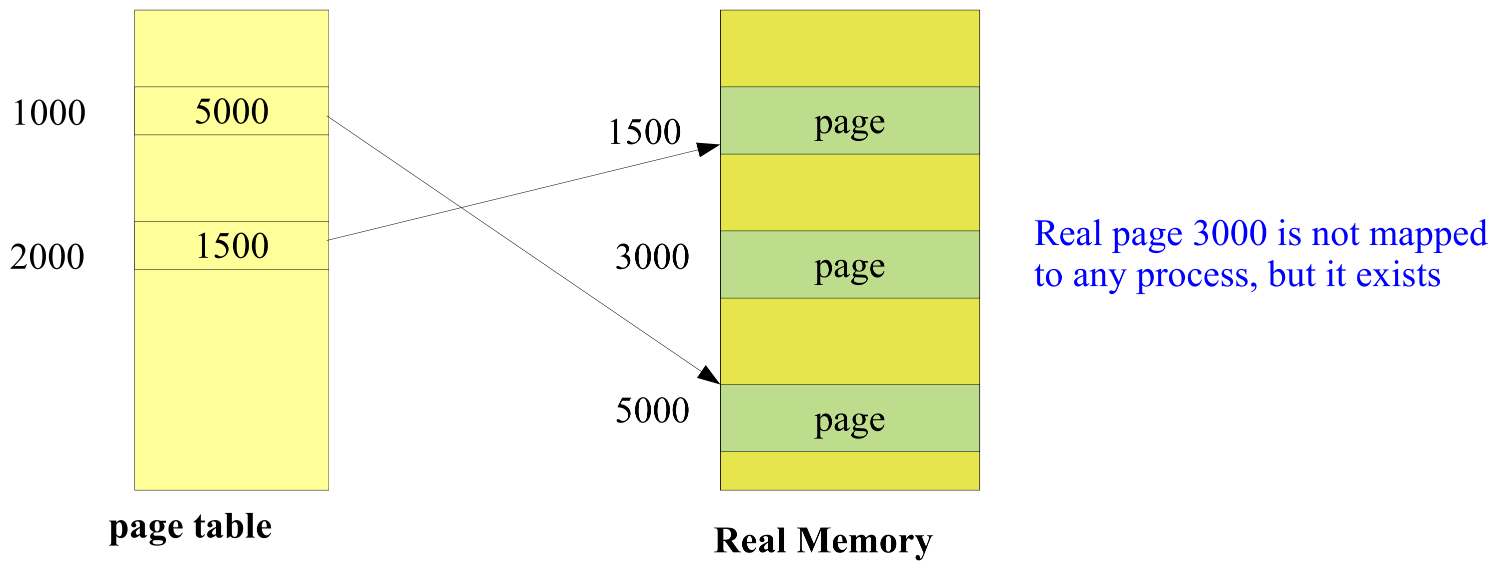
These traps do not change machine resources but accesses priviledged resources, they are emulated
on the guest VM architected state

System Virtual Machines

- Virtualize physical memory
  - Traditionally
    - Physical memory is virtualized through MMUs
    - Providing applications with the illusion of non-shared memory
  - With 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...

System Virtual Machines

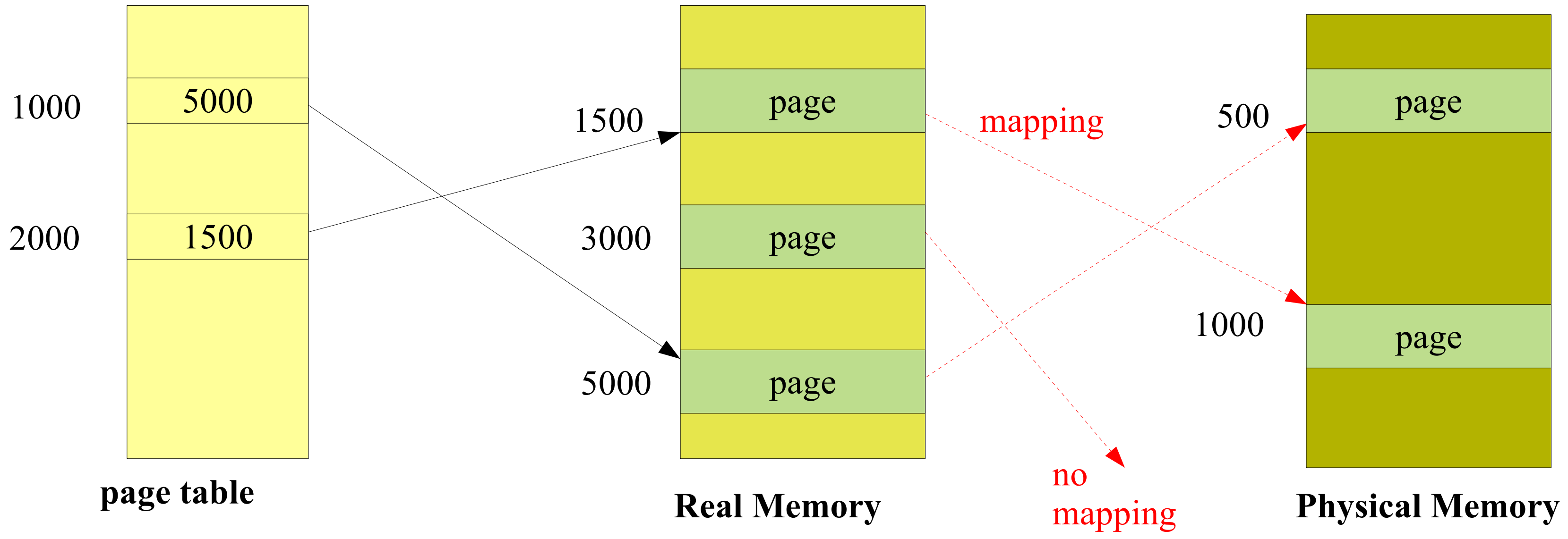
- Virtualize an architected page table
  - A guest VM view
    - **Real memory**
    - **Virtual memory per process**



Virtual memory
for a process P

System Virtual Machines

- Virtualize an architected page table
  - A guest VM view
    - Real memory and virtual memory per process
  - VMM view
    - Introduces **physical memory**
    - Real memory pages are held in physical memory

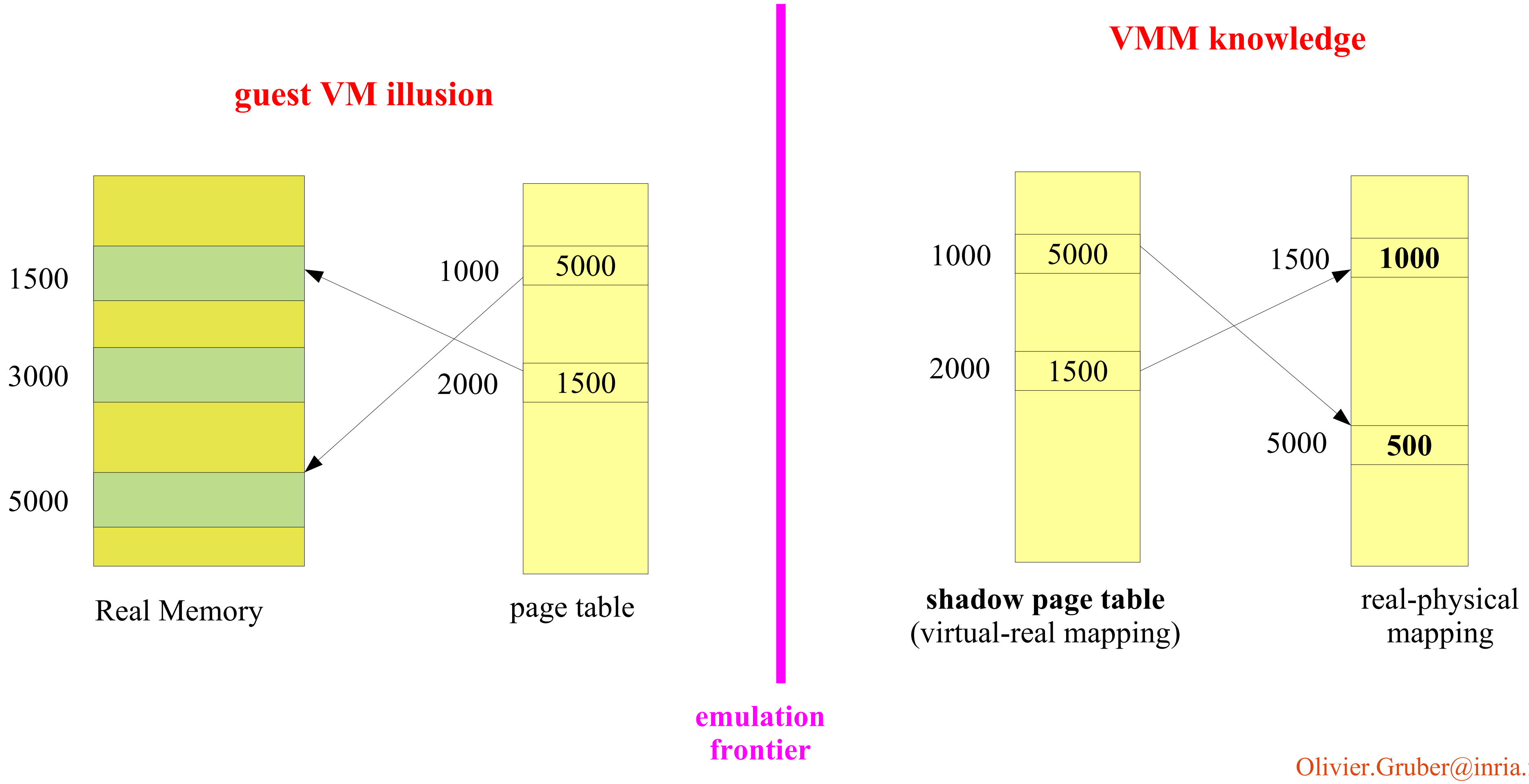


System Virtual Machines

- Virtualize an architected page table
  - A story of make believe through emulation
    - Load and store instructions in the page table are priviledged
    - They will trap when used by the guest VM in user mode
  - The VMM emulates the virtual-to-real-to-physical mapping
    - Each guest VM
      - Track page tables that it manages
      - Setting the page table pointer is priviledged
    - VMM
      - 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

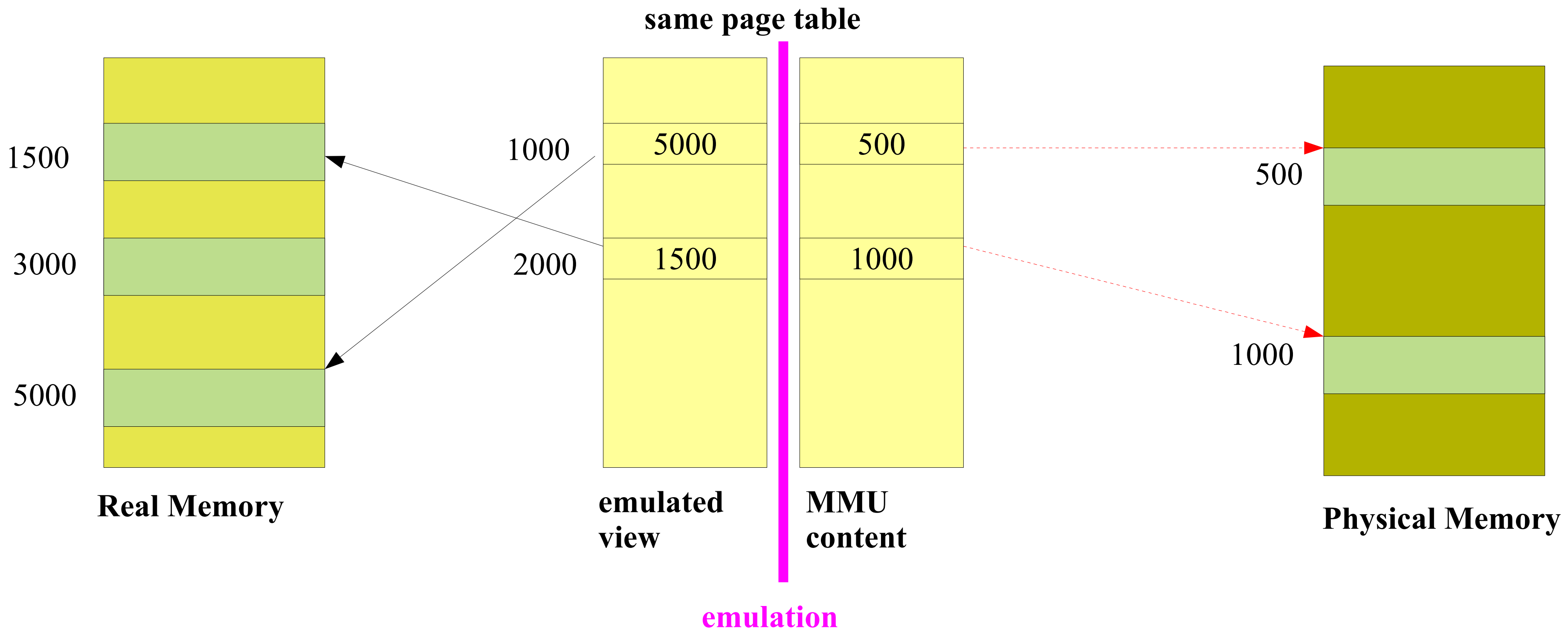
System Virtual Machines

- Virtualize an architected page table
  - Architected state per guest VM
    - Page tables (different virtual-to-real mappings)
    - One real-to-physical mapping



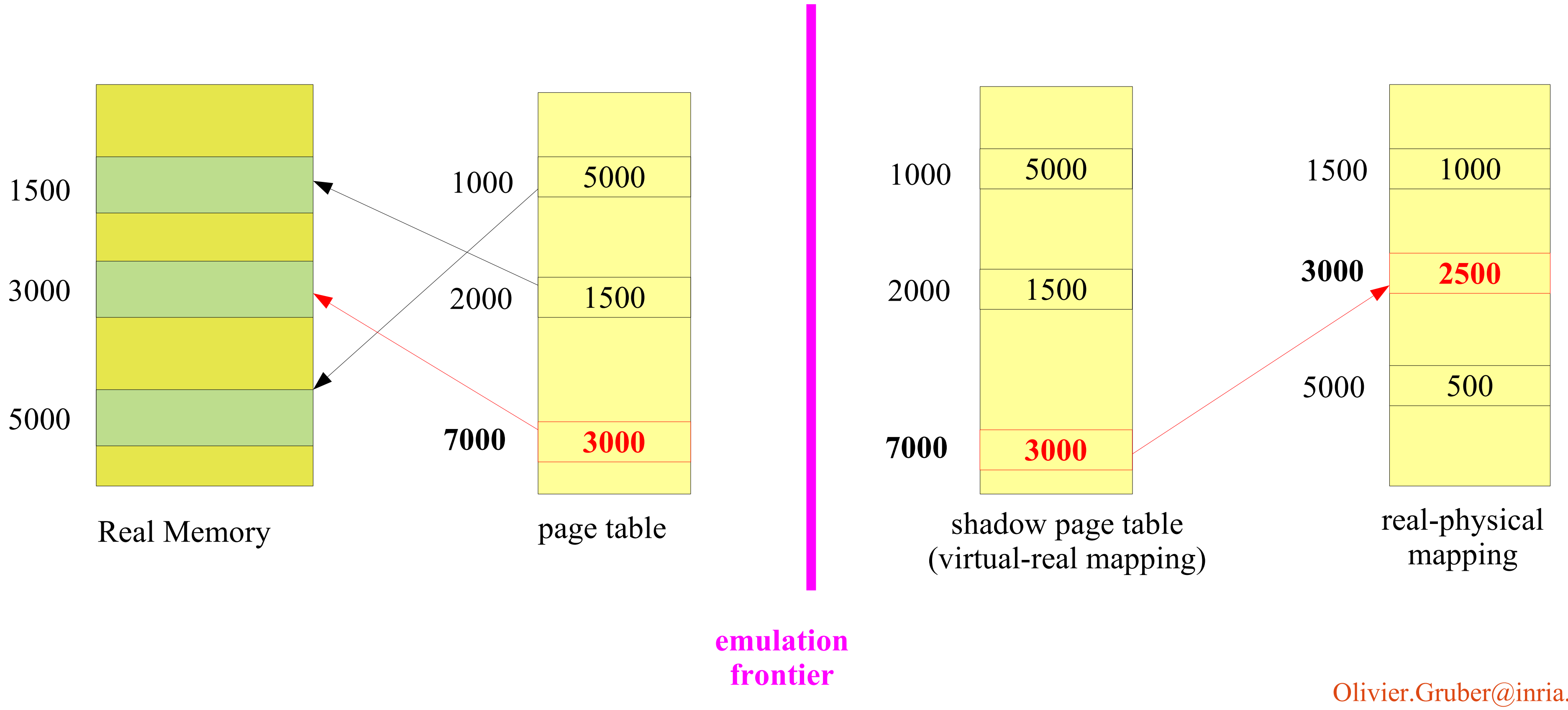
System Virtual Machines

- Virtualize an architected page table
  - 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



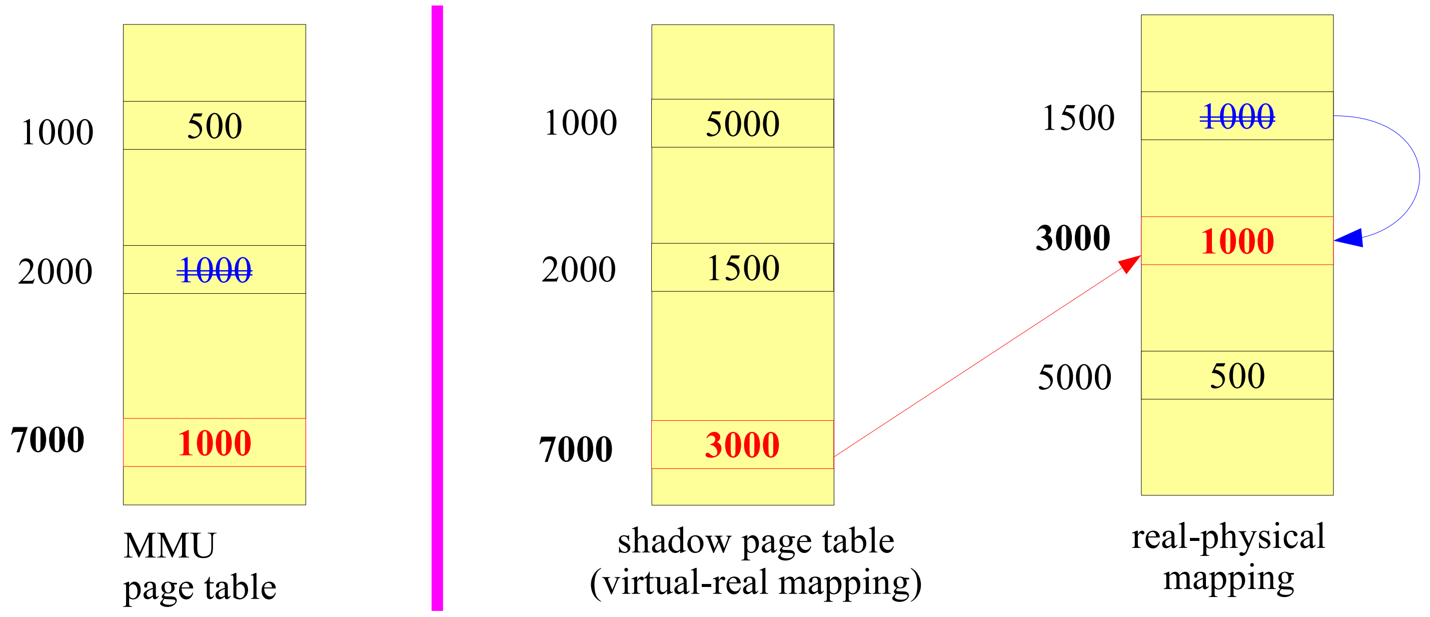
System Virtual Machines

- Virtualize an architected page table
  - 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)



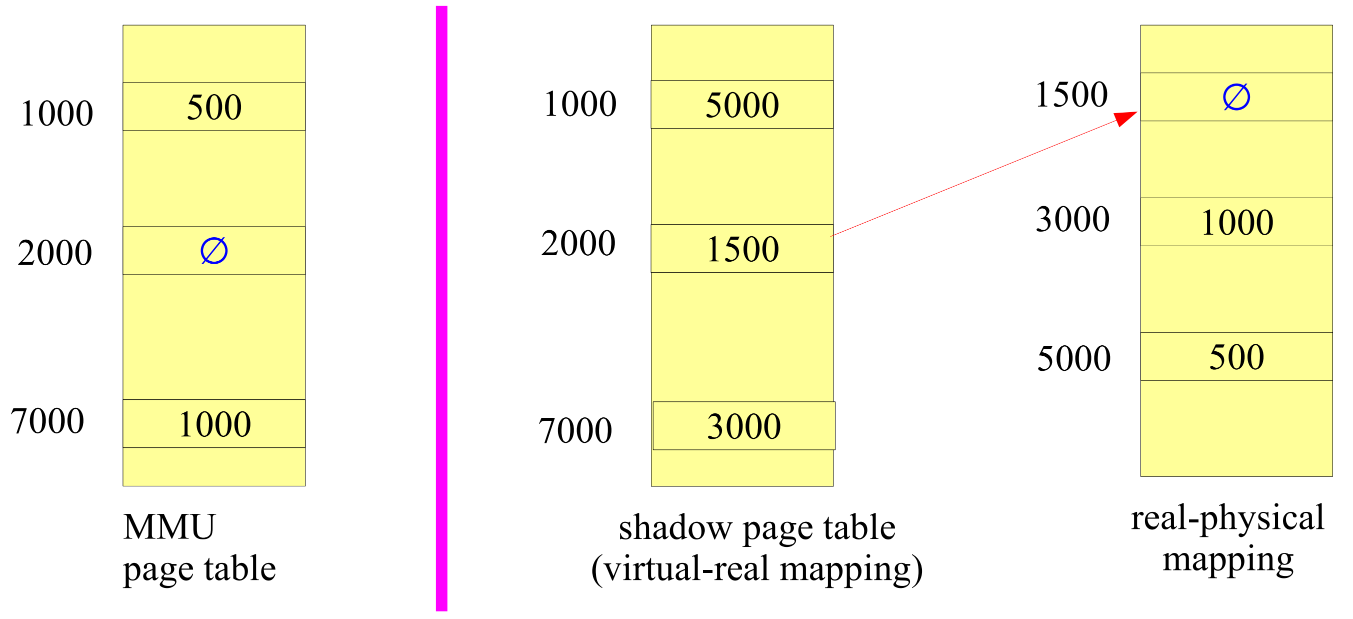
System Virtual Machines

- Virtualize an architected page table
  - 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)



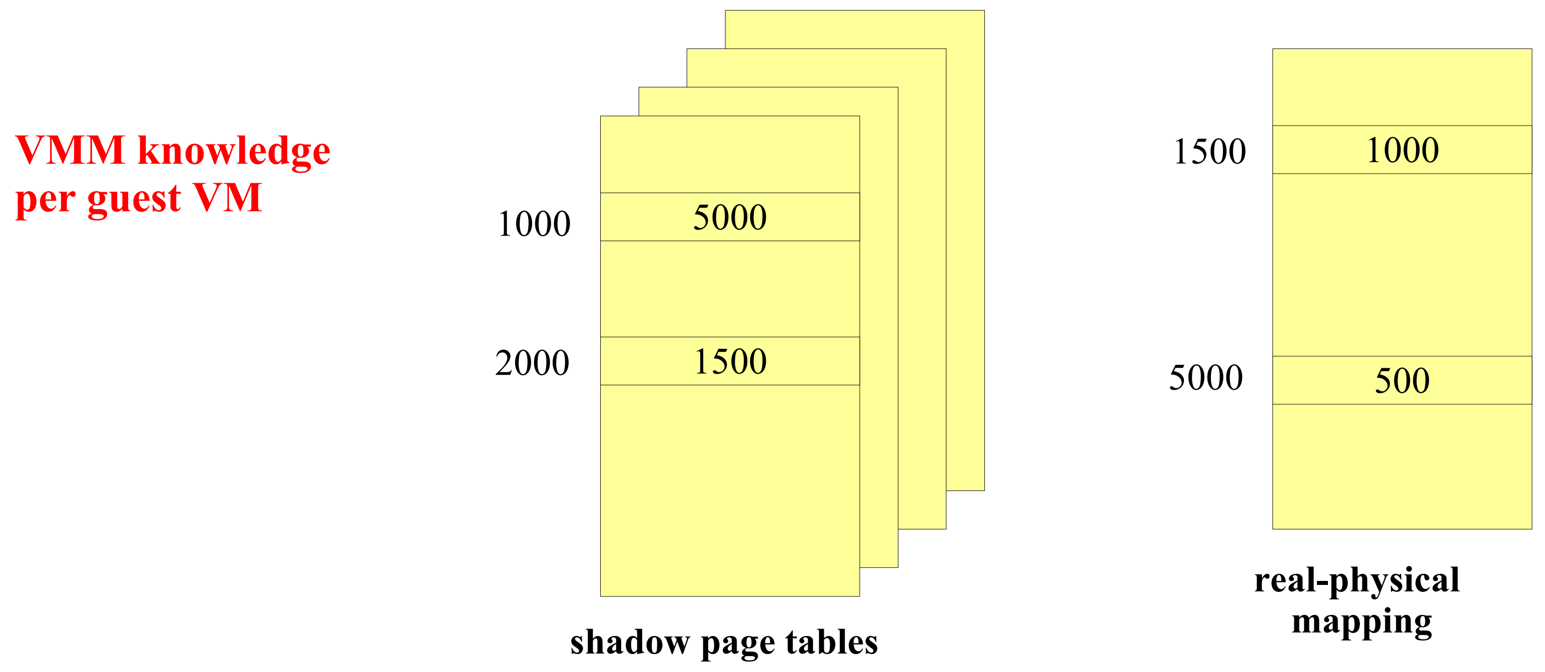
System Virtual Machines

- Virtualize an architected page table
  - 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



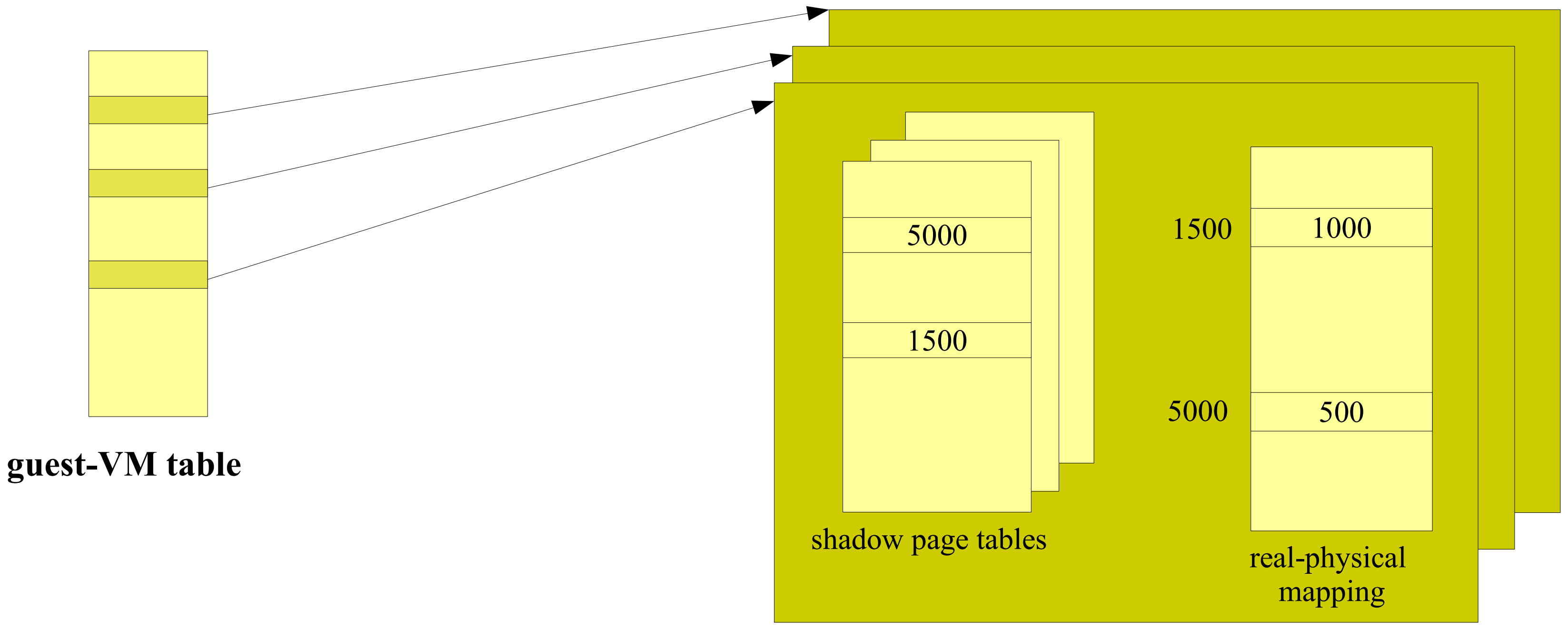
System Virtual Machines

- Virtualize an architected page table
  - 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



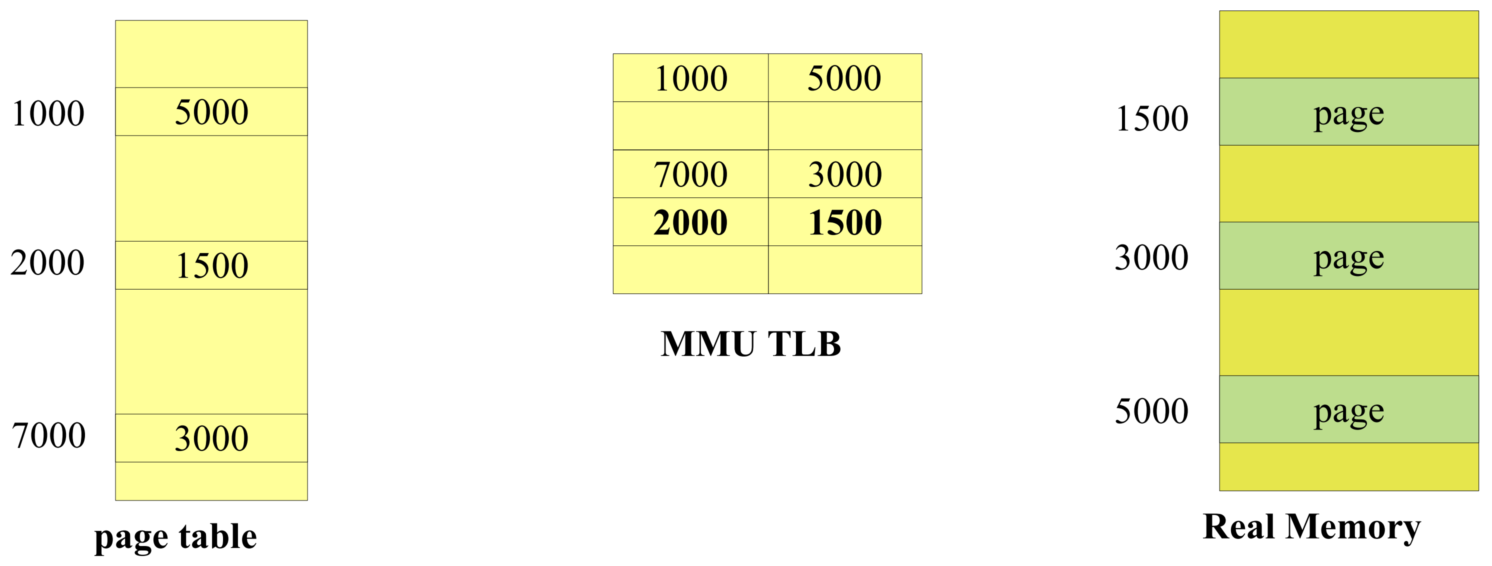
System Virtual Machines

- Virtualize an architected page table
  - Switching between guest VMs
    - Switching the architected state for the new guest VM
    - That includes:
      - the correct page table that the guest VM expects
      - The real-physical mapping



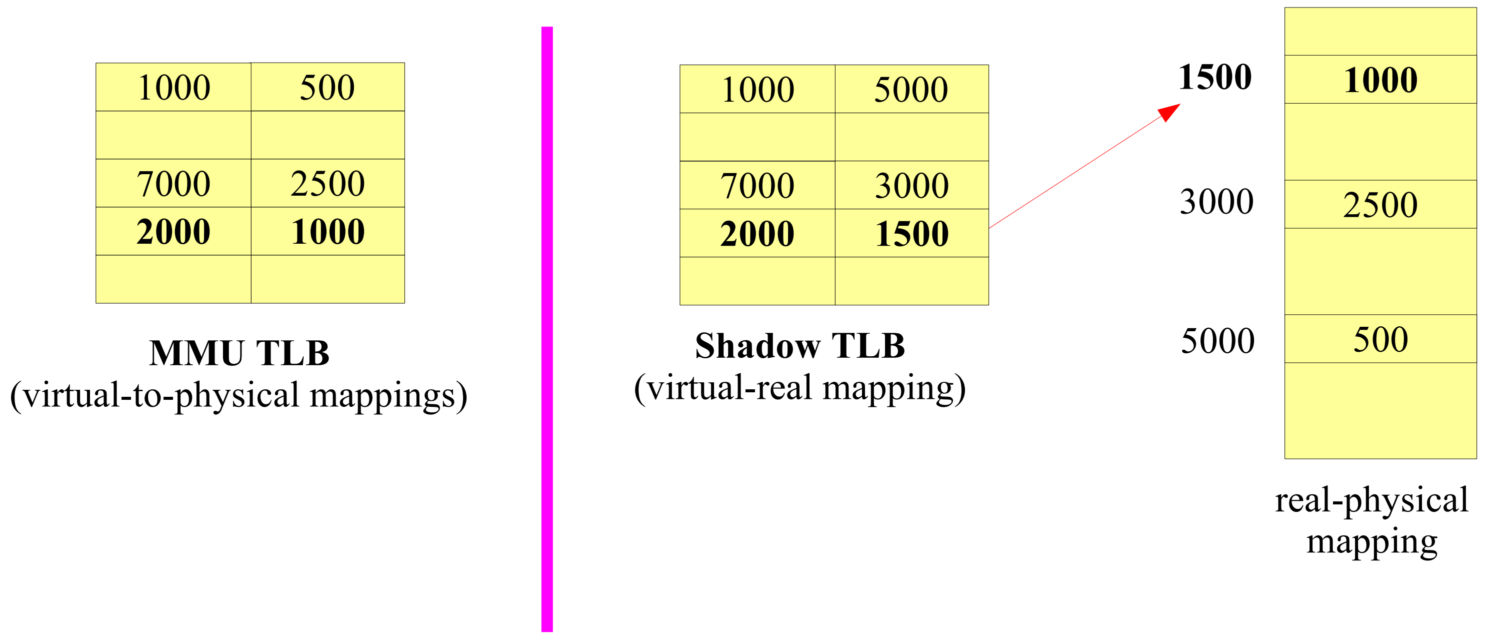
System Virtual Machines

- Virtualize an architected TLB
  - 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

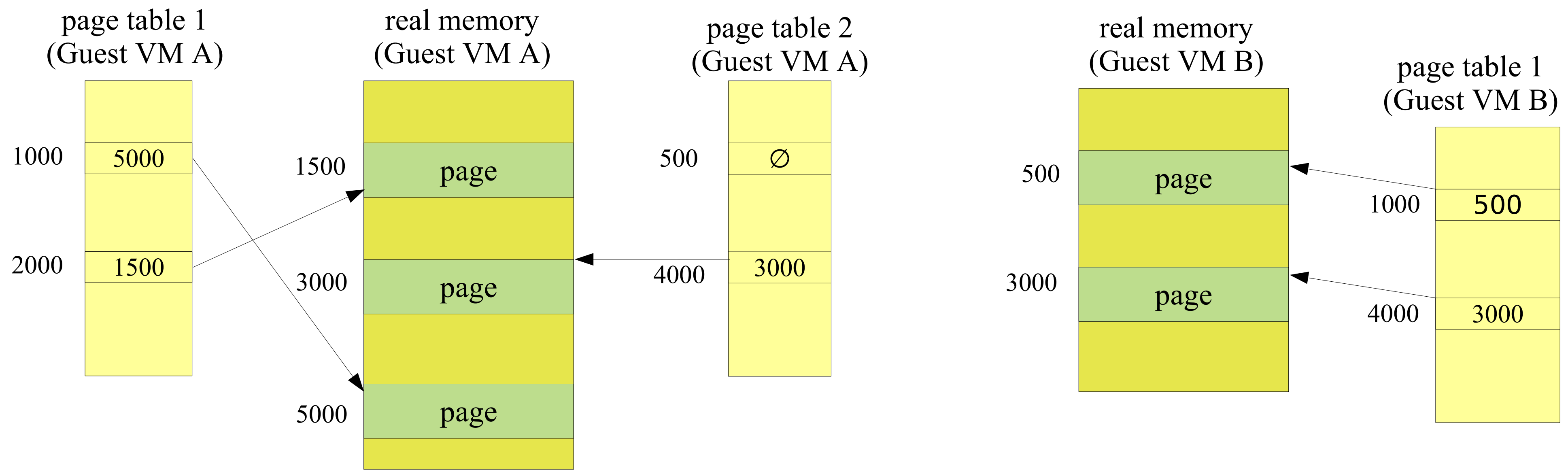


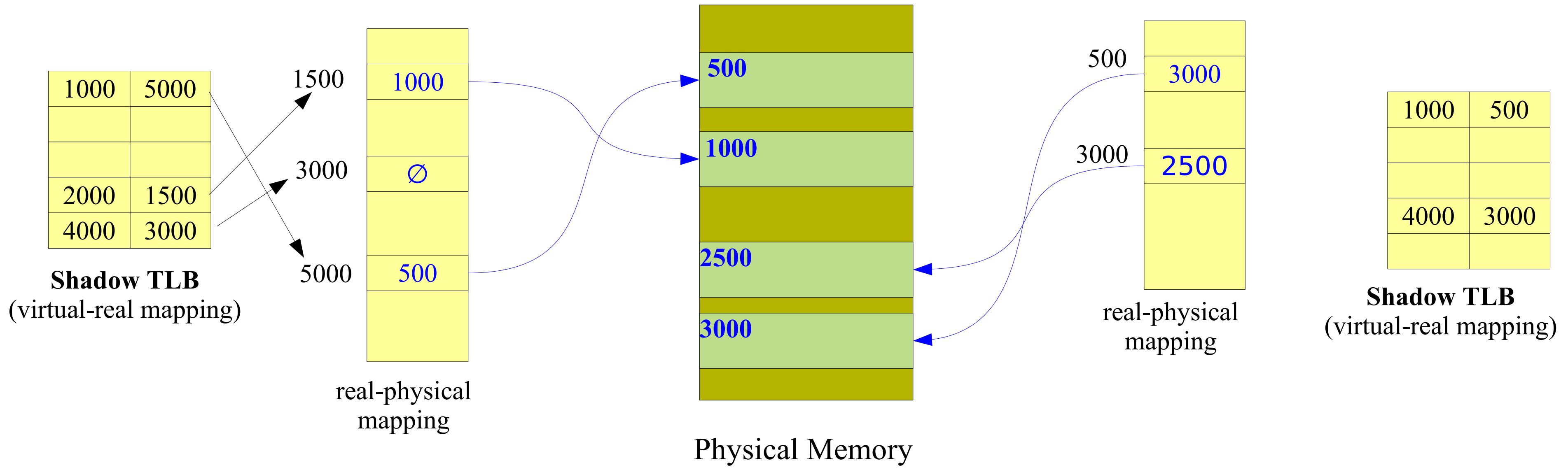
System Virtual Machines

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



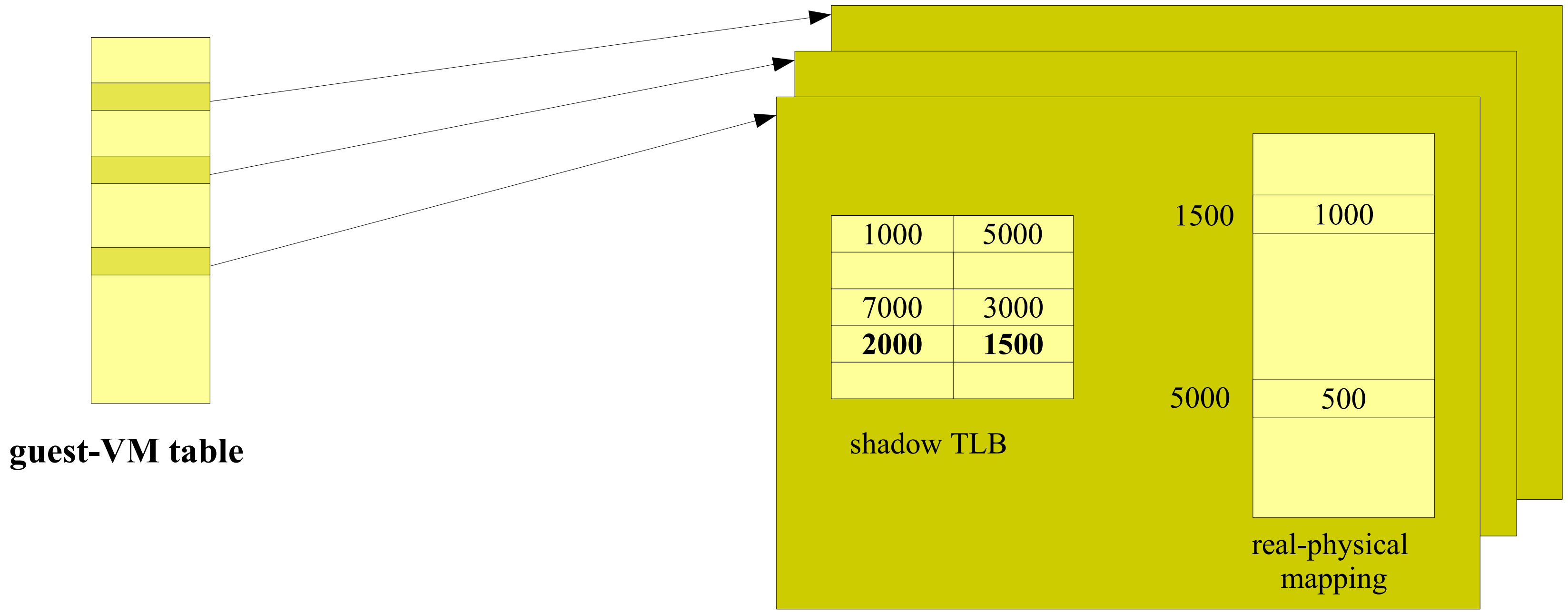
System Virtual Machines





System Virtual Machines

- Virtualize an architected TLB
  - Switching between guest VMs
    - Switching the architected state for the new guest VM
    - Rewrite the entire TLB with **virtual-to-physical** mappings
      - But this is expensive...



System Virtual Machines

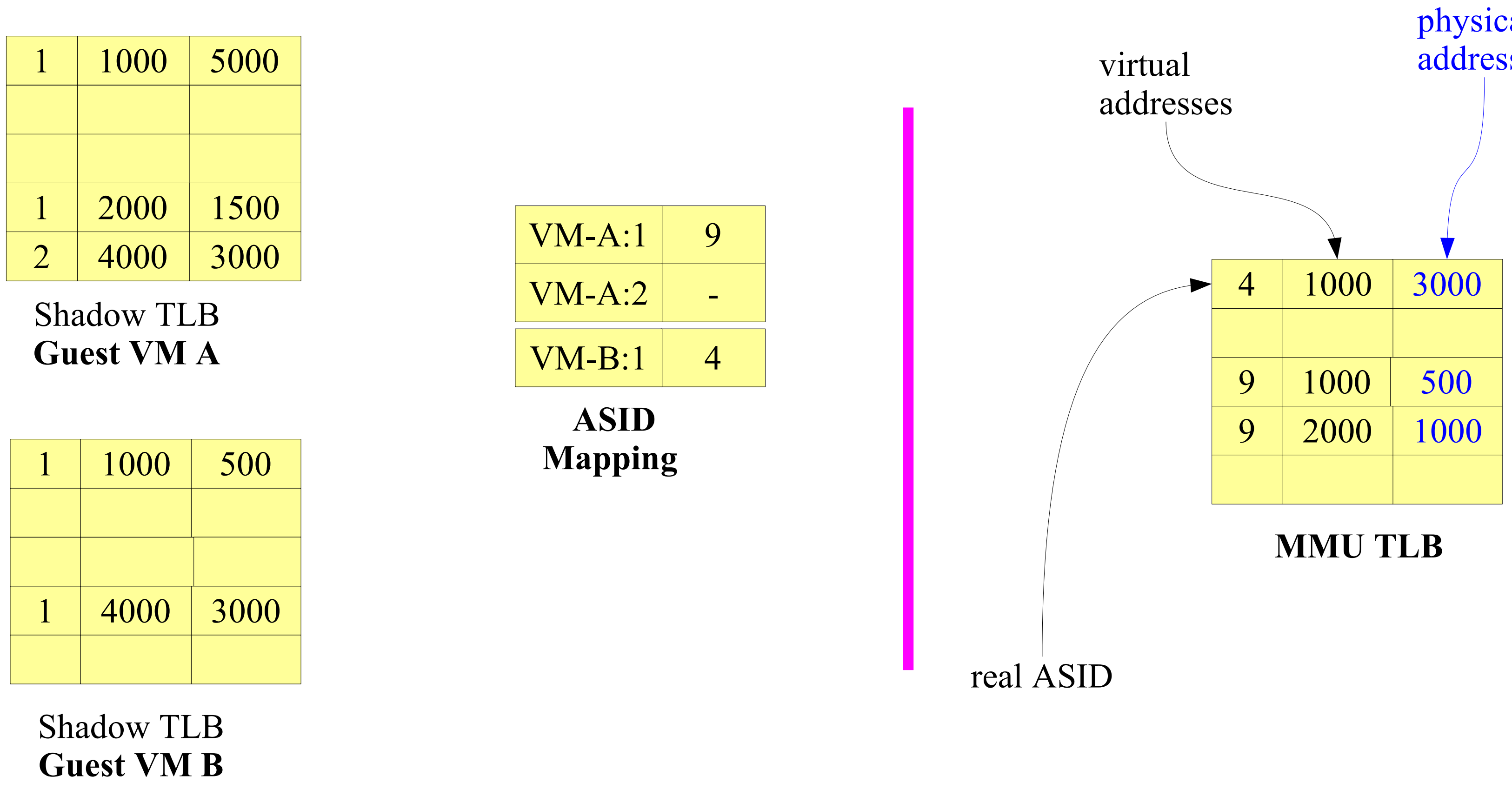
- Virtualize an architected TLB
  - Address Space IDentifier (**ASID**)
    - Included support in architected software-managed TLBs
      - An architected ASID register contains the current ASID
      - ASID register is assigned on address space switch
    - Enables to mix address translations for different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

System Virtual Machines

- Virtualize an architected TLB
  - 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

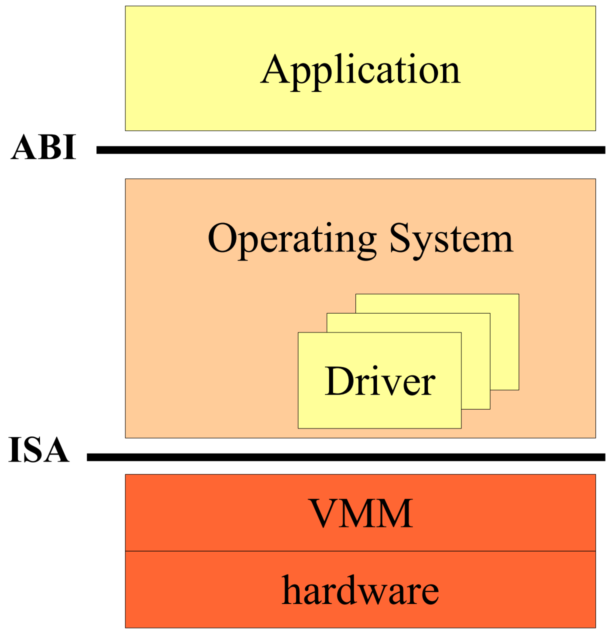


System Virtual Machines

- Virtualize resources
  - 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 constantly growing
  - VMM device families
    - 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

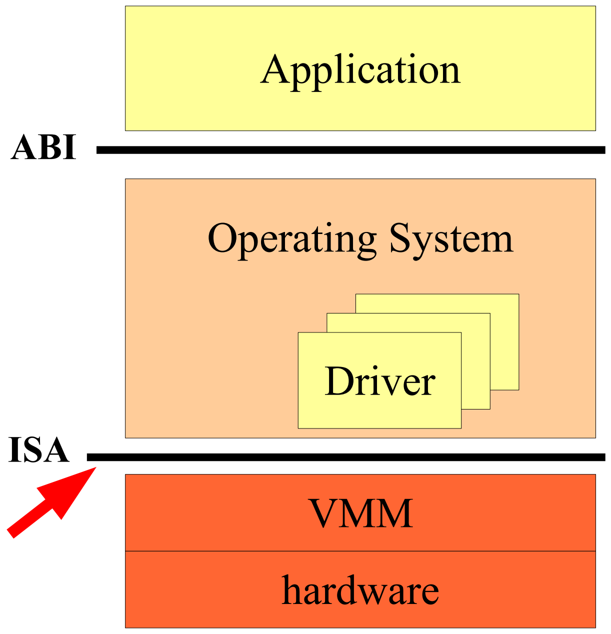
System Virtual Machines

- Virtualize I/O activity
  - Three possible levels
    - At the system call interface (ABI)
    - At the device driver interface
    - At the operation-level interface (ISA)



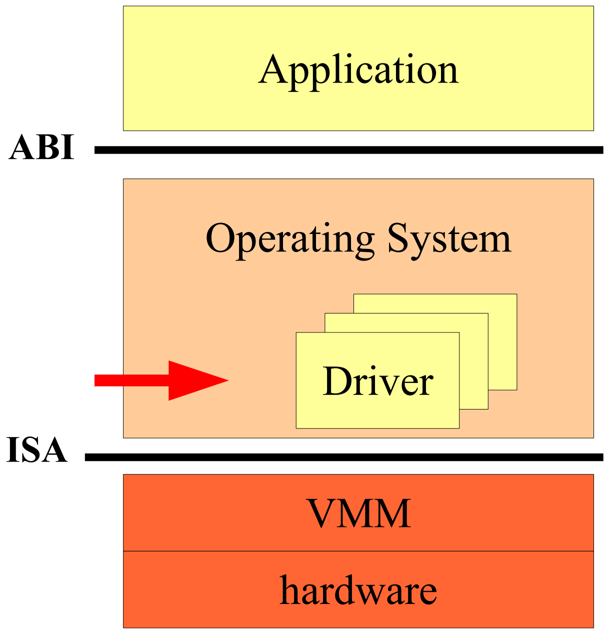
System Virtual Machines

- At the operation-level interface
  - 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 that having to develop all the drivers for a regular OS



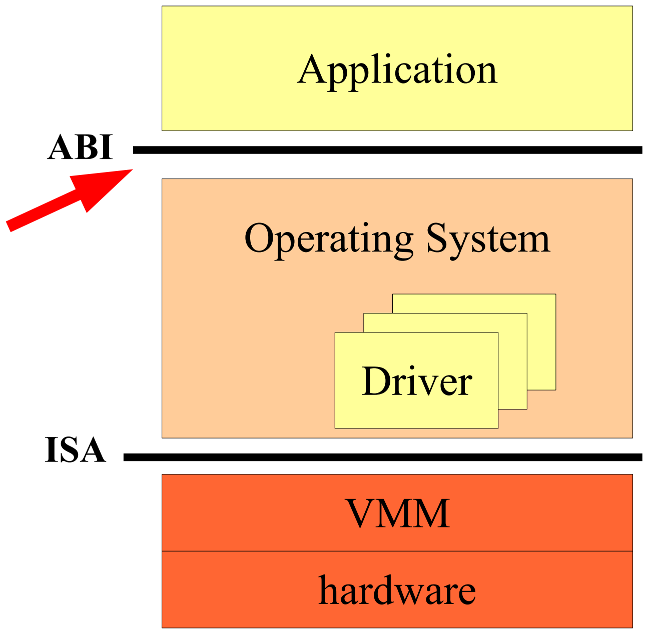
System Virtual Machines

- At the device driver interface
  - A natural interception point
    - Easy to emulate and redirect calls to the driver of the physical device
  - Not general
    - Require some knowledge of the guest OS and of its internal device driver interface
    - Does not work if the VMM is intended to host specific or esoteric operating systems
  - Often practical enough
    - VMM for successful operating systems, such as Windows or Linux
    - The VMM only needs to support a small number of virtual devices (e.g. one type of virtual NIC, one type of virtual disk, etc.)



System Virtual Machines

- 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
    - Require some knowledge of the guest OS and of its internal device driver interface
    - Does not work if the VMM is intended to host specific or esoteric operating systems
  - Daunting task
    - VMM needs an ABI mirroring the guest OS ABIs with many system calls
  - Very OS specific
    - Need precise emulation of the different I/O behaviors of the different guest OS
    - Can only be done if the VMM team has intimate knowledge of the guest operating systems

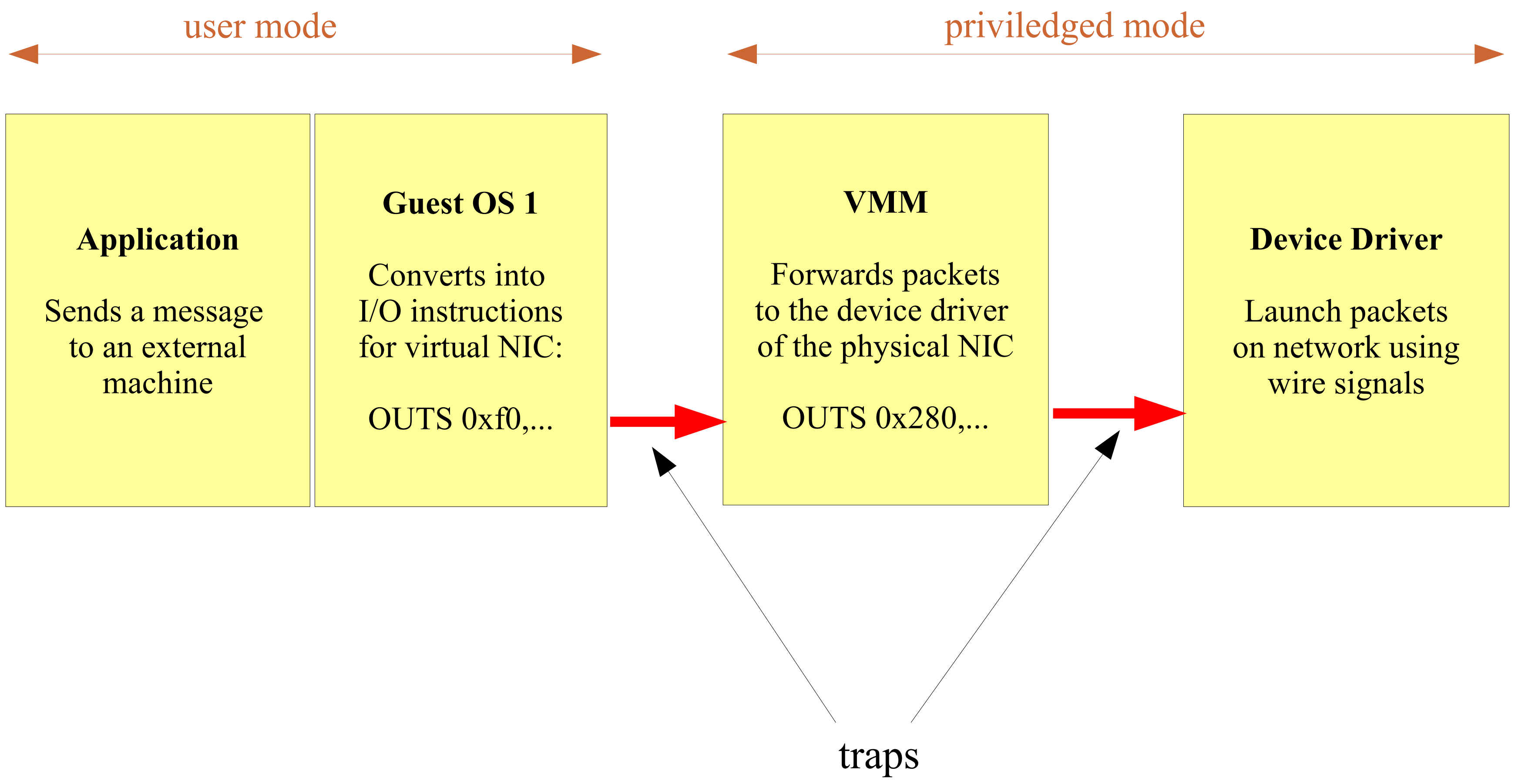


System Virtual Machines

- Example: network virtualization
  - Network Interface Card (NIC)
    - Assume virtual NIC is the same as the physical NIC
    - Assume Intel IA-32
      - With IN/OUT or INS/OUTS instructions
      - On a I/O port, like 0xf0
      - Each port may be set to trap if I/O instructions are attempted
  - 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
      - Traps to the VMM device driver that performs the transfer

System Virtual Machines

- Example: network virtualization



user mode
priviledged mode
Application
Sends a message to an external machine
Guest OS 1
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
traps

System Virtual Machines

- 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
  - Problems
    - 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

System Virtual Machines

- 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

System Virtual Machines

- 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

System Virtual Machines

- 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

System Virtual Machines

- 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

System Virtual Machines

- 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

System Virtual Machines

- 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

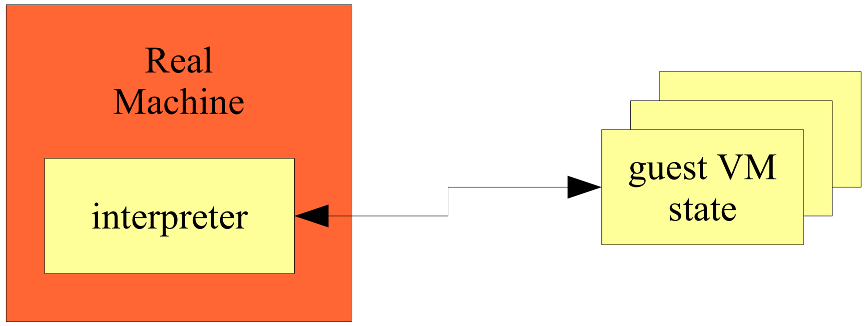
System Virtual Machines

- 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?

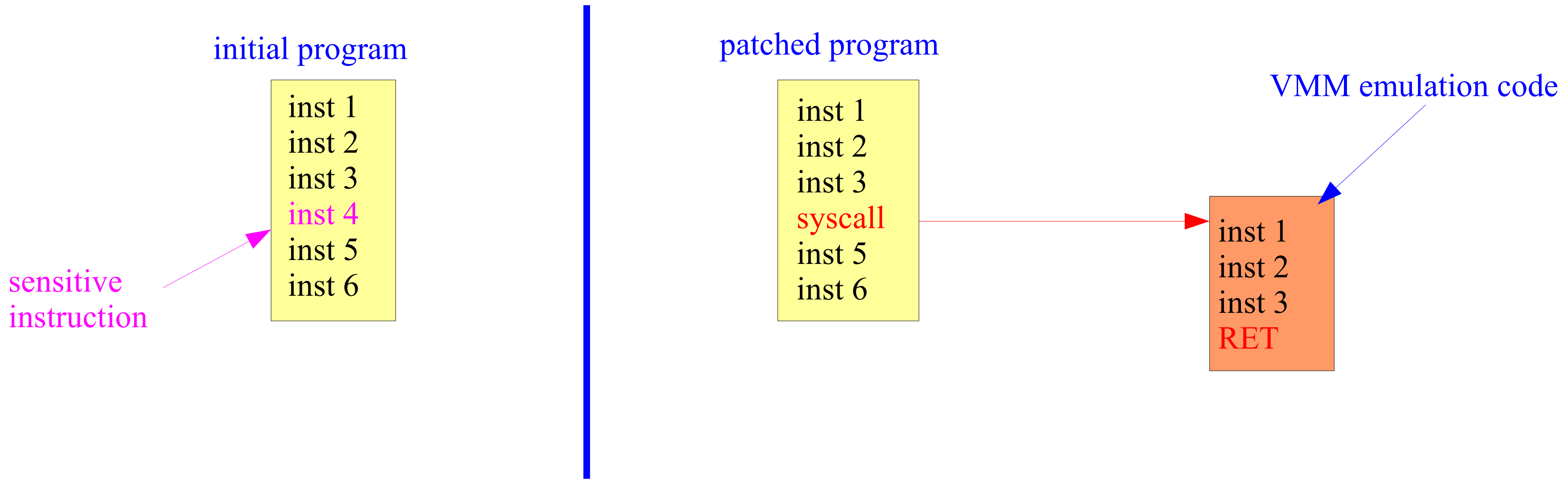
System Virtual Machines

- 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
    - Register moves are now in fact memory moves



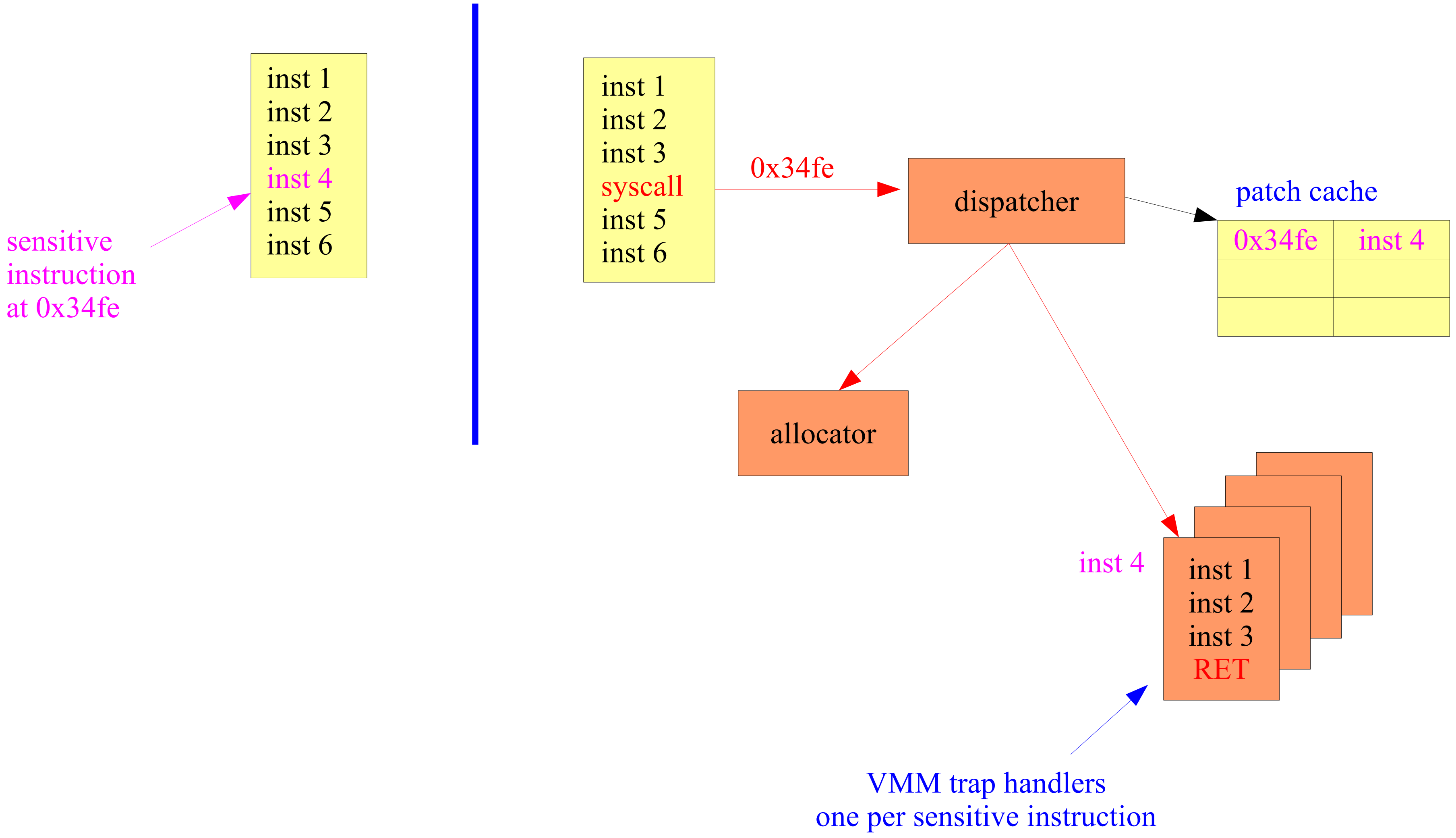
System Virtual Machines

- Binary translation is necessary
  - Execute most of the instructions natively for speed
  - State management occurs
    - Each guest VM has a state block
    - Restored and saved on every switch between guest VMs
  - We must scan and patch sensitive non-priviledged instructions
    - Basic idea



System Virtual Machines

- Binary translation is necessary
  - More complete picture

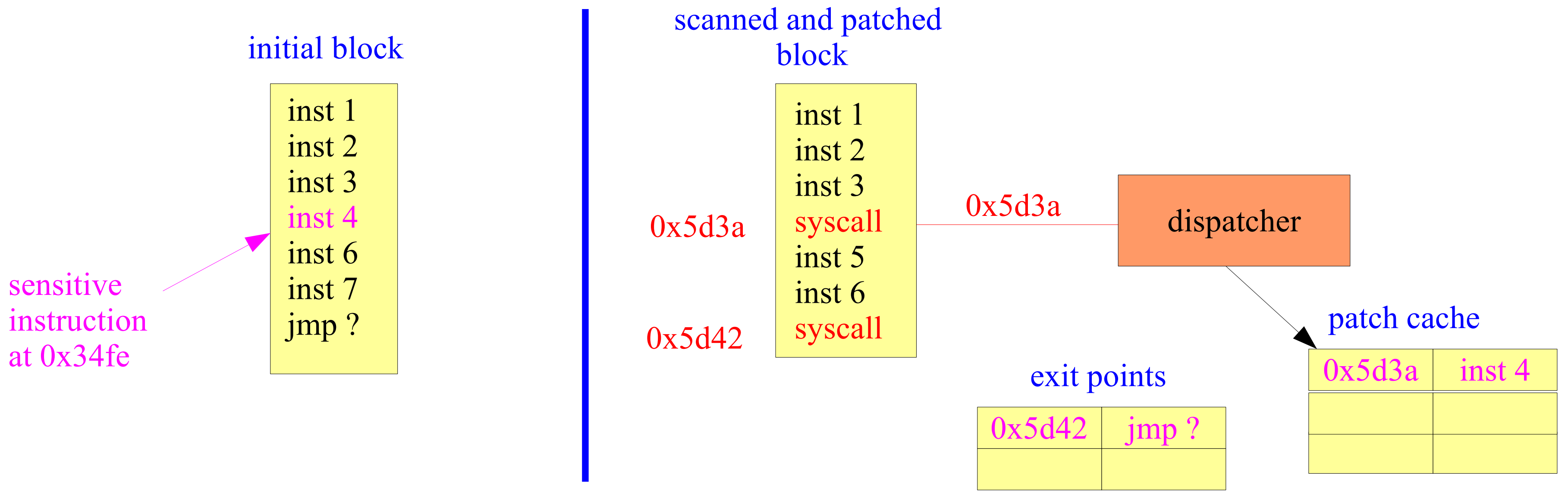


System Virtual Machines

- Discussing patching
  - Can we scan and patch the entire code?
    - No, we cannot because we 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
        - We don't know statically the size of table
        - We probably do not know statically the function pointers either
    - Self-modifying code
      - This happens more than we think
      - High-level VMs typically do that all the time as optimizations
    - Dynamically loaded code
      - Not known statically

System Virtual Machines

- Dynamic scanning and patching
  - We know the entry point
    - We can scan to certain points, typically branch points
  - This produces basic code blocks
    - Code blocks can be scanned and patched
    - The exit point in each code block becomes traps to the VMM

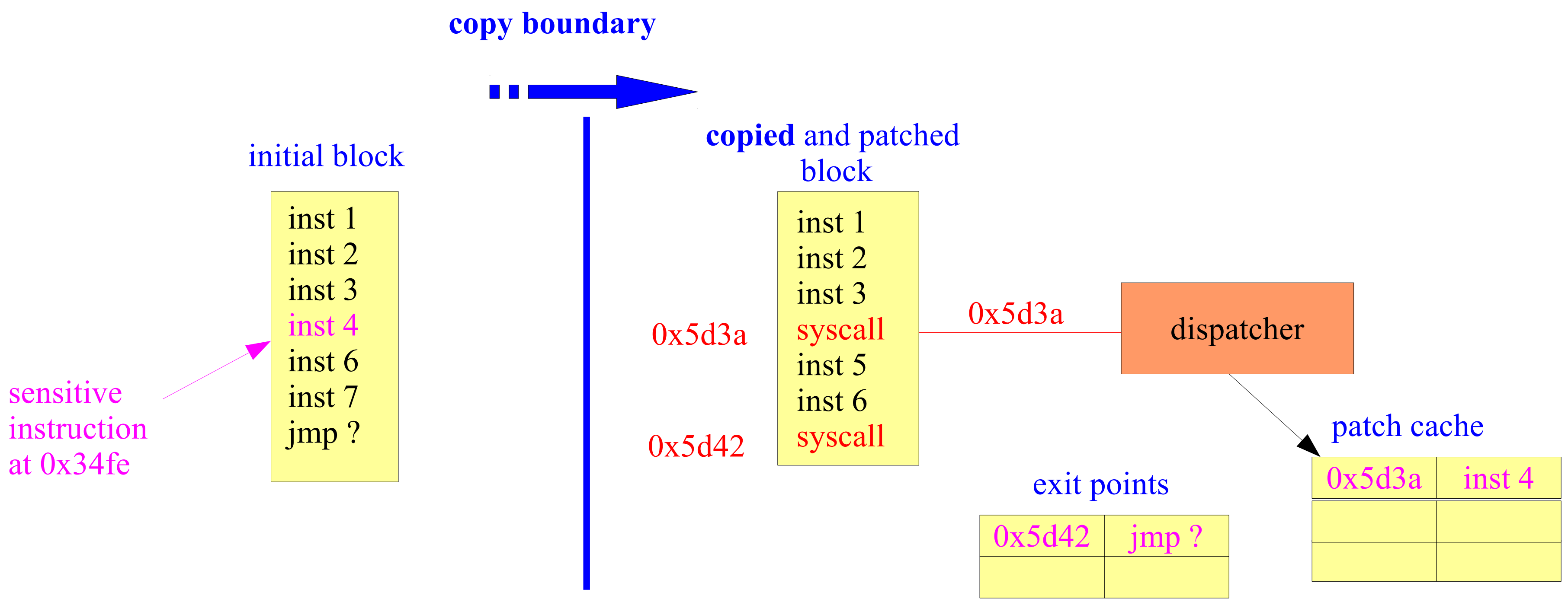


System Virtual Machines

- Dynamic scanning and patching
  - Code blocks are copied
    - **Execution only happens within copied and patched code blocks**
  - Rationales
    - Replaced instructions may be smaller than the syscall instruction
      - So we can't do it in place
      - Depends on the ISA and the trap used
    - 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
      - We need to invalidate the corresponding patched code blocks

System Virtual Machines

- Dynamic scanning and patching
  - 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



System Virtual Machines

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

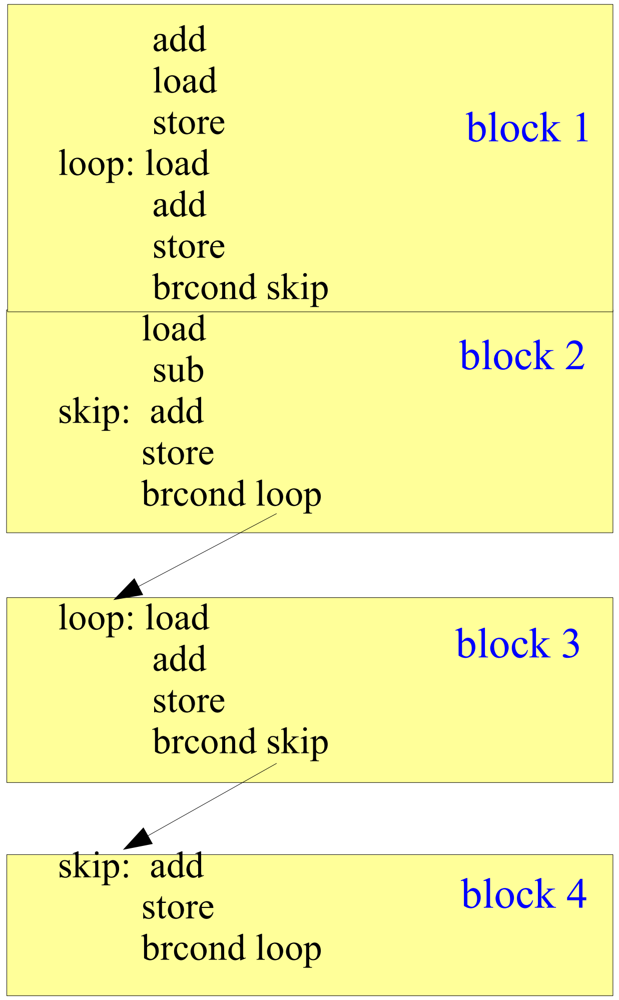
System Virtual Machines

- 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
  - Since we have a code cache
    - We seek fast translation
    - We introduce dynamic code blocks

System Virtual Machines

- 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

```
        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                  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
```

System Virtual Machines

- Dynamic code blocks
  - Slightly larger than static code blocks
    - More suited to optimizing binary translators
  - Introduce a bit of redundancy
    - Same instructions may be in several blocks
  - Faster to generate
    - Just parse instruction streams to next branch
    - Never split existing dynamic blocks on backward branches
    - Just starts producing a dynamic block on a branch miss in the code cache