ECE 5984 Virtualization Technologies

# The Popek and Goldberg Theorem

Pierre Olivier

# Outline

1) Introduction

2) Model

3) Theorem

4) Nested virtualization & hybrid VMs

5) Paging and the theorem

6) Theorem violations

# Outline

1) Introduction

2) Model

3) Theorem

4) Nested virtualization & hybrid VMs

5) Paging and the theorem

6) Theorem violations

# Popek & Goldberg theorem: introduction

- **Paper published in 1974 in *Communications of the ACM***

Popek, Gerald J., and Robert P. Goldberg. "**Formal requirements for virtualizable third generation architectures.**" *Communications of the ACM* 17.7 (1974): 412-421.

- **Defines the requirements for an ISA to be virtualizable**
  - ISA: Instruction set architecture (ex: x86-64, x86-32, aarch64, etc.)
  - Virtualizable: a VMM can be constructed on that architecture in a way that *an OS running on the hardware can also run in a VM*
- **Original idea of the paper: show that some ISA are not virtualizable**
  - DEC PDP-10 taken as a case study

# Popek & Goldberg theorem: introduction

- **Lack of popularity for virtualization at the time the paper was published**
- **Later, VMs become popular (end of 90s)**
  - Intel & AMD explicitly designed ADM-V and Intel VT-X in the 2000s to meet the Popek & Goldberg criteria
    - Hardware support for x86-64 virtualization
- **This is now a seminal paper on virtualization**
  - Can an ISA support a VMM that itself support *arbitrary* guests, relying exclusively on *direct execution*
- ***We'll also learn through this theorem the fundamental principles behind hypervisor operation on virtualizable ISAs***

# Popek & Goldberg theorem: introduction

- **We will explain the theorem as follows:**
  - Explain P&G **simplified CPU model**
    - Simple hardware platform, but still representative of modern CPUs, as a support for the theorem
  - Explain **how a regular, non-virtualized, OS would run on that simplified CPU model**
  - Give the **theorem**: what characteristics an ISA needs to exhibit in order to be able to run a VMM and VMs
  - **Describe a VMM for that simplified CPU model**
    - Which properties it should satisfy to be an actual efficient VMM
    - How it operates
  - Give some **examples of theorem violations** (ISAs not virtualizable)

# Outline

1) Introduction

2) Model

3) Theorem

4) Nested virtualization & hybrid VMs

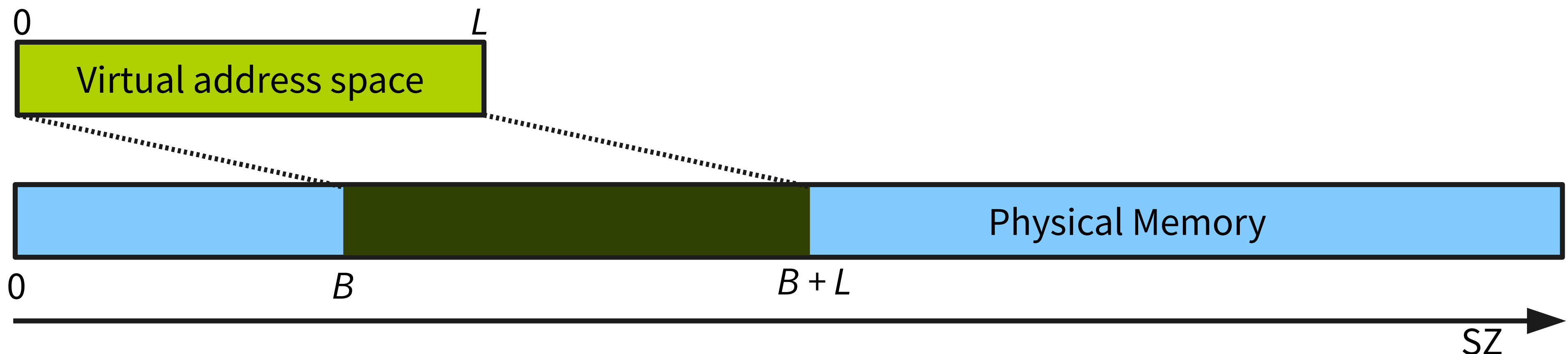
5) Paging and the theorem

6) Theorem violations

# The model

## Simplified CPU definition

- **Authors defines a simplified computer model to be the support for the theorem**
  1) One processor with **2 execution modes: user and supervisor**
  2) **Support for virtual memory** implemented through **segmentation**
     - Single segment: Base $B$, Limit $L$
     - Virtual range $[0, L[$ mapped to physical range $[B, B+L[$
     - (no paging)
  3) **Physical memory** is contiguous, starts at 0, size: $SZ$

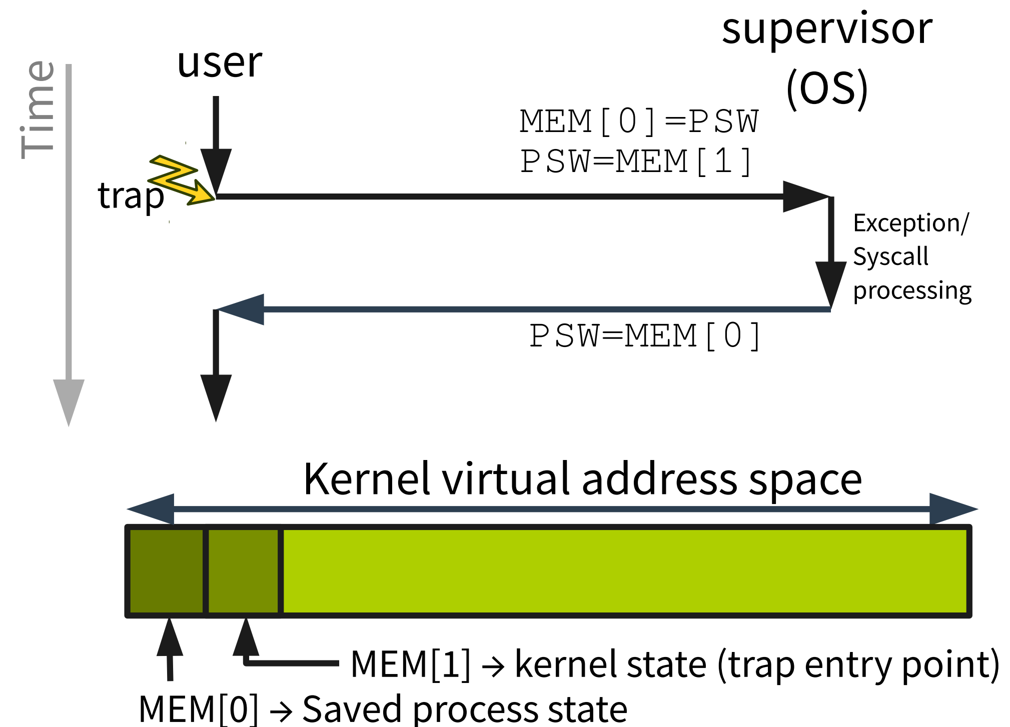


# The model

## Simplified CPU definition (2)

- **Authors defines a simplified computer model to be the support for the theorem (continued)**

4) CPU ***state***: ***Processor Status Word*** (PSW): ***(M, B, L, PC)***
   - Execution level $M = \{s, u\}$ (supervisor or user)
   - Segment register $(B, L)$
   - The current program counter: *PC*
     - Instruction currently executed
5) CPU offers support for saving PSW content in memory `MEM[0]` and loading a new value from `MEM[1]`
   - Action of entering the OS following a trap
6) CPU offers an instruction to load PSW content from a location in memory
   - Exiting the OS after a trap processing
7) No I/O or interrupts for simplicity



# The model

## OS operation without VMM

- This simple model is necessary, and sufficient, to run an OS

1) Kernel runs in $M = s$, applications run in $M = u$
2) Kernel sets trap entry point during initialization
   - `MEM[1] ← (M:s, B:0, L:SZ, PC:trap_entry_point)`
3) Kernel allocates a contiguous range of physical memory for each application defined by (B, L)
4) Kernel launches/resume apps with address space [*B*, *B*+*L*[, currently executing *PC:*
   - `PSW ← (M:u, B:B, L:L, PC:PC)`
5) At the trap entry point, kernel decodes the instruction `MEM[0].PC`, determines the cause of the trap and takes appropriate actions

# The model

## VMM construction & requirements

- Research question posed by Popek & Goldberg:

**Given a computer defined according to the model, under which conditions can a VMM be constructed so that the VMM:**

- can execute one or more VMs;
- is in complete control of the machine at all times;
- supports arbitrary, unmodified, and potentially malicious OS designed for the same architecture; and
- be efficient and show at worst a small performance decrease?

# The model

## VMM construction & requirements (2)

- **The VMM needs to comply with these criteria:**

1) **Equivalence**
   - VM is a duplicate of the underlying physical machine
   - Program (application and OSes) behave similarly running natively and in the VM
     - They run unmodified
2) **Safety**
   - VMM in complete control of the hardware at all time
   - No assumption on guests, they can be malicious
   - VMM must enforce isolation
     - Between VM and the VMM/hardware
     - Between VM themselves
       - No shared state
3) **Performance**
   - Minimal decrease in a virtualized program execution speed

# Outline

1) Introduction

2) Model

3) **Theorem**

4) Nested virtualization & hybrid VMs

5) Paging and the theorem

6) Theorem violations

# The Popek & Goldberg Theorem

- **A few definitions:**
  - Sensitive instructions
    - Control sensitive: instruction *updates the system state*
    - Behavior sensitive: instruction *semantics depends on the value of the system state*
  - Instruction that are not sensitive are named innocuous instructions
  - Privileged instructions
    - Can only be executed in supervisor mode and *traps when executed in user mode*

Theorem:

**For any conventional third-generation computer, a VMM may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions**

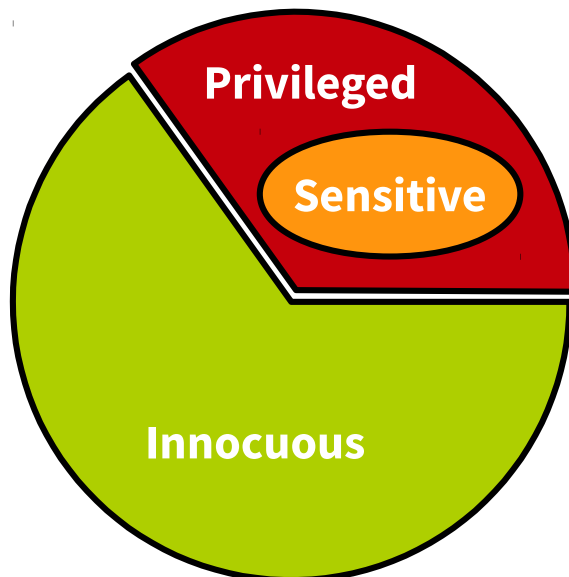
**{*control-sensitive*} ∪ {*behavior-sensitive*} ⊆ {*privileged*}**

# The Popek & Goldberg Theorem (2)

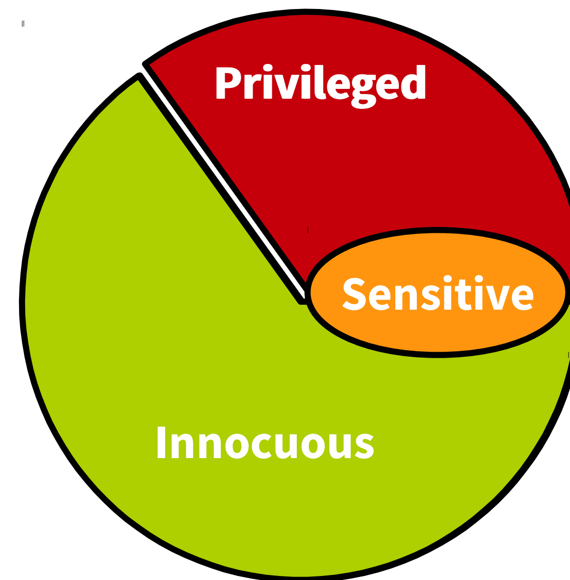
- X86 instruction examples
  - Privileged instruction: **`HLT`**
    - Traps if %cpl != 0
  - Control sensitive: **`LGDT`**
    - Controls x86 segments
  - Behavior sensitive: **`POPF`**
    - Load status (state) register with data from the stack

https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf

Theorem in other words: **all sensitive instructions need to trap in user mode for the ISA to be virtualizable**




Virtualizable ISA




Non-virtualizable ISA

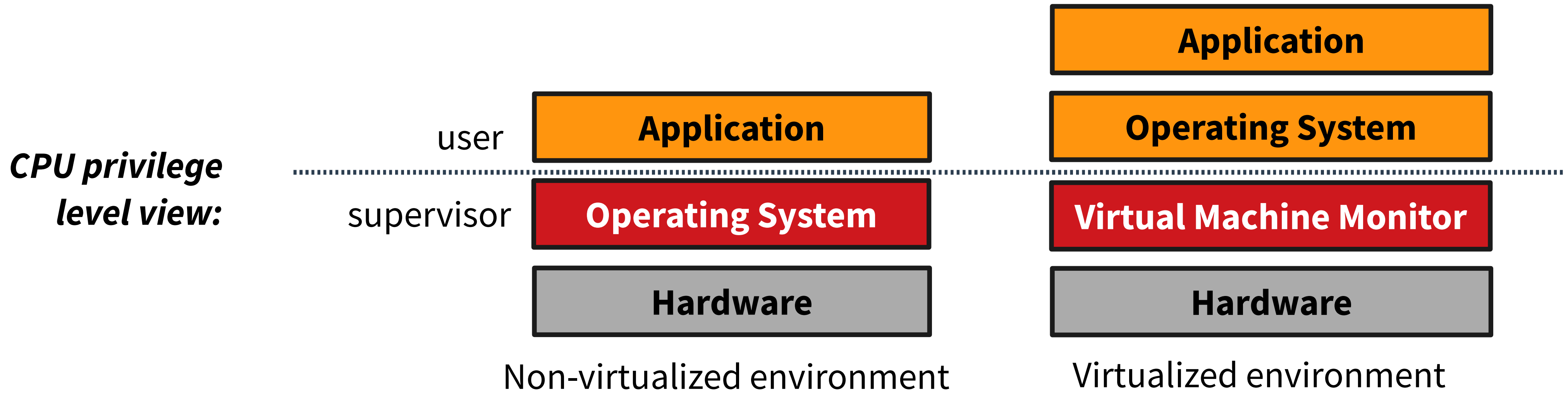
# The Popek & Goldberg Theorem

## VMM operation

- {*control-sensitive*} ∪ {*behavior-sensitive*} ⊆ {*privileged*}
  - Converse holds too: if the criteria is not met, a VMM cannot be constructed for that architecture
    - ➔ If a control-sensitive instruction does not trap, any guest can modify the system state without supervision/check from the VMM
      - For example a guest OS installing an arbitrary page table
  - With trap and emulate (direct execution) the guest OS runs in user mode
    - ➔ If a behavior sensitive instruction does not trap:
      - Guest OS instruction executed with user-level semantics (loss of equivalence)

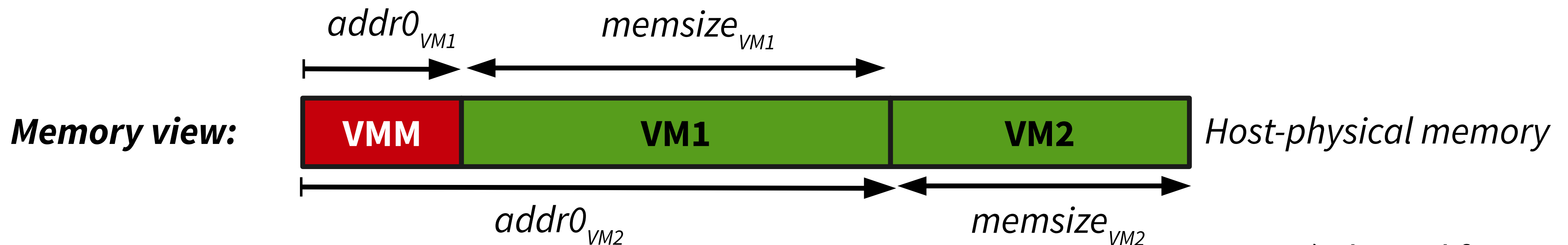
# The Popek & Goldberg Theorem
## VMM operation

- **Under these conditions, VMM operates as follows:**
  1) Only the VMM runs in supervisor mode
     - *Guest OS runs in user mode!*
     - VMM allocates contiguous physical memory for himself, never mapped by guests



# The Popek & Goldberg Theorem
## VMM operation (2)

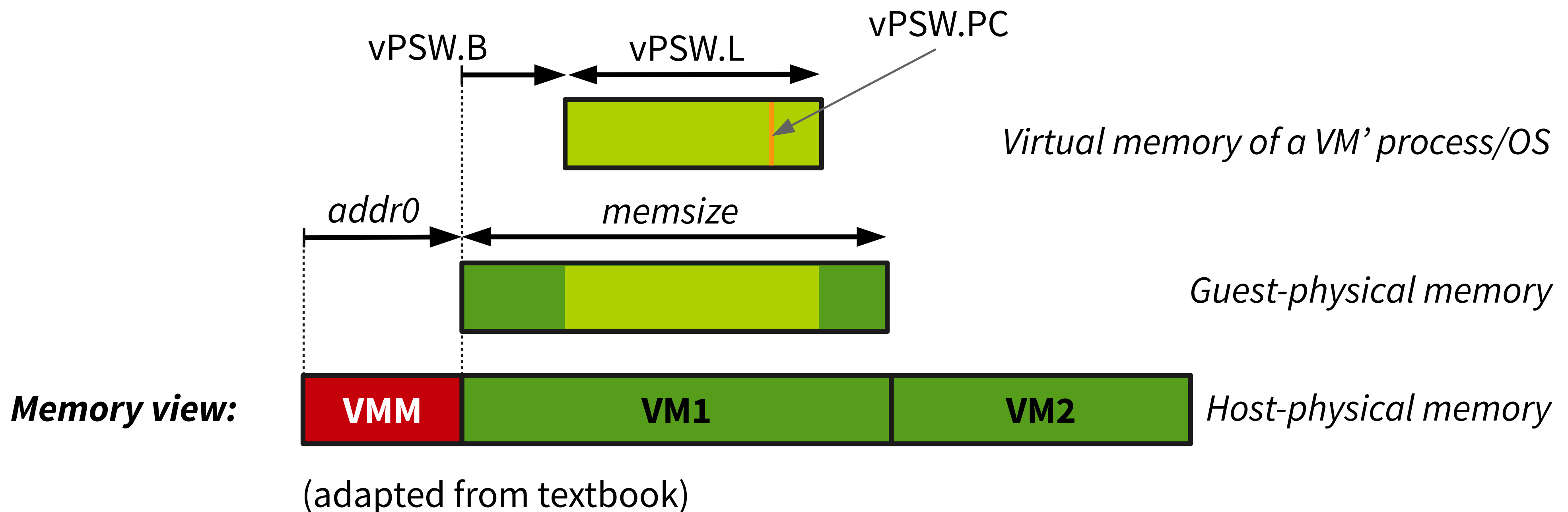
- **Under these conditions, VMM operates as follows (continued):**
  2) VMM allocates contiguous physical memory for VMs
     - Each machine gets a range defined by *addr0* and *memsize*




(adapted from textbook)

# The Popek & Goldberg Theorem
## VMM operation (3)

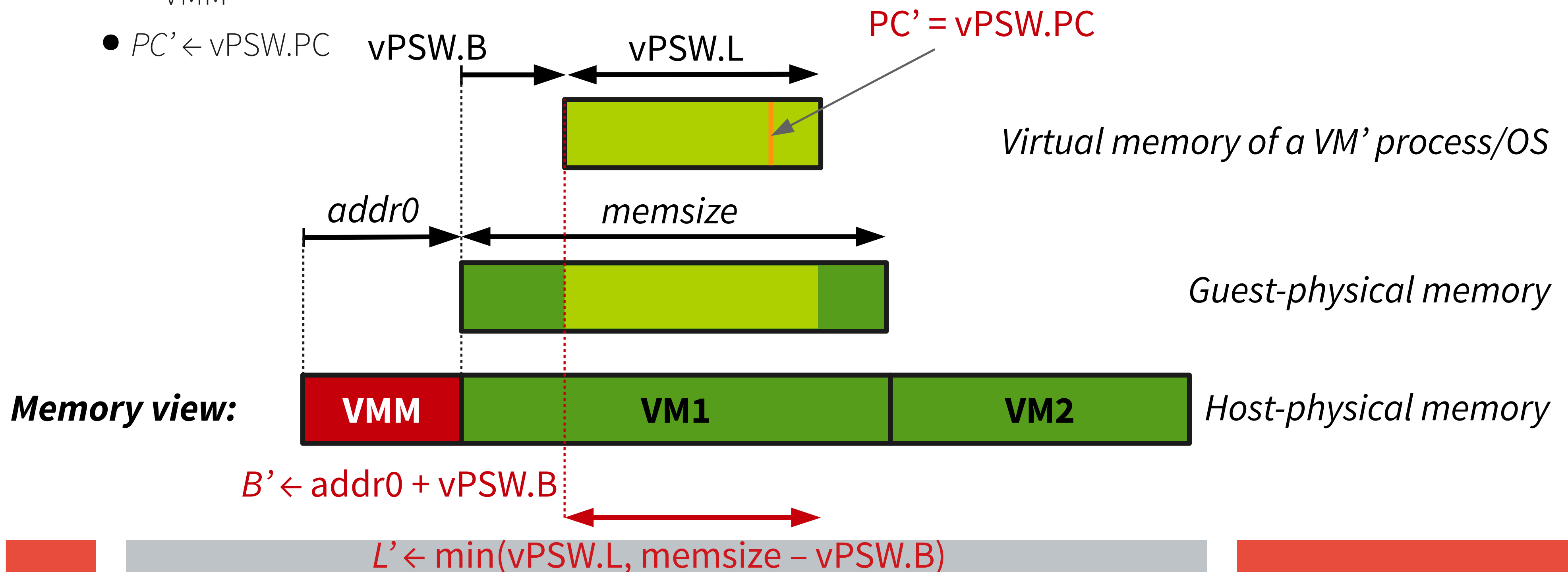
- **Under these conditions, VMM operates as follows (continued):**
  3) VMM keeps in memory the CPU state for each VM, vPSW
     - Consists of (M, B, L, PC)
       - ➔ M: execution mode the VM *thinks it's running on: vm-supervisor vs vm-guest*




(adapted from textbook)

# The Popek & Goldberg Theorem
## VMM operation (4)

- **Under these conditions, VMM operates as follows (continued):**
  - VMM resumes VM execution by loading the hardware PSW ← (*M'*, *B'*, *L'*, *PC'*)
    - *M'* ← *u*
    - *B'* ← addr0 + vPSW.B
    - *L'* ← min(vPSW.L, memsize – vPSW.B)
      - ➔ The *min* ensures that a potentially malicious VM cannot access memory above the limit defined by the VMM
    - *PC'* ← vPSW.PC



# The Popek & Goldberg Theorem

## VMM operation (5)

- **Under these conditions, VMM operates as follows (continued):**

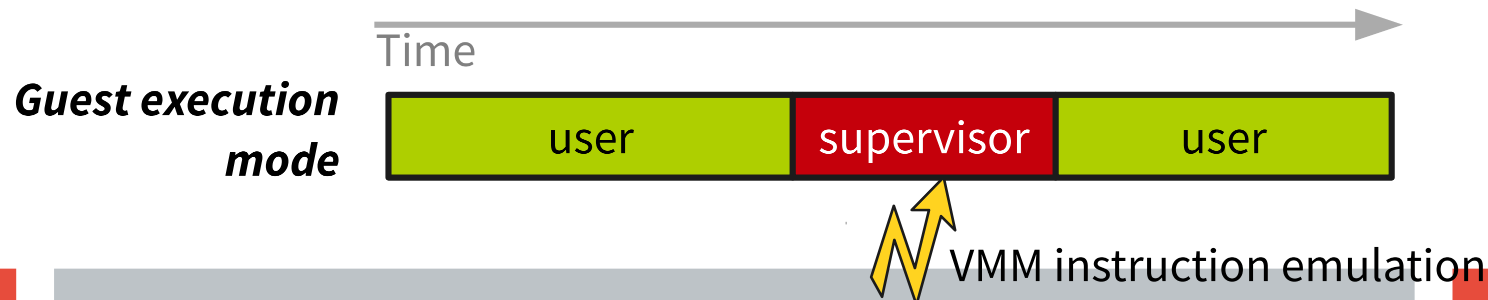
5) VMM update vPSW.PC ← PSW.PC on every trap
   - Note that any try by the VM to modify *M*, *B* or *L* will trap
     ➔ Theorem hypothesis assumes all control-sensitive instruction are also privileged

6) Next, VMM emulates the semantics of the instruction that trapped
   - *If guest OS caused the trap* (vPSW = s), VMM emulates according to the ISA
     ➔ Ex: if the guest OS is trying to update the segment register, the VMM update vPSW.B and vPSW.L

     - Hardware PSW.L and PSW.B will be set accordingly when we return back to VM execution: PSW.B ← addr0 + vPSW.B and PSW.L ← min(vPSW.L, memsize – vPSW.B)

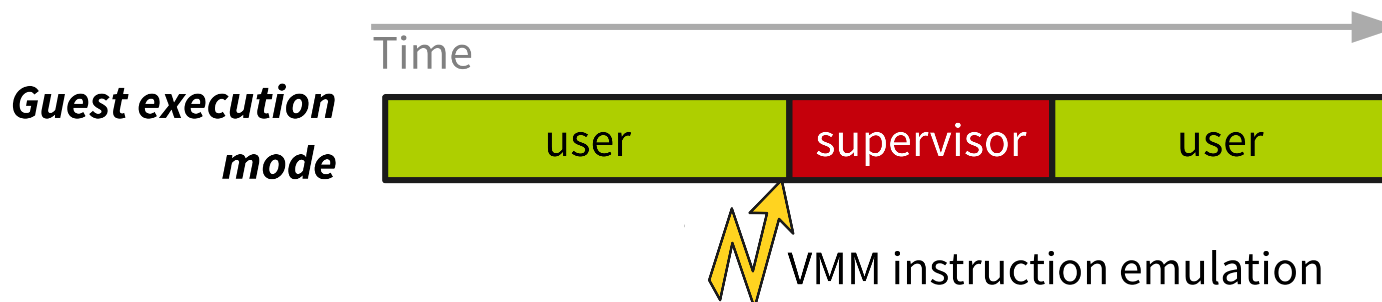
     ➔ Then the VMM ensures the VM will resume at the next instruction: vPSW.PC++

     ➔ Then the VM resumes execution by loading PSW



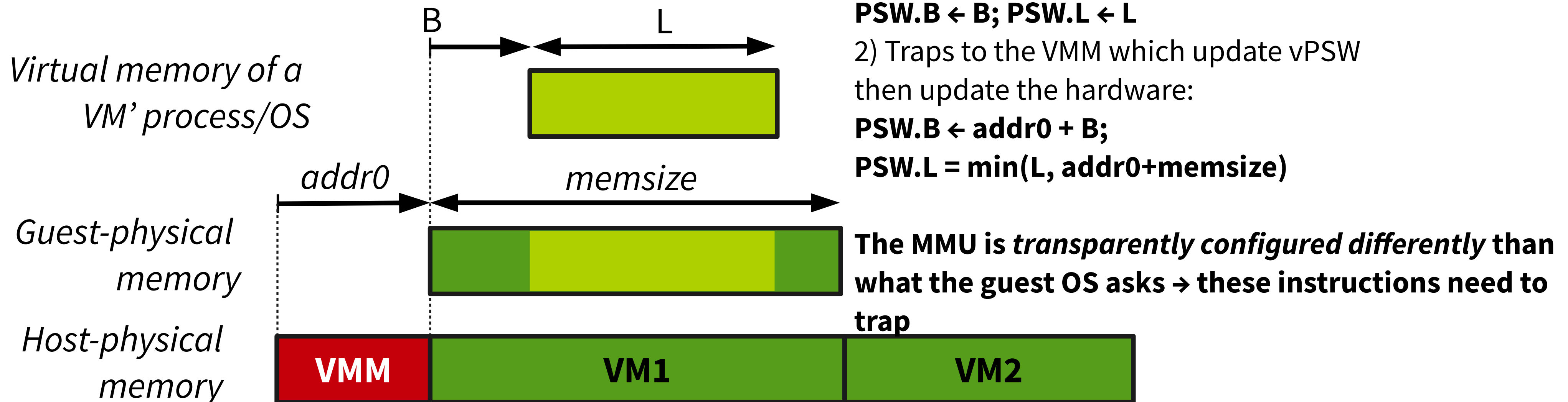
# The Popek & Goldberg Theorem
## VMM operation (6)

- **Under these conditions, VMM operates as follows (continued):**
  6) *If guest application caused the trap* (vPSW = u), VMM emulates according to the ISA
     - ➔ Application is doing a syscall or something illegal: should be handled by the guest OS
     - ➔ MEM[addr0] ← vPSW
       - ➢ Save guest application state in the host-physical location of guest-physical MEM[0]
     - ➔ vPSW ← MEM[addr0 + 1] load guest OS state (OS entry point) from memory
     - ➔ Resumes VM (in guest OS mode based on the updated vPSW)



# The Popek & Goldberg Theorem
## VMM operation (7)

- **Under these conditions, VMM operates as follows (continued):**
  7) According to the theorem hypothesis, all instruction updating the system state (control-sensitive) are privileged, so they will trap
     - Includes instructions updating the virtual to physical mapping
       - ➔ Each of these needs to be **checked** by the VMM to ensure **safety** (isolation)
       - ➔ Each of these needs to be **emulated** to give each VM the illusion of exclusive and full access to physical memory



# The Popek & Goldberg Theorem

## VMM operation (8)

- **Under these conditions, VMM operates as follows (continued):**

7) According to the theorem hypothesis, all instruction updating the system state (control-sensitive) are privileged, so they will trap (continued)

- Includes user / supervisor transition instructions

  ➔ Each of these needs to be **tracked** by the VMM

  - to keep $M = u$ at all times in the VM in to ensure **safety**: VMM in complete control at all times

  - to correctly emulate privileged instruction (behavior-sensitive) according to the current guest privileged level (guest-user or guest-supervisor) to ensure **equivalence**

8) Still according to the hypothesis, behavior-sensitive instruction will also trap

- Ex: reading PSW.M or PSW.B

  ➔ Remember than the actual values are set by the VMM to something different than what the guest OS think they are

  ➔ Need to be emulated by the VMM otherwise this will lead to programs behaving differently on bare-metal vs virtualized: **equivalence** requirements

# The Popek & Goldberg Theorem
## Counter examples

- **Control-sensitive *unprivileged* instructions**
  - Update to the system state that does not trap!
    - Ex: unprivileged switch from supervisor to user mode with **`JRST1`** "return to user" in DEC PDP-10 issued from supervisor mode
- **Behavior-sensitive *unprivileged* instructions reading the system state**
  - In particular instructions reading the system state that do not trap, violates the equivalence criteria
    - Ex: the OS reading PSW.M without a trap to the VMM
      - ➔ OS concludes it is running in user mode…
- **Instructions bypassing virtual memory**
  - If they don't trap, the VM directly access physical memory, possibly outside of the range allocated by the VMM

# Outline


1) Introduction

2) Model

3) Theorem

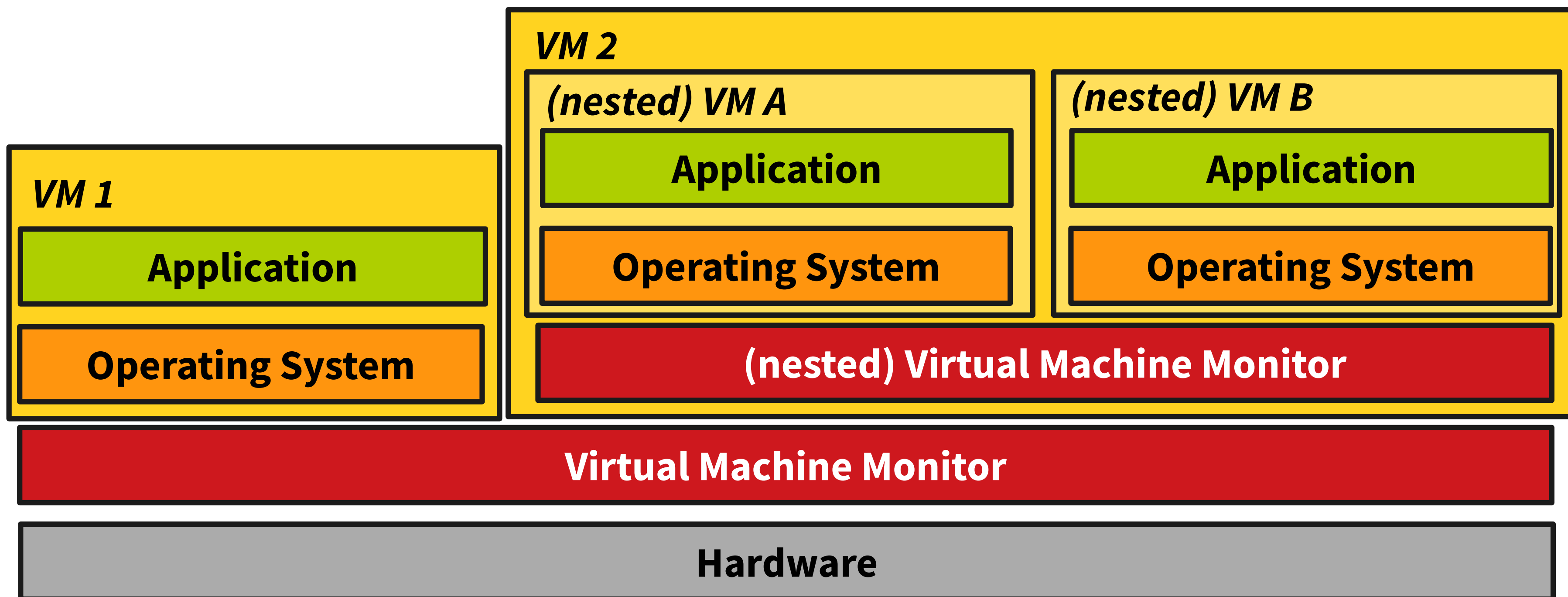
4) **Nested virtualization & hybrid VMs**

5) Paging and the theorem

6) Theorem violations

# Nested virtualization

- **Nested virtualization or recursive virtual machines**
  - Running an hypervisor on top of an hypervisor, within a VM




Example: Xen-blanket

Williams, Dan, Hani Jamjoom, and Hakim Weatherspoon. "The Xen-Blanket: virtualize once, run everywhere." Proceedings of the 7th ACM european conference on Computer Systems. ACM, 2012.

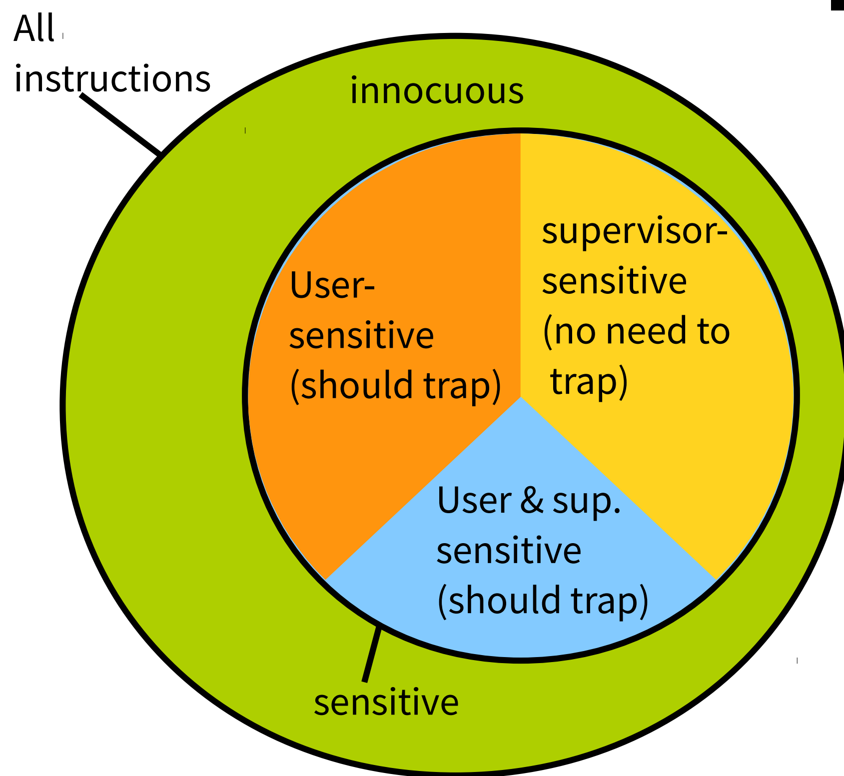
For testing & development, cloud deployment homogenization, security, ...

# Hybrid Virtualization

- **Architecture which fails to meet the P&G criteria because of some specific reasons**
  - Example: JRST 1 in DEC PDP-10
    - Return to user mode from user mode or from supervisor mode without trapping
      - ➔ Control sensitive only when executed in supervisor mode
  - User-sensitive instructions: control/behavior sensitive when executed in user mode
  - Supervisor-sensitive instructions: control/behavior sensitive when executed in supervisor mode
    - JRST 1 is *supervisor-sensitive* but not user-sensitive

# Hybrid Virtualization (2)

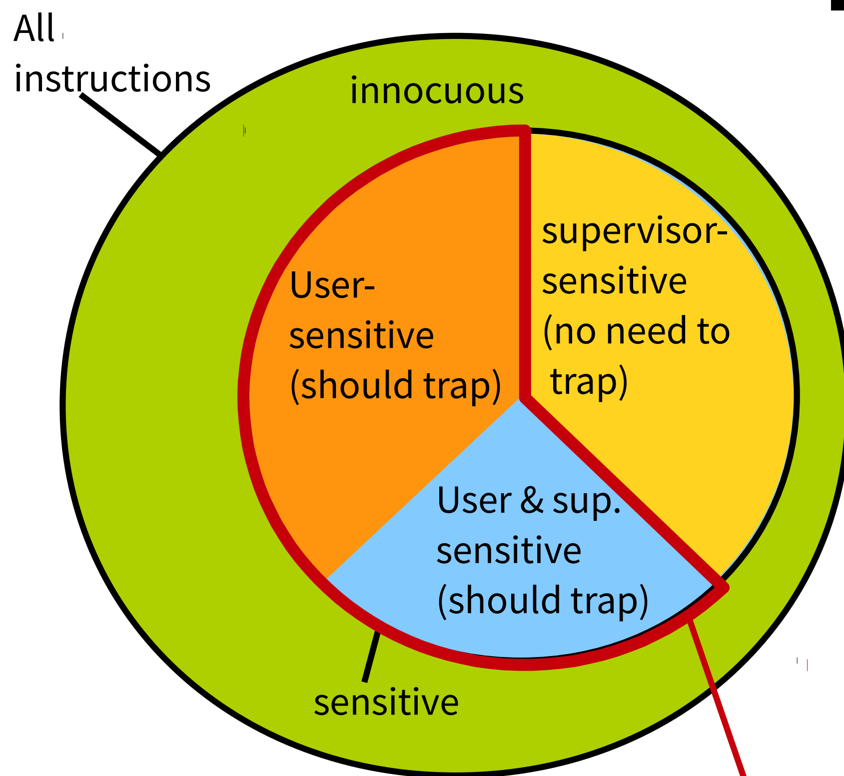
**A hybrid VMM may be constructed for any conventional third-generation computer if the set of user-sensitive instructions is a subset of the set of privileged instructions**




- **When the VM switches to vm-supervisor mode, the VMM interpret all instructions until it switches back to vm-user mode**
  - User-sensitive instructions will trap in vm-user and vm-supervisor and be managed by the VMM
  - Supervisor-sensitive instructions:
    - Will not trap in vm-user, that's okay they are not sensitive in user mode
    - Will be interpreted and emulated in vm-supervisor mode
  - Rationale: time spent in vm-supervisor is low so interpretation does not hurt performance

# Hybrid Virtualization (2)

**A hybrid VMM may be constructed for any conventional third-generation computer if the set of user-sensitive instructions is a subset of the set of privileged instructions**

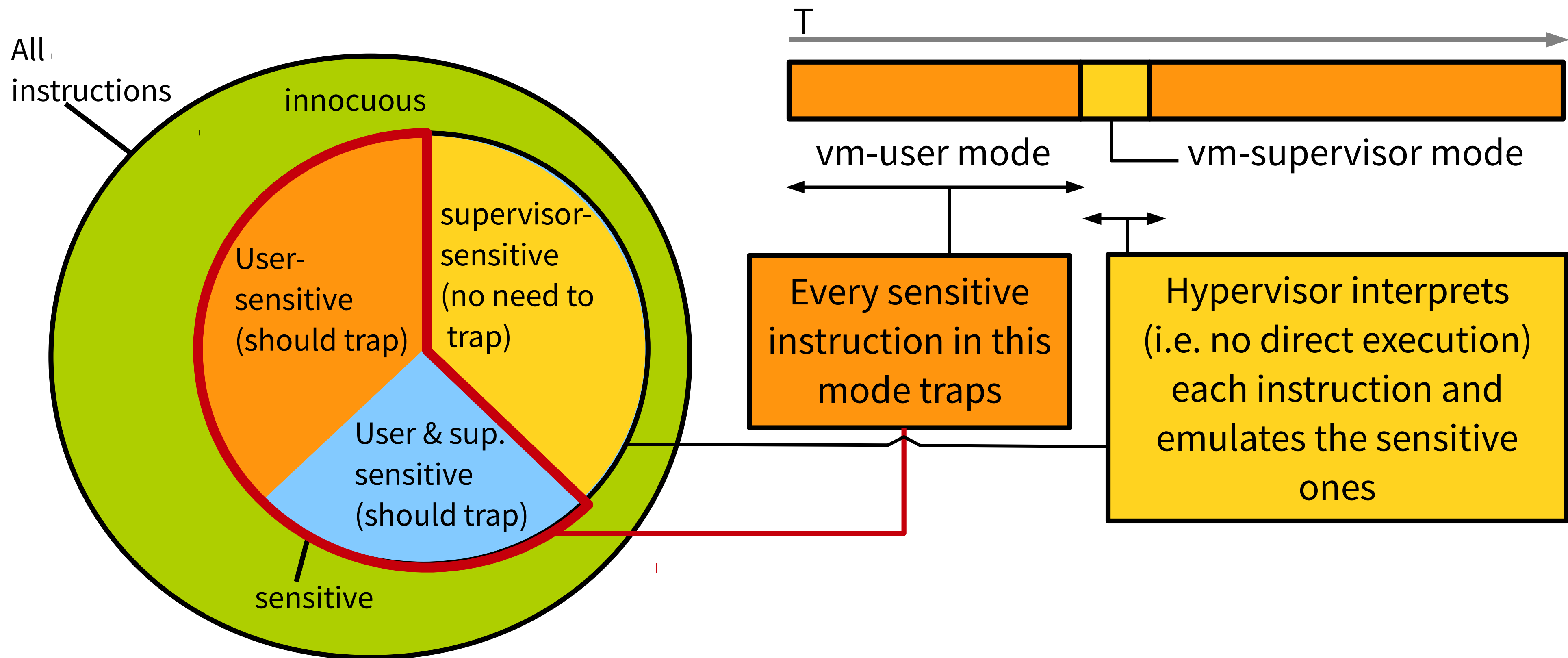



- **When the VM switches to vm-supervisor mode, the VMM *interprets all instructions* until it switches back to vm-user mode**
  - User-sensitive instructions will trap in vm-user and vm-supervisor and be managed by the VMM
  - Supervisor-sensitive instructions:
    - Will not trap in vm-user, that's okay they are not sensitive in user mode
    - Will be interpreted and emulated in vm-supervisor mode
  - Rationale: time spent in vm-supervisor is low so interpretation does not hurt performance

Should be privileged if we want to build a hybrid VMM

# Hybrid Virtualization (2)

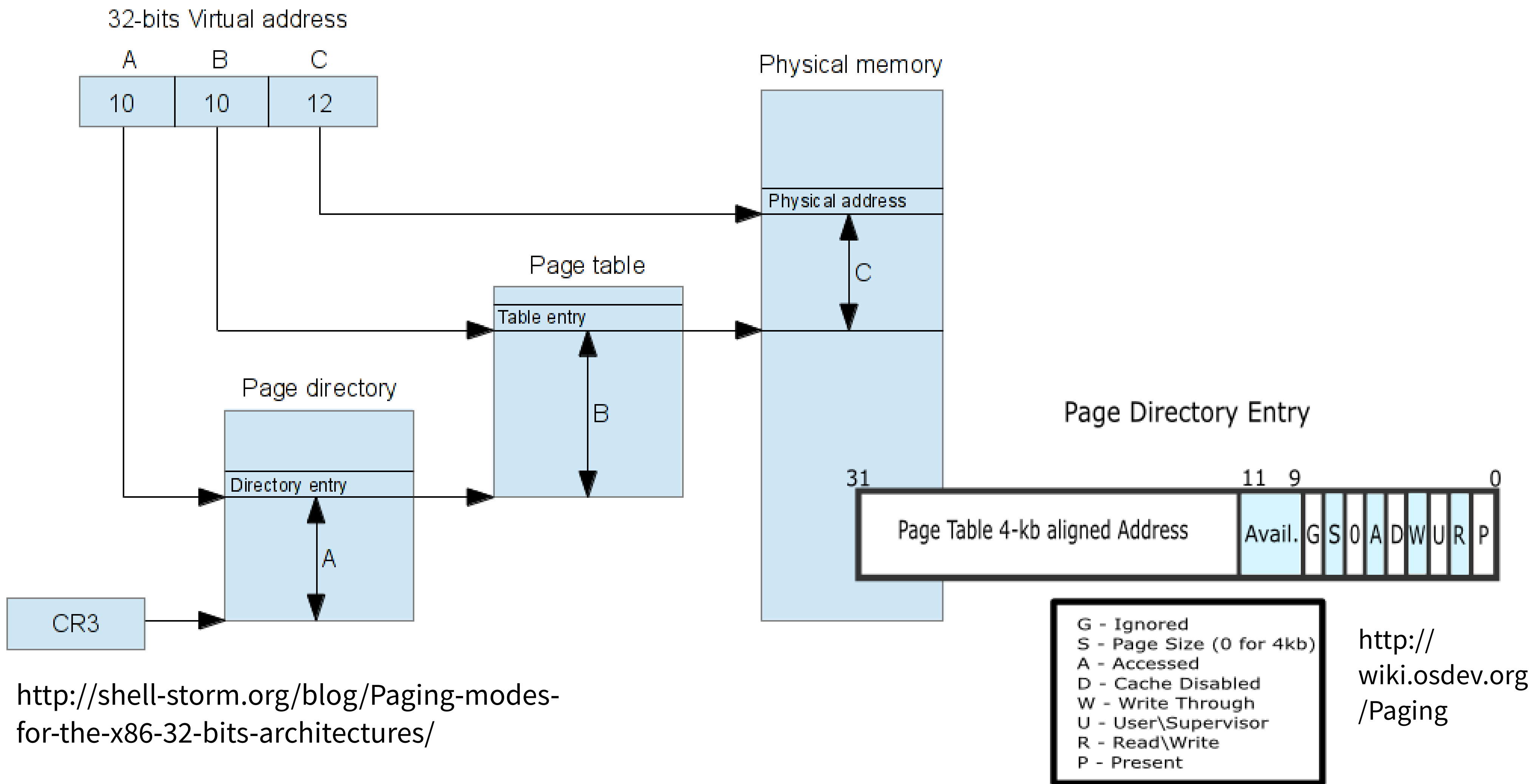
**A hybrid VMM may be constructed for any conventional third-generation computer if the set of user-sensitive instructions is a subset of the set of privileged instructions**



# Outline

1) Introduction
2) Model
3) Theorem
4) Nested virtualization & hybrid VMs
5) **Paging and the theorem**
6) Theorem violations

# Paging

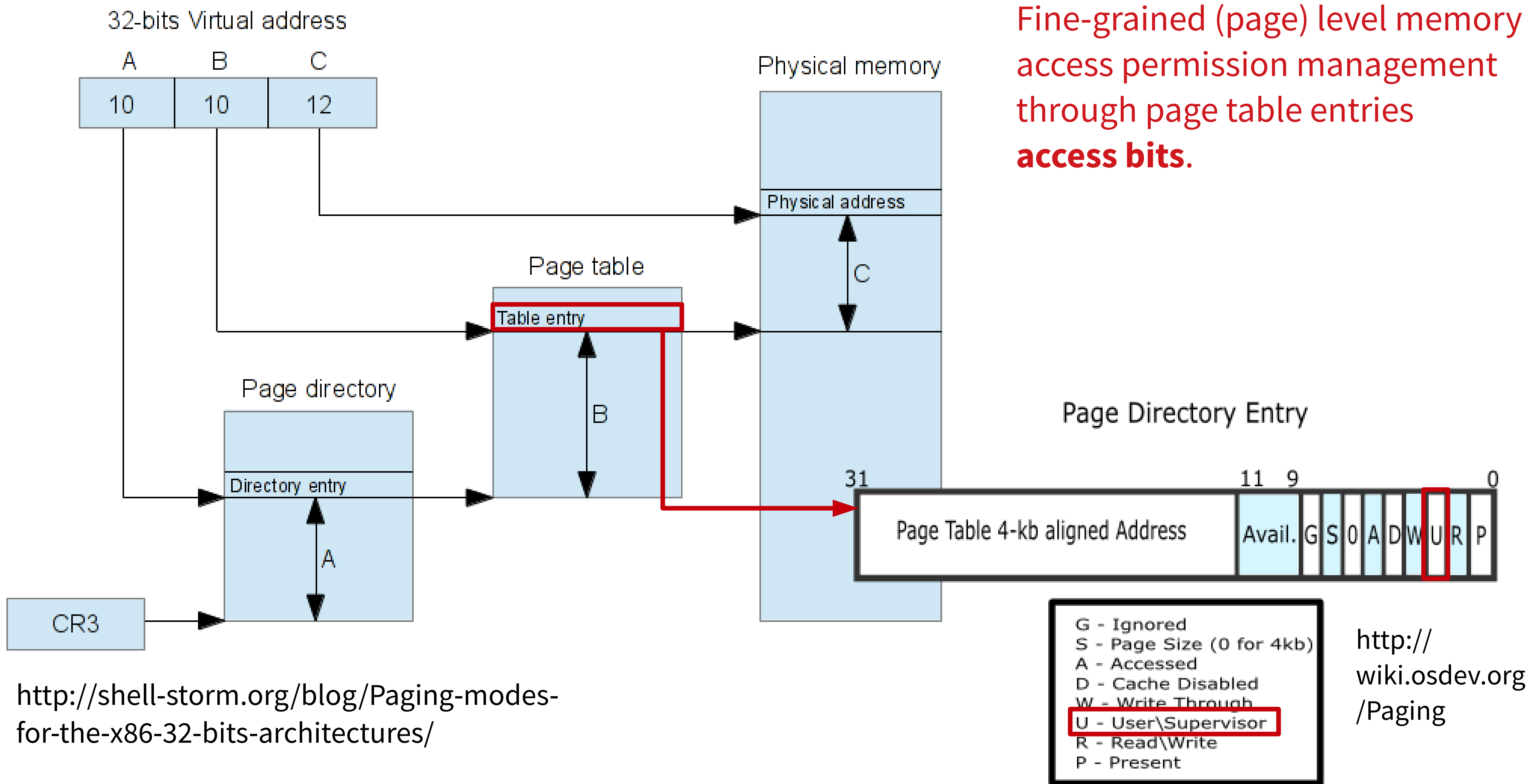



http://shell-storm.org/blog/Paging-modes-for-the-x86-32-bits-architectures/

http://wiki.osdev.org/Paging

# Paging

Fine-grained (page) level memory access permission management through page table entries **access bits**.

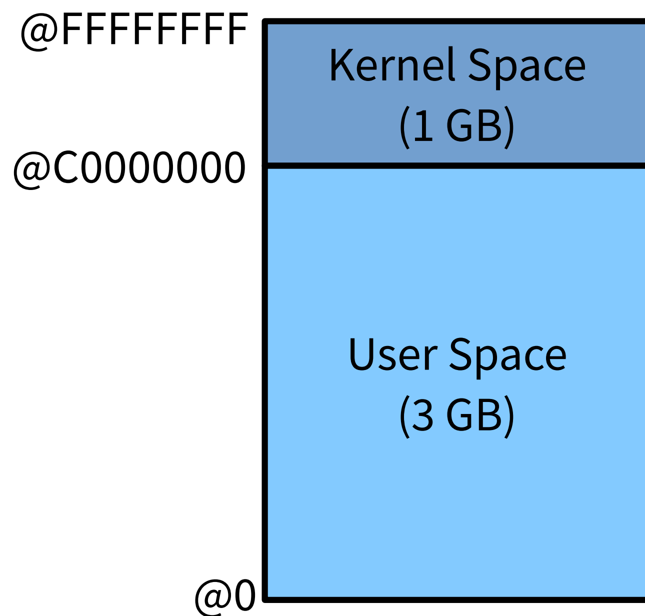



http://shell-storm.org/blog/Paging-modes-for-the-x86-32-bits-architectures/

http://wiki.osdev.org/Paging

# Paging

- **With paging, monolithic OS maps kernel and process in the same address space for performance reasons**
  - No page table switch and no TLB flush
  - Supervisor/user bit in PTEs used to protect OS data/code from userland access




- **Where to put the hypervisor in that linear address space and how to protect it against guest accesses?**
- **How to protect the guest OS, not running in supervisor mode anymore, against guest process accesses?**
- **How to create the guest-physical to host-physical memory mapping according to the guest page table definition?**

# Outline

1) Introduction

2) Model

3) Theorem

4) Nested virtualization & hybrid VMs

5) Paging and the theorem

6) **Theorem violations**

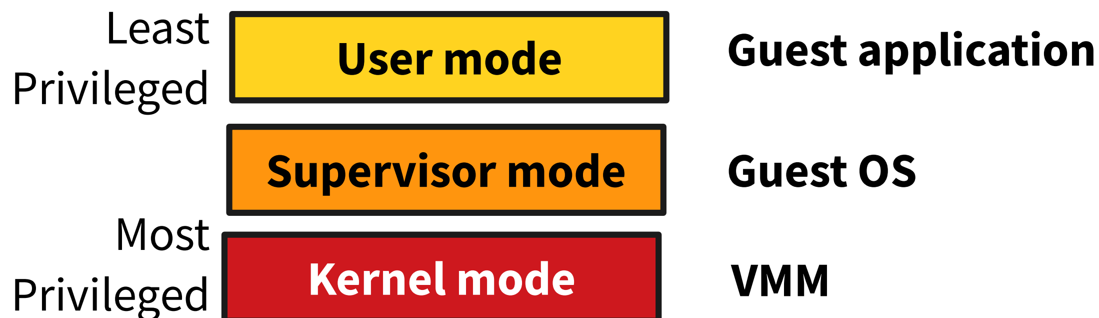
# Theorem violations

- **Direct access to physical memory**
  - Ex: MIPS
- **Location-sensitive instructions**
  - Unprivileged read access to system state
  - Ex: X86-32
- **Behavior and control-sensitive violations**
  - Instructions with different semantics according to the privilege level
  - Ex: X86-32

# Theorem violations

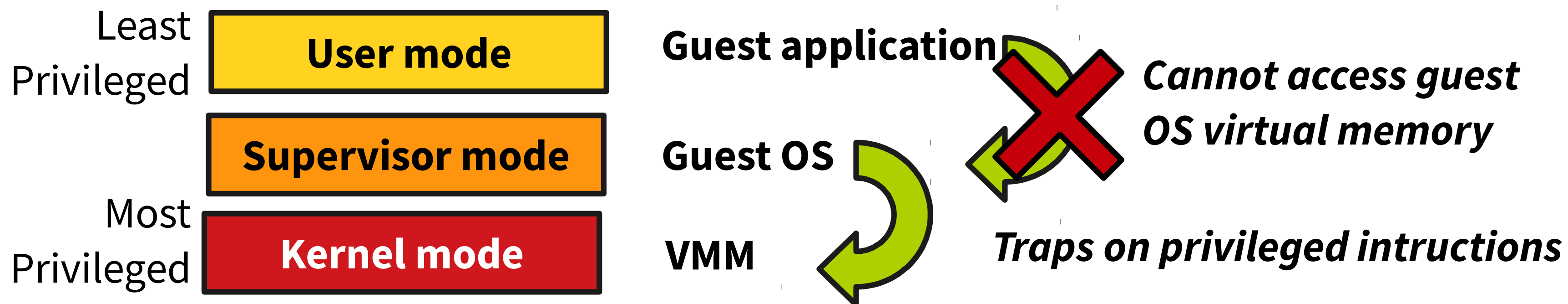
## MIPS

- **MIPS: RISC ISA**
  - 3 execution modes: kernel mode, supervisor mode, user mode
  - Only kernel mode can execute privileged instructions
  - Supervisor mode is user mode + access to additional ranges of virtual memory unavailable from user mode



# Theorem violations
## MIPS

- **MIPS: RISC ISA**
  - 3 execution modes: kernel mode, supervisor mode, user mode
  - Only kernel mode can execute privileged instructions
  - Supervisor mode is user mode + access to additional ranges of virtual memory unavailable from user mode
    - Intuitively, good model for virtualization: we can run everything in the same address space, no need to switch segments and flush TLB on user/OS world switches



# Theorem violations

## MIPS (2)

| Region | Base | Length | Access K,S,U | MMU |
|---|---|---|---|---|
| USEG | 0x0000 0000 | 2 GB | ✓,✓ ✓ | mapped |
| KSEG0 | 0x8000 0000 | 512 MB | ✓,x,x | unmapped |
| KSEG1 | 0xA000 0000 | 512 MB | ✓,x,x | unmapped |
| KSSEG | 0xC000 0000 | 512 MB | ✓,✓,x | mapped |
| KSEG3 | 0xE000 0000 | 512 MB | ✓,x,x | mapped |

Source: textbook

- **Problem: OS compiled for MIPS expect to be able to use KSEG0 and KSEG1**
  - Every memory reference in there would cause a trap if OS run in supervisor mode (not in kernel mode)
    - Violates the efficiency criteria

# Theorem violations

## x86-32

- **Popular CISC ISA**
  - Multiple sensitive and unprivileged instructions
  - More info:
    - Robin, John Scott, and Cynthia E. Irvine. "Analysis of the Intel Pentium's ability to support a secure virtual machine monitor." Proceedings of the 9th USENIX Security Symposium, Denver, CO., 2000.
- **Let's illustrate one x86-32 violation with the "POPF" problem**
  - Source: https://www.cs.cmu.edu/~410-s14/lectures/L30_Virtualization.pdf

# Theorem violations

## x86-32: the POPF issue

- **POPF is behavior sensitive and does not trap**
  - One example of usage is for disabling interrupts

```
PUSHF                      # Push %EFLAGS on the stack
ANDL $0x003FFDFF, (%ESP)  # Clear IF
POPF                       # Load %EFLAGS from stack
```

003FFDFF

Hexadecimal ▾

17776777₈ = 4193791₁₀

```
0000 0000 0000 0000 0000 0000 0000 0000
63                  47                 32
0000 0000 0011 1111 1111 1101 1111 1111
31                  15                  0
```

| Intel x86 FLAGS register[1] | | | |
|---|---|---|---|
| **Bit #** | **Abbreviation** | **Description** | **Category** |
| **FLAGS** | | | |
| 0 | CF | Carry flag | Status |
| 1 | | Reserved, always 1 in EFLAGS [2] | |
| 2 | PF | Parity flag | Status |
| 3 | | Reserved | |
| 4 | AF | Adjust flag | Status |
| 5 | | Reserved | |
| 6 | ZF | Zero flag | Status |
| 7 | SF | Sign flag | Status |
| 8 | TF | Trap flag (single step) | Control |
| 9 | IF | Interrupt enable flag | Control |
| 10 | DF | Direction flag | Control |
| 11 | OF | Overflow flag | Status |
| 12-13 | IOPL | I/O privilege level (286+ only), always 1 on 8086 and 186 | System |
| 14 | NT | Nested task flag (286+ only), always 1 on 8086 and 186 | System |
| 15 | | Reserved, always 1 on 8086 and 186, always 0 on later models | |
| **EFLAGS** | | | |
| 16 | RF | Resume flag (386+ only) | System |
| 17 | VM | Virtual 8086 mode flag (386+ only) | System |
| 18 | AC | Alignment check (486SX+ only) | System |
| 19 | VIF | Virtual interrupt flag (Pentium+) | System |
| 20 | VIP | Virtual interrupt pending (Pentium+) | System |
| 21 | ID | Able to use CPUID instruction (Pentium+) | System |
| 22 | | Reserved | |
| 23 | | Reserved | |
| 24 | | Reserved | |
| 25 | | Reserved | |
| 26 | | Reserved | |
| 27 | | Reserved | |
| 28 | | Reserved | |
| 29 | | Reserved | |
| 30 | | Reserved | |
| 31 | | Reserved | |
| **RFLAGS** | | | |
| 32-63 | | Reserved | |

# Theorem violations

## x86-32: the POPF issue

- **POPF is behavior sensitive and does not trap**
  - One example of usage is for disabling interrupts

```
PUSHF                    # Push %EFLAGS on the stack
ANDL $0x003FFDFF, (%ESP) # Clear IF
POPF                     # Load %EFLAGS from stack
```

- **Works from kernel mode in a non-virtualized OS (it's a privileged operation)**
- **When executed in user mode, *CPU ignores the changes to the privileged EFLAGS bits***
  - *With a P&G-defined VMM, guest OS running in user mode will silently fail to disable interrupts*
    - *No trap, no way for the VMM to emulate*

# Theorem violations

## ARM

- **RISC ISA**
  - Present multiple (24) sensitive but unprivileged instructions
  - Present in Armv6, Armv7, similar issues with Armv8 (aarch64)
- **Examples: LOAD/STOREs user register when in privileged mode**
  - Fail silently (no trap) in user mode
- **More info: see the textbook and this paper:**
  - Christoffer Dall and Jason Nieh, KVM for ARM, *Ottawa Linux Symposium*, 2010

# Further reading


- Popek, Gerald J., and Robert P. Goldberg. "Formal requirements for virtualizable third generation architectures." Communications of the ACM 17.7 (1974): 412-421.
- Irvin, C. E., and J. S. Robin. "Analysis of the Intel Pentium's ability to support a secure virtual machine monitor." Proceedings of the USENIX Security Symposium. USENIX Association. 2000.
- Christoffer Dall and Jason Nieh, KVM for ARM, Ottawa Linux Symposium, 2010