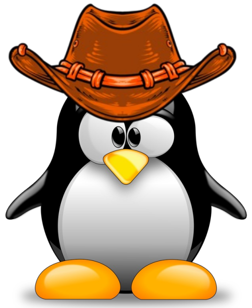


CSE 330: Operating Systems

Adil Ahmad

Lecture #21: System virtualization overview

Recap of FS logging

- All data at system call is first written to in-memory block cache
- Then transferred to the log disk space inside your disk
- Also, FS system calls can happen concurrently

Can you spot the major challenges to ensure smoothness?

Challenges of logging?

- **System call data must fit inside the log disk space**
  - Recall that each write(..) puts all blocks into log regions
- xv6's solutions:
  1. Set an upper bound of all log entries
     - Set log size >= upper bound
  2. Breakdown large writes into smaller transactions
     - Problem: large writes are thus not atomic
     - However, overall system remains in a consistent state

Challenges of logging?

- **Concurrent system call handling**

  1. Must allow different FS system calls from different proc at the same time
  2. Block may be written multiple times while still in-memory
     - Creating a new log entry for each time is wasteful

- xv6's solutions:

  1. Check if a system call's data can fit in log before starting it
     - Else, sleep and wait for log to free up

  2. If the block is already assigned a log entry, return without creating new entry
     - Also called "write absorption" → absorb multiple writes into a single

Pros of xv6’s logging?

- FS internal invariants are maintained
  - Metadata remains consistent (e.g., no two inodes point to same blocks)
- Except for the last few operations, data is preserved on the disk
  - Every system call is written at the end before return
  - No surprises of losing data written *way back*
- Write order is preserved
  - $ echo A > X ; echo B > Y
  - X will be written before Y

Cons of xv6's crash recovery method?

- Several inefficiencies

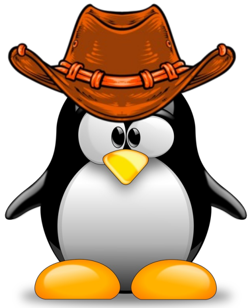
Design choice: ensure either old or new copy is correct. No corruption

  - Every block is written twice (once for log + once for final)

  - Eagerly writes to disk after every write system call
  - Writes each block one-by-one to disk

Implementation choice: *How would you overcome?*

Keep data in-memory for longer periods and batch disk writes



Moving on to system virtualization

What is the meaning of the term ‘virtualization’?

Provide an abstracted or ‘fake’ view of computer resources

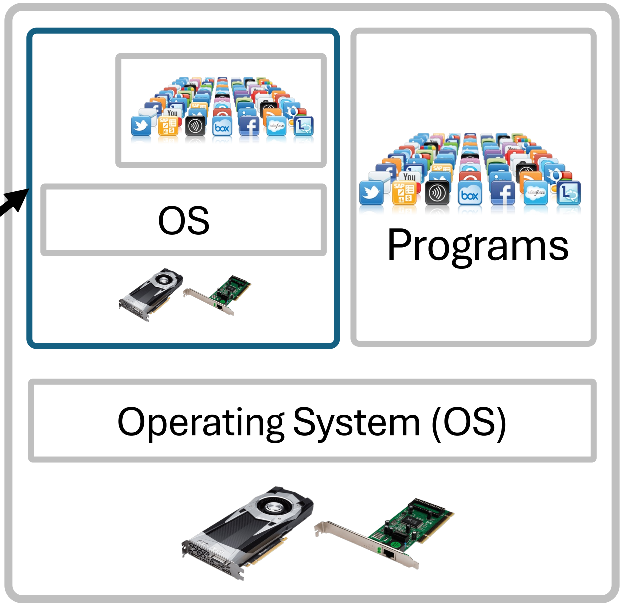
How have we used the concept of ‘virtualization’ so far?

- Memory virtualization for processes
  - Set-up page tables to prevent the process from accessing other regions
- Disk virtualization
  - We leverage the filesystem and block layers to implement access control, etc.

What’s a virtual machine (or a VM)?

A *simulated* instance of a computer inside a computer

A virtual machine



Desired VM properties:

1. Accurate simulation
2. Isolation from ‘host’
3. Fast!

Why do we use VMs in today's world?

A. Cloud computing:

- Each user has their own isolated 'machine'
- Can run different OSs and remain isolated from other users
- Better resource management for cloud providers

B. System development:

- Testing OSs without breaking your own system
- Testing programs built for certain OS/packages

C. System security:

- Many important OS features can be secured by implementing in a higher privileged layer (remember nested kernel?)

Different ways to design a VM today

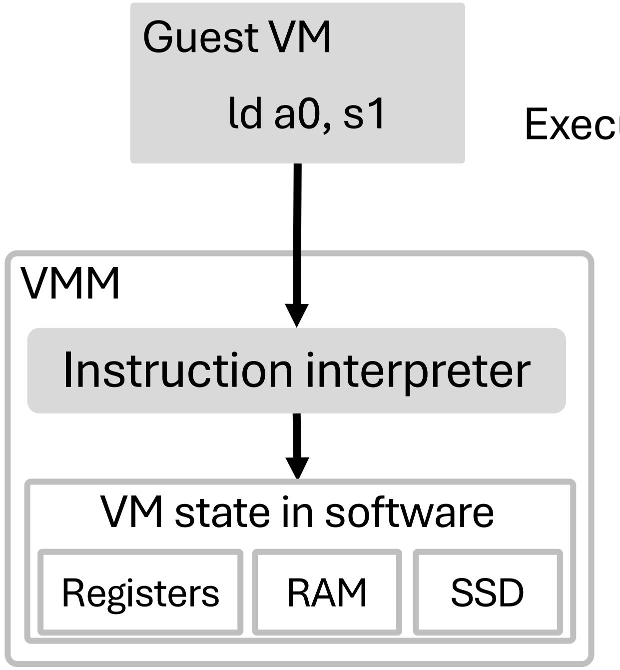
1. Full software-based emulation

2. Trap-and-emulate
   - Also includes para-virtualized (PV) interfaces

3. Hardware-assisted virtualization

1. Full software emulation-based virtualization

Simulate the entire machine's execution in software-mode



Executes any instruction

Transform into a function call to VMM (e.g., using JIT)

Affect VM state that is in memory as data structures

Other state that must be simulated: privileged registers, UART, PMP, etc.

An example of a RISC-V instruction interpretation

- Guest executes address 0xa: “csrr a1,mhartid”
- VMM reads 0xa and finds the binary code: 0xf14025f3
- VMM knows RISC-V instruction convention
  - Instruction is 4-bytes (or 32-bits)
  - Bits 0-6 is instruction opcode (0x73 → CSR instructions)
  - Bits 7-11 is destination register (0xb → a1 register)
  - Bits 15-19 is src1 register (0x0 → zero register/unused)
  - Bits 20-31 is src2 register (0xf14 → mhartid register)
- Based on convention, VMM can keep update simulated registers

What are the advantages of full software emulation?

A. Architecture and OS-agnostic

- Can run any architecture (e.g., RISC-V on x86 using QEMU)
- Do not actually even need support from the Operating System

B. Great for prototyping new devices/hardware

- Testing new CPU features without having real hardware

- Some very famous examples:
  - Intel SGX emulation using QEMU
  - Arm CCA emulation

What are the disadvantages of full software emulation?

A. Pretty slow!

- Must translate *every* instruction as well as maintain internal register/system state (e.g., all registers)

B. Complex VMM design

- Your VMM must have lots of emulation code (e.g., check QEMU's software emulation stack)

- Easy to make mistakes – emulated HW does not mimic actual machine (very common!!)

How can we speed up virtual machine execution?

Allowing VMs to execute instructions on CPU is faster!

- Does not incur the overhead of instruction interpreter
  In fact, we allow a process to execute instructions directly!

- **Why not allow the VM to directly execute its instructions?**
  OS inside the VM might execute a privileged instruction. E.g., load a new page table and defeat memory isolation

- **Is there a solution to this problem?**
  Execute the VM in user-mode (U-mode)?!

2. Trap-and-emulate (T&E) virtualization

- Run the VM as a user-mode process
- User instructions are directly executed by the VM
  - ld a0, s1 → no trap
- Supervisor instructions by the VM OS are trapped to the VMM
  - VMM emulates these just like in full software emulation
- **Advantages of this approach?**
  - Helps avoid the cost of translating *all* instructions
  - Still can build this design in software entirely

What system state must we trap and emulate?

Short answer: any change to privileged CPU state must be T&E-ed

- Page tables (registers)
- Privileged registers (e.g., model-specific registers)
- Interrupt controls (e.g., PLIC/CLINT)
- ..

Understanding the trap semantics in different arch.

How exactly does the VMM trap supervisor instructions?

- RISC-V → easy
  - Automatically handled by the architecture
  - Execute 'csrr' in u-mode – direct trap to the S-mode SW

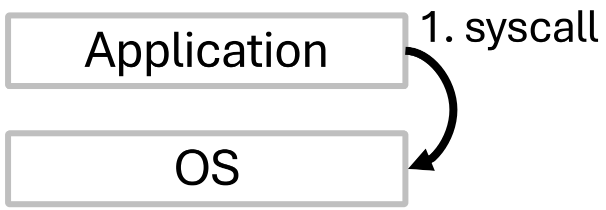
- x86 (32-bit) → hard
  - Executing a privileged instruction → skip the instruction
  - *Solution: para-virtualization.*
    - Make the OS aware of virtualization
    - Change the OS' privileged operations to directly call the VMM using hyper-calls (e.g., VMCALL instruction in x86)

Let’s look at a few important T&E scenarios

1. Executing a system call from U-mode inside the VM
2. Reading a privileged register by the guest OS
3. Making changes to page tables

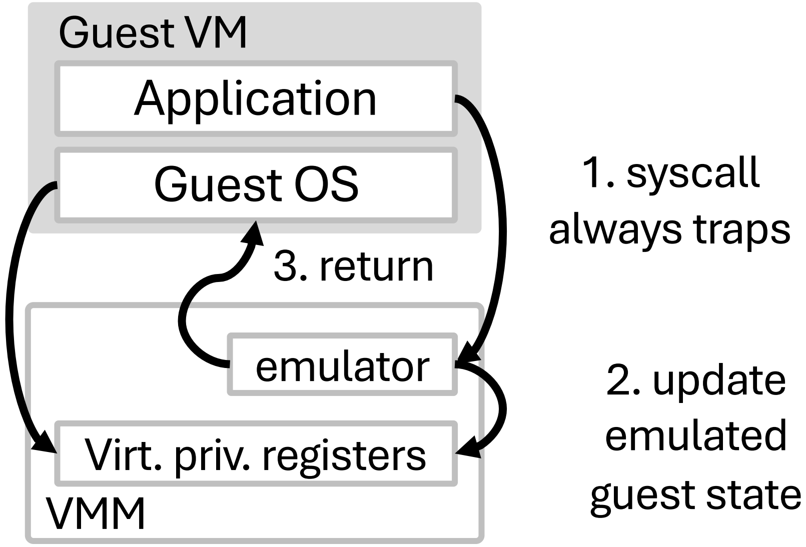
Scenario #1: Executing a system call from the guest

Executing a system call in
a non-virtualized system



Read register
state for syscall

Executing a system call in
trap+emulate system



4. read virtual registers
(next slide)

Scenario #2: Reading from a privileged register from guest

OS tries to read CR3 register. Following happens:

- Trap into the VMM since CR3 is a privileged instruction
- VMM checks what the guest wants by examining guest instruction
- Performs the operation in SW; moves *instruction pointer* to a1
- Moves the instruction; sepc += 4 bytes (4B is instruction size, e.g.,)
- Returns to the guest OS

Scenario #3: Making changes to page tables

Guest OS creates page table and tries to write to CR3 register.

- VMM inspects page table and creates a “shadow”
  - Guest’s page table: guest VA → guest PA (host VA)
  - Current page table: guest PA (host VA) → host PA
  - Shadow page table: guest VA → host PA

- VMM installs shadow to real CR3 after making sure there are no malicious entries (e.g., VM trying to read/write outside)

Disadvantages of trap-and-emulate?

A. Still slow (lots of supervisor-level traps)

- Kernel's performance affects every software's performance
- Can be improved using several techniques:
  - Especially for an 'enlightened' guest; one that knows it's a VM and actively tries to reduce traps

B. Complexity remains

- Privileged operations are the most important operations in a system and their emulation is no trivial matter

How do we speed-up virtualization beyond T&E?

A. Still slow (lots of supervisor-level traps)

- Kernel’s performance affects every software’s performance
- Can be improved using several techniques:
  - Especially for an ‘enlightened’ guest; one that knows it’s a VM and actively tries to reduce traps

B. Complexity remains

- Privileged operations are the most important operations in a system and their emulation is no trivial matter

We should allow the VM to execute ~~all~~ *most* instructions!

- Does not incur the overhead of *any* traps

  Even privileged instructions can be directly executed on the CPU

- **How is this even possible, let alone secure?**
  - Think about a process – save kernel context in TRAPFRAME and move to process execution at ‘sret’ instruction
  - What if we can save the host context and jump to the guest VM’s execution and resume later?

Today’s standard: Hardware-assisted virtualization

- New mode of execution: root mode and non-root mode.
  - Root mode is for host/VMM, non-root mode for guest/VM

- CPUs provide support for isolating the execution of host and VM
  - VMENTER/VMRESUME – instruction to enter/resume a VM from host
  - VMEXIT – exit the VM back to the host

- On VMENTER, host should save all context that it needs for later

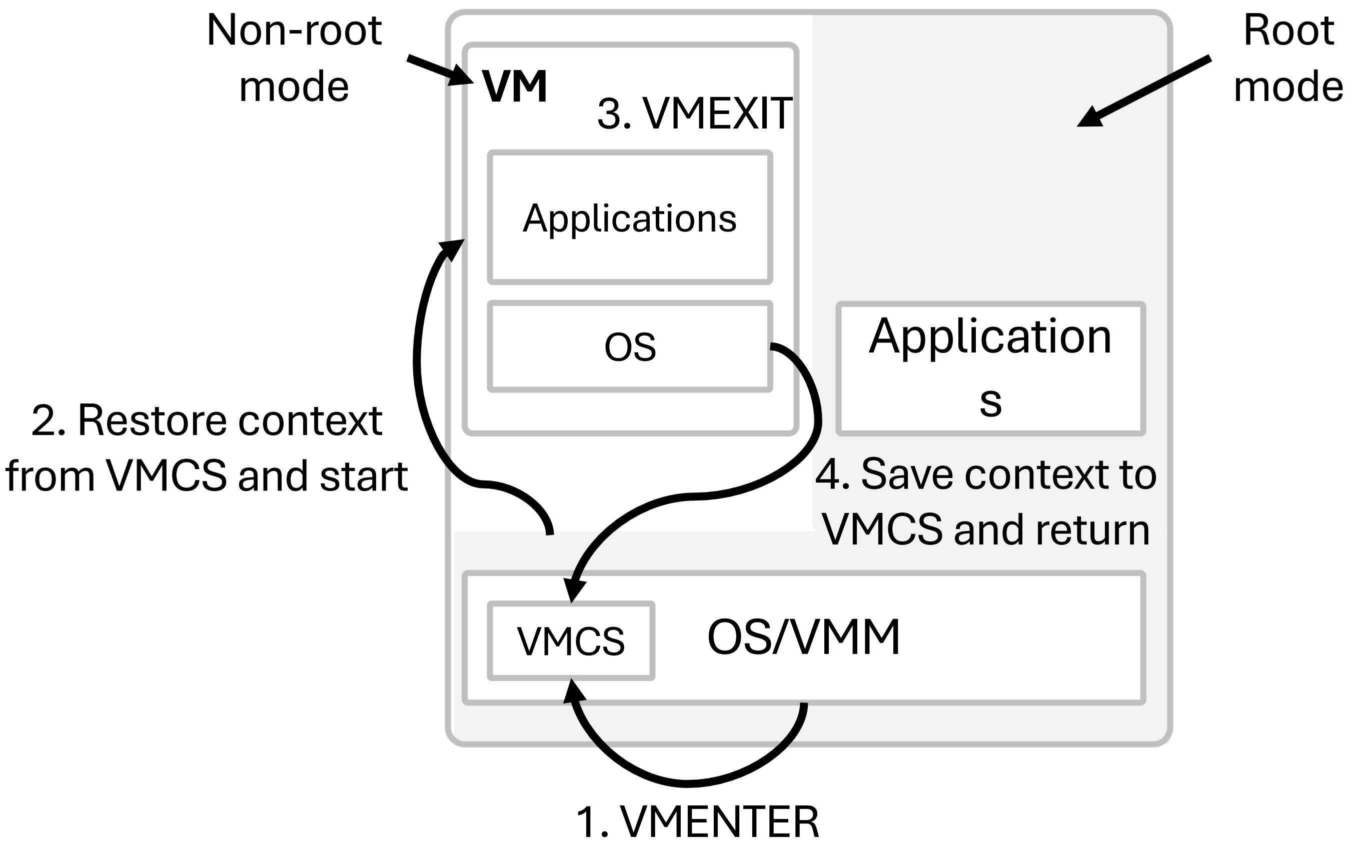
- On VMEXIT, CPU saves guest context in a “TRAPFRAME” called the “Virtual Machine Control Structure” or VMCS (x86)

The context to be saved on exit/entry to VMs

All register contexts including the privileged/unprivileged registers

- Control registers (CRs)
- Segment registers
- Floating point registers, etc.

x86 HW-assisted virtualization illustration



There are three main **security problems**

We need to systematically address the following .

A. Guest could read/write outside its own memory regions

- E.g., using modified page tables

B. Guest could talk to devices that the host wants to isolate.

- Could also refuse to give up control to host at interrupts

C. Guest could adversely impact important system functionality

- Change voltage on a certain CPU core

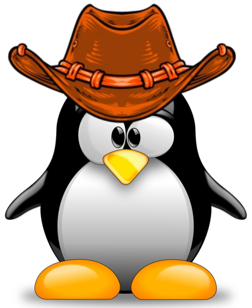
How would you address these problems? (one-by-one)

We need to systematically address the following .

A. Guest could read/write outside its own memory regions
- E.g., using modified page tables

B. Guest could talk to devices that the host wants to isolate.
- Could also refuse to give up control to host at interrupts

C. Guest could adversely impact important system functionality
- Change voltage on a certain CPU core



Questions? Otherwise, see you next class!