Virtualization

Mengjia Yan
Computer Science & Artificial Intelligence Lab
M.I.T.
Abstractions

- Devices
- Materials
- Atoms
Abstractions

Digital design
Combinational and sequential circuits

Devices
Materials
Atoms
Abstractions

Computer architecture
Processors, caches, pipelining

Digital design
Combinational and sequential circuits

Devices
Materials
Atoms
Abstractions

Software

Computer architecture
Processors, caches, pipelining

Digital design
Combinational and sequential circuits

Devices
Materials
Atoms
Abstractions

Software

Computer architecture
Processors, caches, pipelining

Digital design
Combinational and sequential circuits

Devices
Materials
Atoms

Instruction set + memory

Digital circuits

Bits, Logic gates
Abstractions

Computer programs

Virtual machines

Computer systems
Operating systems, virtual memory, I/O

Instruction set + memory

Computer architecture
Processors, caches, pipelining

Digital circuits

Digital design
Combinational and sequential circuits

Bits, Logic gates

Devices
Materials
Atoms
Evolution in Number of Users

IBM 1620
1959

Single User

Runtime loaded with program
Evolution in Number of Users

IBM 1620 1959
Single User
Runtime loaded with program

IBM 360 1960s
Multiple Users
OS for sharing resources
Evolution in Number of Users

IBM 1620
1959
Single User
Runtime loaded with program

IBM 360
1960s
Multiple Users
OS for sharing resources

IBM PC
1980s
Single User
OS for sharing resources
# Evolution in Number of Users

<table>
<thead>
<tr>
<th>Year</th>
<th>System</th>
<th>Users</th>
<th>OS for Sharing Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>1959</td>
<td>IBM 1620</td>
<td>Single User</td>
<td>Runtime loaded with program</td>
</tr>
<tr>
<td>1960s</td>
<td>IBM 360</td>
<td>Multiple Users</td>
<td>OS for sharing resources</td>
</tr>
<tr>
<td>1980s</td>
<td>IBM PC</td>
<td>Single User</td>
<td>OS for sharing resources</td>
</tr>
<tr>
<td>1990s</td>
<td>Cloud Servers</td>
<td>Multiple Users</td>
<td>Multiple OSs</td>
</tr>
</tbody>
</table>
Single-Program Machine

- Hardware executes a single program and has direct and complete access to all hardware resources
Single-Program Machine

- Hardware executes a single program and has direct and complete access to all hardware resources
- The ISA is the interface between software and hardware:
  - Program counter
  - General purpose registers
  - Memory
• Runtime library added to save programming effort and provided *an abstraction to create uniform interface to devices.*
Runtime library added to save programming effort and provided *an abstraction to create uniform interface to devices.*
Multi-Program Machine (1st attempt)
Multi-Program Machine (1\textsuperscript{st} attempt)

Program  
Program  
Runtime Library  
Hardware

Any problems?
Multi-Program Machine (1st attempt)

- Program
- Program
- Runtime Library
- Hardware

Any problems? security
Simple Base and Bound Translation

Load X

Program Address Space

Bound Register

Effective Address

Base Register

Segment Length

≤

Bounds Violation?

Physical Address

Base Physical Address

Main Memory

current segment

November 27, 2023
Simple Base and Bound Translation

Load X

Program Address Space

Main Memory

current segment

Bound Register

Effective Address

Base Register

Base Physical Address

Segment Length

≤

 Bounds Violation?

Physical Address

Load X

Program Address Space

Base Physical Address

Bound Register

Effective Address

Base Register
Introduce a new privileged mode in which the base and bounds registers are visible/accessible.
### Protecting Memory

**Page Table Entry**

<table>
<thead>
<tr>
<th>Valid&lt;31&gt;</th>
<th>Prot&lt;30:27&gt;</th>
<th>Modified&lt;26&gt;</th>
<th>OS&lt;25:21&gt;</th>
<th>PFN&lt;20:0&gt;</th>
</tr>
</thead>
</table>

November 27, 2023
Protecting Memory

Page Table Entry

Valid<31>  Prot<30:27>  Modified<26>  OS<25:21>  PFN<20:0>

TLB Entry

Tag  Valid  Prot  PFN

TLB Fill
Protecting Memory

Page Table Entry

Valid<31>  Prot<30:27>  Modified<26>  OS<25:21>  PFN<20:0>

TLB Entry

Tag  Valid  Prot  PFN

- TLB access checks if protection allows access for current mode
- TLB fills require read/copy page table data -> security sensitive
Operating Systems

- Operating System (OS) goals:
Operating System (OS) goals:

- **Abstraction**: OS hides details of underlying hardware
  - e.g., a process can open and access files instead of issuing raw commands to the disk

- **Resource management**: OS controls how processes share hardware (CPU, memory, disk, etc.)

- **Protection and privacy**: Processes cannot access each other’s data
Operating System Mechanisms

• The OS kernel lets processes invoke system services (e.g., access files or network sockets) via system calls
Operating System Mechanisms

- The OS kernel lets processes invoke system services (e.g., access files or network sockets) via **system calls**

- The OS kernel **schedules processes** into cores
  - Each process is given a fraction of CPU time
  - A process cannot use more CPU time than allowed

Running process

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
<th>Process 1</th>
</tr>
</thead>
</table>

Time
Operating System Mechanisms

- The OS kernel lets processes invoke system services (e.g., access files or network sockets) via system calls.

- The OS kernel schedules processes into cores:
  - Each process is given a fraction of CPU time.
  - A process cannot use more CPU time than allowed.

- The OS kernel provides a private address space to each process:
  - Each process is allocated space in physical memory by the OS.
  - A process is not allowed to access the memory of other processes.
ISA Extensions to Support OS

- Two modes of execution: **user** and **supervisor**
  - OS kernel runs in supervisor mode
  - All other processes run in user mode
ISA Extensions to Support OS

- Two modes of execution: **user** and **supervisor**
  - OS kernel runs in supervisor mode
  - All other processes run in user mode

- **Privileged instructions and registers** that are only available in supervisor mode
ISA Extensions to Support OS

- Two modes of execution: user and supervisor
  - OS kernel runs in supervisor mode
  - All other processes run in user mode
- Privileged instructions and registers that are only available in supervisor mode
- How to transition from user mode to supervisor mode?
ISA Extensions to Support OS

• Two modes of execution: **user** and **supervisor**
  – OS kernel runs in supervisor mode
  – All other processes run in user mode

• Privileged instructions and registers that are only available in supervisor mode

• How to transition from user mode to supervisor mode?
  – **Traps (exceptions)** to safely transition from user to supervisor mode
Process Mode Switching

Trap, e.g., i/o read() or exception

user mode

kernel mode
Process Mode Switching

Trap, e.g., i/o read() or exception

user mode

Switch to kernel mode;
Pass arguments;
Save app state;
Transfer to trap handler

kernel mode
Process Mode Switching

Trap, e.g., i/o read() or exception

user mode

Switch to kernel mode;
Pass arguments;
Save app state;
Transfer to trap handler

kernel mode

Trap handler
Process Mode Switching

Trap, e.g., i/o read() or exception

user mode

Switch to kernel mode;
Pass arguments;
Save app state;
Transfer to trap handler

kernel mode

Trap handler

Must be at fixed addresses
Process Mode Switching

Trap, e.g., i/o read() or exception

Switch to kernel mode;
Pass arguments;
Save app state;
Transfer to trap handler

user mode

kernel mode

 Trap handler

Must be at
fixed addresses

Check arguments;
Find kernel routine addr
Process Mode Switching

Trap, e.g., i/o read() or exception

user mode

Switch to kernel mode;
Pass arguments;
Save app state;
Transfer to trap handler

kernel mode

Trap handler

Must be at fixed addresses

Check arguments;
Find kernel routine addr

Why?
Process Mode Switching

Trap, e.g., i/o read() or exception

user mode

Switch to kernel mode;
Pass arguments;
Save app state;
Transfer to trap handler

kernel mode

Trap handler

Must be at fixed addresses

Check arguments;
Find kernel routine addr

Kernel routine

Why?
Process Mode Switching

Trap, e.g., i/o read() or exception

Switch to kernel mode; 
Pass arguments; 
Save app state; 
Transfer to trap handler

user mode

kernel mode

Trap handler

Must be at fixed addresses

Check arguments; 
Find kernel routine addr

Kernel routine

Why?

Restore app state; 
Return to user
Protection – Single OS

OS Kernel

User Process

Trap

User Process

Trap
Protection – Single OS

Key idea: Provides a strong abstraction that cannot be escaped
Virtual Machines

- The OS gives a Virtual Machine (VM) to each process
  - Each process believes it runs on its own machine...
  - ...but this machine does not exist in physical hardware
Virtual Machines

- The OS gives a Virtual Machine (VM) to each process
  - Each process believes it runs on its own machine...
  - ...but this machine does not exist in physical hardware
Virtual Machines

- The OS gives a Virtual Machine (VM) to each process
  - Each process believes it runs on its own machine...
  - ...but this machine does not exist in physical hardware

---

![Diagram of Virtual Machines and OS Kernel]
Virtual Machines

• The OS gives a Virtual Machine (VM) to each process
  – Each process believes it runs on its own machine...
  – ...but this machine does not exist in physical hardware

OS Kernel (specially privileged process)
Virtual Machines

- A Virtual Machine (VM) is an *emulation* of a computer system
  - Very general concept, used beyond operating systems

![Diagram showing relationships between virtual and physical hardware, processes, and system components.]

**OS Kernel (specially privileged process)**
Virtual Machines Are Everywhere

• Example: Consider a Python program running on a Linux Virtual Machine
Virtual Machines Are Everywhere

• Example: Consider a Python program running on a Linux Virtual Machine
Virtual Machines Are Everywhere

• Example: Consider a Python program running on a Linux Virtual Machine

Python program

Python interpreter (CPython)

Python Language

Implements a Python VM
Virtual Machines Are Everywhere

- Example: Consider a Python program running on a Linux Virtual Machine

<table>
<thead>
<tr>
<th>Python program</th>
<th>Python Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python interpreter (CPython)</td>
<td>Implements a Python VM</td>
</tr>
<tr>
<td>Linux OS kernel</td>
<td>Linux ABI</td>
</tr>
<tr>
<td></td>
<td>Implements a Linux-x86 VM</td>
</tr>
</tbody>
</table>
Virtual Machines Are Everywhere

• Example: Consider a Python program running on a Linux Virtual Machine

- Python program
- Python interpreter (CPython)
- Linux OS kernel
- VirtualBox

Python Language
- Implements a Python VM

Linux ABI
- Implements a Linux-x86 VM

x86 ISA
- Implements an x86 system VM
Virtual Machines Are Everywhere

- Example: Consider a Python program running on a Linux Virtual Machine

<table>
<thead>
<tr>
<th>Python program</th>
<th>Python Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python interpreter (CPython)</td>
<td>Implements a Python VM</td>
</tr>
<tr>
<td>Linux OS kernel</td>
<td>Linux ABI</td>
</tr>
<tr>
<td>VirtualBox</td>
<td>Implements a Linux-x86 VM</td>
</tr>
<tr>
<td>OS kernel (Win/Linux/MacOS/...)</td>
<td>x86 ISA</td>
</tr>
<tr>
<td></td>
<td>Implements an x86 system VM</td>
</tr>
<tr>
<td></td>
<td>Win/Linux/MacOS/... ABI</td>
</tr>
<tr>
<td></td>
<td>Implements an OS-x86 VM</td>
</tr>
</tbody>
</table>
Virtual Machines Are Everywhere

- **Example:** Consider a Python program running on a Linux Virtual Machine

<table>
<thead>
<tr>
<th>Python program</th>
<th>Python Language</th>
<th>Implements a Python VM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python interpreter (CPython)</td>
<td>Linux ABI</td>
<td>Implements a Linux-x86 VM</td>
</tr>
<tr>
<td>Linux OS kernel</td>
<td>x86 ISA</td>
<td>Implements an x86 system VM</td>
</tr>
<tr>
<td>VirtualBox</td>
<td>Win/Linux/MacOS/... ABI</td>
<td>Implements an OS-x86 VM</td>
</tr>
<tr>
<td>OS kernel (Win/Linux/MacOS/...)</td>
<td>x86 ISA</td>
<td>Implements an x86 physical machine</td>
</tr>
<tr>
<td>Hardware (e.g., your laptop)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

November 27, 2023
Implementing Virtual Machines

- Virtual machines can be implemented entirely in software, but at a performance cost
  - e.g., Python programs are 10-100x slower than native Linux programs due to Python interpreter overheads
Implementing Virtual Machines

• Virtual machines can be implemented entirely in software, but at a performance cost
  – e.g., Python programs are 10-100x slower than native Linux programs due to Python interpreter overheads

• We want to support virtual machines with minimal overheads → often need hardware support!
Application-level virtualization

- Programs are usually distributed in a binary format:
  - Encodes the program’s instructions and initial values of data segments.
  - Conforms to the application binary interface (ABI).
Application-level virtualization

• Programs are usually distributed in a *binary format*:
  – Encodes the program’s instructions and initial values of data segments.
  – Conforms to the application binary interface (ABI).

• ABI specifications include
  – Which instructions are available (the ISA)
  – What system calls are possible (I/O, or the *environment*)
  – What state is available at process creation
Application-level virtualization

• Programs are usually distributed in a binary format:
  – Encodes the program’s instructions and initial values of data segments.
  – Conforms to the application binary interface (ABI).

• ABI specifications include
  – Which instructions are available (the ISA)
  – What system calls are possible (I/O, or the environment)
  – What state is available at process creation

• Operating system implements the virtual environment
  – At process startup, OS reads the binary program, creates an environment for it, then begins to execute the code, handling traps for I/O calls, emulation, etc.
Full ISA-Level Virtualization

Run programs for one ISA on hardware with different ISA (for compatibility, platform-independent):
Full ISA-Level Virtualization

Run programs for one ISA on hardware with different ISA (for compatibility, platform-independent):

- Run-time Hardware Emulation
  - IBM System 360 had IBM 1401 emulator in microcode
  - Intel Itanium converted x86 to native VLIW (two software-visible ISAs)
  - ARM cores support 64-bit ARM, 32-bit ARM, 16-bit Thumb
Full ISA-Level Virtualization

Run programs for one ISA on hardware with different ISA (for compatibility, platform-independent):

• Run-time Hardware Emulation
  – IBM System 360 had IBM 1401 emulator in microcode
  – Intel Itanium converted x86 to native VLIW (two software-visible ISAs)
  – ARM cores support 64-bit ARM, 32-bit ARM, 16-bit Thumb

• Run-time Software Emulation (*OS software interprets instructions*)
  – E.g., OS for PowerPC Macs had emulator for 68000 code
Full ISA-Level Virtualization

Run programs for one ISA on hardware with different ISA (for compatibility, platform-independent):

- **Run-time Hardware Emulation**
  - IBM System 360 had IBM 1401 emulator in microcode
  - Intel Itanium converted x86 to native VLIW (two software-visible ISAs)
  - ARM cores support 64-bit ARM, 32-bit ARM, 16-bit Thumb

- **Run-time Software Emulation** (*OS software interprets instructions*)
  - E.g., OS for PowerPC Macs had emulator for 68000 code

- **Static Binary Translation** (*convert at install time, load time, or offline*)
  - IBM AS/400 to modified PowerPC cores
  - DEC tools for VAX->Alpha and MIPS->Alpha
Full ISA-Level Virtualization

Run programs for one ISA on hardware with different ISA (for compatibility, platform-independent):

- **Run-time Hardware Emulation**
  - IBM System 360 had IBM 1401 emulator in microcode
  - Intel Itanium converted x86 to native VLIW (two software-visible ISAs)
  - ARM cores support 64-bit ARM, 32-bit ARM, 16-bit Thumb

- **Run-time Software Emulation** (*OS software interprets instructions*)
  - E.g., OS for PowerPC Macs had emulator for 68000 code

- **Static Binary Translation** (*convert at install time, load time, or offline*)
  - IBM AS/400 to modified PowerPC cores
  - DEC tools for VAX->Alpha and MIPS->Alpha

- **Dynamic Binary Translation** (*non-native to native ISA at run-time*)
  - Sun’s HotSpot Java JIT (just-in-time) compiler
  - Transmeta Crusoe, x86->VLIW code morphing
Partial ISA-level virtualization

Implement part of ISA in software to trade-off between performance and cost (make the common things fast):
Partial ISA-level virtualization

Implement part of ISA in software to trade-off between performance and cost (make the common things fast):

- Expensive but rarely used instructions can cause trap to OS emulation routine:
  - e.g., decimal arithmetic in µVax implementation of VAX ISA
Partial ISA-level virtualization

Implement part of ISA in software to trade-off between performance and cost (make the common things fast):

• Expensive but rarely used instructions can cause trap to OS emulation routine:
  – e.g., decimal arithmetic in µVax implementation of VAX ISA

• Infrequent but difficult operand values can cause trap
  – e.g., IEEE floating-point denormals cause traps in almost all floating-point unit implementations
Partial ISA-level virtualization

Implement part of ISA in software to trade-off between performance and cost (make the common things fast):

• Expensive but rarely used instructions can cause trap to OS emulation routine:
  – e.g., decimal arithmetic in µVax implementation of VAX ISA

• Infrequent but difficult operand values can cause trap
  – e.g., IEEE floating-point denormals cause traps in almost all floating-point unit implementations

• Old machine can trap unused opcodes, allows binaries for new ISA to run on old hardware
  – e.g., Sun SPARC v8 added integer multiply instructions, older v7 CPUs trap and emulate
Motivation for Multiple OSs

Some motivations for using multiple operating systems on a single computer:

• Allows use of capabilities of multiple distinct operating systems
Motivation for Multiple OSs

Some motivations for using multiple operating systems on a single computer:

- Allows use of capabilities of multiple distinct operating systems
- Allows different users to share a system while using completely independent software stacks
Motivation for Multiple OSs

Some motivations for using multiple operating systems on a single computer:

- Allows use of capabilities of multiple distinct operating systems
- Allows different users to share a system while using completely independent software stacks
- Allows for load balancing and migration across multiple machines
Motivation for Multiple OSs

Some motivations for using multiple operating systems on a single computer:

- Allows use of capabilities of multiple distinct operating systems
- Allows different users to share a system while using completely independent software stacks
- Allows for load balancing and migration across multiple machines
Motivation for Multiple OSs

Some motivations for using multiple operating systems on a single computer:

• Allows use of capabilities of multiple distinct operating systems

• Allows different users to share a system while using completely independent software stacks

• Allows for load balancing and migration across multiple machines

• Allows operating system development without making entire machine unstable or unusable
Supporting Multiple OSs

- process_1, ..., process_N
- OS Kernel_1, ..., OS Kernel_K
- Virtual Machine Monitor (VMM/Hypervisor)
- Hardware
Supporting Multiple OSs

- A VMM (aka Hypervisor) provides a system virtual machine to each OS
Supporting Multiple OSs

- A VMM (aka Hypervisor) provides a *system virtual machine* to each OS
- VMM can run directly on hardware (as above) or on another OS
  - Precisely, VMM can be implemented against an ISA (as above) or a process-level ABI. Who knows what lays below the interface...
Virtualization Nomenclature

From (Machine we are attempting to execute)

- Guest
- Client
- Foreign ISA

To (Machine that is doing the real execution)

- Host
- Target
- Native ISA
Virtual Machine Requirements
[Popek and Goldberg, 1974]

• Equivalence/Fidelity: A program running on the VMM should exhibit a behavior essentially *identical* to that demonstrated when running on an equivalent machine directly.

• Resource control/Safety: The VMM must be in complete control of the *virtualized resources*.

• Efficiency/Performance: A statistically dominant fraction of machine instructions must be executed without VMM intervention.
Virtual Machine Requirements
[Popek and Goldberg, 1974]

Classification of instructions into 3 groups:

• Privileged instructions: Instructions that trap if the processor is in user mode and do not trap if it is in a more privileged mode. (previously defined)

• Control-sensitive instructions: Instructions that attempt to change the configuration of resources in the system.

• Behavior-sensitive instructions: Those whose behavior depends on the configuration of resources, e.g., mode

Building an effective VMM for an architecture is possible if the set of sensitive instructions is a subset of the set of privileged instructions.

Run guest-OS code using the trap-and-emulate strategy.
Sensitive instruction handling

Non-VMM mode

Switch to VMM mode; Pass arguments; Save app state

VMM mode

Find handler addr

VMM handler

VMM routine

Restore app state, Return to guest
Protection – Multiple OS

- VMM
- OS Kernel
- User Process
- Trap
- Sensitive

November 27, 2023
Virtual Memory in VMs

Virtual Address Space of Process-1

Physical Address Space

0
1
2
3

1
0
3
2
Virtual Memory in VMs

Virtual Address Space of Process-1

Physical Address Space

App-a

App-b

Guest OS
Virtual Memory in VMs

- Emulate the physical memory for the VM

- Virtual Address Space of App-a inside VM
- Virtual Address Space of App-b inside VM
- Virtual Address Space of Process-1
  -> Emulate the physical memory for the VM

- Physical Address Space
  0
  1
  2
  3
Virtual Memory in VMs

Guest Virtual Address

(gVA)

Virtual Address Space of App-a inside VM

Virtual Address Space of App-b inside VM

Virtual Address Space of Process-1

-> Emulate the physical memory for the VM

Physical Address Space
Virtual Memory in VMs

Guest Virtual Address (gVA)

Virtual Address Space of App-a inside VM

Virtual Address Space of App-b inside VM

Host Virtual Address (hVA)

= Guest Physical Address (gPA)

Virtual Address Space of Process-1

-> Emulate the physical memory for the VM
Virtual Memory in VMs

Guest Virtual Address
(gVA)

Virtual Address Space of App-a inside VM

Virtual Address Space of Process-1
-> Emulate the physical memory for the VM

= Guest Physical Address
(gPA)

Host Virtual Address
(hVA)

= Guest Physical Address
(gPA)

Host Physical Address
(hPA)

Physical Address Space
Nested Page Tables

Guest VA

Guest Page Table Base

Guest Page Table

Page Table

Host Page Table Base

Page Table

Host PA

Guest PA

How many accesses do we need?

Guest PA == Host VA

Index Offset

PTE

PPN Offset

November 27, 2023
Nested Page Tables

Guest VA

Guest Page Table Base

Guest Page

Table Base

Index

Offset

PTE

gPA->hPA

Page Table

PPN

Offset

Guest PA

Guest PA == Host VA

Host Page Table Base

Host Page

Table Base

Index

Offset

PTE

gPA->hPA

Page Table

PPN

Offset

Host PA

How many accesses do we need?

November 27, 2023

November 27, 2023
Nested Page Tables

Guest VA

Guest Page Table Base → PTE → PPN → Offset

Page Table

Index

Offset

gPA->hPA

Guest PA

Guest PA == Host VA

Index

Offset

Host Page Table Base → PTE → PPN → Offset

Page Table

Host PA

How many accesses do we need? 1 -> 3

November 27, 2023

MIT 6.5900 Fall 2023

L23-32
Nested Page Tables (Hierarchical)

Guest VA

Guest Page Table Base

Index 1  Index 2  Offset

L1 Table

PTP

L2 Table

PPN  Offset

Guest PA == Host VA

Host Page Table Base

Index 1  Index 2  Offset

L1 Table

PTP

L2 Table

PPN  Offset

How many accesses do we need?

Guest PA

Host PA

PPN  Offset

November 27, 2023

MIT 6.5900 Fall 2023
Nested Page Tables (Hierarchical)

Guest VA

Guest Page Table Base

Index 1

Index 2

Offset

gPA->hPA

L1 Table

PTP

PTE

PPN

Offset

L2 Table

Guest PA == Host VA

Host Page Table Base

Index 1

Index 2

Offset

Host PA

gPA->hPA

L1 Table

PTP

PTE

PPN

Offset

L2 Table

How many accesses do we need?

Guest PA

November 27, 2023
Nested Page Tables (Hierarchical)

How many accesses do we need? 2 -> 8
How many accesses do we need?
Shadow Page Tables

How many accesses do we need?
Shadow Page Tables

Guest VA

Guest VA

Index 1

Index 2

Offset

Guest Page Table Base

Guest Page Table Base

PTP

PTP

L1 Table

L1 Table

PTE

PTE

L2 Table

L2 Table

Guest PA

Host PA

How many accesses do we need? 2 \rightarrow 2

November 27, 2023

MIT 6.5900 Fall 2023
Shadow Page Tables

What if guest OS changes the guest page table?
# Nested vs Shadow Paging

<table>
<thead>
<tr>
<th></th>
<th>Native</th>
<th>Nested Paging</th>
<th>Shadow Paging</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TLB Hit</strong></td>
<td>VA-&gt;PA</td>
<td>gVA-&gt;hPA</td>
<td>gVA-&gt;hPA</td>
</tr>
<tr>
<td><strong>TLB Miss (max)</strong></td>
<td>4</td>
<td>24</td>
<td>4</td>
</tr>
<tr>
<td><strong>PTE Updates</strong></td>
<td>Fast</td>
<td>Fast</td>
<td>Uses VMM</td>
</tr>
</tbody>
</table>

On x86-64
Supporting Multiple Process Groups

process_1 ... process_N  process_1 ... process_M

Container              Container

OS Kernel

Hardware
Supporting Multiple Process Groups

• A “container” provides a process group virtual machine to each set of processes
Supporting Multiple Process Groups

- A “container” provides a process group virtual machine to each set of processes
- Container can run directly on OS, which provides a specific OS ABI to the processes in container
Container Semantics

• Isolation between containers is maintained by the OS, which supports a virtualized set of kernel calls.
  – Therefore, processes in all containers must target the same OS*

• Per Container Resources
  – Set of processes (each with a virtual memory space)
  – Set of filesystems
  – Set of network interfaces and ports
  – Selected devices

*Or closely related variants
Security and Side Channels

• Hardware isolation mechanisms like virtual memory guarantee that architectural state will not be directly exposed to other processes...and

• ISA and ABI are *timing-independent* interfaces
  – Specify *what* should happen, not *when*

• ...so non-architectural state and other implementation details and timing behaviors (e.g., microarchitectural state, power, etc.) may be used as *side channels* to leak information!
• 6.S984: Datacenter Computing
• Instructor: Christina Delimitrou
• Short description:
  – Datacenter Computing explores the end-to-end stack of modern datacenters, from hardware and OS all the way to resource managers and programming frameworks.
  – The class will also explore cross-cutting issues, such as ML for systems, energy efficiency, availability, security, and reliability.
  – The main deliverable for the course is a semester-long research project on cloud computing, done in groups of 2-3 students. We will provide a list of suggested projects, but students are also encouraged to suggest their own.

• Lecture time: TR1-2:30
Thank you!