Virtualization

Joel Emer
Computer Science & Artificial Intelligence Lab
M.I.T.
Evolution in Number of Users

IBM 1620
1959

Single User
Runtime loaded with program
# Evolution in Number of Users

<table>
<thead>
<tr>
<th>IBM 1620</th>
<th>IBM 360</th>
</tr>
</thead>
<tbody>
<tr>
<td>1959</td>
<td>1960s</td>
</tr>
</tbody>
</table>

**Single User**
- Runtime loaded with program

**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

IBM 1620
1959

IBM 360
1960s

IBM PC
1980s

Cloud Servers
1990s

Single User
Multiple Users
Single User
Multiple Users

Runtime loaded with program
OS for sharing resources
OS for sharing resources
Multiple OSs
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
Single-Program Machine

- Hardware executes a single program and has direct and complete access to all hardware resources.
- The architecture is the interface between software and hardware:
Single-Program Machine

- Hardware executes a single program and has direct and complete access to all hardware resources
- The architecture is the interface between software and hardware:
  - Program counter
Single-Program Machine

- Hardware executes a single program and has direct and complete access to all hardware resources.
- The architecture is the interface between software and hardware:
  - Program counter
  - General purpose registers
Single-Program Machine

- Hardware executes a single program and has direct and complete access to all hardware resources.
- The architecture is the interface between software and hardware:
  - Program counter
  - General purpose registers
  - Memory
Single-Program Machine (with RTL)

- 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)

- The architecture is the interface between software and hardware:
  - Program counter
  - General purpose registers
  - Memory

Any problems?
Simple Base and Bound Translation

Program Address Space

Load X

Bounds Register

Effective Address

Segment Length

≤

Bounds Violation?

Base Register

Physical Address

Physical Address

Base Physical Address

Main Memory

current segment

Segment Length
Introduce a new privileged mode in which the base and bounds registers are visible/accessible.
Protecting Memory

Page Table Entry

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

## TLB Entry

<table>
<thead>
<tr>
<th>Tag</th>
<th>Valid</th>
<th>Prot</th>
<th>PFN</th>
</tr>
</thead>
</table>
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 Fill

- TLB fill is a privileged operation.
- TLB access checks if protection allows access for current mode
Operating Systems

- Operating System (OS) goals:

  ![Diagram showing the relationship between processes, OS Kernel, ISA, and Application Binary Interface (ABI)]
Operating Systems

- 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
Operating Systems

- **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.)
Operating Systems

- **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 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
Operating System Mechanisms

- 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
Operating System Mechanisms

- 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

- 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
Operating System Mechanisms

- 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

- 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
Operating System Mechanisms

• 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

• 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 lets processes invoke system services (e.g., access files or network sockets) via system calls
ISA Extensions to Support OS
ISA Extensions to Support OS

• Virtual memory to provide private address spaces and abstract the storage resources of the machine
ISA Extensions to Support OS

- Virtual memory to provide private address spaces and abstract the storage resources of the machine
- Two modes of execution: user and supervisor
ISA Extensions to Support OS

- **Virtual memory** to provide private address spaces and abstract the storage resources of the machine
- **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

- **Virtual memory** to provide private address spaces and abstract the storage resources of the machine
- **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

- **Virtual memory** to provide private address spaces and abstract the storage resources of the machine
- **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
- **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

kernel mode

Switch to kernel mode;
Pass arguments;
Save app state;
Transfer to 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
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 well-known addresses
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 well-known addresses

Check arguments;
Find kernel routine addr
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 well-known addresses

Check arguments;
Find kernel routine addr

Why?
Process Mode Switching

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

user mode

kernel mode

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

Trap handler

Must be at well-known 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 well-known addresses

Check arguments; Find kernel routine addr

Kernel routine

Why?

Restore app state; Return to user
Protection – Single OS

OS Kernel

User Process

User Process

Trap

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
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
A Virtual Machine (VM) is an *emulation* of a computer system

- Very general concept, used beyond operating systems
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 Language
Virtual Machines Are Everywhere

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

- Python program
- Python interpreter (CPython)
- Python Language
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

   - Python program
   - Python interpreter (CPython)
   - Python Language
   - Implements a Python VM
   - Linux ABI
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

Linux OS kernel

Linux ABI
Virtual Machines Are Everywhere

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

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

- Python Language
  - Implements a Python VM
- Linux ABI
  - Implements a Linux-x86 VM
Virtual Machines Are Everywhere

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

- Python program
- Python interpreter (CPython) Implements a Python VM
- Linux OS kernel Implements a Linux-x86 VM
- Python Language
- Linux ABI
- x86 ISA
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
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

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

Python Language
- Implements a Python VM
- Linux ABI
- x86 ISA
- Win/Linux/MacOS/... ABI

Novemeber 29, 2021
MIT 6.823 Fall 2021
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>
</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>
</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>
</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>Win/Linux/MacOS/... ABI</td>
</tr>
<tr>
<td></td>
<td>Implements an x86 system VM</td>
</tr>
<tr>
<td></td>
<td>Implements an OS-x86 VM</td>
</tr>
<tr>
<td></td>
<td>x86 ISA</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

OS kernel (Win/Linux/MacOS/...)

Hardware (e.g., your laptop)

Python Language
  Implements a Python VM

Linux ABI
  Implements a Linux-x86 VM

x86 ISA
  Implements an x86 system VM

Win/Linux/MacOS/... ABI
  Implements an OS-x86 VM

x86 ISA
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
  - Linux OS kernel
    - Linux ABI
    - Implements a Linux-x86 VM
    - x86 ISA
    - Implements an x86 system VM
  - VirtualBox
    - Win/Linux/MacOS/... ABI
  - OS kernel (Win/Linux/MacOS/...)
  - Implements an OS-x86 VM
    - x86 ISA
    - Implements an x86 physical machine
  - Hardware (e.g., your laptop)
Application-level virtualization

• Programs are usually distributed in a binary format that encodes the program’s instructions and initial values of some data segments. These requirements are called the application binary interface (ABI), which can be virtualized.

• 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
Full ISA-Level Virtualization

Run programs for one ISA on hardware with different ISA
Full ISA-Level Virtualization

Run programs for one ISA on hardware with different ISA

- 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

• 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

• Emulation (*OS software interprets instructions at run-time*)
  – 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

- **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

- **Emulation (OS software interprets instructions at run-time)**
  - 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

- **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

- **Emulation (OS software interprets instructions at run-time)**
  - 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
Partial ISA-level virtualization

Often good idea to implement part of ISA in software:
Partial ISA-level virtualization

Often good idea to implement part of ISA in software:

- 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

Often good idea to implement part of ISA in software:

- 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

Often good idea to implement part of ISA in software:

- 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
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 → need hardware support!
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
- Allows operating system development without making entire machine unstable or unusable
Supporting Multiple OSs

process₁ \cdots \text{process}_N \quad \text{process}_1 \cdots \text{process}_M

OS Kernel₁ \cdots \text{OS Kernel}_K

Virtual Machine Monitor (VMM)

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
[Popec 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.

- 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.
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

Diagram showing the flow of processes and trap mechanisms within a virtual machine monitor (VMM) and OS kernels, with user processes managed by the VMM for protection.
Virtual Memory Operations

TLB can be designed to translate guest virtual addresses (gVA) to a host physical address (hPA), but...
Virtual Memory Operations

TLB can be designed to translate guest virtual addresses (gVA) to a host physical address (hPA), but...

- TLB misses are a ‘sensitive’ operation
Virtual Memory Operations

TLB can be designed to translate guest virtual addresses (gVA) to a host physical address (hPA), but...

- TLB misses are a ‘sensitive’ operation
- TLB misses happen very, very frequently
Virtual Memory Operations

TLB can be designed to translate guest virtual addresses (gVA) to a host physical address (hPA), but...

• TLB misses are a ‘sensitive’ operation
• TLB misses happen very, very frequently

• So how expensive are TLB fills?
Nested Page Tables

Guest VA

Index

Offset

Guest Page Table Base

PTE

Page Table

PPN

Offset

Guest PA
Nested Page Tables

Guest VA

Index Offset

Guest Page Table Base

Guest PA == Host VA

Index Offset

Host Page Table Base

Host PA

Page Table

PPN Offset

PTE

Guest PA

PPN Offset

PTE

Page Table
Nested Page Tables (Hierarchical)
Nested Page Tables (Hierarchical)

Guest VA

Guest Page Table Base

Index 1

Index 2

Offset

L1 Table

PTP

PTE

L2 Table

PPN

Offset

Guest PA

November 29, 2021

MIT 6.823 Fall 2021

L23-29
Nested Page Tables (Hierarchical)

Guest VA

Guest Page Table Base

Index 1 | Index 2 | Offset

PTP

L1 Table

L2 Table

Guest PA == Host VA

Host Page Table Base

Index 1 | Index 2 | Offset

PTP

L1 Table

L2 Table

Guest PA

Host PA

Host PA

Guest PA
Shadow Page Tables

- Guest VA
- Guest Page Table Base
- Index 1
- Index 2
- Offset
- PTP
- L1 Table
- PTE
- L2 Table
- PPN
- Offset
- Guest PA
Shadow Page Tables

Guest VA

Guest Page Table Base

Index 1

Index 2

Offset

L1 Table

PTP

L2 Table

PTE

Guest PA

PTP

Index 1

Index 2

Offset

PPN

Guest VA

Shadow Page Table Base

L1 Table

PTP

L2 Table

PTE

Host PA

PPN
## 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\(_M\)

Hardware

OS Kernel

Container

Container

ISA

ABI
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
Security and Side Channels

- Hardware isolation mechanisms like virtual memory guarantee that architectural state will not be directly exposed to other processes...and
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*
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!
Thank you!