Virtualization

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

Computer architecture
- Instruction set + memory
- Processors, caches, pipelining

Digital design
- Digital circuits
- Combinational and sequential circuits

Devices
- Bits, Logic gates
- Materials
- Atoms
## Evolution in Number of Users

<table>
<thead>
<tr>
<th>Year</th>
<th>Type</th>
<th>Platform</th>
<th>Users</th>
<th>OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1959</td>
<td>Single User</td>
<td>IBM 1620</td>
<td>Single User</td>
<td>Runtime</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>loaded with</td>
</tr>
<tr>
<td>1960s</td>
<td>Multiple Users</td>
<td>IBM 360</td>
<td>Multiple Users</td>
<td>OS for</td>
</tr>
<tr>
<td>1980s</td>
<td>Single User</td>
<td>IBM PC</td>
<td>Single User</td>
<td>OS for</td>
</tr>
<tr>
<td>1990s</td>
<td>Multiple Users</td>
<td>Cloud Servers</td>
<td>Multiple Users</td>
<td>OSs</td>
</tr>
</tbody>
</table>
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
Single-Program Machine (with RTL)

• Runtime library added to save programming effort and provided an abstraction to create uniform interface to devices.
Multi-Program Machine (1\textsuperscript{st} attempt)

Program

Program

Runtime Library

Hardware

Any problems? security
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

```
process_1  process_2  ...  process_N
```

```
OS Kernel
```

```
Hardware
```

```
Application Binary Interface (ABI)
```

```
ISA
```

December 5, 2022
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**
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> |

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
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

1. Trap, e.g., i/o read() or exception
2. Switch to kernel mode; Pass arguments; Save app state; Transfer to trap handler
3. Check arguments; Find kernel routine addr
4. Must be at fixed addresses
5. Why?
6. Trap handler
7. Kernel routine
8. Restore app state; Return to user
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
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

- 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**
- Implements an x86 physical machine
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 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

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

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
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
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.
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.

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

Sensitive instruction

Non-VMM mode

Switch to VMM mode;
Pass arguments;
Save app state

VMM mode

VMM handler

Find handler addr

VMM routine

Restore app state,
Return to guest
Protection – Multiple OS

Diagram showing a VMM (Virtual Machine Monitor) with two OS kernels, each containing sensitive information. There are user processes connected to each OS kernel, and arrows indicating traps and denials.
Virtual Memory in VMs

- App-a
- App-b
- Guest OS

Virtual Address Space of Process-1

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
</table>

Physical Address Space

- 1
- 0
- 3
- 2
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)

Host Physical Address (hPA)

Virtual Address Space of Process-1

Physical Address Space

0
1
2
3

1
0
3
2
Nested Page Tables

Guest VA

Index          Offset

Guest Page Table Base

PTE

Page Table

Guest PA

Index          Offset

Guest PA == Host VA

Guest Page Table Base

PTE

Host Page Table Base

Page Table

PPN          Offset

Host PA

PPN          Offset

How many accesses do we need?
Nested Page Tables

How many accesses do we need? 1 => 3
Nested Page Tables (Hierarchical)

How many accesses do we need?
Nested Page Tables (Hierarchical)

Guest VA

Guest Page Table Base

Index 1

Index 2

Offset

gPA->hPA

Guest Page Table Base

Index 1

Index 2

Offset

gPA->hPA

Guest PA == Host VA

Host Page Table Base

Index 1

Index 2

Offset

How many accesses do we need? 2 => 8

Guest PA

How many accesses do we need? 2 => 8

Host PA

PPN

Offset

PPN

Offset

23 - 34
Shadow Page Tables

Guest VA

Guest Page Table Base

Index 1  Index 2  Offset

L1 Table

PTP

L2 Table

PTE

Guest PA

Host PA

How many accesses do we need? 2 => 2

Guest VA

Index 1  Index 2  Offset

Shadow Page Table Base

L1 Table

PTP

L2 Table

PTE

PPN  Offset

How many accesses do we need? 2 => 2
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

- 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
Coming Spring 2023 ...

- **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:** two 1.5-hour meetings per week
Thank you!