# Virtual Machines

## CS 423 - University of Illinois

Wade Fagen-Ulmschneider
(Slides built from Adam Bates and Tianyin Xu previous work on CS 423.)

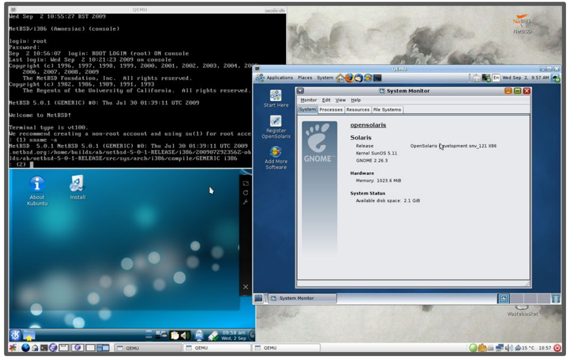
# Big Idea: The OS is an illusionist

- **So Far, the OS makes it appear that every process has:**
  - exclusive, continuous access to the **CPU**,
  - a large, nearly infinite unbounded amount of **RAM**,
  - *...but secretly swaps the resources between many processes...*
- Do we really need more abstraction??

# Big Idea: The OS is an illusionist

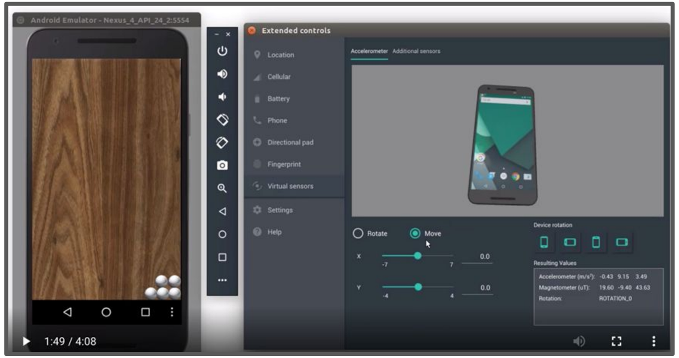

## Hardware Platform Virtualization

Running hardware platform-specific binaries on different hardware.



## Operating System Virtualization

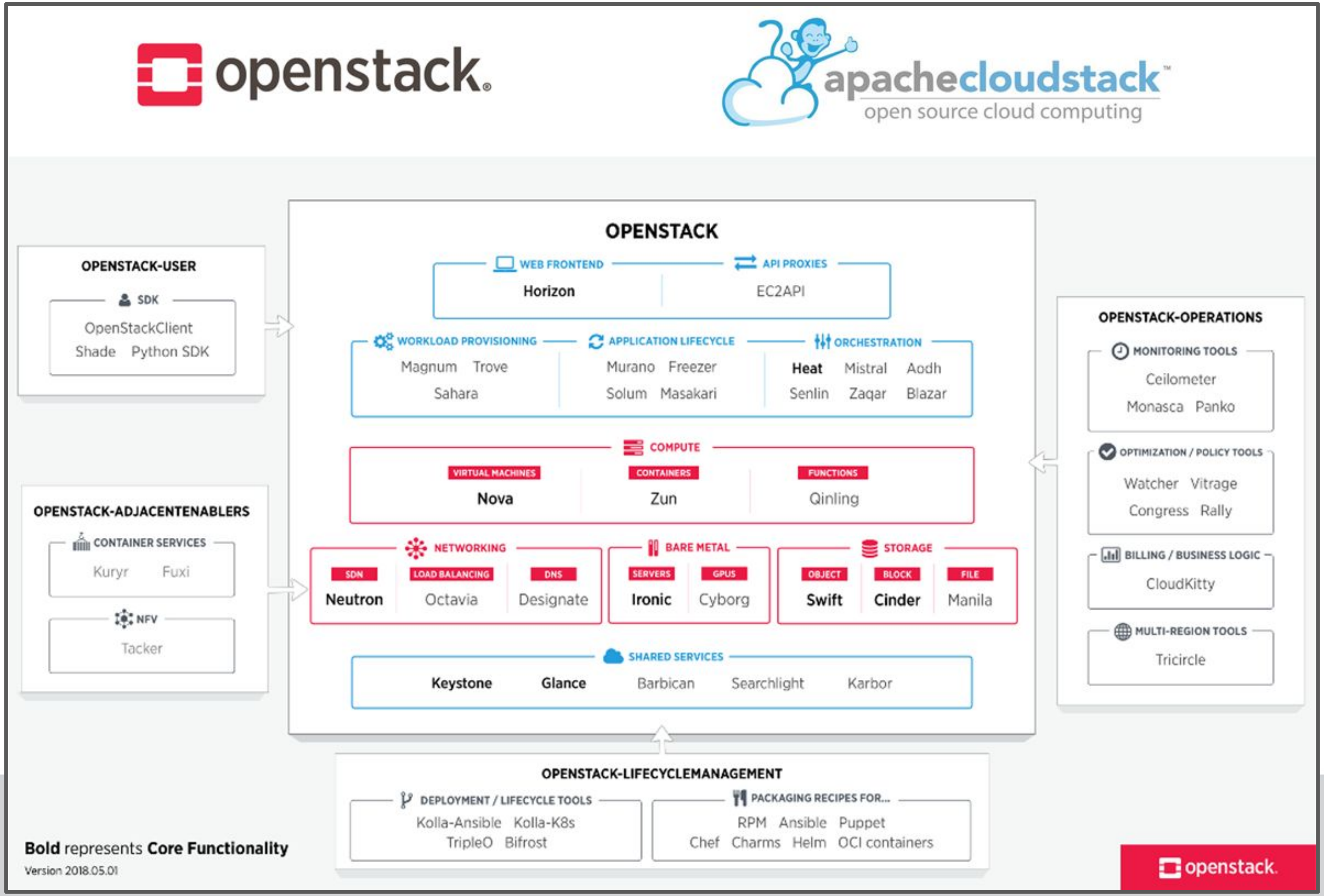
Running guest operating systems within a host operating system environment (VirtualBox)




## Hardware Virtualization

Mobile development is full of hardware virtualization to test mobile apps in various environments.

# The Entire Cloud: On Your Laptop



# Virtualization

★ The goal of all virtualization is to map a **virtual system** onto a **host system**:

- All virtual states **S** can be represented on the host system as **V(S)**
- For all sequence of translations between **S1 ⇒ S2**, there's a sequence of operations that map **V(S1) ⇒ V(S2)**.

# Key Interfaces to Virtualization

- ★ Application Level Interfaces (APIs)
  - ex: libc
- ★ Application Binary Interfaces (ABIs)
  - user-level instructions
  - system calls
- ★ Hardware-Software Interfaces
  - Instruction Set Architectures (ISAs)

# A Virtual “Machine”

★ In virtualization, a “**machine**” is **any entity that provides an interface**:

- **Language Virtualization**
  - Machine := Entity that provides the API
- **Process Virtualization**
  - Machine := Entity that provides the ABI
- **System Virtualization**
  - Machine := Entity that provides the ISA

- **Language Virtualization**
  - Machine := Entity that provides the API
  - Software := Compiler/Interpreter
    - Example: Java Virtual Machine (JVM)
- **Process Virtualization**
  - Machine := Entity that provides the ABI
  - Software := Runtime
    - Example: Windows Subsystem for Linux (WSL)
- **System Virtualization**
  - Machine := Entity that provides the ISA
  - Software := Virtual Machine Monitor

# Process/Language Virtual Machines

## CS 423 - University of Illinois

Wade Fagen-Ulmschneider

(Slides built from Adam Bates and Tianyin Xu previous work on CS 423.)

- **Language Virtualization**
  - Machine := Entity that provides the API
  - Software := Compiler/Interpreter
    - Example: Java Virtual Machine (JVM)
- **Process Virtualization**
  - Machine := Entity that provides the ABI
  - Software := Runtime
    - Example: Windows Subsystem for Linux (WSL)
- **System Virtualization**
  - Machine := Entity that provides the ISA
  - Software := Virtual Machine Monitor

# Example 1: Emulation

- **Emulation** allows one ABI to run on top of another:
  - **Ex:** Early emulation focused on running Windows apps (IA-32) on top of MacOS (PowerPC).
    - Specifically: Running an app compiled for IA-32/Windows on MacOS/PowerPC.
    - Modern emulation often focuses on virtualizing phone interfaces (ARMv8).

  - **Approach 1: Interpreters** -- Read one instruction at a time, update host state using a [set] of host instructions.

  - **Approach 2: Translation** -- Translate the binary instructions to host instructions in one step; run the translated binary.

# Example 2: Binary Optimization

★ **Optimizations** usually involve running an ABI on top of itself for purposes of analysis/profiling.

- **Ex:** **`valgrind`** is a utility that replaces all memory-related library calls to profile memory usage.
- Allows the implementation of optimizations found through runtime-execution.

# Example 3: Language Virtual Machines

- **Language VMs** involve implementing a single API on top of a set of diverse ABIs.
  - **Ex:** `javac` compiles Java code to an intermediate form *(Java Source Code ⇒ Java Bytecode)*
  - Runtime interpreters interpret the bytecode on different ABIs.
  - Not just Java; Microsoft has the “Common Language Interface (CLI)” for the .NET languages; and others exist.

# System Virtual Machines

## CS 423 - University of Illinois

Wade Fagen-Ulmschneider

(Slides built from Adam Bates and Tianyin Xu previous work on CS 423.)

- **Language Virtualization**
  - Machine := Entity that provides the API
  - Software := Compiler/Interpreter
    - Example: Java Virtual Machine (JVM)
- **Process Virtualization**
  - Machine := Entity that provides the ABI
  - Software := Runtime
    - Example: Windows Subsystem for Linux (WSL)
- **System Virtualization**
  - Machine := Entity that provides the ISA
  - Software := Virtual Machine Monitor

# System VMs

- ★ Implement a VMM (ISA emulation) **on bare hardware**:
  - Most efficient,
  - Must support hardware emulation (drivers), and
  - Replaces any OS hosted on the bare hardware.

- ★ Implement a VMM **on top of a host OS**:
  - Less efficient,
  - Leverages the OS drivers and hardware abstractions, and
  - Easy to install on top of the host OS.

# System VMs

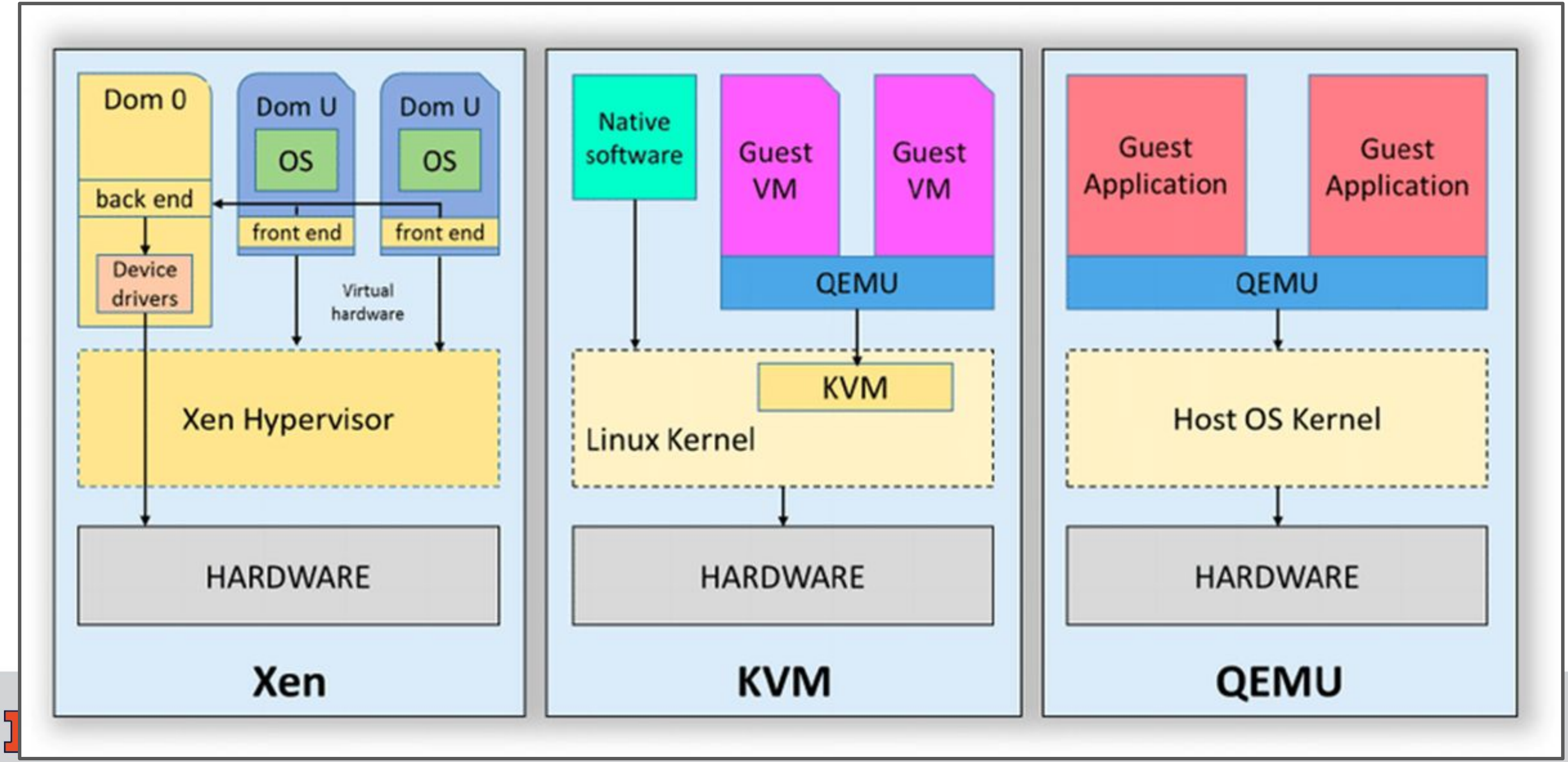
- ★ Implement a VMM (ISA emulation) **on bare hardware**:
  - Most efficient
  - M… ardware emulation (drivers), and
  - Replaces any OS hosted on the bare hardware.

**Type 1 Hypervisor**
(Runs at “Ring -1”; need for hardware support.)

- ★ Implement a VMM **on top of a host OS**:
  - Less efficient
  - L… ers and hardware abstractions, and
  - Easy to install on top of the host OS.

**Type 2 Hypervisor**
(Runs at “Ring 1” on x64; less dependent on specific hardware support.)

# System VMs



Dom 0
back end
Device drivers
Dom U
OS
front end
Dom U
OS
front end
Virtual hardware
Xen Hypervisor
HARDWARE
Xen
Native software
Guest VM
Guest VM
QEMU
KVM
Linux Kernel
HARDWARE
KVM
Guest Application
Guest Application
QEMU
Host OS Kernel
HARDWARE
QEMU

# Emulator Design

## CS 423 - University of Illinois

Wade Fagen-Ulmschneider

(Slides built from Adam Bates and Tianyin Xu previous work on CS 423.)

# Emulator Design

- ★ **Goal:** Emulate guest ISA on a host ISA
  - Need: Simulations of guest data structures
    - Guest memory layout (stack, heap, etc)
    - Guest CPU layout (registers, flags, etc)
  - Need: Simulation of binary instructions

# Emulator Design: Binary Instructions

- ★ **Need:** Simulation of binary instructions
- ★ **Solution:** Basic interpretation could switch on opcode:

```
instruction = sourceCode[PC]
opcode = extract_opcode(instruction)
switch (opcode) {
  case OPCODE1: emulate_OPCODE1(); break;
  case OPCODE2: emulate_OPCODE2(); break;
  /* ... */
}
```

# Emulator Design: Binary Instructions

- ★ **Need:** Simulation of binary instructions
- ★ **Solution:** Use functors (function pointers) to interpret opcode

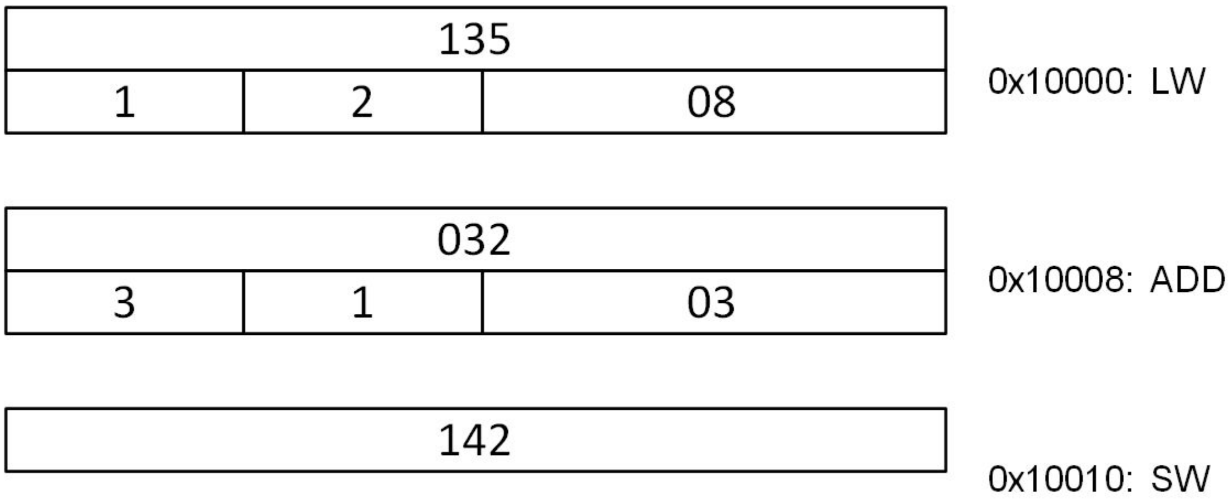
```
instruction = sourceCode[PC]
opcode = extract_opcode(instruction)
emulation = GUEST_TO_HOST_CODE[opcode]
emulation(instruction)
```

# Ex: MIPS

0x1000: LW r1, 8(r2)

0x1004: ADD r3, r3, r1

0x1008: SW r3, 0(r4)



# Ex: MIPS

```
0x1000: LW    r1, 8(r2)
0x1004: ADD   r3, r3, r1
0x1008: SW    r3, 0(r4)
```

| Routine_address07 | | |
|---|---|---|
| 1 | 2 | 08 |

0x10000: LW

| Routine_address08 | | |
|---|---|---|
| 3 | 1 | 03 |

0x10008: ADD

| Routine_address37 | | |
|---|---|---|
| 3 | 4 | 00 |

0x10010: SW

# Opcode Extraction

- ★ Opcodes often have options and may rely on combining several bits ranges.
- ★ **Option 1 - Emulate:** Program the logic of the opcode in software (may be very slow/complex, one opcode could have many paths).
- ★ **Option 2 - Pre-Decoding:** Pre-extract opcode+operand combinations for all instructions and create separate segments for various operands.

# Why not direct translation?

**Q:** Why not just read the source binary and translate it statically one instruction at a time to a target binary?

# Why not direct translation?

**Q:** Why not just read the source binary and translate it statically one instruction at a time to a target binary?

1. **Code discovery and binary translation:**
   a. How to tell whether something is code or data?
   b. We encounter a jump instruction: Is word after the jump instruction code or data?

2. **Code location problem:**
   a. How to map source program counter to target program counter?
   b. Can we do this without having a table as long as the program for instruction-by-instruction mapping?