Background and Virtualization Basics

Yiying Zhang
Administravía

• Form project group by next Wed (discussion topic created on Canvs)

• Paper summary starts next Monday (submit to Canvas by 9:35am! late: 0 point)

• Class attendance tracking from next Monday (we’ll have more discussion)

• Mostly will continue online teaching for a few more weeks after 1/18

• First quiz will be given on 1/19 in class over Canvas (3 quizzes in total, no exam)
Outline

• Review on core OS and architecture concepts
• Different forms of virtualization
• Basic virtualization approaches
Typical OS Structure
Typical OS Structure

Application

Libraries

Portable OS Layer

Machine-dependent layer

Written by programmer
Compiled by programmer
User library class
Typical OS Structure

- Application
  - Written by programmer
  - Compiled by programmer
  - User library class
- Libraries
  - Written by gurus
  - Provided pre-compiled
  - Input to linker
  - Can also be resolved after
- Portable OS Layer
- Machine-dependent layer
Typical OS Structure

Application

Libraries

Portable OS Layer

Machine-dependent layer

Written by programmer
Compiled by programmer
User library class

Written by gurus
Provided pre-compiled
Input to linker
Can also be resolved after

“Guts” of system calls
“High-level” code
Typical OS Structure

Application

Libraries

Portable OS Layer

Machine-dependent layer

“Guts” of system calls
“High-level” code

System initialization
Device drivers
Kernel/user mode switching
Interrupt and exception
Processor management

Written by programmer
Compiled by programmer
User library class

Written by gurus
Provided pre-compiled
Input to linker
Can also be resolved after
Dual-Mode Operation

- OS manages shared resources
- OS protects programs from other programs
  → OS needs to be “privileged”
Dual-Mode Operation

- OS manages shared resources
- OS protects programs from other programs
  ➔ OS needs to be “privileged”

- Dual-mode operation of hardware
  - Kernel mode – can run *privileged* instructions
  - User mode – can only run *non-privileged* instructions
Transition between User/Kernel Modes

Diagram:
- User process:
  - User process executing
  - Calls system call
  - Return from system call
- Kernel:
  - Trap (mode bit = 0)
  - Execute system call
  - Return (mode bit = 1)
- User mode (mode bit = 1)
- Kernel mode (mode bit = 0)
Interrupt

- A mechanism for coordination between concurrently operating units of a computer system (e.g. CPU and I/O devices) to respond to specific conditions within a computer
- Results in transfer of flow of control (to interrupt handler in the OS), forced by hardware
- Hardware interrupts
  - I/O devices: NIC, keyboard, etc.
  - Timer
- Software interrupts
  - Exception: a software error (e.g., divided by zero)
  - System call
Handling Interrupts

- Incoming interrupts are **disabled** (at this and lower priority levels) while the interrupt is being processed to prevent a lost interrupt.

- Interrupt architecture must **save the address** of the interrupted instruction.

- Interrupt **transfers control** to the interrupt service routine:
  - generally, through the interrupt vector, which contains the addresses of all the service routines.

- If interrupt routine modifies process state (register values):
  - **save the current state** of the CPU (registers and the program counter) on the system stack.
  - **restore the state** before returning.

- Interrupts are **re-enabled** after servicing current interrupt.

- Resume the interrupted instruction.
Different OS Structures

- User-Mode
  - Monolithic Kernel
  - MicroKernel
  - ExoKernel (Library OS)

- Kernel-Mode
Monolithic kernel vs Microkernel

- What was the main idea?
- What were the problems?
ExoKernel

- Only does protection and multiplexing
- Present hardware resources directly to users

Diagram showing layers of the ExoKernel system, including:
- Apps
- Library Operating System (Replaceable)
- Kernel
- HW

- Mosaic
  - WWW
  - POSIX
  - TCP

- Barnes-Hut
  - Distributed Shared Memory
  - IPC
  - VM
  - Traps
  - Vess

- Expose allocation/revocation/names/information, and Support Protection

- Disk
- TLB
- Network
- Memory
- Frame buffer
What Is An ISA?

• ISA (instruction set architecture)
  • A well-defined hardware/software interface

• The “contract” between software and hardware
  • Functional definition of operations, modes, and storage locations supported by hardware
  • Precise description of how to invoke, and access them

• No guarantees regarding
  • How operations are implemented
  • Which operations are fast and which are slow and when
  • Which operations take more power and which take less
 Compatibility

- No-one buys new hardware... if it requires new software
  - IBM did this for mainframes; Intel for PCs
  - ISA must remain compatible, no matter what
    - x86 arguably one of the worst ISAs EVER, but survives
    - As does IBM’s 360/370/390 (the first “ISA family”)

- **Backward compatibility**
  - New processors must support old programs
    - Can’t drop features, but can deprecate and emulate
    - Very important

- **Forward (upward) compatibility**
  - Old processors must support new programs (with software help)
    - New processors redefine only previously-illegal opcodes
    - Allow software to detect support for specific new instructions
    - Old processors emulate new instructions in low-level software
x86

• x86 was first 16-bit chip by ~2 years
  • IBM put it into its PCs because there was no competing choice
  • Rest is historical inertia and “financial feedback”

• x86 is "Difficult to explain and impossible to love"

• Complex architecture due to "growth"
  • Typical of many older ISAs, e.g. IBM 360/370/390
  • Started as 16-bit microprocessor (later, 32-bits; later x86-64)
  • Upward compatible from 8080 (accumulator-based)
x86 Outside = RISC Inside

- 1993: Intel wanted out-of-order execution in Pentium Pro
- OOO was very hard to do with a coarse grain ISA like x86
- Their solution? Translate x86 to RISC μops in hardware
  
  ```
  push $eax
  is translated (dynamically in hardware) to
  store $eax [$$esp-4]
  addi $esp,$esp,-4
  ```

- Processor maintains x86 ISA for external compatibility
- But executes RISC µISA for internal implementability
  - Translation itself is proprietary, but 1.6 uops per x86 insn
- Given translator, x86 almost as easy to implement as RISC
Emulation/Binary Translation

- Compatibility is still important but definition has changed
  - Less necessary that processor ISA be compatible
  - As long as some combination of ISA + software translation layer is
  - Advances in emulation, binary translation have made this possible
- **Binary-translation**: transform static image, run native
- **Emulation**: unmodified image, interpret each dynamic insn
  - Typically optimized with just-in-time (JIT) compilation
- Examples
  - FX!32: x86 on Alpha
  - IA32EL: x86 on IA64
  - Rosetta: PowerPC on x86
- Downside: performance overheads
Virtual ISAs

- Java, C#, and some other high-level languages use an ISA-like interface
  - JVM uses a stack-based bytecode
  - C# has the CLR (common language runtime)
  - Higher-level than machine ISA
    - Design for translation (not direct execution)
- Goals:
  - Portability (abstract away the actual hardware)
  - Target for high-level compiler (one per language)
  - Source for low-level translator (one per ISA)
  - Flexibility over time
Outline

- Review on core OS and architecture concepts
- Different forms of virtualization
- Basic virtualization approaches
Interaction between Different Layers

API
ABI
ISA

Hardware

System ISA
User ISA

OS

Libraries

System Calls

Applications

API – application programming interface
ABI – application binary interface
ISA – instruction set architecture
Interaction between Different Layers

Which layer should virtualization be at?

API – application programming interface
ABI – application binary interface
ISA – instruction set architecture
Five Levels of Virtualization Options

1. **Application level**
   - JVM / .NET CLR / Panot

2. **Library (user-level API) level**
   - WINE/ WABI/ LxRun / Visual MainWin / vCUDA

3. **Operating system level**
   - Jail / Virtual Environment / Ensim’s VPS / FVM

4. **Hardware abstraction layer (HAL) level**
   - VMware / Virtual PC / Denali / Xen / L4 / Plex 86 / User mode Linux / Cooperative Linux

5. **Instruction set architecture (ISA) level**
   - Bochs / Crusoe / QEMU / BIRD / Dynamo
Type 1 and Type 2 Hypervisor (VMM)

![Diagram showing the location of type 1 and type 2 hypervisors.](image)

Figure 7-1. Location of type 1 and type 2 hypervisors.

*Operating Systems, 2016, Meni Adler, Danny Hendler & Amnon Meisels*
Virtualization Principles

Popek and Goldberg’s virtualization principles in 1974:

- **Fidelity.** Software on the VMM executes identically to its execution on hardware, barring timing effects.

- **Performance.** An overwhelming majority of guest instructions are executed by the hardware without the intervention of the VMM.

- **Safety.** The VMM manages all hardware resources.
Outline

- Review on core OS and architecture concepts
- Different forms of virtualization
- Basic virtualization approaches
Virtualization Approach 1: Complete Machine Emulation (Hosted Interpretation)

- VMM implements the complete hardware architecture in software
- VMM steps through VM’s instructions and update emulated hardware as needed

```c
while(1){
    curr_instr = fetch(virtHw.PC);
    virtHw.PC += 4;
    switch(curr_instr){
        case ADD:
            int sum = virtHw.regv[cwrr_linst reconciliation reg0] +
                    virtHw.regv[cwrr_linst reconciliation reg1];
            virtHw.regv[cwrr_linst reconciliation reg0] = sum;
            break;
        case SUB: //...etc...
```
Complete Machine Emulation (Hosted Interpretation)

• Pros
  • Easy to handle all types of instructions (can enforce policy when doing so)
  • Provides complete isolation (no guest instructions runs directly on hardware)
  • Can debug low-level code (e.g., boot code) in the guest

• Cons
  • Emulate a modern processor is difficult
  • Violates performance requirement (*it is really slow!*
Protection Rings

- More privileged rings can access memory of less privileged ones
- Calling across rings can only happen with hardware enforcement
- Only Ring 0 can execute privileged instructions
- Rings 1, 2, and 3 trap when executing privileged instructions
- Usually, the OS executes in Ring 0 and applications execute in Ring 3

Image Source: https://commons.wikimedia.org/wiki/File:CPU_ring_scheme.svg
Virtualization Approach 2:
Direct Execution with Trap-and-Emulate
Virtualization Approach 2: Direct Execution with Trap-and-Emulate

• Idea: execute most guest instructions natively on hardware (assuming guest OS runs on the same architecture as real hardware)
Virtualization Approach 2: Direct Execution with Trap-and-Emulate

• Idea: execute most guest instructions natively on hardware (assuming guest OS runs on the same architecture as real hardware)

• Applications run in ring 3 (can’t access memory owned by guest OS (ring1))
Virtualization Approach 2: Direct Execution with Trap-and-Emulate

- Idea: execute most guest instructions natively on hardware (assuming guest OS runs on the same architecture as real hardware)

- Applications run in ring 3 (can’t access memory owned by guest OS (ring1))

- Guest OS runs in ring 1 (can’t access memory owned by VMM (ring 0))
Virtualization Approach 2: Direct Execution with Trap-and-Emulate

- Idea: execute most guest instructions natively on hardware (assuming guest OS runs on the same architecture as real hardware)

- Applications run in ring 3 (can’t access memory owned by guest OS (ring 1))

- Guest OS runs in ring 1 (can’t access memory owned by VMM (ring 0))

Goldberg (1974) two classes of instructions

- privileged instructions: those that trap when in user mode

- sensitive instructions: those that modify or depends on hardware configs
Virtualization Approach 2: Direct Execution with Trap-and-Emulate

- Idea: execute most guest instructions natively on hardware (assuming guest OS runs on the same architecture as real hardware)

- Applications run in ring 3 (can’t access memory owned by guest OS (ring1))

- Guest OS runs in ring 1 (can’t access memory owned by VMM (ring 0))

- Cannot allow guest OS to run sensitive instructions directly!

Goldberg (1974) two classes of instructions

- *privileged instructions*: those that trap when in user mode

- *sensitive instructions*: those that modify or depends on hardware configs
Virtualization Approach 2: Direct Execution with Trap-and-Emulate

- Idea: execute most guest instructions natively on hardware (assuming guest OS runs on the same architecture as real hardware)

- Applications run in ring 3 (can’t access memory owned by guest OS (ring 1))

- Guest OS runs in ring 1 (can’t access memory owned by VMM (ring 0))

- Cannot allow guest OS to run *sensitive instructions* directly!

Goldberg (1974) two classes of instructions

- *privileged instructions*: those that trap when in user mode

- *sensitive instructions*: those that modify or depends on hardware configs
Virtualization Approach 2: Direct Execution with Trap-and-Emulate

- Idea: execute most guest instructions natively on hardware (assuming guest OS runs on the same architecture as real hardware)
- Applications run in ring 3 (can’t access memory owned by guest OS (ring 1))
- Guest OS runs in ring 1 (can’t access memory owned by VMM (ring 0))
- Cannot allow guest OS to run *sensitive instructions* directly!
- When guest OS executes a *privileged instruction*, will trap into VMM

Goldberg (1974) two classes of instructions
- *privileged instructions*: those that trap when in user mode
- *sensitive instructions*: those that modify or depends on hardware configs
Virtualization Approach 2: Direct Execution with Trap-and-Emulate

- Idea: execute most guest instructions natively on hardware (assuming guest OS runs on the same architecture as real hardware)

- Applications run in ring 3 (can’t access memory owned by guest OS (ring1))

- Guest OS runs in ring 1 (can’t access memory owned by VMM (ring 0))

- Cannot allow guest OS to run **sensitive instructions** directly!

- When guest OS executes a **privileged instruction**, will **trap** into VMM

- When guest applications generates a software interrupt, will **trap** into VMM

Goldberg (1974) two classes of instructions

- **privileged instructions**: those that trap when in user mode
- **sensitive instructions**: those that modify or depends on hardware configs
Trap-and-Emulate

• Goal: hand off sensitive operations to the VMM

• Reality: privileged operations trap to VMM

• VMM emulates the effect of privileged operations on virtual hardware provided to the guest OS
  • VMM controls how the VM interacts with physical hardware
  • VMM fools the guest OS into thinking that it runs at the highest privilege level

• Performance implications
  • Almost no overhead for non-privileged instructions
  • Large overhead for privileged instructions
open:
push dword mode
push dword flags
push dword path
mov eax, 5
push eax
int 80h

<table>
<thead>
<tr>
<th>Process</th>
<th>Hardware</th>
<th>Operating System</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Execute instructions (add, load, etc.)</td>
<td>3. Switch to kernel mode; Jump to trap handler</td>
<td>4. In kernel mode; Handle system call; Return from trap</td>
</tr>
<tr>
<td>2. System call: Trap to OS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Switch to user mode; Return to user code</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6. Resume execution (@PC after trap)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure B.1: Executing a System Call
## System Calls with Virtualization

<table>
<thead>
<tr>
<th>Process</th>
<th>Operating System</th>
<th>VMM</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1. System call:</strong></td>
<td>Trap to OS</td>
<td><strong>2. Process trapped:</strong> Call OS trap handler (at reduced privilege)</td>
</tr>
<tr>
<td><strong>2. OS trap handler:</strong></td>
<td>Decode trap and execute appropriate syscall routine; When done: return from trap</td>
<td></td>
</tr>
<tr>
<td><strong>3. Resume execution:</strong></td>
<td>(@PC after trap)</td>
<td><strong>4. OS tried return from trap:</strong> Do real return from trap</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure B.2: System Call Flow Without Virtualization**

<table>
<thead>
<tr>
<th>Process</th>
<th>Operating System</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1. System call:</strong></td>
<td>Trap to OS</td>
</tr>
<tr>
<td><strong>2. OS trap handler:</strong></td>
<td>Decode trap and execute syscall; When done: issue return-from-trap</td>
</tr>
<tr>
<td><strong>3. Resume execution:</strong></td>
<td>(@PC after trap)</td>
</tr>
</tbody>
</table>

**Figure B.3: System Call Flow with Virtualization**
x86 Difficulties

Popek and Goldberg’s Theorem (1974)

– A machine can be virtualized (using trap-and-emulate) if every sensitive instruction is privileged.
x86 Difficulties

Popek and Goldberg’s Theorem (1974)

– A machine can be virtualized (using trap-and-emulate) if every sensitive instruction is privileged.

• Not all sensitive instructions are privileged with x86 for many years, i.e., non-virtualizable processor

• These instructions do not trap and behave differently in kernel and user mode

• Example: `popf`
  • Pops 16 bits from top of the stack to the `%eflags` register
  • Bit 9 of `%eflags` masks interrupts (i.e., enables/disables interrupts)
  • `popf` is not privileged. What happens if guest OS (ring 1) runs `popf` to `%eflags`?
  • In Ring 0, `popf` can set bit 9, but CPU silently ignores `popf` when running in Ring 1
  • What should happen is a trap so that VMM can emulate interrupts (change which interrupts to forward to guest OS)
Trap-and-Emulate

• Pros and Cons?
Virtualization Approach 3: Direct Execution with Binary Translation

- VMM dynamically rewrites instructions
- So that non-virtualizable instructions can trap to VMM
- VMware’s business
- More next lecture
Virtualization Approach 4:
Direct Execution with Hardware-Assisted Virtualization

- Adds a new mode so that sensitive operations could all trap
- Other hardware support to make virtualization easier/faster
- More next lecture
Virtualization Approach 5: Direct Execution with Paravirtualization

• Full virtualization (no guest OS modification)
  • Tricky and has performance overhead

• Para-virtualization: modified guest OS
  • Change (rewrite) guest OS to remove sensitive but unprivileged instructions and to use other tricks to make virtualization faster
    • Guest OS works with hypervisor (i.e., knows that it is a VM) and has some exposure to hardware
    • e.g., guest OS informs hypervisor of page table changes
    • e.g., guest OS directly calls hypervisor on system calls (hypercalls)
  • Guest applications are still unmodified
  • Pros and Cons?
Other Virtualization Approaches

- **Container**: Essentially just a group of processes with some additional features (isolated namespace, isolated resources, etc.) (e.g., Docker)

- **Unikernel**: LibraryOS designed for a single application, running on hypervisor (as a VM) or host OS (as a process)

- **Sandboxing**: Limit what the applications (and libOS) can do (e.g., gVisor)

- **Language-based**: Running applications written in a high-level language on language runtimes (e.g., JVM)
Virtualization Approaches Summary

• Hosted interpretation
  • Interpret each instruction, super slow (e.g., Virtual PC on Mac)

• Direct execution with trap-and-emulate
  • Requires a virtualizable processor and only works for the same architecture

• Direct execution with binary translation
  • Works with non-virtualizable processor, but implementing VMM is tricky

• Direct execution with hardware-assisted virtualization
  • Needs new generation of hardware (which is the norm now), mode switching is still not optimized

• Direct execution with paravirtualization
  • Good performance and works with non-virtualizable processors, but require guest OS changes

• OS-level virtualization, library-level, language (app)-level, unikernels, etc.
  • More lightweight and faster to start, but less secure
Review: Regular Virtual Memory System

Virtual address

VPage # | offset
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Page-map L4 base addr (CR3)

Page-map L4 table

Page-directory pointer table

Page-directory table

Page-table entry

Page-table

Physical address

Main memory

Hit

Miss

Real page table

Physical page frame number | Page offset
Review: Software-controlled TLB

- On a TLB hit, MMU checks the valid bit
  - If valid, perform address translation
  - If invalid (e.g. page not in memory), MMU generates a page fault
    - OS performs page fault handling
    - Restart the faulting instruction

- On a TLB miss, HW raises exception, traps to the OS
Review: Software-controlled TLB

- On a TLB hit, MMU checks the valid bit
  - If valid, perform address translation
  - If invalid (e.g. page not in memory), MMU generates a page fault
    - OS performs page fault handling
    - Restart the faulting instruction

- On a TLB miss, HW raises exception, traps to the OS
  - OS parses page table and loads PTE into TLB
    - Needs to replace if TLB is full
  - Same as in a hit...
Review: Hardware-controlled TLB

- On a TLB hit, MMU checks the valid bit
  - If valid, perform address translation
  - If invalid (e.g. page not in memory), MMU generates a page fault
    - OS performs fault handling
    - Restart the faulting instruction

- On a TLB miss
  - MMU parses page table and loads PTE into TLB
Review: Hardware-controlled TLB

- On a TLB hit, MMU checks the valid bit
  - If valid, perform address translation
  - If invalid (e.g. page not in memory), MMU generates a page fault
    - OS performs fault handling
    - Restart the faulting instruction

- On a TLB miss
  - MMU parses page table and loads PTE into TLB
  - Needs to replace if TLB is full
  - Same as hit …
Virtualizing Memory

• Extra level of memory addressing
Virtualizing Memory

- TLB miss flow with software-managed TLB

![Figure B.5: TLB Miss Flow without Virtualization](image)

<table>
<thead>
<tr>
<th>Process</th>
<th>Operating System</th>
</tr>
</thead>
</table>
| 1. Load from mem
TLB miss: Trap |
| 2. OS TLB miss handler:
Extract VPN from VA;
Do page table lookup;
If present and valid:
get PFN, update TLB;
Return from trap |

3. Resume execution
(@PC of trapping instruction);
Instruction is retried;
Results in TLB hit

![Figure B.6: TLB Miss Flow with Virtualization](image)

<table>
<thead>
<tr>
<th>Process</th>
<th>Operating System</th>
<th>Virtual Machine Monitor</th>
</tr>
</thead>
</table>
| 1. Load from mem
TLB miss: Trap |
| 2. VMM TLB miss handler:
Call into OS TLB handler (reducing privilege) |
| 3. OS TLB miss handler:
Extract VPN from VA;
Do page table lookup;
If present and valid,
get PFN, update TLB |
| 4. Trap handler:
Unprivileged code trying
to update the TLB;
OS is trying to install
VPN-to-PFN mapping;
Update TLB instead with
VPN-to-MFN (privileged);
Jump back to OS (reducing privilege) |
| 5. Return from trap |
| 6. Trap handler:
Unprivileged code trying
to return from a trap;
Return from trap |

7. Resume execution
(@PC of instruction);
Instruction is retried;
Results in TLB hit
Difficulty in Virtualizing Hardware-Managed TLB

- Hardware-managed TLB
  - Hardware does page table walk on each TLB miss
- Hypervisor doesn’t have chance to intercept on TLB misses
- More on memory virtualization next Wed
I/O Virtualization

• Goal
  • Multiplexing device across guest VMs

• Challenges
  • Each guest OS has its own device driver
  • How can one device be controlled by multiple drivers?
  • What if one guest OS tries to format its disk?
Possible Solutions of I/O Virtualization

- Direct access: VM exclusively owns a device
- Device emulation: VMM emulates device in software
- Para-virtualization: split driver into guest part and host part
- Hardware assisted: hardware devices offer isolated “virtual interfaces”
- More on I/O virtualization later