Background and Virtualization Basics

Yiying Zhang
Administravvia

- Form project group by next Friday
  - email TA your group info (name, student ID, email address, chosen project topic)

- Paper summary starts next Monday (submit to Canvas by 10am! late: 0 point)

- Class attendance tracked from next Monday

- Course Piazza created (e.g., used to find project partners)
Outline

• Review on core OS and architecture concepts
• Different forms of virtualization
• Basic virtualization approaches
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**
  - "Guts" of system calls
  - "High-level" code

- **Machine-dependent layer**
  - System initialization
  - Device drivers
  - Kernel/user mode switching
  - Interrupt and exception
  - Processor management
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
Different OS Structures

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

- Kernel-Mode
  - OS
  - App
  - Logic
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

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

Hardware resources:
- Disk
- TLB
- Network
- Memory
- Frame buffer
<table>
<thead>
<tr>
<th>OS Functionalities</th>
<th>Virtualized Functionalities</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Process management</td>
<td>• Virtualize sensitive instructions</td>
</tr>
<tr>
<td>• Virtual memory system</td>
<td>• Virtualized physical memory</td>
</tr>
<tr>
<td>• File and storage system</td>
<td>• Virtual disk</td>
</tr>
<tr>
<td>• Networking</td>
<td>• Virtual network interface</td>
</tr>
<tr>
<td>• Other I/O systems</td>
<td>• Other virtualized I/O systems</td>
</tr>
<tr>
<td>• Command-interpreter system</td>
<td>• Command-interpreter system</td>
</tr>
</tbody>
</table>
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

• **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)
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
Outline

• Review on core OS and architecture concepts

• Different forms of virtualization

• Basic virtualization approaches
Interaction between Different Layers

Which layer should virtualization be at?

API – application programming interface
ABI – application binary interface
ISA – instruction set architecture
Figure 7-1. Location of type 1 and type 2 hypervisors.
Outline

• Review on core OS and architecture concepts
• Different forms of virtualization
• Basic virtualization approaches
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.
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_regs[curr_instr.reg0] +
                      virtHw_regs[curr_instr.reg1];
            virtHw_regs[curr_instr.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

- 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!
  - Those that touches hardware configurations
Trap-and-Emulate

• Goal: hand off sensitive operations to the VMM

• VMM emulates the effect of sensitive 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
Example of Trap-and-Emulate: (Review) Transition between User/Kernel Modes
Example of Trap-and-Emulate: (Review) Flow of a System Call

1. System call (User→Kernel Mode)
2. Check parameters
3. Call service routine
4. Service Routine call utilities
Reschedule/Return to user
Example of Trap-and-Emulate: (Review) Regular System Call

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:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trap to OS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Switch to kernel mode; Jump to trap handler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. 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
Example of Trap-and-Emulate: System Calls with Virtualization

<table>
<thead>
<tr>
<th>Process</th>
<th>Operating System</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. System call:</td>
<td>Trap to OS</td>
</tr>
<tr>
<td></td>
<td>2. OS trap handler:</td>
</tr>
<tr>
<td></td>
<td>Decode trap and execute appropriate syscall routine;</td>
</tr>
<tr>
<td></td>
<td>When done: return from trap</td>
</tr>
<tr>
<td>3. Resume execution</td>
<td>(atPC after trap)</td>
</tr>
<tr>
<td></td>
<td>figure B.2: System Call Flow Without Virtualization</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process</th>
<th>Operating System</th>
<th>VMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. System call:</td>
<td>Trap to OS</td>
<td>2. Process trapped:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call OS trap handler</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(at reduced privilege)</td>
</tr>
<tr>
<td>3. OS trap handler:</td>
<td>Decode trap and execute syscall; When done: issue return-from-trap</td>
<td></td>
</tr>
<tr>
<td>5. Resume execution</td>
<td>(@PC after trap)</td>
<td>4. OS tried return from trap:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Do real return from trap</td>
</tr>
</tbody>
</table>

figure B.3: System Call Flow with Virtualization
Trap-and-Emulate

• Pros and Cons?

• Performance implications
  • Almost no overhead for non-privileged instructions
  • Large overhead for privileged instructions
x86 Difficulties

Popek and Goldberg (1974) two classes of instructions

- **privileged instructions**: those that trap when in user mode
- **敏感指令**: those that modify or depends on hardware configs
  - A machine can be virtualized (using trap-and-emulate) if every sensitive instruction is privileged.

- Reality: only privileged operations trap to VMM
- 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 emulates interrupts (change which interrupts to forward to guest OS)
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 week
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