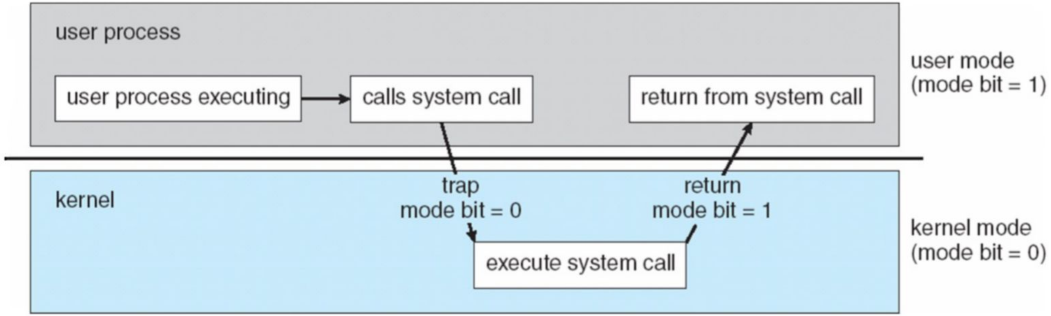
Virtualization:
Concepts and Applications

Kent Hall

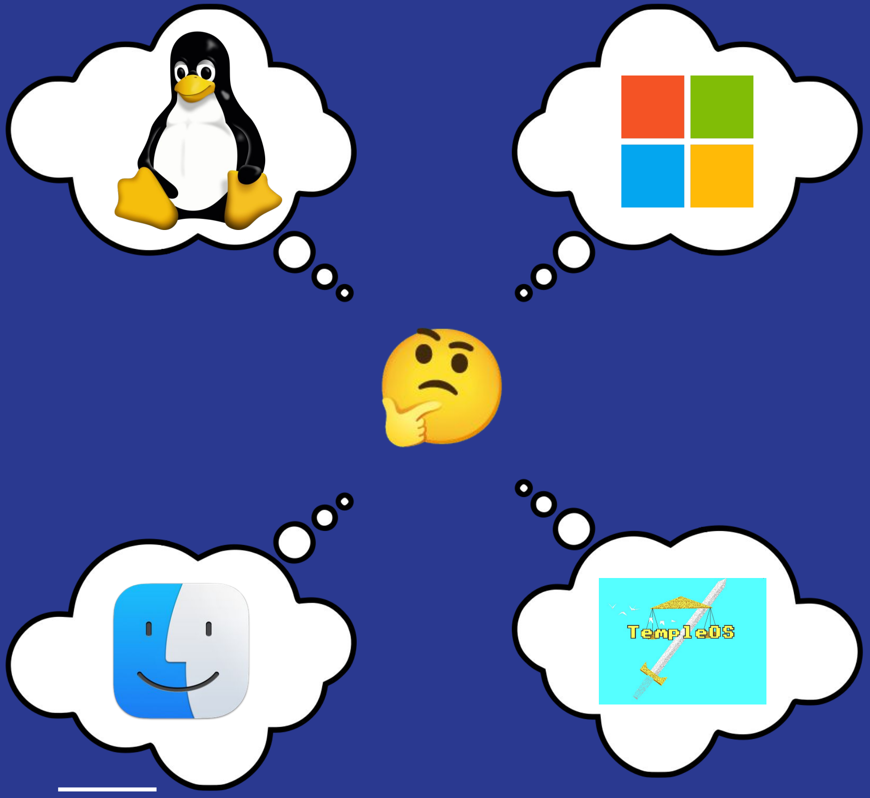
A Refresher

- ➢ Think user/kernel mode
- ➢ CPU can only execute a subset of instructions in user mode, must elevate to kernel mode (e.g. via a system call) to perform privileged operations
- ➢ User programs are compiled to operate within the bounds of user mode; executing a privileged instruction will get trapped and likely killed by the OS



How do we run a program which expects to be able to perform privileged operations?

e.g. another OS



Demo: Running a program with privileged instructions

CPU Emulation

Computers are turing-complete, why not write a program which does everything a CPU does to “emulate” a machine that runs our privileged executable code?

```
struct virtual_cpu {
      int mode;
      int interrupts_enabled;
      ...
};
```

```
struct virtual_cpu vcpu;
...
while (fread(&instruction, sizeof(instruction), os_executable)) {
      if (instruction == MASK_INTERRUPTS)
            vcpu.interrupts_enabled = 0;
      if (instruction == UNMASK_INTERRUPTS)
            vcpu.interrupts_enabled = 1;
      if (instruction == SOFTWARE_INTERRUPT_INSTRUCTION)
            if (vcpu.mode == 1)  // virtual CPU is in user mode
                  vcpu.mode = 0; // we will enter kernel mode
      ...
}
```

CPU Emulation

👍

- Emulator is just a normal user program, basically an interpreter for machine code
- Regardless of our host machine, we can run a guest compiled for any architecture, privilege, etc.

👎

- Really, really slow compared to executing instructions natively on the CPU
- We're emulating every single instruction, not just the privileged ones; far from optimal when the guest architecture is the same as the host machine

Trap-and-Emulate

The guest operating system largely executes in user-mode, but unlike emulation, instructions are executed directly on the CPU; in the event we run into any privileged instructions, we trap to kernel-mode for emulating the operation and maintaining CPU state for the guest's "virtual" CPU.

user-mode (guest OS)

```
load reg0, 5        ✅
load reg1, 6        ✅
add reg0, reg1      ✅
mask_interrupts     ❌
```

jump to trap handler

kernel-mode (host OS / VMM / hypervisor)

```
if (fault_instruction == MASK_INTERRUPTS)
      vcpu.interrupts_enabled = 0;
...
return_to_vcpu();
```

jump to right after the emulated instruction

```
next_instruction    ❓
```

Trap-and-Emulate

👍

- ➢ Much faster than full emulation, we only emulate the “sensitive” instructions that could interact with host OS state
- ➢ Non-privileged instructions are executed natively on the CPU, reducing overhead greatly

👎

- ➢ Only possible if the CPU actually traps on every “sensitive” instruction
- ➢ Major architectures, such as x86 and ARM, do not

To Trap or Not to Trap: x86 Example

```
// guest disabled interrupts earlier
long flags;
spin_lock_irqsave(lk, flags);    // does PUSHF
...
spin_lock_irqrestore(lk, flags); // enables interrupts??
```

user-mode (guest OS)

```
PUSHF    ✅
POP EAX  ✅
```

guest execution continues having read host CPU state

kernel-mode (host OS / VMM / hypervisor)

oopsie, we never trap!

```
if (fault_instruction == PUSH_FLAGS) {
      int val = vcpu.interrupts_enabled;
      do_push_stack(vcpu, val);
}
...
return_to_vcpu();
```

x86 ISA includes many “sensitive” instructions which can be executed at the user privilege level. One such instruction is PUSHF, which reads some CPU state flags and pushes them onto the stack. These flags include `IF`, the interrupt enable flag.

Virtualizing the Non-Virtualizable (before it was cool)

1974

Popek and Goldberg

formally define: “sensitive” instructions must all trap for an architecture to be “virtualizable”[1]

1998

Intel Corporation

engineers are convinced that their processors cannot be virtualized in any practical sense[2]

???

1999

VMware

releases the first x86 virtualization product, a desktop hypervisor software[3]

1. https://en.wikipedia.org/wiki/Popek_and_Goldberg_virtualization_requirements
2. https://personal.utdallas.edu/~sridhar/ios/ref/virt_book.pdf
3. https://en.wikipedia.org/wiki/VMware_Workstation

Virtualizing the Non-Virtualizable (before it was cool)

VMware had to get creative to enable virtualization on x86; they employed binary translation, essentially patching over the actual guest binary to enable emulation of any sensitive instructions.

```
MOV EAX, 5      ✅
MOV EBX, 6      ✅
ADD EAX, EBX    ✅
PUSHF           ✅
```

no trap 😔

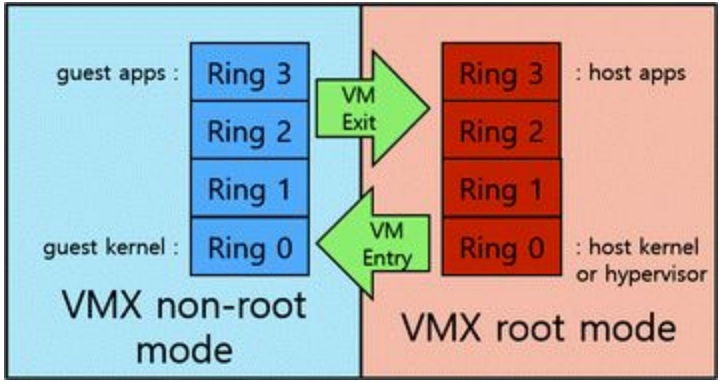
```
MOV EAX, 5      ✅
MOV EBX, 6      ✅
ADD EAX, EBX    ✅
...
INT X           ❌
```

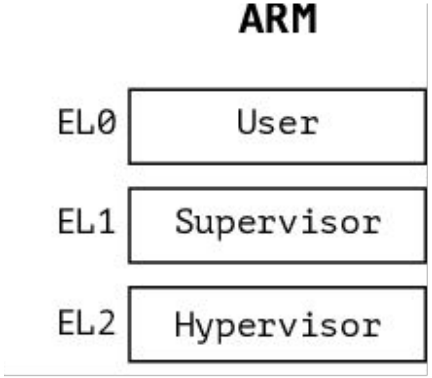
we trap 😃

Hardware Support

If there's no way under the current ISA design to trap sensitive instructions to our higher privilege level, and we need to maintain backwards compatibility with all the existing executable code out there, let's just slap on an additional layer of privilege!

- ❏ Intel VT-x: a new CPU mode that is completely orthogonal to the existing privilege levels: root vs. non-root mode
- ❏ ARM Virtualization Extensions: an additional level of privilege, EL2, which sits on top of the existing privilege levels





Hardware Support

intel

- ➢ Orthogonal non-root mode maintains “shadow” state which parallels all root mode CPU state
- ➢ Shadow state is stored in one giant “virtual machine control structure” (VMCS), and must be restored/saved upon entry/exit to root mode all at once

arm

- ➢ Additional EL2 level of privilege with its own independent state
- ➢ Can selectively save/restore CPU state when entering from / returning to lower privilege, allowing the hypervisor more fine-tune control

x86 Example Revisited

non-root kernel-mode
(guest OS)

root kernel-mode
(host OS / VMM / hypervisor)

```
// guest disabled interrupts earlier
long flags;
spin_lock_irqsave(lk, flags);    // does PUSHF
...
spin_lock_irqrestore(lk, flags); // intrs stay disabled!
```

PUSHF ✅

POP EAX ✅

guest execution continues having read shadow CPU state

we never trap— but this is ok!

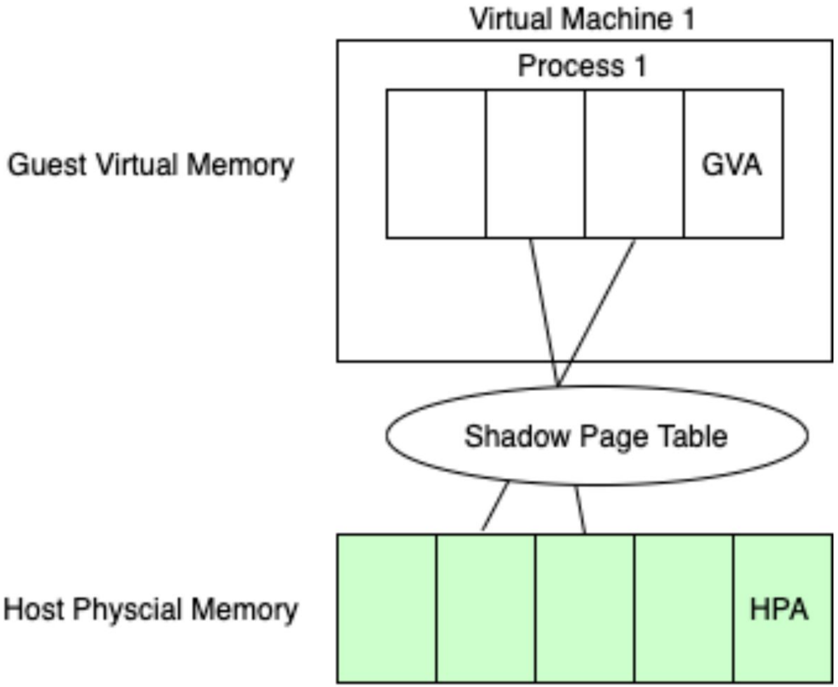
```
if (fault_instruction == PUSH_FLAGS) {
        int val = vcpu.interrupts_enabled;
        do_push_stack(vcpu, val);
}
...
return_to_vcpu();
```

PUSHF will read the *shadow* CPU state flags which were restored from VMCS upon VMRESUME.

GUEST STATE AREA				
CR0	CR3		CR4	
DR7				
RSP	RIP		RFLAGS	
CS	Selector	Base Address	Segment Limit	Access Right
SS	Selector	Base Address	Segment Limit	Access Right
DS	Selector	Base Address	Segment Limit	Access Right
ES	Selector	Base Address	Segment Limit	Access Right
FS	Selector	Base Address	Segment Limit	Access Right
GS	Selector	Base Address	Segment Limit	Access Right
LDTR	Selector	Base Address	Segment Limit	Access Right
TR	Selector	Base Address	Segment Limit	Access Right
GDTR	Selector	Base Address	Segment Limit	Access Right
IDTR	Selector	Base Address	Segment Limit	Access Right
IA32_DEBUGCTL	IA32_SYSENTER_CS	IA32_SYSENTER_ESP	IA32_SYSENTER_EIP	
IA32_PERF_GLOBAL_CTRL	IA32_PAT	IA32_EFER	IA32_BNDCFGS	
SMBASE				
Activity state	Interruptibility state			
Pending debug exceptions				
VMCS link pointer				
VMX-preemption timer value				
Page-directory-pointer-table entries	PDPTE0	PDPTE1	PDPTE2	PDPTE3
Guest interrupt status				
PML index				

The Problem with Paging

Guest OS wants to set up its page tables to map virtual addresses to physical memory, but the VMM obviously can't allow it to have free reign over actual memory. Instead, the illusion must be created by trapping any guest attempts at page table configuration and maintaining "shadow page tables".

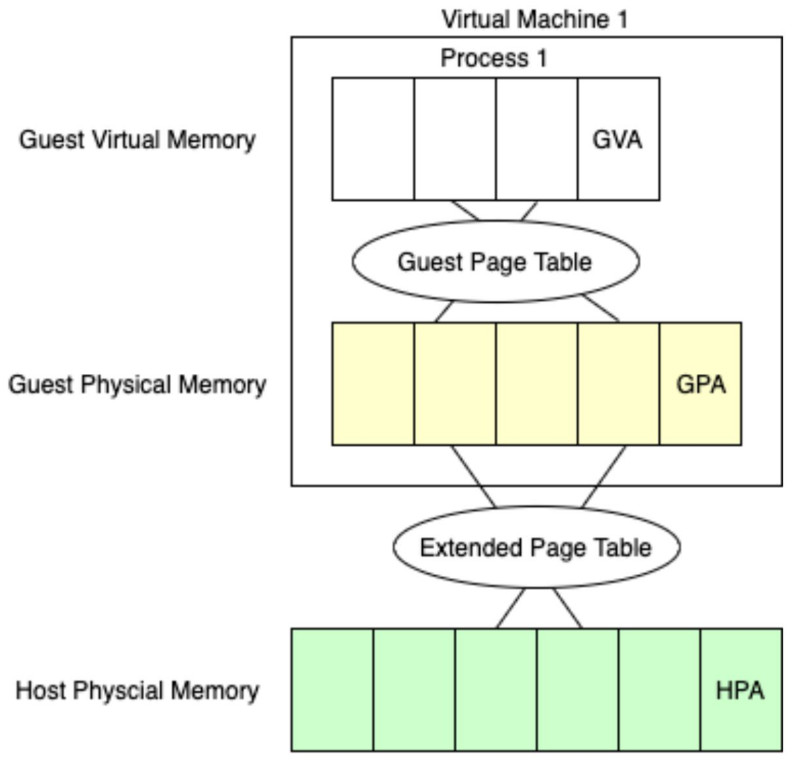


Second Level Address Translation

Instead of the VMM trapping everything and doing the heavy-lifting in software, let's let the guest manage its own page tables and have the MMU handle an additional level of translation—from guest physical to actual physical addresses—in hardware.

CPU designers: might as well, we're extending the ISA to better facilitate virtualization anyway.

- ❏ Intel Extended Page Tables (EPT)
- ❏ ARM Stage-2 Page Tables

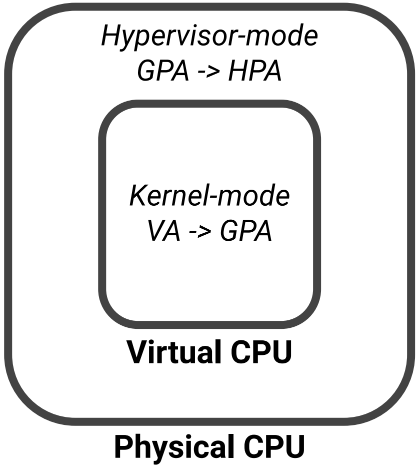


Recap: Modern Day Virtualization

Modern CPUs facilitate virtualization of guest operating systems by adding another layer of privilege and MMU translation; like how user/kernel-mode context switching isolates user programs, but for guest operating systems.

Probably shouldn't be necessary in an ideal world, but multiplexing operating systems onto the same physical hardware has become necessary.

- Fragmented ecosystems: how do you run Windows programs on a Mac?
- Additional layer of hardware-level isolation: vulnerabilities in a massive, popular piece of software like Windows are more often found/exploited than a smaller, narrowly-focused hypervisor



CPUCEPTION

blog.ryujinx.org/the-impossible-port-macos/

Implementing ARM compatibility is a big-deal for normal software, and emulators are far from normal. But, as we alluded to earlier, one of the niceties for Switch Emulation in particular is that the host and guest system both execute the same code. Dedicated optimizations for Apple Silicon have thus been made:

- ARM Hypervisor - Allows native ARMv8 code execution with **no translation.** The overhead cost of emulating the game code is almost entirely removed, accessing the full potential of the chip.

developer.apple.com/documentation/hypervisor

The Hypervisor framework has the following requirements:

Supported hardware

> The Hypervisor framework requires hardware support to virtualize hardware resources. On Apple silicon, that includes the Virtualization Extensions. On Intel-based Mac computers, the framework supports machines with an Intel VT-x feature set that includes Extended Page Tables (EPT) and Unrestricted Mode.

Brief Aside: Nintendo Switch Emulation!

Hypervisors Today

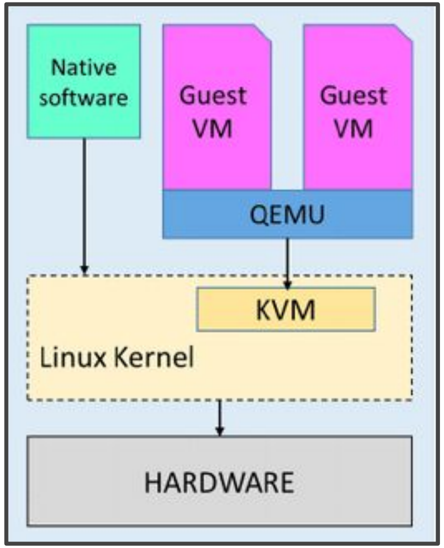
Type I

Type II

Linux Kernel-based Virtual Machine (KVM)

A Linux kernel module for exposing hypervisor functionality via an `ioctl()` interface on a pseudo-device at */dev/kvm*—just like the HW7 Farfetch'd optional part!

It abstracts over the the virtualization features of various CPUs to enable running a virtual CPU on any supported processor.



QEMU

Demo: Running a program with privileged instructions (successfully)

Thanks!