Virtual Machines
虛擬機器
CSIE 5310

Prof. Shih-Wei Li

Department of Computer Science and Information Engineering
National Taiwan University

Review: ISA Virtualizability

- Defined by Popek and Goldberg in their 1974's paper about conditions that an ISA can efficiently support virtual machines
  - Made a few assumptions about machines: a CPU includes system/user modes, system ISA, etc.
  - Applicable to modern day CPUs

Instruction Set Properties

- Popek and Goldberg categorized instructions of an ISA into various types:
  - Privileged Instructions
  - Sensitive Instructions:
    - Control Sensitive Instructions
    - Behavioral Sensitive Instructions
- Sensitive instructions specify instructions that interact with hardware

.

Privileged Instructions

- Executions in user mode cause a trap to the system mode; executions in the system mode do not trap
- Examples:
  - CPU halt instructions
  - Load PSW (LPSW) and Set CPU Timer (SPT) in IBM System/370

Case Study: LPSW in System/370

- LPSW updates the processor status word (PSW) from values in memory
- PSW contains bits that determine the state of the CPU
  - The P bit specifies whether the CPU is in user or system mode.
  - PSW also specifies the instruction address (or program counter).
- Why LPSW is designed as a privileged instruction?

Control Sensitive Instructions

- Instructions that change the configuration of resources in the system
  - LPSW and SPT instructions mentioned earlier
  - Instructions that change page tables; ex: mov CR3, MSR TTBR
- Examples:
  - Modify control registers (ex: CR0 in x86, SCTLR in Arm)
  - Return from the kernel to user mode (eret in Arm)

Behavior Sensitive Instructions

- Instructions whose behaviors depend on the configuration of resources
  - **LRA in System 370:** takes a virtual address, translates it, and saves the corresponding real address in a specified general-purpose register. The behaviors depend on the address space mapping. Arm provides a similar **AT** instruction.
  - **POPF in x86:** pop the flag registers from a stack held in memory. The interrupt enabled flag register can only be modified when executing in privilege mode, but NOOP in user mode

Theorems from Popek & Goldberg in 74’ (1)

- ***Theorem 1:*** A VMM may be constructed if that computer's set of sensitive instructions is a subset of the set of privileged instructions.
  - The computer (with its ISA) is (efficiently) ***virtualizable*** if true
  - Implementing VMMs using “***trap-and-emulate***” — run the entire VMs in user mode and VMMs in the privileged mode
    - Non-privileged instructions to execute natively
    - Privileged instructions cause traps to the VMM upon execution:
      - The VMMs can handle the traps for emulating these instructions

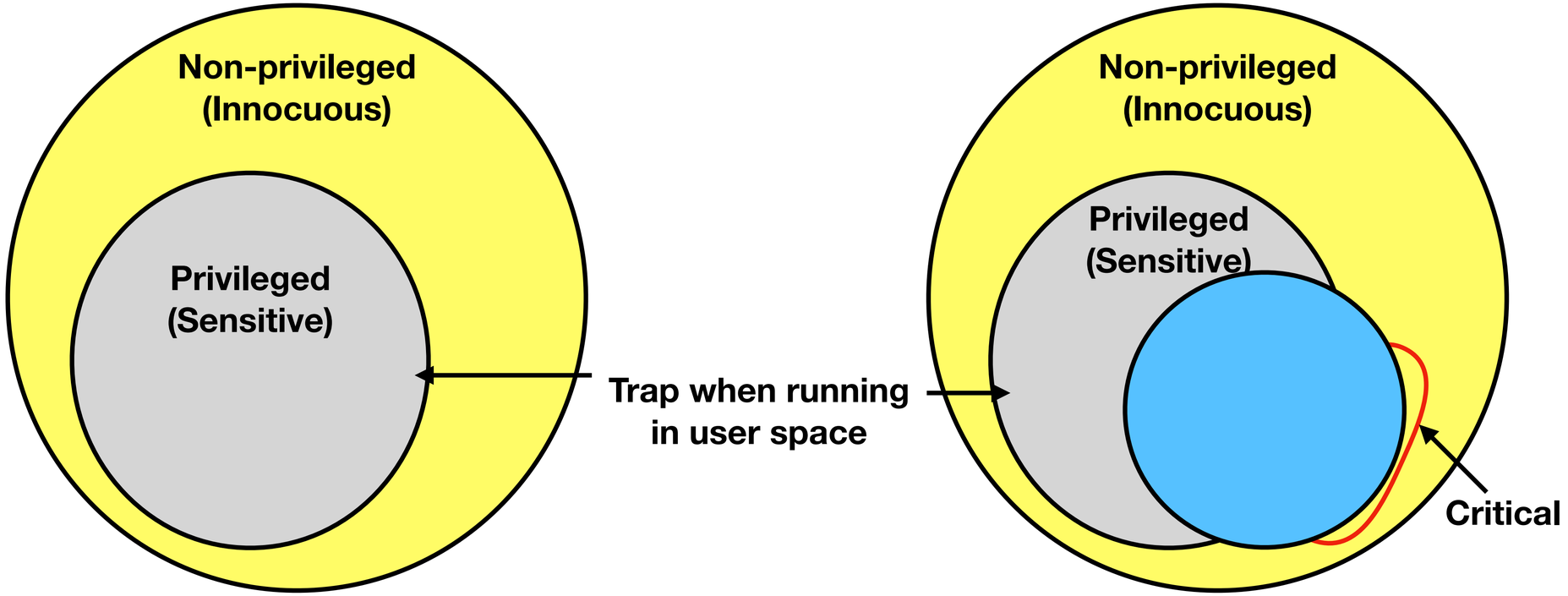
Theorems from Popek & Goldberg in 74’ (2)

- ***Theorem 2:*** a computer is recursively virtualizable, if (a) it is virtualizable and (b) a VMM without any timing dependencies can be constructed for it
  - (b) is important because recursive virtualization can be much slower
  - We will go back to this later when we discuss nested virtualization

Can we simply implement VMMs using trap-and-emulate?

Non-Privileged Sensitive Instructions

- Modern ISAs include non-privileged sensitive instructions (**critical instructions**) — these instructions don't trap when executing in user mode: x86 and Arm include critical instructions



Critical Instructions in ISAs

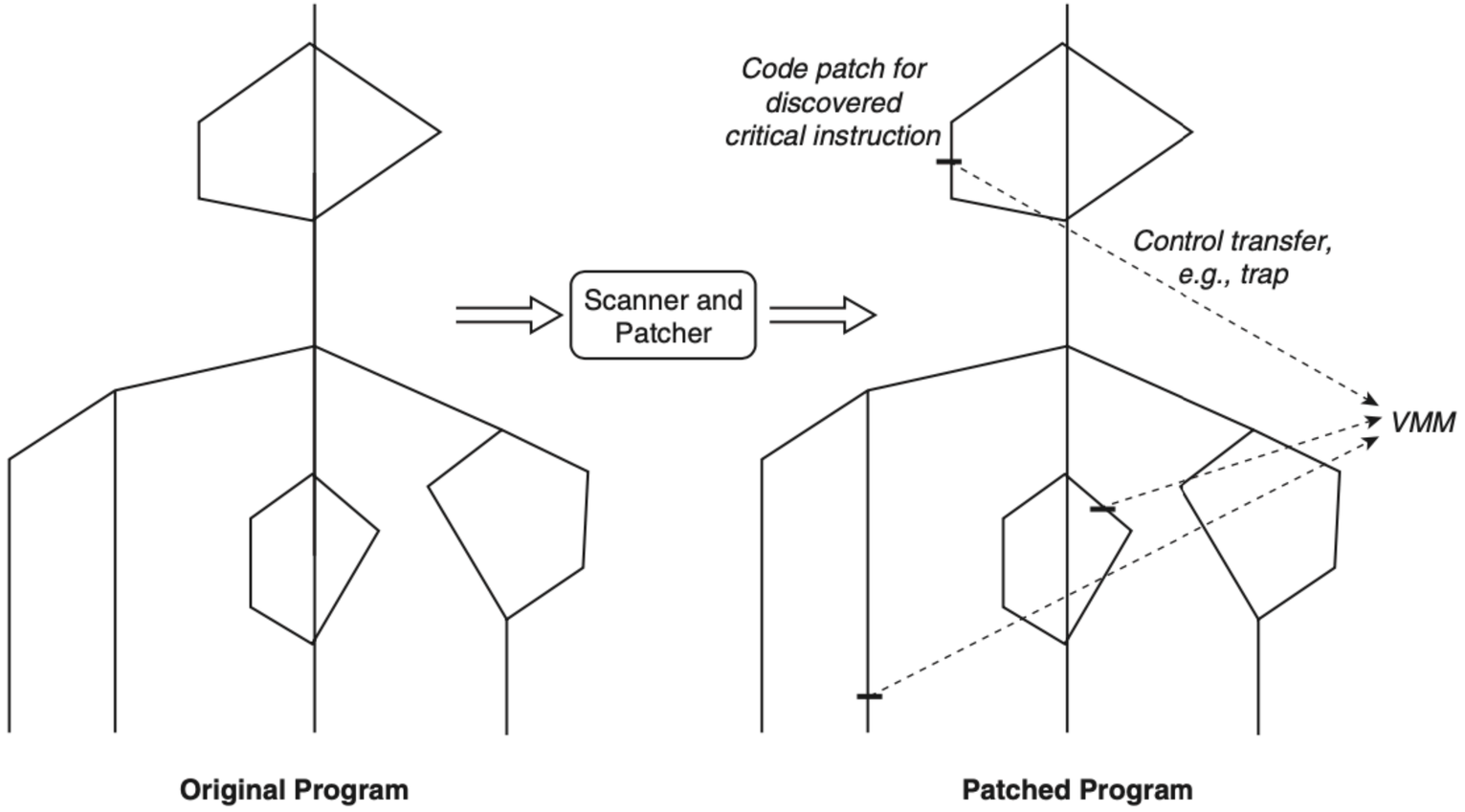
- x86-32 critical instructions:

Group (x86-32)	Instructions
Access to interrupt flags	pushf, popf, iret
Visibility into segment descriptors	lar, verr, verw, lsl
Segment manipulation	pop <seg>, push <seg>, mov <seg>
Read-only access to privileged state	sgdt, sldt, sidt, smsw
Interrupt and gate	fcall, longjump, retfar, str, int <n>

Handling Critical Instructions

- Force these problematic instructions to trap
  - Supporting **Full virtualization:** use live patching or binary translation at runtime to replace problematic instructions and emulate them
  - Supporting **Para virtualization:** search for problematic instructions in the source code and replace them with calls to VMM

Patching Critical Instructions

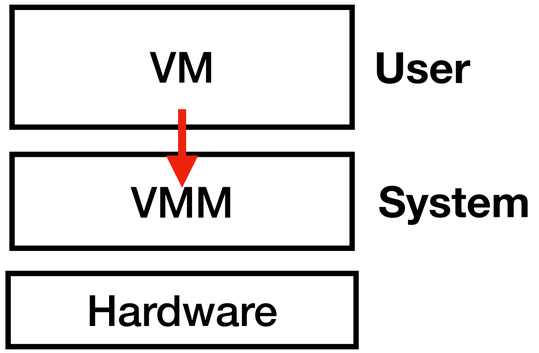


Case Study: Trap Critical Instructions

//CPS from Arm modifies current state register (CPSR) to enable interrupts
cpsie if

Result in NOOP when executing in user mode!

//Replace CPS with Arm's system call instruction (syscall) to trap to the VMM — encode the instruction so the VMM can distinguish from others
svc #CPSIE_IF



Adopting trap and emulate (1)

- Sensitive instructions are widely used in OS kernels
- Q: Can you name a few spots?

Adopting trap and emulate (2)

- Problem: trapping frequently results in huge performance overhead:
  - What is the cost here?

Ask yourself: design hypervisor to support trap-and-emulate

Agenda

- Introduction to System VMs
- CPU Virtualization
  - **Hardware Support for Virtualization**

Hardware Support for Virtualization

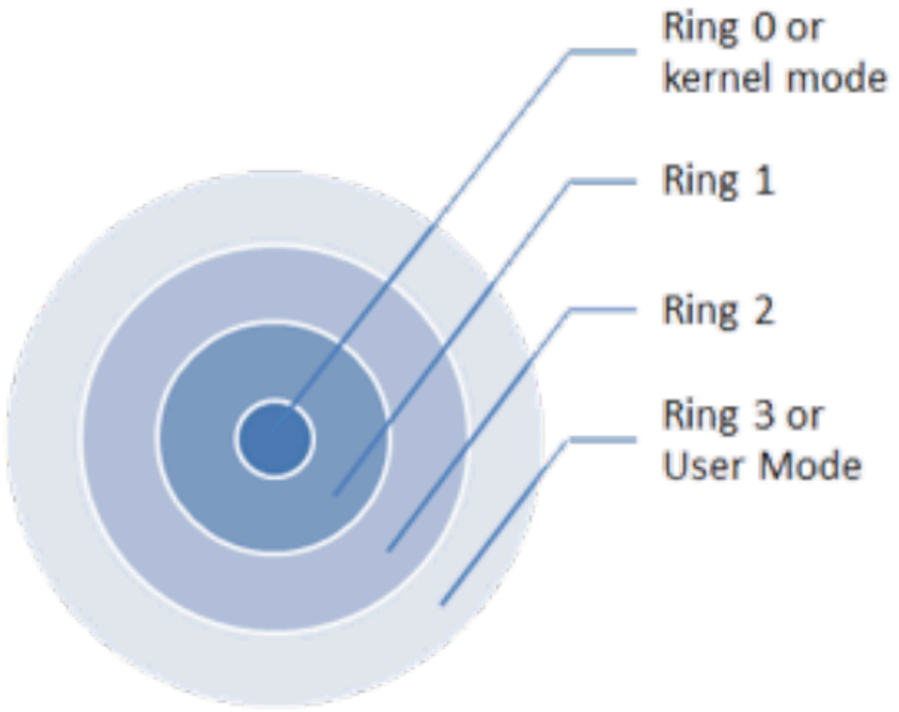
- Modern CPU architectures provide virtualization hardware support
  - x86, Arm, and other architectures introduced hardware support
- Support most of the sensitive instructions without trapping
  - VMs can execute most of the sensitive instructions natively in kernel mode
  - Must ensure the hypervisors retain complete hardware control and modify sensitive resources: some sensitive instructions still trap to the hypervisor
- Support full virtualization
  - No modification required in OS kernels

ARM Virtualization Extensions

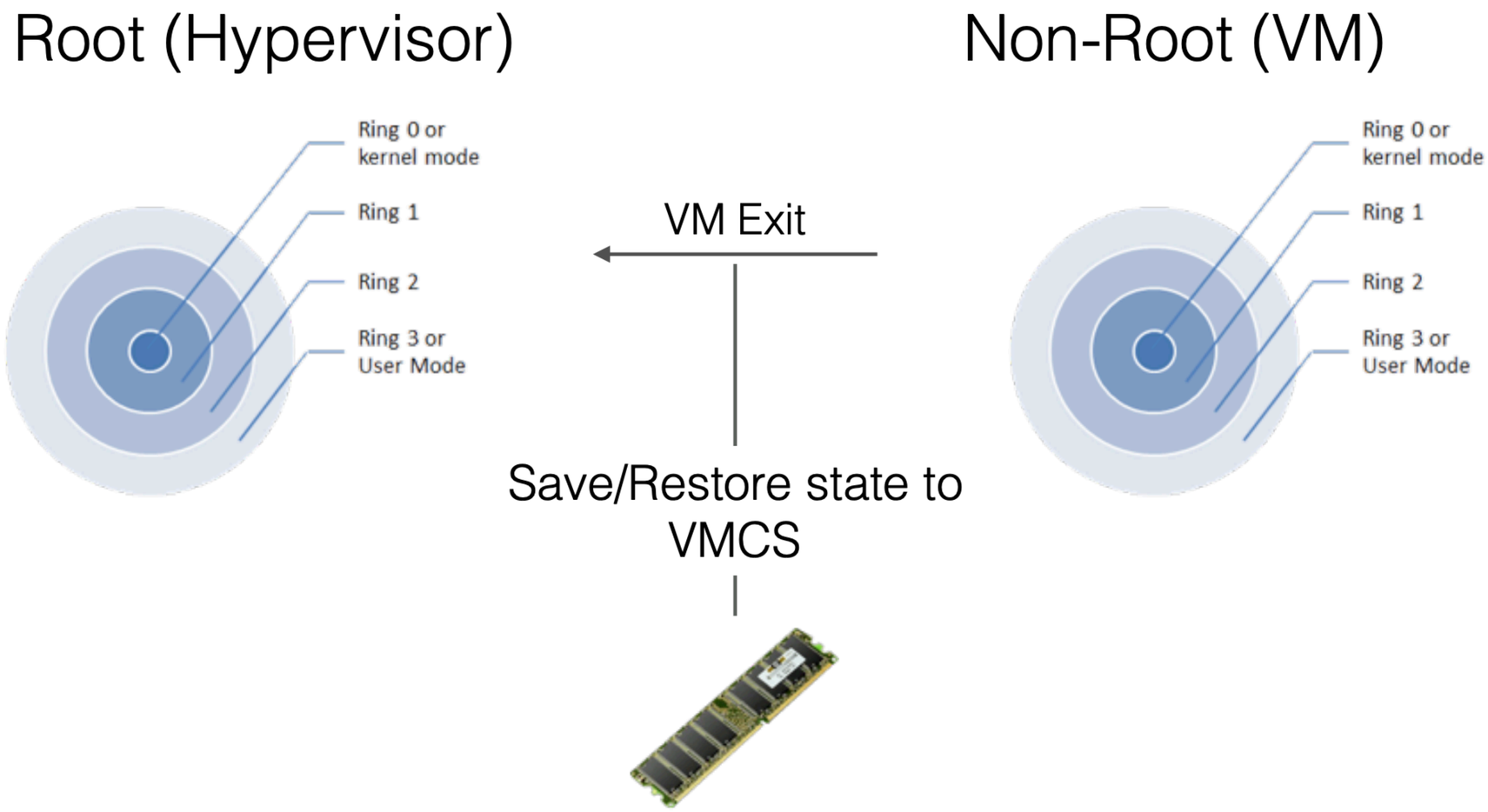
intel Virtual-Machine Extensions

Case Study: Intel VT-x (1)

- Intel x86 includes 4 protection rings with different privileges
  - Ring 0: kernel mode
  - Ring 3: user mode



Case Study: Intel VT-x (2)



Case Study: Intel VT-x (3)

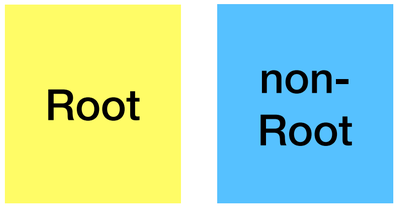
- Also called Intel Virtual Machine Extension (VMX)
  - Check if your system supports VMX: in Linux, you can grep for “vmx” from the output of “cat /proc/cpuinfo”
- Provides new execution context to run CPUs: ***root/non-root operations***
  - Root operation allows full hardware control — for running VMMs
  - Non-root operation is restricted — for running VMs in a virtualized environment; VMs cannot compromise VMM states
- The root/non-root operations are orthogonal to the exiting rings: both have the same set of the protection rings

Case Study: Intel VT-x (4)

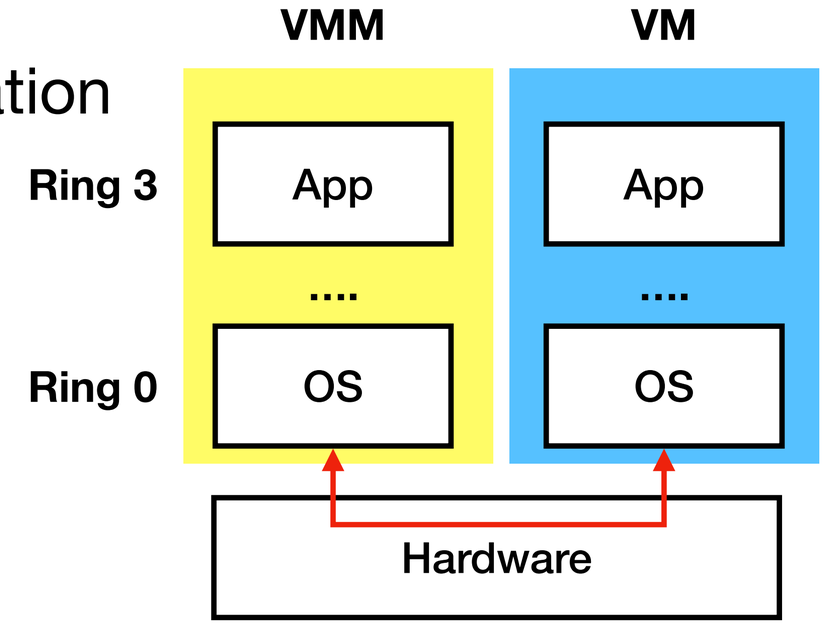
- Most of the VM's sensitive instructions can run natively in Ring 0 of Non-Root operation
  - Do not need to trap during context switches — improve performance
  - Some still cause traps to VMM
  - Support Full Virtualization (no guest OS modification is needed)

Case Study: Intel VT-x (5)

- Both VMM and VMs share the same four rings
  - VM exits: switches from non-root to root operation
  - VM enters: switches from root to non-root operation



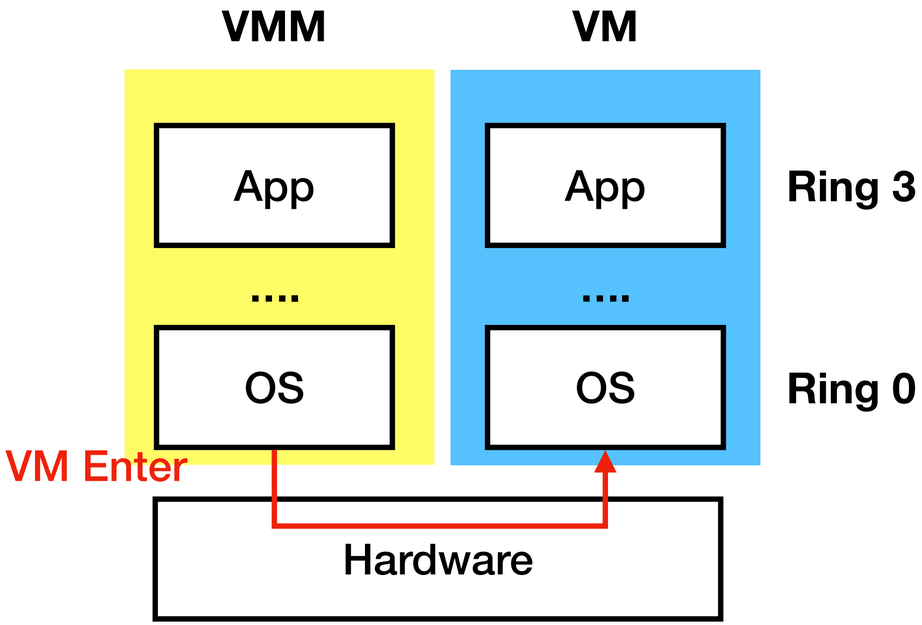
Virtual-Machine Extension



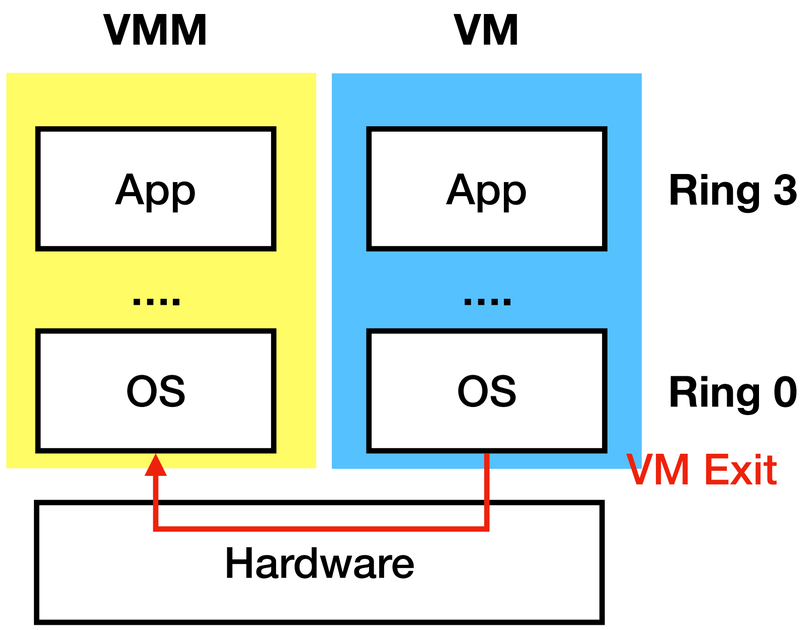
Case Study: Intel VT-x (6)

- VMX provides a ***Virtual Machine Control Structure (VMCS)***
  - An in-memory data structure (analogous to struct vcpu) that is per-CPU
  - Include fields to store the VMM and VM states (registers, control info)
    - The VMX hardware ***automatically context switches*** hardware states using the VMCS
    - VMCS does not include the general purpose registers; what is the implication?
  - Contain VM exit information (e.g., VM exit reasons)
  - Allow the hypervisor to configure fine-grain control VM execution on a CPU:
    - Example: to trap certain CPU operations, control memory accesses, etc

Case Study: Intel VT-x (7)



VMX saves VMM states to VMCS and restores VM states from VMCS to the hardware



VMX saves VM states to VMCS and restores VMM states from VMCS to the hardware

Case Study: Intel VT-x (8)

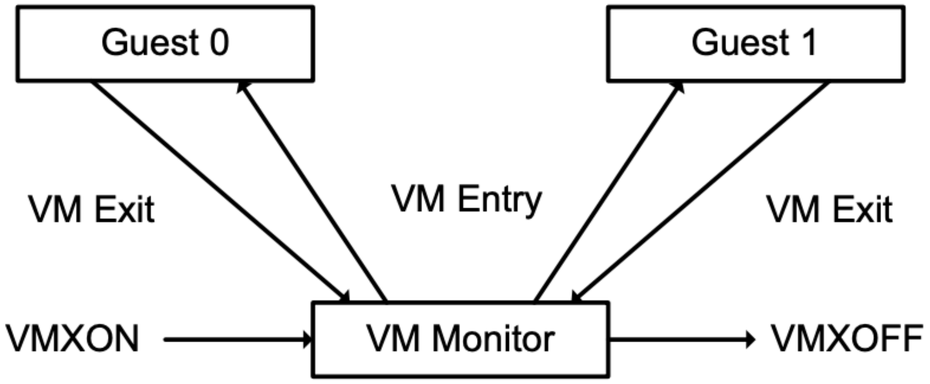
- VMX provides new instructions for VMCS management
  - VMMs use *VMCLEAR* to initialize a VMCS
  - VMMs use *VMPTRLD* to load a VMCS (base) to a given CPU
  - VMMs can use *VMREAD/VMWRITE* to access fields in the VMCS

```
static __always_inline void __vmcs_writel(unsigned long field, unsigned long value)
{
        vmx_asm2(vmwrite, "r"(field), "rm"(value), field, value);
}
```

vmwrite in KVM

Case Study: Intel VT-x (9)

- VMX provides a set of instructions to the hypervisors to execute VMs
  - Software enters VMX operation using *VMXON*; exits VMX operation using *VMXOFF*
  - VMM enters the VM using *VMLAUNCH* (on the first enter) and *VMRESUME* (on the following enters)



Arm CPUs

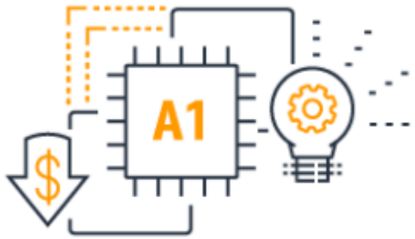
ARM Servers

ARM®

Virtualization Extensions

Arm CPUs: support for virtualization (1)

- Arm servers supported on Amazon AWS - the a1 instances



Leap to Graviton2

AWS Graviton processor
- First Arm processor in AWS
- First-class citizen in EC2
- ~5 Billion transistors
- 16nm

AWS Graviton2 processor
- 4x the vCPUs
- 7x CPU performance
- ~2x performance/vCPU
- ~30 Billion transistors
- 7nm

Achieve up to 34% better price/performance with AWS Lambda Functions powered by AWS Graviton2 processor

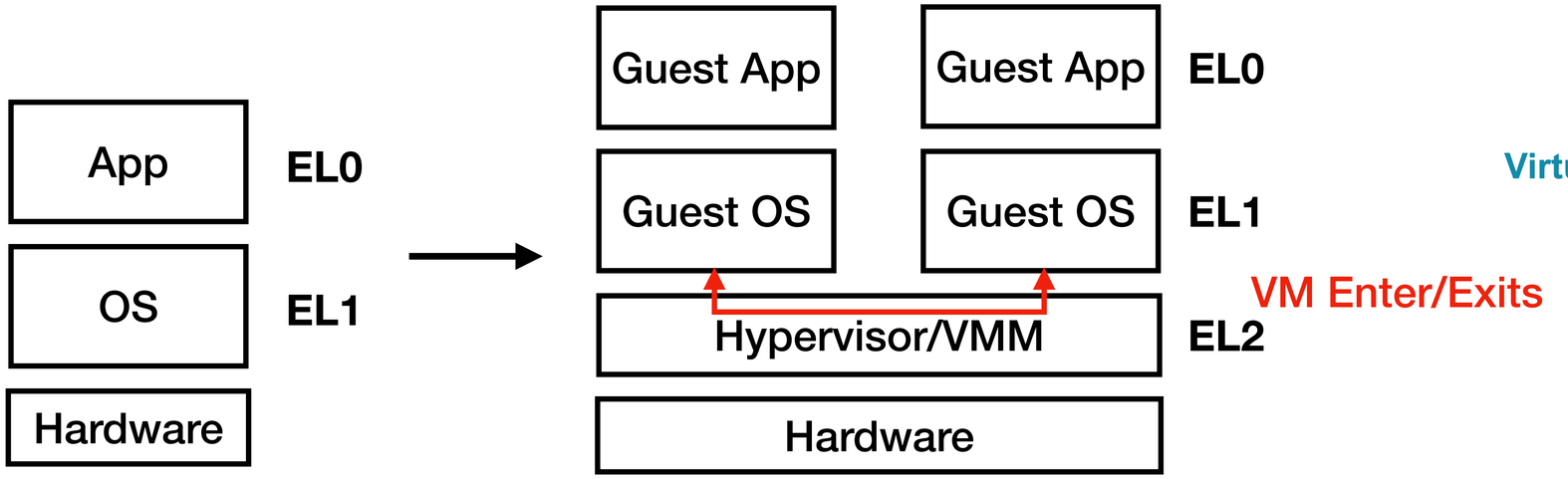
Posted On: Sep 29, 2021

Arm CPUs: support for virtualization (2)

- Architectural updates in mid 2010s for supporting server workloads
  - Larger address space with 64-bit support
  - Back work compatible with 32-bit support (Armv7 and before)
  - Improved support for SIMD and floating point computation

Case Study: Arm VE (Virtualization Extensions) (1)

- Armv8 adds a new CPU mode, EL2 (Hyp mode) to run hypervisors
  - Different approach compared to Intel VMX
  - Allow guest OS and Apps to run in EL1 and EL0, respectively

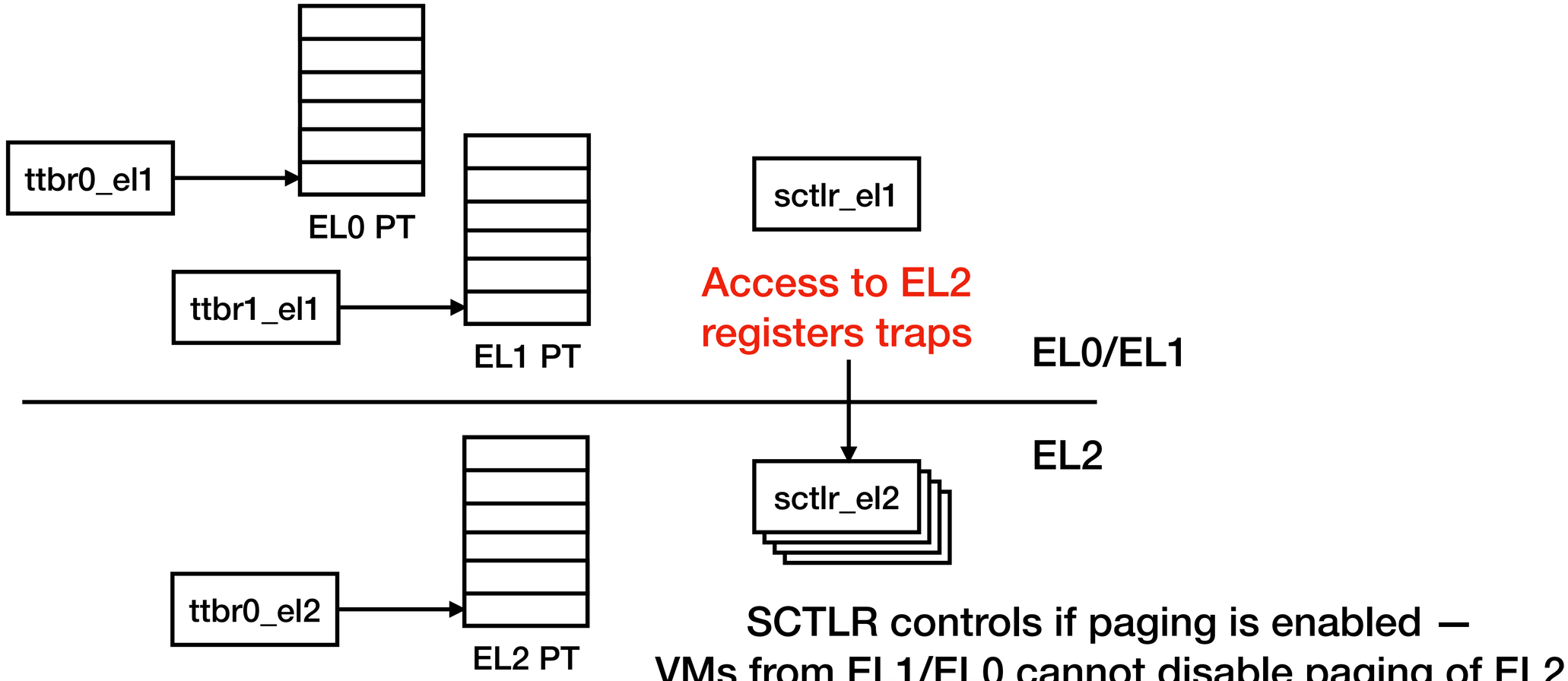


ARM
Virtualization Extensions

Case Study: Arm VE (Virtualization Extensions) (2)

- Arm VE supports features to isolate VMM running in EL2 from EL0/EL1
- How is isolation achieved?
  - EL2 has a separate set of system registers and address space — preventing VMs from accessing the VMM's registers and memory
  - EL2 registers control the behaviors of EL1/EL0, including instruction execution, memory access, and interrupt handling
    - EL2 registers are only accessible by the code executing in EL2
  - Unlike Intel VMX, Arm VE ***does not*** automatically context-switch hardware states; task the hypervisor (running in EL2) to context switches VM states

Case Study: Arm VE (Virtualization Extensions) (3)



Armv8: Call to the privileged software

`SVC` - Supervisor call

Causes an exception targeting EL1.

Used by an application to call the OS.

`HVC` - Hypervisor call

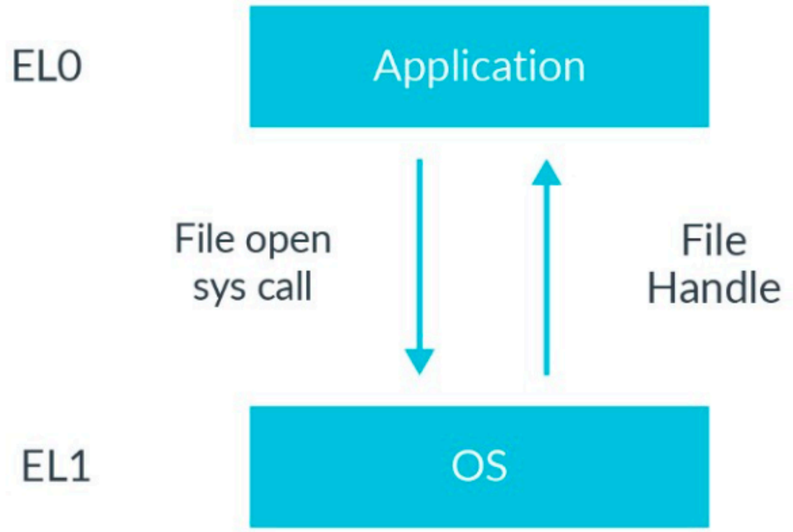
Causes an exception targeting EL2.

Used by an OS to call the hypervisor, not available at EL0.

`SMC` - Secure monitor call

Causes an exception targeting EL3.

Used by an OS or hypervisor to call the EL3 firmware, not available at EL0.



From: Armv8-A-Instruction-Set-Architecture

Hardware-assisted Virtualization

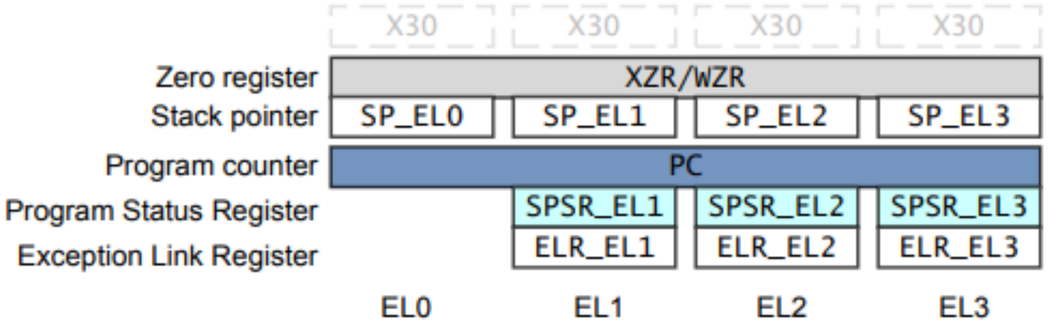
- Most of the VM's instructions can run on the hardware without trapping to the VMM
  - The performance-critical system VMs (e.g., in the cloud) nowadays do so
  - Can we virtualize hypervisor mode (aka root operation or EL2)?
- VMs executing sensitive instructions still require VMM intervention
  - Arm's WFI/WFE, which puts the CPU into low power state
  - Instructions that access I/O devices, ex: MMIO instructions

Armv8: CPU registers (1)

- 31 General Purpose Registers (GPRs):
  - Each of them can be used as 64-bit (Xn) register or 32-bit (wn) register
- Program Counter (PC), Stack Pointer (SP), Exception Link Register (ELR)

Armv8: CPU registers (2)

- Process Status Register (PSTATE, SPSR)
  - PSTATE includes conditional flags (Z, C, ..), Exception Masks (A, I, F), and the current Execution states (EL0/EL1/EL2, 32 or 64-bit)
  - SPSR holds the value of PSTATE on exceptions



Armv8: CPU registers (3)

- Arm system registers: are used to configure the processor and to control systems such as the MMU and exception handling:
  - Translation Table (Page Table) base (ttbr)
  - Paging/traps control (sctlr/tcr), memory attributes (mair)
  - Specify the exception vector (vbar), exception syndromes (esr), page fault address (far)
- Some system registers are banked in different modes, some are unique in a specific mode
  - ESR (exception syndrome register) exists in kernel/hyp modes
  - HCR (hypervisor control register) exists in hyp mode

Armv8: CPU registers - Accessing System Registers (1)

```
MRS    Xd, <system register>
```

reads the system register into `Xd`.

```
MSR    <system register>, Xn
```

write `Xn` to the system register.

System registers are specified by name, for example `SCTLR_EL1`:

```
MRS    X0, SCTLR_EL1
```

reads `SCTLR_EL1` into `X0`.

From: Armv8-A-Instruction-Set-Architecture

Armv8: CPU registers - Accessing System Registers (2)

System register names end with `_ELx`. The `_ELx`specifies the minimum privilege necessary to access the register. For example:

```
SCTLR_EL1
```

requires EL1 or higher privilege.

```
SCTLR_EL2
```

requires EL2 or higher privilege.

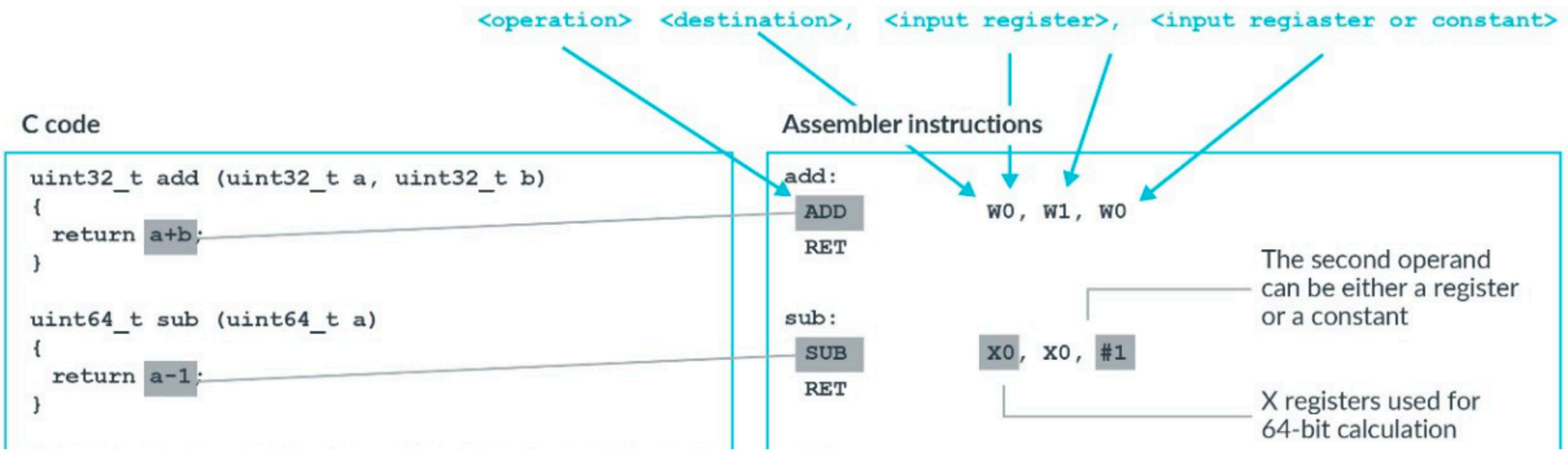
```
SCTLR_EL3
```

requires EL3 privilege

Attempting to access the register with insufficient privilege results in an exception.

From: Armv8-A-Instruction-Set-Architecture

Armv8: Arithmetic Operations



From: Armv8-A-Instruction-Set-Architecture

Armv8: Loads and Stores

```
LDR<Sign><Size>     <Destination>, [<address>]
STR<Size>     <Destination>, [<address>]
```

```
LDR    W0, [<address>]
```

This instruction loads 64 bits fro

```
LDR    X0, [<address>]
```

The <Size> field allows you to lo
instruction stores the bottom by

```
STRB   W0, [<address>]
```

This instruction stores the botto

```
STRH   W0, [<address>]
```

Finally, this instruction stores th

```
STRW   X0, [<address>]
```

8.4 Load pair and store pair

So far, we have discussed the load and store of a single register. A64 also has load (`LDP`) and store pair (`STP`) instructions.

These pair instructions transfer two registers to and from memory. The first instruction loads `[X0]` into `W3`, and loads `[X0 + 4]` into `W7`:

```
LDP    W3, W7, [X0]
```

This second instruction stores `D0` into `[X4]` and stores `D1` to `[X4 + 8]`:

```
STP    D0, D1, [X4]
```

Load and store pair instructions are often used for pushing, and popping off the stack. This first instruction pushes `X0` and `X1` onto the stack:

```
STP    X0, X1, [SP, #-16]!
```

This second instruction pops `X0` and `X1` from the stack:

```
LDP    X0, X1, [SP], #16
```

Remember that in AArch64 the stack-pointer must be 128-bit aligned.

From: Armv8-A-Instruction-Set-Architecture

Armv8: Program Flow

C	Typical output from a compiler

`if (a == 5)`
`  b = 5;`
	`  CMP W0, #5`
	`  B.NE skip`
	`  MOV W8, #5`
	`skip:`
	`…`

`while (a != 0)`
`{`
`  b = b + c;`
`  a = a - 1;`
`}`
	`loop:`
	`  CBZ W8, skip`
	`  ADD W9, W9, W10`
	`  SUB W8, W8, #1`
	`  B loop`
	`skip:`
	`…`

C	Branching

`if (a != 0)`
`  b = b + 1;`
	`CMP  W0, #0`
	`  B.EQ else`
	`  ADD W1, W1, #1`
	`else:`
	`…`

`if (a == 0)`
`  y = y + 1;`
`else`
`  y = y - 1;`
	`CMP  W0, #0`
	`  B.NE else`
	`  ADD W1, W1, #1`
	`  B end`
	`else:`
	`  SUB W1, W1, #1`
	`end:`
	`…`

From: Armv8-A-Instruction-Set-Architecture

Virtualizing Armv8 CPU in KVM

```
struct user_pt_regs {
        __u64           regs[31];
        __u64           sp;
        __u64           pc;
        __u64           pstate;
};
```

```
enum vcpu_sysreg {
        __INVALID_SYSREG__,   /* 0 is reserved as an invalid value */
        MPIDR_EL1,      /* MultiProcessor Affinity Register */
        CSSELR_EL1,     /* Cache Size Selection Register */
        SCTLR_EL1,      /* System Control Register */
        ACTLR_EL1,      /* Auxiliary Control Register */
        CPACR_EL1,      /* Coprocessor Access Control */
        ZCR_EL1,        /* SVE Control */
        TTBR0_EL1,      /* Translation Table Base Register 0 */
        TTBR1_EL1,      /* Translation Table Base Register 1 */

        NR_SYS_REGS     /* Nothing after this line! */
};
```

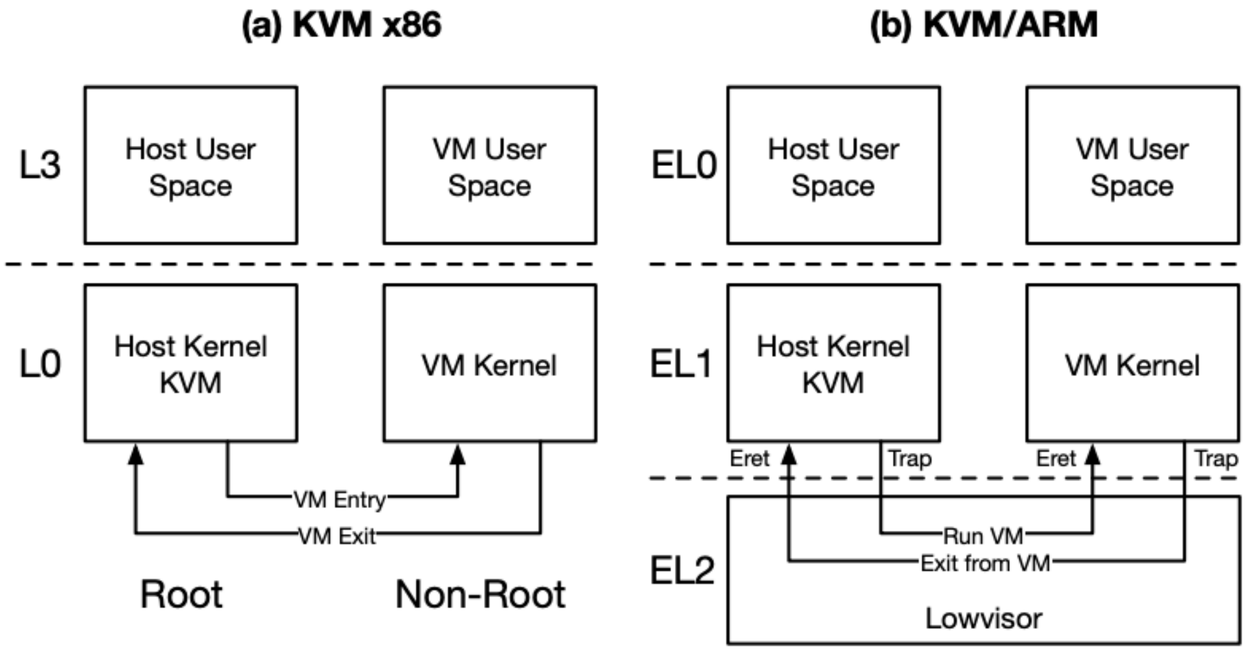
```
struct kvm_cpu_context {
        struct user_pt_regs regs;       /* sp = sp_el0 */

        u64     spsr_abt;
        u64     spsr_und;
        u64     spsr_irq;
        u64     spsr_fiq;

        struct user_fpsimd_state fp_regs;

        u64 sys_regs[NR_SYS_REGS];

        struct kvm_vcpu *__hyp_running_vcpu;
```

Per Virtual CPU state in KVM for Arm

Virtualizing Armv8 CPU in KVM

- KVM context (world) switches the hardware CPU states on VM enter/exit
  - On VM exit; KVM saves VM GPRs from hardware in assembly code first; then saves VM system registers in C code
  - On VM enter; KVM restores VM system registers to hardware in C code; then restores VM GPRs assembly
- Question: Why the order?

KVM with Virtualization Hardware Support



From: https://www.usenix.org/system/files/conference/atc17/atc17-dall.pdf

KVM for Arm

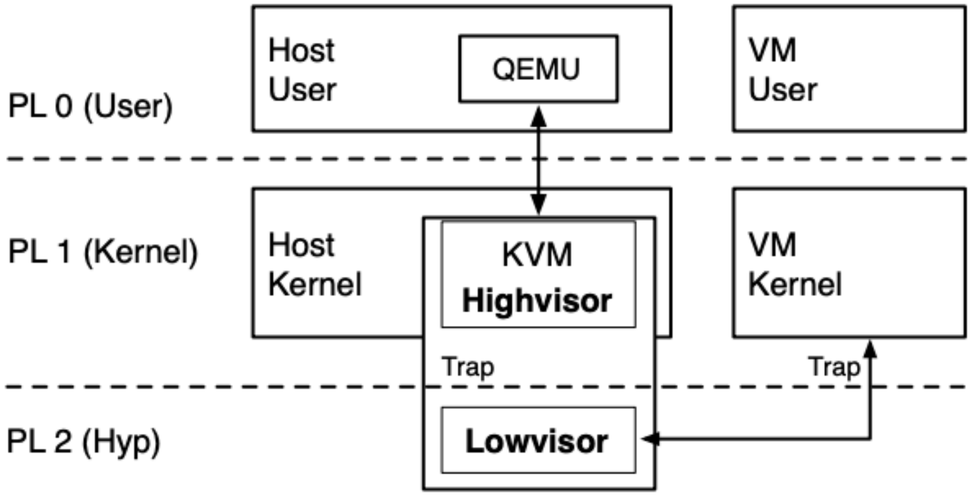
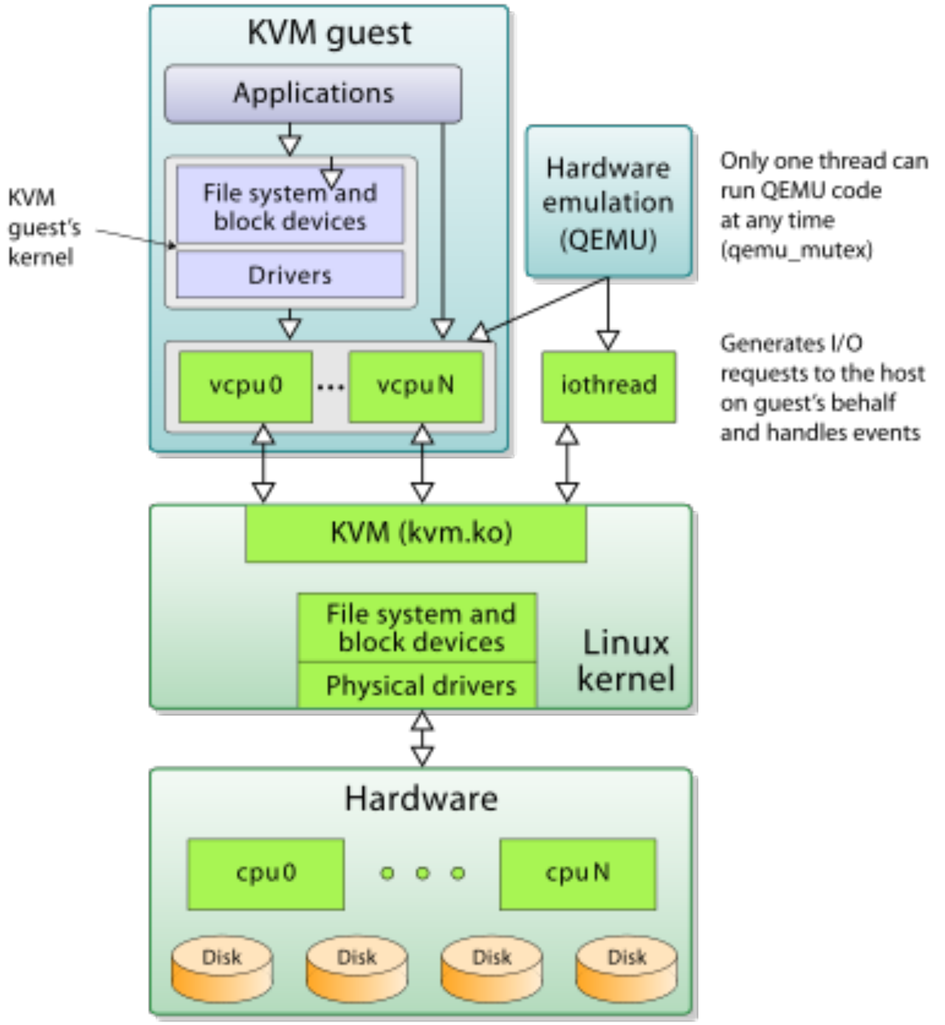


Figure 2: KVM/ARM System Architecture



From: KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor, ASPLOS 14