Copyright Notice

These slides are distributed under the Creative Commons Attribution 3.0 License

• You are free:
  • to share—to copy, distribute and transmit the work
  • to remix—to adapt the work

• under the following conditions:
  • **Attribution:** You must attribute the work (but not in any way that suggests that the author endorses you or your use of the work) as follows:
    
    “Courtesy of Gernot Heiser, UNSW Sydney”

The complete license text can be found at http://creativecommons.org/licenses/by/3.0/legalcode
Virtual Machine (VM)

“A VM is an efficient, isolated duplicate of a real machine” [Popek&Goldberg 74]

- **Duplicate**: VM should behave identically to the real machine
  - Programs cannot distinguish between real or virtual hardware
  - Except for:
    - Fewer resources (potentially different between executions)
    - Some timing differences (when dealing with devices)
- **Isolated**: Several VMs execute without interfering with each other
- **Efficient**: VM should execute at speed close to that of real hardware
  - Requires that most instruction are executed directly by real hardware

**Hypervisor aka virtual machine monitor (VMM):**
Software layer implementing the VM
Types of Virtualisation

- **Process**
- **Operating System**
- **Processor**

**Virtualisation Layer**

- **VM**
- **Process**
- **OS**
- **Hypervisor**
- **Operating System**
- **Processor**

**Platform VM or System VM**

- Type-1: “Bare metal”
- Type-2: “Hosted”

**OS API**

**Programming Language**

- **Java Program**
- **Java VM**

**Plus anything else you want to sound cool!**
Why Virtual Machines?

• Historically used for easier sharing of expensive mainframes
  • Run several (even different) OSes on same machine
    • called guest operating system
  • Each on a subset of physical resources
  • Can run single-user single-tasked OS in time-sharing mode
    • legacy support

Obsoleted by 1980s
Why Virtual Machines?

- Heterogenous concurrent guest OSes
  - eg Linux + Windows
- Improved isolation for consolidated servers: QoS & Security
  - total mediation/encapsulation:
    - replication
    - migration/consolidation
    - checkpointing
    - debugging
- Uniform view of hardware

Would not be needed if OSes provided proper security & resource management!
Why Virtual Machines: Cloud Computing

- Increased utilisation by sharing hardware
- Reduced maintenance cost through scale
- On-demand provisioning
- Dynamic load balancing through migration

Cloud Provider Data Centre

H/W

Hypervisor

OS

App

B.com

OS

H/W

App

A.com

OS

H/W

App
Hypervisor aka Virtual Machine Monitor

- Software layer that implements virtual machine
- Controls resources
  - Partitions hardware
  - Schedules guests
    - “world switch”
  - Mediates access to shared resources
    - e.g. console, network

**Implications:**
- Hypervisor executes in *privileged* mode
- Guest software executes in *unprivileged* mode

- Privileged guest instructions trap to hypervisor
Native vs Hosted Hypervisor

- Hosted VMM besides native apps
  - Sandbox untrusted apps
  - Convenient for running alternative OS on desktop
  - Leverage host drivers

Overheads:
- Double mode switches
- Double context switches
- Host not optimised for exception forwarding
Virtualisation Mechanics: Instruction Emulation

- Traditional *trap-and-emulate* (T&E) approach:
  - guest attempts to access physical resource
  - hardware raises exception (trap), invoking HV’s exception handler
  - hypervisor emulates result, based on access to virtual resource

Guest

```assembly
ld r0, curr_thrd
ld r1, (r0, ASID)
mv CPU_ASID, r1
ld sp, (r1, kern_stk)
```

VMM

```assembly
lda r1, vm_reg_ctxt
ld r2, (r1, ofs_r0)
sto r2, (r1, ofs_ASID)
```

Most instructions do not trap
- prerequisite for efficient virtualisation
- requires VM ISA (almost) same as processor ISA
Trap & Emulate Requirements

• **Privileged instruction:** when executed in user mode will *trap*

• **Privileged state:** determines resource allocation
  • Incl. privilege level, PT ptr, exception vectors…

• **Sensitive instruction:**
  • *control sensitive:* change privileged state
  • *behaviour sensitive:* expose privileged state
    • eg privileged instructions which NO-OP in user state

• **Innocuous instruction:** not sensitive

---

**No-op is insufficient!**

• Some inherently sensitive, e.g. set interrupt level
• Some context-dependent, e.g. store to page table

---

**T&E virtualisable HW:**
All sensitive instructions are privileged

Can run unmodified guest binary
"Impure" Virtualisation

- Support non-T&E hardware
- Improve performance

- Modify binary: *binary translation* (VMware)
- Modify hypervisor "ISA": *para-virtualisation*
Virtualisation vs Address Translation

Virtual Memory → Virtual Page Table → Guest Physical Memory

Virtual Memory → Virtual Page Table → Guest Physical Memory

Virtual Memory → Page Table → Guest Physical Memory

Two levels of address translation!

Must implement with single MMU translation!
Virtualisation Mechanics: Shadow Page Table

Virt_PT_ptr (Software)

Guest

PT_ptr (Hardware)

VMM

User

ld r0, adr

Guest virtual address

(Virtual) guest page table

Physical address

data

Hypervisor's guest memory map

Shadow (real) guest page table, translations cached in TLB

Guest physical address

Memory

Guest (Virtual) page table

Hypervisor's guest memory map

Physical address

data

Virt_PT_ptr (Software)
Mechanics: Shadow Page Table

Hypervisor must shadow (virtualize) PT updates by guest:
- trap guest writes to guest PT
- translate guest PA in guest (virtual) PTE using memory map
- insert translated PTE in shadow PT

Shadow PT has TLB semantics (i.e. weak consistency) ⇒
Update at synchronisation points:
- page faults
- TLB flushes

SPT is a virtual TLB
- similar semantics
- can be incomplete

Virt_PT_ptr (Software)

PT_ptr (Hardware)

Memory

data

User

Guest

VMM

ld r0, adr

Guest virtual address

Guest physical address

Physical address

Used by VMware
Mechanics: Lazy Shadow Update

User

Guest OS

add mapping in GPT

add another mapping; return to user

Hypervisor

write-protect GPT

unprotect GPT & mark dirty

update dirty shadow; write-protect GPT

access new page ...

...
Mechanics: Lazy Shadow Update

User

Guest OS

- Invalidate mapping in GPT
- Invalidate another mapping; flush TLB
- return to user

Hypervisor

- write-protect GPT
- unprotect GPT & mark dirty
- update dirty shadow; write-protect GPT; flush TLB

return to user
Mechanics: Real Guest Page Table

VMM maintains guest PT

On guest PT access must translate (virtualise) PTEs:
- store: guest “PTE” → real PTE
- load: real PTE → guest “PTE”

Each guest PT access traps!
Mechanics: Optimised Guest Page Table

- Guest translates PTE on read from PT
  - Linux PT-access wrappers help
  - Guest batches PR updates
  - hypercalls to reduce overhead

- Guest "knows" it's virtualised

- Used by original Xen

- Guest virtual address

- Physical address
Mechanics: Guest Self-Virtualisation

Example: Interrupt-enable in virtual PSR
- guest and VMM agree on VPSR location
- VMM queues guest IRQs when disabled in VPSR

```assembly
# Virtual State in Guest
VPSR: 0

# Physical State Register
PSR: 0

mov r1, #VPSR
ldr r0, [r1]
or r0, r0, #VPSR_ID
sto r0, [r1]
```
Mechanics: Device Models

- Emulated
- Split (para-virtualised)
- Pass-through
Mechanics: Emulated Device

Each device access must be trapped and emulated
- unmodified native driver
- high overhead!
- may not actually work, violate device timing constraints
Mechanics: Split Driver

Simplified, high-level device interface
- small number of hypercalls
- new (but very simple) driver
- low overhead
- must port drivers to hypervisor

virtio: Linux I/O virtualisation interface

“Para-virtualized” driver

Virtual device: simple interface
Mechanics: Driver OS (Xen Dom0)

- Leverage native drivers
- No driver porting
- Must trust complete driver guest!
- Huge trusted computing base (TCB)!
Mechanics: Pass-Through Driver

Unmodified native driver
- Must trust driver (and guest) for DMA
  - except with hardware support: I/O MMU
- Can’t share device between VMs
  - except with hardware support: recent NICs

“Self-virtualising” devices:
- Single-root I/O virtualisation (SRIOV)
- NIC presenting multiple, isolated virtual NIC interfaces
x86 Virtualisation Extensions: VT-x

New processor mode: VT-x root mode
- orthogonal to protection rings
- entered on virtualisation trap

Traditional x86 behaviour

Guest Kernel

Hypervisor

Kernel entry
VM exit
Arm Virtualisation Extensions (1)

**EL₂ aka “hyp mode”**

- **Normal world**
  - EL₀
  - User mode
  - Kernel modes

- **Secure world**
  - EL₁
  - User mode
  - Kernel modes
  - EL₂
  - Monitor mode

**New privilege level**
- Strictly higher than kernel (EL₁)
- Virtualizes or traps *all* sensitive instructions
- Presently only available in Arm TrustZone “normal world”
- Next ISA revision supports it also in “secure world”
Arm Virtualisation Extensions (2)

Configurable Traps

User mode

Kernel mode

Hyp mode

Native syscall

Virtual syscall

Can configure traps to go directly to guest OS

Big performance boost!

x86 similar

User mode

Kernel mode

Virtual syscall

Trap to guest
Arm Virtualisation Extensions (3)

Emulation

1) Load faulting instruction:
   • Compulsory L1-D miss!

2) Decode instruction
   • Complex logic

3) Emulate instruction
   • Usually straightforward

```
IR
mv CPU_ASID, r1
ld r1, (r0, ASID)
mv CPU_ASID, r1
ld sp, (r1, kern_stk)
```

```
L1 I-Cache
```

```
L1 D-Cache
mv CPU_ASID, r1
...
```

```
L2 Cache
ld r1, (r0, ASID)
mv CPU_ASID, r1
ld sp, (r1, kern_stk)
```
Arm Virtualisation Extensions (3)

Emulation

No x86 equivalent

1) HW decodes instruction
   - No L1 miss
   - No software decode

2) SW emulates instruction
   - Usually straightforward

IR

L1 I-Cache

L1 D-Cache

L2 Cache

mv CPU_ASID,r1

ld r1,(r0,ASID)
mv CPU_ASID,r1
ld sp,(r1,kern_stk)

mv CPU_ASID,r1

ld r1,(r0,ASID)
mv CPU_ASID,r1
ld sp,(r1,kern_stk)

...
Arm Virtualisation Extensions (4)

2-stage translation

- Hardware PT walker traverses both PTs
- PT walker loads combined (guest-virtual to physical) mapping into TLB
- eliminates “virtual TLB”

x86 similar (EPTs)
Arm Virtualisation Extensions (4)

2-stage translation cost

- On page fault walk twice number of page tables!
- Can have a page miss on each, requiring PT walk
- $O(n^2)$ misses in worst case for n-level PT
- Worst-case cost is massively worse than for single-level translation!

Trade-off:
- fewer traps
- simpler implementation
- higher TLB-miss cost up to 50% of run-time!
Arm Virtualisation Extensions (5)

Virtual Interrupts

- 2-part IRQ controller
  - global “distributor”
  - per-CPU “interface”
- New H/W “virt. CPU interface”
  - Mapped to guest
  - Used by HV to forward IRQ
  - Used by guest to acknowledge
- Halves hypervisor invocations for interrupt virtualization

x86: issue only for legacy level-triggered IRQs
**Arm Virtualisation Extensions (6)**

**System MMU (I/O MMU)**

- Devices use virtual addresses
- Translated by *system MMU*
  - elsewhere called I/O MMU
  - translation cache, like TLB
  - reloaded from I/O page table

**Physical Memory**

- guest accesses device registers
- no hypervisor invocation

**x86 different (VT-d)**

**Many ARM SoCs different**
RISC-V H Extension (Draft v0.6)

Add virtual U+S modes
- Extra registers for VM state
- Re-direct VS traps to S
- 2-stage address translation
- VIRQ injection

V = 1
- Virtual U mode
- Virtual S mode

V = 0
- User mode
- Supervisor mode
- Machine mode

Trap

Hypervisor
World Switch

**x86**
- VM state is $\leq 4$ KiB
- Save/restore done by hardware on VMexit/VMentry
- Fast and simple

**Arm**
- VM state is 488 B
- Save/restore done by hypervisor
- Selective save/restore
  - Eg traps w/o world switch

**RISC-V (draft)**
- VM state $\approx 80$ B
- Save/restore done by hypervisor
- Selective save/restore
  - Eg traps w/o world switch
Hybrid Hypervisor-OSes

Huge TCB, contains full Linux system (kernel and userland)!

Idea: Turn OS into hypervisor by running in VT-x root mode, pioneered by KVM

Often falsely called a “Type-2” hypervisor

Non-Root

VM
Guest apps
Guest kernel

Root

VM
Guest apps
Guest kernel

Hypervisor

Linux kernel “Host”
Drivers

Native Linux apps

Reuse Linux drivers!

Ring 3

Ring 0

VM exit
Why Still Have an OS?

Frequently single app (server) per VM

Library OS, linked to app
- Also “unikernel”
- Everything in virtual kernel mode
- Examples: Mirage, rump kernel
Fun and Games with Hypervisors

- Time-travelling virtual machines [King ‘05]
  - debug backwards by replaying VM from checkpoint, log state changes
- SecVisor: kernel integrity by virtualisation [Seshadri ‘07]
  - controls modifications to kernel (guest) memory
- Overshadow: protect apps from OS [Chen ‘08]
  - make user memory opaque to OS by transparently encrypting
- Turtles: Recursive virtualisation [Ben-Yehuda ‘10]
  - virtualize VT-x to run hypervisor in VM
- CloudVisor: mini-hypervisor underneath Xen [Zhang ‘11]
  - isolates co-hosted VMs belonging to different users
  - leverages remote attestation (TPM) and Turtles ideas
- Containers (Docker etc):
  - Example of OS API virtualisation