Virtualisation
@GernotHeiser
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

Type-1: "Bare metal"
Platform VM or System VM

Type-2: "Hosted"

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
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

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

Exception

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

VMM

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 mode, 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

Can run unmodified guest binary

T&E virtualisable HW:
- All sensitive instructions are privileged
"Impure" Virtualisation

- Support non-T&E hardware
- Improve performance

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

Insert trap – “hypercall”

```
ld   r0, curr_thrd
ld   r1, (r0,ASID)
trap
ld   sp, (r1,kern_stk)
```

Insert in-line emulation code

```
ld   r0, curr_thrd
ld   r1, (r0,ASID)
jmp  fixup_15
ld   sp, (r1,kern_stk)
```

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

- Virtual Memory
  - Virtual Page Table
  - Guest Physical Memory
    - Page Table
    - Two levels of address translation!
  - Virtual Page Table

- Virtual Memory
  - Virtual Page Table
  - Guest Physical Memory
    - Page Table
    - Must implement with single MMU translation!
Virtualisation Mechanics: Shadow Page Table

(Virtual) guest page table
Shadow (real) guest page table, translations cached in TLB
Hypervisor's guest memory map

User
Guest
VMM
Memory

Id r0, adr
Virt_PT_ptr (Software)
PT_ptr (Hardware)

Guest virtual address
Guest physical address
Physical address

data

Guest physical address
Guest virtual address
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

Is virtual TLB
- similar semantics
- can be incomplete
Mechanics: Lazy Shadow Update

**User**
- access new page
  ...

**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
Mechanics: Lazy Shadow Update

User

Guest OS

Hypervisor

continue

invalidate mapping in GPT

invalidate another mapping; flush TLB

return to user

write-protect GPT

unprotect GPT & mark dirty

update dirty shadow; write-protect GPT; flush TLB

flush TLB
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 virtual address
- Guest PT

VMM
- PT_ptr (Hardware)
- VMM PT
- Physical address

Guest
- Virt_PT_ptr (Software)
- Guest PT

User
- ld r0, adr

Pare-virtualised guest "knows" it's virtualised

Used by original Xen
Mechanics: Guest Self-Virtualisation

Minimise traps by holding some virtual state inside guest

Example: Interrupt-enable in virtual PSR

- guest and VMM agree on VPSR location
- VMM queues guest IRQs when disabled in VPSR

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

Emulated

Split (para-virtualised)

Pass-through

VM

OS

Device Driver

Apps

Emulation

VMM

Device

VM

OS

Virtual Driver

Apps

Device Driver

VMM

Device

VMM

Device
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

“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

Direct device access by guest
x86 Virtualisation Extensions: VT-x

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

Non-Root
- Ring 3
- Ring 2
- Ring 1
- Ring 0

Root
- Ring 3
- Ring 2
- Ring 1
- Ring 0

Guest Kernel

Hypervisor

VM exit

Kernel entry
Arm Virtualisation Extensions (1)

EL₂ aka “hyp mode”

New privilege level
- Strictly higher than kernel (EL₁)
- Virtualizes or traps all sensitive instructions
- Presently only available in Arm TrustZone “normal world”

<table>
<thead>
<tr>
<th>Mode</th>
<th>EL₀</th>
<th>EL₁</th>
<th>EL₂</th>
<th>EL₃</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>User mode</td>
<td>Kernel modes</td>
<td>User mode</td>
<td>Monitor mode</td>
</tr>
<tr>
<td>“Normal world”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>“Secure world”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Arm Virtualisation Extensions (2)

Configurable Traps

User mode

Kernel mode

Hyp mode

Virtual syscall

User mode

Kernel mode

x86 similar

Can configure traps to go directly to guest OS

Big performance boost!

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

```
mv CPU_ASID, r1
ld r1, (r0, ASID)
```

```
ld sp, (r1, kern_stk)
```

```
ld r1, (r0, ASID)
mv CPU_ASID, r1
```

```
ld sp, (r1, kern_stk)
```

```
ld r1, (r0, ASID)
mv CPU_ASID, r1
```

```
ld sp, (r1, kern_stk)
```

```
ld r1, (r0, ASID)
```

```
ld sp, (r1, kern_stk)
```

```
ld r1, (r0, ASID)
```

```
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

R2

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”
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

- Can do pass-through I/O safely
  - guest accesses device registers
  - no hypervisor invocation

- Many ARM SoCs different

- x86 different (VT-d)
World Switch

**x86**
- VM state is ≤ 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
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 exit

Hypervisor

Reuse Linux drivers!

Linux kernel
“Host”

Drivers

Native Linux apps

Linux demons

Native Linux apps

Drivers

VM exit

Non-Root

VM
Guest apps
Guest kernel

VM
Guest apps
Guest kernel
Fun and Games with Hypervisors

• Time-travelling virtual machines [King ‘05]
  • debug backwards by replay 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

... and many more..