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:
    o Fewer resources (and potentially different between executions)
    o 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: Software implementing the VM

“Real machine”: Modern usage more general, “virtualise” any API

Types of Virtualisation

- **Platform VM or System VM**
  - Processor
    - Operating System
  - Hypervisor
    - Virtualization Layer
  - OS API
  - Programming Language

- **OS-level VM**
  - Processor
    - Operating System
  - Hypervisor
    - Virtualization Layer
  - OS API
  - Programming Language

- **Type-1 Bare metal**
  - Processor
    - Operating System
  - Hypervisor
    - Virtualization Layer
  - OS API
  - Programming Language

- **Type-2 Hosted**
  - Processor
    - Java Program
  - OS-level VM
  - Operating System
  - Java VM
  - Programming Language

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
- Gone out of fashion in 80’s
  - Time-sharing OSes common-place
  - Hardware too cheap to worry...

Why Virtual Machines?

- Renaissance in recent years for improved isolation
- Server/desktop virtual machines
  - Improved QoS and security
  - Uniform view of hardware
  - Complete encapsulation
    - Replication
    - Migration/consolidation
    - Checkpointing
    - Debugging
  - Different concurrent OSes
    - Eg Linux + Windows
  - Total mediation
- Would be mostly unnecessary
  - … if OSes were doing their job!

Why Virtual Machines?

- Core driver today is Cloud computing
  - Increased utilisation by sharing hardware
  - Reduced maintenance cost through scale
  - On-demand provisioning
  - Dynamic load balancing through migration

Why Virtual Machines?

- Embedded systems: integration of heterogenous environments
  - RTOS for critical real-time functionality
  - Standard OS for GUIs, networking etc
- Alternative to physical separation
  - Low-overhead communication
  - Size, weight and power (SWaP) reduction
  - Consolidate complete components
    - Including OS,
    - Certified
    - Supplied by different vendors
    - Legacy support
  - “Dual-persona” phone
  - Secure domain on COTS device
Hypervisor aka Virtual Machine Monitor

- Program that runs on real hardware to implement the virtual machine
- Controls resources
  - Partitions hardware
  - Schedules guests
    - "world switch"
  - Mediates access to shared resources
    - e.g. console
- Implications
  - Hypervisor executes in *privileged* mode
  - Guest software executes in *unprivileged* mode
  - Privileged instructions in guest cause a trap into hypervisor
  - Hypervisor interprets/emulates them
  - Can have extra instructions for *hypercalls*

Native vs. Hosted VMM

Native/Classic/Bare-metal/Type-I
- Hosted VMM beside native apps
  - Sandbox untrusted apps
  - Convenient for running alternative OS on desktop
  - Leverage host drivers

Hosted/Type-II
- Less efficient
  - Double node switches
  - Double context switches
  - Host not optimised for exception forwarding

Virtualization 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
- Most instructions do not trap
  - prerequisite for efficient virtualisation
  - requires VM ISA (almost) same as processor ISA

<table>
<thead>
<tr>
<th>Guest</th>
<th>Hypervisor</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld r0, curr_thrd</td>
<td></td>
</tr>
<tr>
<td>ld r1, (r0, ASID)</td>
<td></td>
</tr>
<tr>
<td>mv CPU_ASID, r1</td>
<td></td>
</tr>
<tr>
<td>ld sp, (r1, kern_stk)</td>
<td></td>
</tr>
<tr>
<td>ld r1, vm_reg_ctxt</td>
<td></td>
</tr>
<tr>
<td>ld r2, (r1, ofs_r0)</td>
<td></td>
</tr>
<tr>
<td>sto r2, (r1, ofs_ASID)</td>
<td></td>
</tr>
</tbody>
</table>

Trap-and-Emulate Requirements

Definitions:
- **Privileged instruction**: traps when executed in user mode
  - Note: NO-OP is insufficient!
- **Privileged state**: determines resource allocation
  - Includes privilege mode, addressing context, exception vectors...
- **Sensitive instruction**: control- or behaviour-sensitive
  - control sensitive: changes privileged state
  - behaviour sensitive: exposes privileged state
    - incl instructions which are NO-OPs in user but not privileged state
- **Innocuous instruction**: not sensitive
- Some instructions are inherently sensitive
  - eg TLB load
- Others are context-dependent
  - eg store to page table
Trap-and-Emulate Architectural Requirements

- **T&E virtualisable**: all sensitive instructions are privileged
  - Can achieve accurate, efficient guest execution
  - VMM controls resources
  - Virtualized execution indistinguishable from native, except:
    - resources more limited (smaller machine)
    - timing differences (if there is access to real time clock)

- **Recursively virtualisable**:
  - run hypervisor in VM
  - possible if hypervisor not timing dependent, overheads low

Guest

<table>
<thead>
<tr>
<th>ld  r0, curr_thrd</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld  r1, (r0,ASID)</td>
</tr>
<tr>
<td>mv  CPU_ASID, r1</td>
</tr>
<tr>
<td>ld  sp, (r1,kern_stk)</td>
</tr>
</tbody>
</table>

Hypervisor

<table>
<thead>
<tr>
<th>lds r1, vm_reg_ctxt</th>
</tr>
</thead>
<tbody>
<tr>
<td>lds r2, (r1,ofs_r0)</td>
</tr>
<tr>
<td>sto r2, (r1,ofs_ASID)</td>
</tr>
</tbody>
</table>

Impure Virtualization

- Virtualise other than by T&E of unmodified binary
  - Two reasons:
    - Architecture not T&E virtualisable
    - Reduce virtualisation overheads
  - Change guest OS, replacing sensitive instructions
    - by trapping code ("hypercalls")
    - by in-line emulation code

Guest

<table>
<thead>
<tr>
<th>ld  r0, curr_thrd</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld  r1, (r0,ASID)</td>
</tr>
<tr>
<td>trc</td>
</tr>
<tr>
<td>ld  sp, (r1,kern_stk)</td>
</tr>
</tbody>
</table>

Hypervisor

<table>
<thead>
<tr>
<th>ld  r0, curr_thrd</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld  r1, (r0,ASID)</td>
</tr>
<tr>
<td>jmp</td>
</tr>
<tr>
<td>ld  sp, (r1,kern_stk)</td>
</tr>
</tbody>
</table>

Para-Virtualization

- New(ish) name, old technique
  - coined by Denali [Whitaker '02], popularised by Xen [Barham '03]
  - Mach Unix server [Gold '90], L4Linux [Härtig '97], Disco [Bugnion '97]
- Idea: manually port guest OS to modified (more high-level) ISA
  - Augmented by explicit hypervisor calls ("hypercalls")
    - higher-level ISA to reduce number of traps
    - remove unvirtualisable instructions
    - remove “messy” ISA features which complicate
  - Generally outperforms pure virtualisation, binary re-writing

Binary Translation

- Locate sensitive instructions in guest binary, replace on-the-fly by emulation or trap/hypercall
  - pioneered by VMware
  - detect/replace combination of sensitive instruction for performance
  - modifies binary at load time, no source access required
- Looks like pure virtualisation!
- Very tricky to get right (especially on x86!)
  - Assumptions needed about sane guest behaviour
  - “Heroic effort” [Orran Krieger, then IBM, later VMware] 😊

<table>
<thead>
<tr>
<th>ld  r0, curr_thrd</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld  r1, (r0,ASID)</td>
</tr>
<tr>
<td>av  CPU_ASID, r1</td>
</tr>
<tr>
<td>ld  sp, (r1,kern_stk)</td>
</tr>
</tbody>
</table>

Hardware

Guest

<table>
<thead>
<tr>
<th>ld  r0, curr_thrd</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld  r1, (r0,ASID)</td>
</tr>
<tr>
<td>jmp</td>
</tr>
<tr>
<td>ld  sp, (r1,kern_stk)</td>
</tr>
</tbody>
</table>
Virtualization Overheads

- VMM must maintain virtualised privileged machine state
  - processor status
  - addressing context
  - device state
- VMM needs to emulate privileged instructions
  - translate between virtual and real privileged state
  - eg guest ↔ real page tables
- Virtualisation traps are expensive
  - >1000 cycles on some Intel processors!
  - Better now, Skylake is ≈110 cyc in 64-bit mode
- Some OS operations involve frequent traps
  - STI/CLI for mutual exclusion
  - frequent page table updates during fork()
  - MIPS KSEG addresses used for physical addressing in kernel

Virtualization and Address Translation

Virtual Memory

Guest Physical Memory

Virtual Memory

Guest Physical Memory

Virtual Memory

Two levels of address translation!

Must implement with single MMU translation!

Virtualization Mechanics: Shadow Page Table

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

Used by VMware

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

Shadow PT as virtual TLB
- similar semantics
- can be incomplete: LRU translation cache
Virtualisation Semantics: Lazy Shadow Update

User
- access new page...

Guest OS
- add mapping to GPT
- add mappings... return to user

Hypervisor
- write-protect GPT
- unprotect GPT & mark dirty
- update dirty shadow
- write-protect GPT

Virtualisation Mechanics: Real Guest PT

- On guest PT access must translate (virtualize) PTEs
  - store: translate guest “PTE” to real PTE
  - load: translate real PTE to guest “PTE”
- Each guest PT access traps!
  - including reads
  - high overhead

Hypervisor maintains guest PT

Virtualisation Mechanics: Optimised Guest PT

- Guest translates PTEs itself when reading from PT
  - supported by Linux PT-access wrappers
- Guest batches PT updates using hypercalls
  - reduced overhead

Para-virtualized guest “knows” it is virtualized

Used by original Xen

User
- ld r0, addr

Guest OS
- add mapping to GPT

Hypervisor
- invalidate mapping in GPT
- invalidate mapping
- flush TLB
- write-protect GPT
- unprotect GPT & mark dirty
- update dirty shadow
- write-protect GPT
- flush TLB

Guest PT
- para-virtualized

HV PT
- physical address

Memory
- data
Virtualization Techniques

• Impure virtualisation methods enable new optimisations
  – avoid traps through ability to control the ISA
  – changed contract between guest and hypervisor
• Example: virtualised guest page table
  – lazy update of virtual state (TLB semantics)
• Example: virtual interrupt-enable bit (in virtual PSR)
  – requires changing guest’s idea of where this bit lives
  – hypervisor knows about VM-local virtual state
    o eg queue virtual interrupt until guest enables in virtual PSR

```
VPSR
[0]
PSR
[0]

mov r1,#VPSR
ldr r0,[r1]
orr r0,r0,#VPSR_ID
sto r0,[r1]
```

Virtualization Mechanics: 3 Device Models

- Emulated
- Split
- Pass-through

Virtualization Mechanics: Emulated Device

- Each device access must be trapped and emulated
  - unmodified native driver
  - high overhead!

Virtualization Mechanics: Split Driver (Xen speak)

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

"Para-virtualized driver"
Virtualization Mechanics: Driver OS (Xen Dom0)

- Leverage Driver-OS native drivers
  - no driver porting
  - must trust complete Driver OS guest!
  - huge trusted computing base (TCB)!

Virtualization Mechanics: Pass-Through Driver

- Unmodified native driver
- Can’t share device between VMs
  - newer NICs support sharing
- Must trust driver (and guest)
  - unless have hardware support (I/O MMU)

Modern Architectures Not T&E Virtualisable

- Examples:
  - x86: many non-virtualizable features
    - e.g. sensitive PUSH of PSW is not privileged
    - segment and interrupt descriptor tables in virtual memory
    - segment description expose privileged level
  - MIPS: mostly ok, but
    - kernel registers k0, k1 (for save/restore state) user-accessible
    - performance issue with virtualising KSEG addresses
  - ARM: mostly ok, but
    - some instructions undefined in user mode (banked registers, CPSR)
    - PC is a GPR, exception return is MOVS to PC, doesn’t trap

- Addressed by virtualization extensions to ISA
  - x86, Itanium since ~2006 (VT-x, VT-i, AMD-V), ARM since ‘12
  - additional processor modes and other features
  - all sensitive ops trap into hypervisor or made innocuous (shadow state)
    - eg guest copy of PSW

x86 Virtualization Extensions (VT-x)

- New processor mode: VT-x root mode
  - orthogonal to protection rings
  - entered on virtualisation trap
ARM Virtualization Extensions (1)

Hyp mode

New privilege level
- Strictly higher than kernel
- Virtualizes or traps all sensitive instructions
- Only available in ARM TrustZone "normal world"

EL0
User mode

EL1
Kernel modes

EL2
Kernel modes

EL3
Monitor mode

"Normal world"  "Secure world"

ARM Virtualization Extensions (2)

Configurable Traps

x86 similar

User mode

Kernel mode

Native syscall

Can configure traps to go directly to guest OS

Big performance boost!

Virtual syscall

Virtual syscall

Trap to guest

ARM Virtualization 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)

ld sp, (r1, kern_stk)

L1 I-Cache

L1 D-Cache

R2

mv CPU_ASID, r1

L2 Cache

L1 I-Cache

L1 D-Cache

L2 Cache

Emulation Support

1) HW decodes instruction
   - No L1 miss
   - No software decode
2) SW emulates instruction
   - Usually straightforward

IR

mv CPU_ASID, r1

ld r1, (r0, ASID)

ld sp, (r1, kern_stk)

R2

R3

No x86 equivalent

No x86 equivalent
ARM Virtualization Extensions (4)

2-stage translation

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

• 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 50% in extreme cases!

ARM Virtualization Extensions (5)

Virtual Interrupts

- ARM has 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 Virtualization 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 same page table

Many ARM SoCs different

• Can do pass-through I/O safely
  – Guest accesses device registers
  – No hypervisor invocation
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 software (hypervisor)
  - Selective save/restore
    - Eg traps w/o world switch

Hybrid Hypervisor OSes

- Idea: turn standard OS into hypervisor
  - ... by running in VT-x root mode
  - eg: KVM ("kernel-based virtual machine")
- Can re-use Linux drivers etc
- **Huge trusted computing base!**
- Often falsely called a Type-2 hypervisor

Variant: **VMware MVP**
- ARM hypervisor
  - pre-HW support
  - re-writes exception vectors in Android kernel to catch virtualization traps in guest

ARM: seL4 vs KVM [Dall&Nieh '14]

- **seL4**
  - Full world switch!
- **KVM**
  - Lowvisor
  - Native Linux apps
  - QEMU
Virtualisation Cost (KVM)

<table>
<thead>
<tr>
<th>Component</th>
<th>ARM LoC</th>
<th>x86 LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM exit+entry</td>
<td>27</td>
<td>821</td>
</tr>
<tr>
<td>Hypercall</td>
<td>5,326</td>
<td>1,638</td>
</tr>
<tr>
<td>I/O Kernel</td>
<td>5,990</td>
<td>3,291</td>
</tr>
<tr>
<td>I/O User</td>
<td>10,119</td>
<td>12,218</td>
</tr>
<tr>
<td>EOI+ACK</td>
<td>427</td>
<td>2,305</td>
</tr>
</tbody>
</table>

Source: [Dall&Nieh, ASPLOS’14]

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!

Hypervisors vs Microkernels

- Both contain all code executing at highest privilege level
  - Although hypervisor may contain user-mode code as well
    o privileged part usually called “hypervisor”
    o user-mode part often called “VMM”
- Both need to abstract hardware resources
  - Hypervisor: abstraction closely models hardware
  - Microkernel: abstraction designed to support wide range of systems
- What must be abstracted?
  - Memory
  - CPU
  - I/O
  - Communication

What’s the difference?

<table>
<thead>
<tr>
<th>Resource</th>
<th>Hypervisor</th>
<th>Microkernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>Virtual MMU (vMMU)</td>
<td>Address space</td>
</tr>
<tr>
<td>CPU</td>
<td>Virtual CPU (vCPU)</td>
<td>Thread or scheduler activation</td>
</tr>
</tbody>
</table>
| I/O          | • Simplified virtual device  
  • Driver in hypervisor  
  • Virtual I/O (vIRQ) | • I/O interface to user-mode driver  
  • Interrupt IPC |
| Communication | Virtual NIC, with driver and network stack | High-performance message-passing IPC |

Just page tables in disguise
Just kernel-scheduled activities
Real Difference?
Modelled on HW, Re-uses SW
Minimal overhead, Custom API

Difference to traditional terminology!
Closer Look at I/O and Communication

- Communication is critical for I/O
  - Microkernel IPC is highly optimised
  - Hypervisor inter-VM communication is frequently a bottleneck

Hypervisors vs Microkernels: Drawbacks

**Hypervisors:**
- Communication is Achilles heel
  - more important than expected
    - critical for I/O
  - plenty improvement attempts in Xen

**Microkernels:**
- Not ideal for virtualization
  - API not very effective
    - L4 virtualization performance close to hypervisor
    - effort much higher
  - Needed for legacy support
  - No issue with H/W support?

- Most hypervisors have big TCBs
  - infeasible to achieve high assurance of security/safety
  - in contrast, microkernel implementations can be proved correct

**Trad. L4 requires kernel-scheduled threads for more than exploiting parallelism**
- Kernel imposes policy
- Alternatives exist
  - K42 scheduler activations
  - new sel4 MCS model