Virtual Machine (VM)

“A VM is an efficient, isolated duplicate of a real machine”

• Duplicated: VM should behave identically to the real machine
  – Programs cannot distinguish between real or virtual hardware
  – Except for:
    • Fewer resources (and 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 instructions are executed directly by real hardware

**Hypervisor** aka virtual-machine monitor: Software implementing the VM

Types of Virtualization

<table>
<thead>
<tr>
<th>Type</th>
<th>Operating System</th>
<th>Process VM</th>
<th>OS-level VM</th>
<th>Process VM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform VM</td>
<td>Operating System</td>
<td>Hypervisor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS</td>
<td>Processor</td>
<td>Processor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hypervisor</td>
<td>Operating System</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Processor</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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
- checkpointing
- debugging
- Different concurrent OSes
  - e.g. Linux + Windows
- Total mediation
- Would be mostly unnecessary
  - ... if OSes were doing their job!

Hypervisor

- 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

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

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
    - … by simply running guest binary on hypervisor
    - 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
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
- Two approaches
  - binary translation: change binary
  - para-virtualisation: change ISA

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” [Oran Krieger, then IBM, later VMware] 😊

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!
- Some OS operations involve frequent traps
  - STI/CLI for mutual exclusion
  - frequent page table updates during fork()
Virtualization Techniques

• Impure virtualisation methods enable new optimisations
  – due to ability to control the ISA
• Example: Maintain some virtual machine state inside the VM
  – eg interrupt-enable bit (in virtual PSR)
  – requires changing guest’s idea of where this bit lives
  – hypervisor knows about VM-local virtual state
    • eg queue virtual interrupt until guest enables in virtual PSR

Virtualization and Address Translation

Two levels of address translation!

Virtualization Mechanics: Shadow Page Table

Virtualization Techniques

• Example: Lazy update of virtual machine state
  – virtual state is kept inside hypervisor
  – shadowed by copy inside VM
  – allow temporary inconsistency between primary and shadow
  – synchronise on next forced hypervisor invocation
    • actual trap
    • explicitly hypervisor call when physical state must be updated
• Example: guest enables FPU, handled lazily by hypervisor:
  • guest sets virtual FPU-enable bit
  • hypervisor synchronises on virtual kernel exit
• More examples later
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

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

Virtualisation Semantics: Lazy Shadow Update

User
- invalidate mapping in GPT
- invalidate mapping...
- flush TLB
- continue

Guest OS
- write-protect GPT
- unprotect GPT & mark dirty
- update dirty shadow
- write-protect GPT

Hypervisor
- add mapping to GPT
- add mappings...
- update dirty shadow
- write-protect GPT
- access new page
- continue

Virtualization 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

User
- ld r0, adr
- Guest virtual address

Guest OS
- Guest PT
- Hypervisor maintains guest PT

Hypervisor
- HV PT
- Physical address
- data
**Virtualization 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

**Virtualization Mechanics: 3 Device Models**

- **Emulated**
  - Device register accesses

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

- **Pass-through**
  - Used by original Xen

**Virtualization Mechanics: Emulated Device**

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

**Virtualization Mechanics: Split Driver (Xen speak)**

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

- Leverage Driver-OS native drivers
  - no driver porting
  - must trust complete Driver OS guest!

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 in MOV to PC, doesn’t trap
- Addressed by virtualization extensions to ISA
  - x86, Itanium since ~2006 (VT-x, VT-i), 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

Virtualization Mechanics: Pass-Through Driver

- Unmodified native driver
- Must trust driver (and guest)
  - unless have hardware support (I/O MMU)

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 “non-secure” mode

ARM Virtualization Extensions (2)

**Configurable Traps**

- x86 similar
  - Can configure traps to go directly to guest OS

ARM Virtualization Extensions (3)

**Emulation**

1. Load faulting instruction
   - Compulsory L1-D miss!
2. Decode instruction
   - Complex logic
3. Emulate instruction
   - Usually straightforward

Emulation Support

- HW decodes instruction
  - No L1 miss
  - No software decode
- SW emulates instruction
  - Usually straightforward
ARM Virtualization Extensions (4)

2-stage translation
- Hardware PT walker traverses both PTs
- Loads combined (guest-virtual to physical) mapping into TLB

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

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

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

<table>
<thead>
<tr>
<th>Hypervisor</th>
<th>ISA</th>
<th>Type</th>
<th>Kernel (kLOC)</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>OKL4</td>
<td>ARMv7</td>
<td>para-virtualization</td>
<td>9.8</td>
<td>0</td>
</tr>
<tr>
<td>Prototype</td>
<td>ARMv7</td>
<td>pure virtualization</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>Nova</td>
<td>x86</td>
<td>pure virtualization</td>
<td>9</td>
<td>27</td>
</tr>
</tbody>
</table>

- Size (& complexity) reduced about 40% wrt to para-virtualization
- Much smaller than x86 pure-virtualization hypervisor
  - Mostly due to greatly reduced need for instruction emulation

Overheads (Estimated)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Pure virtualization</th>
<th>Para-virtualization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Instruct</td>
<td>Cycles (est)</td>
</tr>
<tr>
<td>Guest system call</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Hypervisor entry + exit</td>
<td>120</td>
<td>650</td>
</tr>
<tr>
<td>IRQ entry + exit</td>
<td>270</td>
<td>900</td>
</tr>
<tr>
<td>Page fault</td>
<td>356</td>
<td>1500</td>
</tr>
<tr>
<td>Device emul.</td>
<td>249</td>
<td>1040</td>
</tr>
<tr>
<td>Device emul. (accel.)</td>
<td>176</td>
<td>740</td>
</tr>
<tr>
<td>World switch</td>
<td>2824</td>
<td>7555</td>
</tr>
</tbody>
</table>

- No overhead on regular (virtual) syscall – unlike para-virtualization
- Invoking hypervisor 500–1200 cycles (0.6–1.5 µs) more than para
- World switch in ~10 µs compared to 0.25 µs for para
  ⇒ Trade-offs differ

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

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
    • privileged part usually called “hypervisor”
    • 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 IRQ (vIRQ)
  • Interrupt IPC
| I/O          | Virtual NIC, with
  driver and network stack
  • IPC interface to user-mode driver
  • Interrupt IPC |
| Communication| High-performance message-passing IPC |

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

Microkernels:

- Not ideal for virtualization
  • API not very effective
    - L4 virtualization performance close to hypervisor
    - effort much higher
  - Virtualization needed for legacy
- L4 model uses kernel-scheduled threads for more than exploiting parallelism
  • Kernel imposes policy
  • Alternatives exist, eg. K42 uses scheduler activations