Virtual Machine (VM)

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

- Duplicate: VM should behave identically to the real machine
  - Programs cannot distinguish between execution on real or virtual hardware
  - Except for:
    - Fewer resources available (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 instruction are executed directly by real hardware

Hypervisor aka virtual-machine monitor: Software implementing the VM

Types of Virtualization

<table>
<thead>
<tr>
<th>Type-1</th>
<th>Type-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hosted</td>
<td>Native</td>
</tr>
</tbody>
</table>

Programming Language

- Java
  - Java API
  - Java Program
  - Java VM

OS API

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

Platform VM or System VM

- Type-1
- Type-2
- OS-level VM
- Process VM
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 commonplace
  - 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
    - Migration
    - Replication
    - 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 intercepts/emulates them
  - Can have extra instructions for hypercalls

Embedded systems: integration of heterogeneous environments
- RTOS for critical real-time functionality
- Standard OS for GUIs, networking etc
- Alternative to physical separation
  - Low-overhead communication
  - Cost reduction
Native vs. Hosted VMM

Native/Classic/Bare-metal/Type-I

Hosted/Type-II

> Hosted VMM can run besides native apps
  - Sandbox untrusted apps
  - Convenient for running alternative OS on desktop
> Less efficient
  - Twice number of mode switches
  - Twice number of context switches
  - Host not optimised for exception forwarding

Native/Classic/Bare-metal/Type-I

Guest OS

App

Hypervisor

Hardware

Hosted/Type-II

Guest OS

App

Hypervisor

Hardware

Virtualization Mechanics: Instruction Emulation

> Traditional "trap and emulate" approach:
  - guest attempts to access physical resource
  - hardware raises exception (trap), invoking hypervisor's exception handler
  - hypervisor emulates result, based on access to virtual resource
> Most instructions do not trap
  - makes efficient virtualization possible
  - requires that VM ISA is (almost) same as physical processor ISA

Guest

Exception

Hypervisor

ld r0, curr_thrd
ld r1, (r0, ASID)
mv CPU_ASID, r1
ld sp, (r1, kern_stk)
lda r1, vm_reg_ctxt
ld r2, (r1, ofs_r0)
sto r2, (r1, ofs_ASID)

ld r0, curr_thrd
ld r1, (r0, ASID)
mv CPU_ASID, r1
ld sp, (r1, kern_stk)
lda r1, vm_reg_ctxt
ld r2, (r1, ofs_r0)
sto r2, (r1, ofs_ASID)

Trap-and-Emulate Requirements

Definitions:

> Privileged instruction: executes in privileged mode, traps in user mode
  - Note: trap is required, NO-OP is insufficient!
> Privileged state: determines resource allocation
  - Includes privilege mode, addressing context, exception vectors, …
> Sensitive instruction: control-sensitive or behaviour-sensitive
  - control sensitive: changes privileged state
  - behaviour sensitive: exposes privileged state
    - includes instructions which are NO-OPs in user but not privileged mode
> Innocuous instruction: not sensitive

Note:

- Some instructions are inherently sensitive
  - e.g. TLB load
- Others are sensitive in some context
  - e.g. store to page table

Trap-and-Emulate Architectural Requirements

Trap-and-emulate virtualizable if all sensitive instructions are privileged

> Can then achieve accurate, efficient guest execution
  - by simply running guest binary on hypervisor
> VMM controls resources
> Virtualized execution is indistinguishable from native, except:
  - Resources more limited (running on smaller machine)
  - Timing is different (if there is an observable time source)
> Recursively virtualizable machine:
  - VMM can be built without any timing dependence

Guest

Exception

Hypervisor

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

- Used for two reasons:
  - Architecture not trap-and-emulate virtualizable
  - Reduce virtualization overheads
- Change the guest OS, replacing sensitive instructions
  - by trapping code (hypercalls)
  - by in-line emulation code
- Two standard approaches:
  - Binary translation: modifies binary
  - Para-virtualization: changes ISA

Binary Translation

- Locate sensitive instructions in guest binary and replace on-the-fly by emulation code or hypercall
  - pioneered by VMware
  - can also detect combinations of sensitive instructions and replace by single emulation
  - doesn't require source, uses unmodified native binary
    - in this respect appears like pure virtualization!
  - very tricky to get right (especially on x86!)
    - “heroic effort” [Orran Krieger, then IBM later VMware ;-) ]
  - needs to make some assumptions on sane behaviour of guest

Para-Virtualization

- New name, old technique
  - Mach Unix server [Golub et al. 90], L4Linux [Härtig et al. 97], Disco [Bugnion et al. 97]
  - Name coined by Denail [Whitaker et al. 02], popularised by Xen [Barham et al. 03]
- Idea: manually port the guest OS to modified ISA
  - Augment by explicit hypervisor calls (hypercalls)
    - Use more high-level API to reduce the number of traps
    - Remove un-virtualizable instructions
    - Remove “messy” ISA features which complicate virtualization
  - Generally out-performs pure virtualization and binary-rewriting
- Drawbacks:
  - Significant engineering effort
  - Needs to be repeated for each guest-ISA-hypervisor combination
  - Para-virtualized guest needs to be kept in sync with native guest
  - Requires source

Virtualization Overheads

- VMM needs to maintain virtualized privileged machine state
  - processor status
  - addressing context
  - device state...
- VMM needs to emulate privileged instructions
  - translate between virtual and real privileged state
    - e.g. guest ↔ real page tables
- Virtualization traps are be expensive on modern hardware
  - can be 100s of cycles (1150 cycles round-trip on latest Intel x86 processors)
- Some OS operations involve frequent traps
  - STI/CLI for mutual exclusion
  - frequent page table updates during fork()...
  - MIPS KSEG address used for physical addressing in kernel
Virtualization Techniques

- Impure virtualization methods enable new optimisations
  - due to the ability to control the ISA
  - E.g. maintain some virtual machine state inside VMM:
    - e.g. interrupt-enable bit (in virtual PSR)
    - requires changing guest’s idea of where this bit lives
    - hypervisor knows about VMM-local virtual state and can act accordingly
      - e.g. queue virtual interrupt until guest enables in virtual PSR

- E.g. lazy update of virtual machine state
  - virtual state is kept inside hypervisor
  - keep copy of virtual state inside VM
  - keep copy of virtual state inside VM
  - allow temporary inconsistency between local copy and real VM state
  - synchronise state on next forced hypervisor invocation
    - actual trap
    - explicit hypercall when physical state must be updated
      - Example: guest enables FPU
        - no need to invoke hypervisor at this point
        - hypervisor syncs state on virtual kernel exit

Virtualization and Address Translation

Two levels of address translation!

Guest Physical Memory

Must implement with single MMU translation!

Virtualization Mechanics: Shadow Page Table
**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

**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

**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
- 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

Virtualization Mechanics: Driver OS (Xen Dom0)

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

Virtualization Mechanics: Pass-Through Driver

- Unmodified native driver
- Must trust driver (and guest)
  - unless have hardware support (IO MMU)
Non-Virtualizable Architectures

- x86: lots of 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
- Itanium: mostly virtualizable, but
  - interrupt vector table in virtual memory
  - THASH instruction exposes hardware page tables address
- MIPS: mostly virtualizable, but
  - kernel registers k0, k1 (needed to save/restore state) user-accessible
  - performance issue with virtualizing KSEG addresses
- ARM: mostly virtualizable, but
  - some instructions undefined in user mode (banked registers, CPSR)
  - PC is a GPR, exception return in MOVVS to PC, doesn’t trap
- Most others have problems too
- Modern trend are virtualization extensions to ISA
  - x86, Itanium since ~2006 (VT-x, VT-i)
- Case study: ARM
  - announced ’10, samples ’11, products ’12

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
- Note: different from x86
  - VT-x “root” mode is orthogonal to x86 protection rings

ARM Virtualization Extensions (2)

**Configurable Traps**

- User mode
- Kernel mode
- Hyp mode

- Native syscall
- 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

1d 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 Virtualization Extensions (3)

**Emulation Support**

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

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

```
mv CPU ASID, r1
ld sp, (r1, kern_stk)
```

L1 I-Cache

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

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

L2 Cache

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

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

---

ARM Virtualization Extensions (4)

2-stage translation

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

```
ld r0, adr
```

```
1st PT ptr (Hardware)
```

```
Guest virtual address
```

```
Guest page table
```

```
Hypervisor's guest memory map
```

```
2nd PT ptr (Hardware)
```

```
Guest physical address
```

```
Physical address
```

---

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 entries for interrupt virtualization

```
CPU Interface
```

```
Virt. CPU Interf
```

```
Hypervisor
```

```
Distributor
```

```
Guest
```

```
Distributor
```

```
CPU Interface
```

```
Virt. CPU Interf
```

```
Hypervisor
```

```
Guest
```

```
Distributor
```

```
CPU Interface
```

```
Virt. CPU Interf
```

```
Hypervisor
```

```
Guest
```

```
Distributor
```
Hypervisor Size

<table>
<thead>
<tr>
<th>Hypervisor</th>
<th>ISA</th>
<th>Type</th>
<th>Kernel</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>OKL4</td>
<td>ARMv7</td>
<td>para-virtualization</td>
<td>9.8 kLOC</td>
<td>0</td>
</tr>
<tr>
<td>Prototype</td>
<td>ARMv7</td>
<td>pure virtualization</td>
<td>6 kLOC</td>
<td>0</td>
</tr>
<tr>
<td>Nova</td>
<td>x86</td>
<td>pure virtualization</td>
<td>9 kLOC</td>
<td>27 kLOC</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-virtualiz.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Instruct</td>
<td>Cycles (est)</td>
</tr>
<tr>
<td>Guest system call</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Hypervisor entry + exit</td>
<td></td>
<td>120</td>
</tr>
<tr>
<td>IRQ entry + exit</td>
<td></td>
<td>270</td>
</tr>
<tr>
<td>Page fault</td>
<td></td>
<td>356</td>
</tr>
<tr>
<td>Device emul.</td>
<td></td>
<td>249</td>
</tr>
<tr>
<td>Device emul. (accel.)</td>
<td></td>
<td>176</td>
</tr>
<tr>
<td>World switch</td>
<td></td>
<td>2824</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

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)  | • IPC interface to user-mode driver
  • Interrupt IPC |
| Communication | Virtual NIC, with driver and network stack | High-performance message-passing IPC |

Difference to traditional terminology!

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

- Fundamentally, both provide similar abstractions
- Optimised for different use cases
  - Hypervisor designed for virtual machines
    - API is hardware-like to ease guest ports
  - Microkernel designed for multi-server systems
    - Seems to provide more OS-like abstractions

**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

More on this later!