Virtual Machines

A virtual machine (VM) is an efficient, isolated duplicate of a real machine (PC7A).

**Duplicate:** VM should behave identically to the real machine
- Programs can run dialogically between executing on a real or virtual hardware
- Except for:
  - Some instructions executed (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 a speed close to that of hardware
- Requires that most instructions are executed directly by real hardware

Types of Virtual Machines

- Contemporary use of the term VM is more general
- Call virtual machines even if there is no correspondence to an existing real machine
  - Eg. new virtual machine
  - Can be seen as virtualizing at the API level
  - Also called process-VM [W95]

We only concern ourselves with virtualizing at the ISA level
- ISA = Instruction set architecture (hardware-software interface)
- Also called native VM
- Will later see subclasses of this.

Virtual Machines, Simulators and Emulators

**Simulator**
- Provides a reasonably accurate software model of a machine
  - Can run on any hardware
  - But typically slow (order of 1000 slowdown)

**Emulator**
- Provides a functional model of hardware (and possibly SW)
  - High fulfillment
  - (Reasonably fast (order of 10 slowdown)

**Virtual machine**
- Matches a machine exactly and efficiently
  - (Optimal solution)

Needs to be run on the physical machine if simulators (more or less)

Boundaries are becoming soft, eg some simulators approaching VM performance

Virtual Machine Monitor (VMM), aka Hypervisor

- Program that runs on real hardware to implement the virtual machine
- Controls resources
  - Partitions hardware
  - Schedules guests
  - Mediates access to shared resources
  - Performs virtual switch

**Implications:**
- Hypercall executes in privileged mode
- Guest software executes in unprivileged mode
- Redgeded instructions in guest cause a trap into hypervisor
- Hypercall instructions are like real CPU instructions
  - Can have extra instructions for Hypervisor
  - Execution of hypervisor APIs that are not machine instructions
Why Virtual Machines?

- Historically used for easier sharing of expensive mainframes
  - Run several (even different) OSes on one machine
  - Each on a subset of physical resources
  - Cool but single-user single-tasked OS in time-sharing system
  - Would work well between VMs
- Gone out of fashion in 80's
  - Mainframe become too cheap to worry...

Native vs. Hosted VMM

Native/Classic/Bare-metal Type-1
- Can run unmodified apps
- Single OS
- Less efficient

Hosted Type-2
- Hosted VM can run neben native apps
- Sandbox untrusted apps
- Run several OSs
- Less efficient
  - Guest privileged instruction traps into OS, forwarded to hypervisor
  - Return to guest requires native OS system call

Requirements for Virtualisation

Definitions
- Privileged instruction: executes in privileged mode, traps in user mode
- Some traps required; 64-bit OS is insufficient
- Privileged state: determines resource allocation
- Includes privilege mode, addressing control, exception vectors, ...

Sensitive instruction: control-sensitive or behaviour-sensitive
- Behaviour sensitive: if privileged state
- Includes instructions which are in privileged mode
- Incorrect instruction: not sensitive

VMM Types

- Classic: as above
- Hosted: e.g. VMware GSX Server
- Whole-system: virtual hardware and operating system
  - Needs no re-creation
  - E.g. virtual PC for Institutions
- Physically partitioned: allocate actual processors to each VM
- Logically partitioned: time-share processors between VMs
- Co-designed: hardware specifically designed for VM
  - E.g. Transmeta Crusoe, IBM boxes
- Frankly: no enforcement of partitioning
  - Guest at same privilege level as hypervisor
  - Really above all, "virtualisation"

Requirements for Virtualisation

An architecture is virtualisable if all sensitive instructions are privileged (suitable for pure virtualisation)

- Can then achieve accurate, efficient guest execution
  - Guest sensitive instruction trap and are emulated by VM
  - Guest's privilege instructions are executed directly
  - VM controls resources
Requirements for Virtualisation

- Characteristics of pure virtualisation:
  - Execution is indistinguishable from native, except...
  - Resources are more limited...
  - Effectively running on virtual machine...
  - Timing is different...
  - Inevitable due to the observable real-time source...
  - Hard to communicate with external world (networks)...
  - In practice hard to completely virtualise time...

- Remotely virtualised machine:
  - If VM can be built without any timing dependence...

Virtualisation Overheads

- VMM needs to maintain virtualised privileged machine state...
  - Processor state...
  - Addressing context...
- VMM needs to simulate privileged instructions...
  - Synchronise virtual and real privileged state...
  - E.g., shadow page tables to simulate hardware...
- Frequent virtualisation traps can be expensive...
  - SVVU for mutual exclusion...
  - Frequent page table updates...
  - MIPS cids address used for physical addressing in kernel...

Unvirtualisable Architectures

- X86: lots of unvirtualisable features...
  - E.g., native PV of PVM is not privileged...
  - Segment and descriptor table in virtual memory...
- Risc: mostly virtualisable, but...
  - Interrupt vector is virtual memory...
- Taint: instruction depends on hardware page table address...
- MIPS: mostly virtualisable, but...
  - Kernel registers (X) (privileged to observation only) user accessible...
- Performance issue with virtualisation (RISC) addresses...
- ARM: mostly virtualisable, but...
  - Some instruction overhead is user mode (new trap: GPP)... PC is a GPR, except return is MOV to PC, doesn't trap...
- Most others have problems too...

Impure Virtualisation

- Used for two reasons:
  - Unvirtualisable architectures...
  - Performance problem of virtualisation...
- Two standard approaches:
  - Pure virtualisation...
  - Binary translation...

Paravirtualisation

- Name: old technique...
  - Used on IBM (z/vm)...
  - Use of hypervisor in VM...
  - Need to change to ISAs or make changes in VM...
- Manually port the guest OS to modified ISA...
  - Augment by explicit hypervisor calls...
  - Idea: to provide more high-level API to reduce the number of traps...
- Remove unvirtualisable instructions...
- Remove “kernel” ISA features which complicate virtualisation...

- Drawbacks:
  - Significant engineering effort...
  - Needs to be repeated for each guest, ISA, hypervisor combination...
  - Complex guest stack is locked in vmpx with native guest...
  - Requires source...

Binary Translation

- Locate unvirtualisable instruction in guest binary and replace on-the-fly by emulation code or hypervisor...
  - Processor by instructions on H-VISA...
  - Can also detect conditional hits and instructions by hardware...
  - Guest does not require source...
  - May redefine its native emulation in user space for efficiency...
  - Very tricky to get right (especially on ARM...
  - Needs to make some assumptions on some behaviour of guest...
Virtualisation Techniques: Memory

- Shadow page tables
  - Guest accesses shadow PT
  - VIDL behaves changes (e.g., 0000 to 0001) and synch with real PT
- Can-overcommit memory (similar to virtual memory paging)
- Note: Fast expose hardware page tables (at least some versions)

- Memory reclamation: Reclaiming (VMware ESX Server)
  - Load exploding pseudonode, driver into guest
  - To extend, build driver imposes physical memory from guest
  - VIDL can then remove host memory
  - Guest determines which pages to release

- Page sharing
  - VIDL device pages with identical content
  - Enables sparse (i.e., white) mapping to single page per shadow PT

- Significant savings when running many identical guest Oses

Virtualisation Techniques: Devices

- Drivers in VMX
  - Mobile portal legacy drivers

- Host drivers
  - For hosted VMDK

- Legacy drivers in separate driver VM
  - E.g., separate Linux “kernel-off” for each device (LINUS)
  - Are privileged “banned” guest

- Drivers in guest
  - Requires virtualizing device registers
  - Very expensive, no re-sharing of devices

- Virtualisation-friendly devices with guest drivers
  - VM-based cachex and bcache (working set)
  - Safe device access by guest if physical memory access is restricted (IO-MMU)

Pre-Virtualisation

- Combines advantages of pure and para-virtualisation
- Multi-stage process
  - Decompress, and sensitive instruction with NOPs and
  - Checkpoint
  - Delay, encoding, and sensitive memory operations
  - E.g., PT, TSS, and cache
  - BIOS, and encoding sensitive memory operations
  - Life estimation (not VMXVM) to cache 1 byte
  - In load: compilation of NIP-padded instructions
  - By initialization

- Features:
  - Signal reduced engineering effort
  - Single binary runs on bare metal as well
  - On hypervisors
  - Requires source (as does normal para-virtualization)
  - Performance may require some para-virtualization

Case study: TrustZone — ARM Virtualisation

ARM virtualisation extensions introduce:

- New processor mode: Security
  - Secure memory (PC, LR)
  - Guest runs in kernel mode
  - Unavailable instructions are not a problem
  - New architectural instruction: SMI
  - Secure monitor mode
  - New processor state: Security
  - Permitting of resources
  - Memory and devices trained Security
  - In secure mode, processor can access to all resources
  - Background mode, processor has access to various resources only
  - Atomic: switched world (scramble or measure)
  - Optional hypervisor switches memory (para-virtualised) guests

Hardware Virtualisation Support

- Intel VT-x/VT-i virtualisation support for x86/Latium [LINU3+]
  - Introduces new processor mode: RDTSC
  - RDTSC instruction on CPU processor
  - Very expensive (like RDTSC on Core processors)
  - VT (Latium) also reduces virtual address space size for harvest

- Similar AMD (Pitfire), PowerPC, ARM (TrustZone)

- Arm is virtualisation of unmodified legacy OSes

Other uses of virtualisation

- Checkpoint & restart
  - Can be used for debugging, including executing backtraces in time
  - nix: fast from last checkpoint, rolled traces, revert trace...
  - Migrate live system images
  - nix: for load balancing and power management in clusters
  - Take your work home — without heating a laptop around

- Multiple OSes
  - Linux and Windows or a Mac
  - Legacy OS version (DP image for old apps that don’t run on Vmdk)

- OS development, obviously
  - Develop on same box you’re working on

- Ship complete OS image with application
  - Avoids some configuration dependencies
  - Also for security (run on trusted OS image)
  - Sounds like Java...
**Why Virtualisation in Embedded Systems?**

- Heterogenous OS environments
- Legacy protection
- License separation
- Security

**Why Virtualisation: Heterogenous Environments**

- Typical use: RTOS and high-level OS on same core
  - Recall of growing ES complexity
  - RTOS environment for RT part
  - Maintain legacy environment
  - High-level OSes not real-time capable
  - High-level OS for applications
  - Well-defined OS API
  - GUI, 3rd-party apps
  - e.g. Linux, WinCE
  - Alternative to multiframe chips
  - Cost reduction for low-end systems

**Why Virtualisation: License Separation**

- Linux is under GPL
  - All code in Linux kernel
    - Linux GPL
  - Includes loads drivers
- Hypervisor encapsulates GPL
  - RT can unload
  - Can simulate additional VMs for other code...
  - Stub driver forwards IO requests

**Why Virtualisation: Security**

- Protect against exploits
- Modern software attacked by UI exploits
  - Compromised application OS can compromise RT side
  - Could have serious consequences
    - e.g. wireless device (jacking)
- Virtualisation protects
  - Separate app and system code into different VMs

**Limitations of Virtualisation**

- Pure hypervisor provides strong partitioning of resources
  - Good for strict isolation
- This is not really what you want in an embedded system
- Subsystem of an embedded system need to cooperate
- Need controlled, high-performance sharing of resources
  - Shared memory for high-speed data communication
  - Shared devices with low-latency access
- Need integrated scheduling across virtual machines
  - High-level OS (see-effort VM) should be lower per than real-time threads
- However, some threads in real-time subsystems are background activities
- Need more than just a hypervisor!
Hypervisors vs Microkernels

- Microkernels have been used as hypervisors for a long time
  - Mach Unix (89), 4.4BSD (94)
- Hypervisors have more visibility than microkernels
  - Both encapsulate subsystems
    - What's the difference?
      - Microkernels are generic
      - Hypervisors are only meant to support VMs running guest OSes

Microkernel as a Hypervisor

- Has all advantages of a pure hypervisor:
  - Provide isolation (where needed)
  - Run arbitrary guest OSes (high-level and RTOSes)
- Supports efficient sharing
  - High-performance IPC mechanism
  - Shared memory regions
- Supports interkernel networking
  - Application OS VM scheduled as a unit (with a single microkernel process)
  - RT threads directly scheduled by microkernel (without interlock protocol)
- Can have same or higher, some at lower, solo RTOS environment

Hypervisor vs. Microkernel

- Xen vs. L4 on Pentium 4 running Linux 2.6.9
  - Decrease shows in guest OS

<table>
<thead>
<tr>
<th>System</th>
<th>Kernel Sample</th>
<th>Native load</th>
<th>Native exec</th>
<th>Guest load</th>
<th>Guest exec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>1201</td>
<td>61.0/2</td>
<td>24.4</td>
<td>155.6</td>
<td>30</td>
</tr>
<tr>
<td>Linux on Site</td>
<td>2019</td>
<td>63.4</td>
<td>24</td>
<td>155.6</td>
<td>27</td>
</tr>
<tr>
<td>Xen</td>
<td>156.6</td>
<td>22.0</td>
<td>32.0</td>
<td>155.7</td>
<td>27</td>
</tr>
</tbody>
</table>

- Xeon base performance is better
- Network performance shows there is optimisation potential

Sharing Devices

- Requires high-performance IPC
- Hypervisor + fast IPC + Microkernel?
**Integrated Scheduling**

**Other Microkernel Advantages: Native Environment**

- Microkernel suitable for a native OS environment
  - Hypervisor only means to support a guest OS
  - Microkernel powerful enough to support native OS environment

- Microkernel minimizes trusted computing base
  - No guest OS required for simple applications
  - E.g., trusted hypervisor
  - In user protected domain
  - Two TCB instances:
    - dom0 guest
    - complete Linux

**Other Microkernel Advantages: Hybrid Systems**

- Co-existence of monolithic and componentized subsystems
  - Legacy support
  - Successive migration
  - Component by time...

- Other Microkernel Advantages: Native Environment

  - Untrusted
  - Trusted

  - dom0
  - complete Linux