# Virtual Machines

Gernot Heiser

COMP9242 2005/S2 Week 6



## Virtual Machine

*A virtual machine (VM) is an* efficient, isolated duplicate *of a real machine* [?].

**Duplicate:** VM should behave identical to the real machine (programs cannot distinguish between execution on real or virtual hardware), except for:

- ➜ less 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 a speed close to that of real hardware

- ➜ requires that most instructions are executed directly by real hardware



## Virtual Machines, Simulators and Emulators

**Simulator** provides an *functionally accurate* software model of a machine

- ☑ may run on any hardware
- ☒ is typically slow (order of 1000 slowdown)

**Emulator** provides a *behavioural* model of a machine (and possibly its software)

- ☒ not fully accurate
- ☑ reasonably fast (order of 10 slowdown)

**Virtual machine** models a machine exactly and efficiently

- ☑ minimal slowdown
- ☒ needs to be run on the physical machine it virtualises



## 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
  - ➜ e.g. *Java virtual machine*
  - ➜ can be viewed as virtualising at the ABI level
  - ➜ also called *process VM*
- We only concern ourselves with virtualising at the ISA level
  - ➜ ISA = *instruction-set architecture* (hardware-software interface)
  - ➜ also called *system VM* [?]
  - ➜ will later see subclasses of this

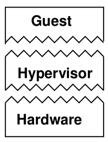


## Why virtual machines?

- Historically used for easier sharing of expensive mainframes
- Gone out of fashion in 80's
- Renaissance in recent years for improved isolation [?]
  - improved QoS and security
  - uniform view of hardware
  - complete encapsulation
    - ➜ for replication, migration, checkpoint/restart, debugging
  - total mediation
- Isn't that exactly the job description of an OS????
  - ➜ This really means that mainstream OSes suck!



## Virtual Machine Monitor (VMM) aka Hypervisor

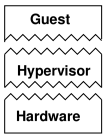



- Program that runs on real hardware to implement the virtual machine
- Controls resources and schedules guests
- Implications:
  - ➜ hypervisor executes in privileged mode
  - ➜ guest software executes in unprivileged mode
  - ➜ *privileged instructions* in guest cause a trap into the hypervisor
  - ➜ hypervisor interprets/emulates them

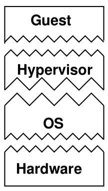


## Native vs. Hosted VMM

**Native/classic/bare-metal**




**Hosted**




- Hosted VMM can run besides native apps
  - sandbox untrusted apps
  - run second OS
  - less efficient:
    - ☒ guest privileged instruction traps into OS, forwarded to hypervisor
    - ☒ return to guest requires a native SO system call



## VMM Types

**Classic** as above

**Hosted** e.g. VMware GSX Server

**Whole-system** virtual hardware and operating system

- ➜ really an emulation
- ➜ e.g. Virtual PC (for Macintosh)

**Physically partitioned** allocate actual processors to each VM

**Logically partitioned** time-share processors between VMs

**Co-designed** hardware specifically designed for VMM

- ➜ e.g. Transmeta Crusoe, IBM i-Series

## Requirements for Virtualisation

### Definitions

**Privileged instruction** executes in privileged mode, traps in user mode

- ➜ Note: trap is required, NOOP 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 NOOPs in user but not privileged mode

**Innocuous instruction** not sensitive



## Requirements for Virtualisation

An architecture is *virtualizable* if all sensitive instructions are privileged (called *pure virtualisation*)

- Can then achieve accurate, efficient guest execution
  - ➜ guest's sensitive instructions trap and are emulated by VMM
  - ➜ guest's innocuous instructions are executed directly
  - ➜ VMM controls resources
- Virtualised execution is indistinguishable from native, except:
  - ➜ resources more limited (running on *smaller* machine)
  - ➜ timing is different (if there is an observable time source)
- Architecture is *recursively virtualizable* if VMM without timing dependency can be built



## Virtualisation Overheads

- VMM needs to maintain virtualised privileged machine state
  - ➜ processor status
  - ➜ addressing context
- VMM needs to simulate privileged instructions
  - ➜ synchronise virtual and real privileged state as appropriate
  - ➜ e.g. *shadow page tables* to virtualize hardware
- Frequent virtualisation traps can be expensive
  - ➜ STI/CLI for mutual exclusion
  - ➜ frequent page table updates
  - ➜ MIPS `KSEG` addresses used for physical addressing in kernel



## Unvirtualisable Architectures

- x86: lots of unvirtualisable features
  - ➜ e.g. sensitive PUSH of PSW is not privileged
  - ➜ segment and interrupt descriptor tables in virtual memory
  - ➜ segment descriptors expose privileged level
- Itanium: mostly virtualizable, but
  - ➜ interrupt vector table in virtual memory
  - ➜ THASH instruction exposes hardware page table address
- MIPS: mostly virtualizable, but
  - ➜ kernel registers `k0`, `k1` (needed to save/restore state) user-accessible
  - ➜ performance issue with virtualising `KSEG` addresses
- Most others have problems too

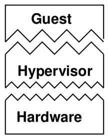


## Impure Virtualisation

- Used for two reasons:
  - ➜ unvirtualisable architectures
  - ➜ performance problems of virtualisation
- Two standard approaches:
  - ① paravirtualisation
  - ② binary translation



## Paravirtualisation




- New name, old technique
  - ➜ Used in Mach Unix server, L4Linux [?], Disco [?]
  - ➜ Name coined by Denali [?], popularised by Xen [?]
- Manually port the guest OS to modified ISA
  - ➜ augmented by explicit hypervisor calls (*hypercalls*)
  - ☑ idea is to provide more high-level API to reduce the number of traps
  - ☑ remove unvirtualisable instructions
  - ☑ remove "messy" ISA features which complicate virtualisation
- Drawbacks:
  - ☒ significant engineering effort
  - ☒ needs to be repeated for each guest, ISA, hypervisor combination
  - ☒ paravirtualised guest needs to be kept in sync with native guest
  - ☒ requires source



## Binary Translation

- Locate unvirtualisable 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
  - ☑ may (safely) do some emulation in user space for efficiency
  - ☒ very tricky to get right (especially on x86!)
  - ☒ needs to make some assumptions on sane behaviour of guest



## Virtualisation Techniques: Memory

- Shadow page tables
  - ➜ Guest accesses shadow PT
  - ➜ VMM detects changes (e.g. making them R/O) and syncs with real PT
  - ➜ can over-commit memory (similar to virtual-memory paging)
  - ➜ Note: Xen exposes hardware page tables
- Memory reclamation: *Ballooning* (VMware ESX Server)
  - ➜ load cooperating pseudo-device driver into guest
  - ➜ to reclaim, balloon driver requests physical memory from guest
  - ➜ VMM can then reuse that memory
  - ➜ guest determines which pages to release
- Page sharing
  - ➜ VMM detects pages with identical content
  - ➜ establishes (copy-on-write) mappings to single page via shadow PT
  - ➜ significant savings when running many identical guest OSes

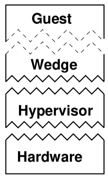
## Virtualisation Techniques: Devices

- Drivers in VMM
  - ➜ maybe ported legacy drivers
- Host drivers
  - ➜ for hosted VMMs
- Legacy driver in separate driver VM
  - ➜ e.g. separate Linux guest for each device [**?**]
- Virtualisation-friendly devices
  - ➜ IBM channel architecture
  - ➜ can safely be accessed by guest drivers if physical memory access is restricted (I/O-MMU)



## Previrtualisation

- Combines advantages of pure and paravirtualisation
- Multi-stage process
  1. During build, pad sensitive instructions with NOPs and keep record
  2. During profiling run, trap sensitive memory operations (e.g. PT accesses) and record
  3. Redo build, also padding sensitive memory operations
  4. Link emulation library (*wedge*) with guest
  5. At load time, replace NOP-padded instructions by emulation code




- Features:
  - ☑ significantly reduced engineering effort
  - ☑ single binary runs on bare metal as well as *all* hypervisors
  - ☒ requires source
  - ☒ performance may require some paravirtualisation
  - ➜ See `http://l4ka.org/projects/virtualization/afterburn/`



## Hardware Support

- Intel VT-x/VT-i: virtualisation support for x86/Itanium [**?**]
  - – introduces new processor mode: *root mode* for hypervisor
  - – if enabled, all sensitive instructions in non-root mode trap to root mode
  - – VT-i (Itanium) also reduces virtual address-space size for non-root
- ARM Trustzone
  - ➜ similar super-privileged mode
  - ➜ also memory and device segregation
- Aim is virtualisation of unmodified legacy OSes



## References