Virtual Machines

COMP9242
2007/S2 Week 5
UNSW
Overview

• Introduction: What are virtual machines
• Why virtualisation?
• Virtualisation approaches
• Hardware support for virtualisation
• Why virtualisation in embedded systems?
• Hypervisors vs microkernels
A virtual machine (VM) is an efficient, isolated duplicate of a real machine [PG74]

**Duplicate:** VM should behave identically 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 hardware
- Requires that most instructions are executed directly by real hardware
Virtual Machines, Simulators and Emulators

**Simulator**

➔ Provides a *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 hardware (and possibly S/W)
☒ 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 (more or less)

Boundaries are becoming soft, eg some simulators approaching VM performance
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 [SN05]
• 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
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 (devices, console)
  → Performs *world switch*

• 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 *hypermrefs*
    • invocation 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 same machine
  ➔ Each on a subset of physical resources
  ➔ Can run *single-user single-tasked OS* in time-sharing system
  ➔ “World switch” between VM

• Gone out of fashion in 80’s
  ➔ Hardware became too cheap to worry...
Why Virtual Machines?

- Renaissance in recent years for improved isolation [RG05]
- Server/desktop virtual machines
  - Improved QoS and security
  - Uniform view of hardware
  - Complete encapsulation (replication, migration, checkpointing, debugging)
  - Different concurrent OSes
    - eg Linux and Windows
  - Total mediation
- Isn't that the job of the OS?
- Do mainstream OSes suck beyond redemption?
Native vs. Hosted VMM

- 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 OS 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

**Pseudo:** no enforcement of partitioning
  - Guests at same privilege level as hypervisor
  - Really abuse of term “virtualisation”
Requirements for Virtualisation

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
Requirements for Virtualisation

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

- Can then achieve accurate, efficient guest execution
  - Guest’s sensitive instruction trap and are emulated by VMM
  - Guest’s innocuous instruction are executed directly
  - VMM controls resources

```
Guest  Exception  VMM
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)
```
Requirements for Virtualisation

• Characteristic of pure virtualization is
  – Execution is indistinguishable from native, except:
  – Resources are more limited
    • effectively running on \textit{smaller} machine
  – Timing is different
    • noticeable only if there is an observable real time source
      – real-time clock
      – devices communicating with external world (network)
    • in practice hard to completely virtualize time

• \textbf{Recursively virtualizable} machine:
  – If VMM can be built without any timing dependence
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 address used for physical addressing in kernel
Unvirtualisable Architectures

• X86: lots of unvirtualizable features
  ➔ E.g. sensitive PUSH of PSW is not privileged
  ➔ Segment and interrupt descriptor tables in virtual memory
  ➔ Segment description expose privilege 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 virtualising KSEG addresses

• ARM: mostly virtualizable, but
  ➔ Some instructions undefined in user mode (banked regs, CPSR)
  ➔ PC is a GPR, exception return is MOVPS to PC, doesn’t trap

• Most others have problems too
Impure Virtualisation

- Used for two reasons:
  - Unvirtualisable architectures
  - Performance problems of virtualisation

- Two standard approaches:
  1. para-virtualisation
  2. binary translation

```c
ld r0, curr_thrd
ld r1,(r0,ASID)
mv CPU_ASID, r1
ld sp,(r1,kern_stk)
```

```c
ld r0, curr_thrd
ld r1,(r0,ASID)
trap
ld sp,(r1,kern_stk)
```

```c
ld r0, curr_thrd
ld r1,(r0,ASID)
jmp fixups_15
ld sp,(r1,kern_stk)
```
Paravirtualisation

• New name, old technique
  ➔ Used in Mach Unix server [GDFR90], L4Linux [HHL+97], Disco [BDGR97]
  ➔ Name coined by Denali project [WSG02], popularised by Xen [DBF+03]

• Manually port the guest OS to modified ISA
  ➔ Augment 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 instruction in guest binary and replace on-the-fly by emulation code or hypercall
  - Pioneered by VMware on x86 [RG05]
  - 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 (at least some versions do)

• 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-white) 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 drivers in separate driver VM
  ➔ E.g. separate Linux “driver OS” for each device (LUSG04)
  ➔ Xen privileged “domain 0” gest

• Drivers in guest
  ➔ Requires virtualizing device registers
  ➔ Very expensive, no sharing of devices

• Virtualisation-friendly devices with guest drivers
  ➔ IBM channel architecture (mainframes)
  ➔ Safe device access by guest if physical memory access is restricted (I/O-MMU)
Pre-Virtualisation

• Combines advantages of pure and para-virtualisation

• Multi-stage process

① During built, pad sensitive instruction with NOPs and keep record
② During profiling run, trap sensitive memory operations (e.g. PT accesses) and record
③ Redo built, also padding sensitive memory operations
④ Link emulation lib (in-place VMM or “wedge”) to 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 (as does normal para-virtualisation)
☒ Performance may require some para-virtualisation

See http://l4ka.org/projects/virtualization/afterburn/ [LUC+05]
Hardware Virtualisation Support

• Intel VT-x/VT-i: virtualisation support for x86/Itanium [UNR+05]
  ➔ Introduces new processor mode: *root mode* for hypervisor
  ➔ If enabled, all sensitive instructions in non-root mode trap to root mode
    • very expensive traps (700+ cycles on Core processors)
  ➔ VT-i (Itanium) also reduces virtual address-space size for non-root

• Similar AMD (Pacifica), PowerPC, ARM (TrustZone)

• Aim is virtualisation of unmodified legacy OSes
Case study: TrustZone — ARM Virtualisation Extensions

ARM virtualisation extensions introduce:

- New processor mode: monitor
  - Banked registers (PC, LR)
  - Guest runs in kernel mode
  - Unvirtualisable instructions are no problem
- New privileged instruction: SMI
  - Enters monitor mode
- New processor state: secure
- Partitioning of resources
  - Memory and devices marked secure or insecure
- In secure mode, processor has access to all resources
- In insecure mode, processor has access to insecure resources only
- Monitor switches world (secure - insecure)
- Optional hypervisor switches insecure (para-virtualised) guests
Other uses of virtualisation

• Checkpoint & restart
  – Can be used for debugging, including executing backwards in time
    • re-run from last checkpoint, collect traces, revert trace…

• Migrate live system images
  – nice for load balancing and power management in clusters
  – take your work home — without hauling a laptop around

• Multiple OSes
  – Linux and Windows on a Mac
  – Legacy OS version (XP image for old apps that don't run on Vista)

• 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
  ➔ Result 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 multicore chips
  ➔ Cost reduction for low-end systems
Why Virtualisation: License Separation

• Linux is under GPL
  → All code in Linux kernel becomes GPLed
  → Includes loaded drivers

• Hypervisor encapsulates GPL
  → RT side unaffected
  → Can introduce additional VMs for other code...
  → Stub driver forwards IO requests
Why Virtualisation: Security

- Protect against exploits
- Modem software attacked by UI exploits
  - Compromised application OS could compromise RT side
  - Could have serious consequences
e.g. for wireless devices (jamming)
- Virtualisation protects
  - Separate apps and system code into different VMs
Why Virtualisation: Security

- Multiple cores offer insufficient protection
  - Cores share memory
    - compromised OS can attack OSes on other cores

- Virtualisation protects assets
  - Provided OS is de-privileged
  - Pseudo-virtualization buys nothing

- Digital Rights Management
  - Encapsulate media player in own VM
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-bandwidth communication
  ➔ Shared devices with low-latency access
- Need integrated scheduling across virtual machines
  ➔ High-level OS (best-effort VM) must be lower prio than real-time threads
  ➔ However, some threads in real-time subsystem are background activities
- Need more than just a hypervisor!
Hypervisors vs Microkernels

- Microkernels have been used as hypervisors for a long time
  - Mach Unix ('90), L4Linux ('97)

- Hypervisors have more visibility than microkernels

- Both encapsulate subsystems
  - Are hypervisors microkernels done right? [HWF+05]

- What's the difference?
  - Microkernels are generic
  - Hypervisors are only meant to support VMs running guest OSes
Microkernel as a Hypervisor

- Microkernel as a hypervisor half-way between native and hosted VMM?
  - However, para-virtualisation may also benefit from in-place emulation
  - E.g. save mode switches by virtualising PSR inside guest address space
Microkernel as a Hypervisor

- Has all advantages of a pure hypervisor:
  - Provide isolation (where needed)
  - Run arbitrary guest OSes (high-level and RTOS)

- Supports efficient sharing
  - High-performance IPC mechanism
  - Shared memory regions
  - Support for device sharing

- Supports interleaved scheduling
  - Application OS VM scheduled as a unit (with a single microkernel prio)
  - RT threads directly scheduled by microkernel (with individual prios)
  - Can have some at higher, some at lower prio than app OS environment
Hypervisor vs. Microkernel

- No other code in kernel mode
- Specialised, legacy guest OS only
- VMM completely in kernel (?)
- Variety of mechanisms
- Smaller? (Xen is 50–100kLOC!)
- Guest communicate via virtual NW
- Strong subsystem partitioning

- No other code in kernel mode
- Generic, guest OS & native apps
- VMM partially in guest AS
- Minimal mechanism
- Small (L4≈10kloc)
- Guest communicate via IPC
- continuum: partitioned - integrated

• Microkernel can be seen as a generalisation of a hypervisor
  - Do we pay with performance?
  - See also [HWF⁺05, HUL06]
Hypervisor vs. Microkernel: Performance

- Xen vs. L4 on Pentium 4 running Linux 2.6.9
- Device drivers in guest OS

<table>
<thead>
<tr>
<th>System</th>
<th>Kernel Compile</th>
<th>Netperf send</th>
<th>Netperf receive</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (s)</td>
<td>CPU (%)</td>
<td>O/H (%)</td>
</tr>
<tr>
<td>native</td>
<td>209</td>
<td>98.4</td>
<td>0</td>
</tr>
<tr>
<td>Linux on Xen</td>
<td>219</td>
<td>97.8</td>
<td>5</td>
</tr>
<tr>
<td>Linux on L4</td>
<td>236</td>
<td>97.9</td>
<td>13</td>
</tr>
</tbody>
</table>

- Xen base performance is better
  - ... but more intrusive changes to Linux
  - Network performance shows that there is optimisation potential
Sharing Devices

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

VM1 (Linux) Priority

System Priority

VM2 (RTOS) Priority

Linux RT task

RT5
RT4
RT3

RT2

Linux

Background1

Background2

Background task

Linux Background task
Other Microkernel Advantages: Native Environment

• Microkernel suitable for a native OS environment
  ➔ Hypervisor only meant to support a guest OS
  ➔ Microkernel powerful enough to support native OS environment

• Microkernel minimises trusted computing base
  ➔ No guest OS required for simple applications
  ➔ E.g. trusted crypto app
    • run in own protection domain
  ➔ Xen TCB includes dom-0 guest
    (complete Linux!)

![Diagram showing the relationship between Legacy App, OK Linux, Sensitive App, Trusted Service, Device Driver, and OKL4.]
Other Microkernel Advantages: Hybrid Systems

- Co-existence of monolithic and componentised subsystems
  - Legacy support
  - Successive migration
    - componentise over time...