COMP9242
Advanced Operating Systems
S2/2013 Week 8: Virtualization
Copyright Notice

These slides are distributed under the Creative Commons Attribution 3.0 License

• You are free:
  – to share—to copy, distribute and transmit the work
  – to remix—to adapt the work
• under the following conditions:
  – Attribution: You must attribute the work (but not in any way that suggests that the author endorses you or your use of the work) as follows:
    • “Courtesy of Gernot Heiser, [Institution]”, where [Institution] is one of “UNSW” or “NICTA”

The complete license text can be found at http://creativecommons.org/licenses/by/3.0/legalcode
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 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 instruction are executed directly by real hardware

Hypervisor aka virtual-machine monitor: Software implementing the VM
Types of Virtualization

- **Platform VM or System VM**
  - *Type-1 “Native”*
  - *Type-2 “Hosted”*

- **Operating System**
- **Processor**

- **Hypervisor**
- **Virtualize Layer**

- **OS API**

- **Programming Language**
- **Java Program**
  - *Java VM*

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 the hardware
  - Complete encapsulation
    - replication
    - migration
    - checkpointing
    - debugging
  - Different concurrent OSes
    - eg Linux + Windows
  - Total mediation
- Would be mostly unnecessary
  - ... if OSes were doing their job!

Gernot prediction of 2004: 2014 OS textbooks will be identical to 2004 version except for s/process/VM/g
Why Virtual Machines?

- 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
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

Guest

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

Exception

Hypervisor

```
lda r1, vm_reg_ctxt
ld   r2, (r1, ofs_r0)
sto r2, (r1, ofs_ASID)
```
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

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

```
Hypervisor
lda  r1, vm_reg_ctxt
ld   r2, (r1,ofs_r0)
sto  r2, (r1,ofs_ASID)
```
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” [Orran Krieger, then IBM, later VMware] 😊
Para-Virtualization

• New(ish) name, old technique
  – coined by Denali [Whitaker ‘02], popularised by Xen [Barham ‘03]
  – Mach Unix server [Golub ‘90], L4Linux [Härtig ‘97], Disco [Bugnion ‘97]

• Idea: manually port guest OS to modified (more high-level) ISA
  – Augmented by explicit hypervisor calls (hypercalls)
    • higher-level ISA to reduce number of traps
    • remove unvirtualisable instructions
    • remove “messy” ISA features which complicate
  – Generally outperforms pure virtualisation, binary re-writing

• Drawbacks:
  – Significant engineering effort
  – Needs to be repeated for each guest-ISA-hypervisor combination
  – Para-virtualised guests must be kept in sync with native evolution
  – Requires source
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()
  - MIPS KSEG addresses used for physical addressing in kernel
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

```
    mov r1,#VPSR
    ldr r0,[r1]
    orr r0,r0,#VPSR_ID
    sto r0,[r1]
```
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
    - explicity hypercall 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

![Diagram of VPSR and PSR registers with trap and mov, ldr, orr, sto instructions]
Virtualization and Address Translation

Two levels of address translation!

Virtual Memory → Virtual Page Table → Guest Physical Memory

Virtual Memory → Virtual Page Table → Guest Physical Memory

Virtual Memory → Virtual Page Table → Guest Physical Memory

Must implement with single MMU translation!
Virtualization Mechanics: Shadow Page Table

User
1d r0, adr

Guest virtual address

(Virtual) guest page table

Shadow (real) guest page table, translations cached in TLB

Guest physical address

Hypervisor's guest memory map

PT ptr (Hardware)
Hypervisor

PT ptr (Software)
Guest OS

Virt

User level

Guest OS

Hypervisor

Physical address

Memory

data

ld r0, adr

Guest virtual address

(Virtual) guest page table

Shadow (real) guest page table, translations cached in TLB

Guest physical address

Hypervisor's guest memory map

PT ptr (Hardware)
Hypervisor

PT ptr (Software)
Guest OS
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

Hypervisor

Virt PT ptr

PT ptr

Memory

Guest

Virtual address

Guest physical address

Physical address

data

User

ld r0, adr

Guest OS

Virt PT ptr

Graduated Centre forста
Virtualisation Semantics: Lazy Shadow Update

- User:
  - access new page
  - continue

- Guest OS:
  - add mapping to GPT
  - add mappings...

- Hypervisor:
  - write-protect GPT
  - unprotect GPT & mark dirty
  - update dirty shadow write-protect GPT
Virtualisation Semantics: Lazy Shadow Update

User

Guest OS

Hypervisor

continue

invalidate mapping in GPT

flush TLB

write-protect GPT

unprotect GPT & mark dirty

update dirty shadow

write-protect GPT

flush TLB
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

Para-virtualized guest “knows” it is virtualized

User

ld r0, adr

Guest virtual address

Guest OS

Guest PT

Hypervisor

Guest PT

HV PT

Physical address

Memory

data

Used by original Xen
Virtualization Mechanics: 3 Device Models

- **Emulated**
  - VM$_1$
  - OS
  - Device Driver
  - Emulation
  - Hypervisor
  - Device

- **Split**
  - VM$_1$
  - OS
  - Device Driver
  - Virtual Driver
  - Hypervisor
  - Device

- **Pass-through**
  - VM$_1$
  - OS
  - Device Driver
  - Hypervisor
  - Device
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 (I/O MMU)
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 MOVS 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
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

Kernel mode

User mode

Native syscall

Can configure traps to go directly to guest OS

Virtual syscall

Trap to guest

User mode

Kernel mode

Virtual syscall

Hyp mode

x86 similar
ARM Virtualization Extensions (3)

Emulation

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

```
mv CPU_ASID, r1

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

```
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)
```

No x86 equivalent
ARM Virtualization Extensions (4)

2-stage translation

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

![Diagram showing 2-stage translation process with labels and annotations](image)

- User
  - `ld r0, adr`
- Guest virtual address
- 1st PT ptr (Hardware)
- Guest OS
- Guest page table
- (Virtual) guest page table
- Hypervisor's guest memory map
- Guest physical address
- 2nd PT ptr (Hardware)
- Hypervisor
- Physical address
- Memory
- data
2-stage translation cost

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

x86: issue only for legacy level-triggered IRQs
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</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>Paravirtualiz.</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 paravirtualization
- 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

Variant: VMware MVP
- ARM hypervisor
  - pre-HW support
- re-writes exception vectors in Android kernel to catch virtualization traps in guest
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

Difference to traditional terminology!
### 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>
<tr>
<td>I/O</td>
<td>Simplified virtual device</td>
<td>IPC interface to user-mode driver</td>
</tr>
<tr>
<td></td>
<td>• Driver in hypervisor</td>
<td>• Interrupt IPC</td>
</tr>
<tr>
<td></td>
<td>• Virtual IRQ (vIRQ)</td>
<td></td>
</tr>
<tr>
<td>Communication</td>
<td>Virtual NIC, with driver and network stack</td>
<td>High-performance message-passing IPC</td>
</tr>
</tbody>
</table>

- Similar abstractions
- Optimised for different use cases

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