COMP9242
Advanced Operating Systems
S2/2011 Week 4: 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 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

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

- **Operating System**
  - VM Process
  - OS

- **Hypervisor**
  - Processor

- **Virtualiz. Layer**
  - Process VM

- **Process VM**
  - Processor

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

- **OS API**

- **“Platform” (HW/SW Interface)**
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 hardware
  • Complete encapsulation
    - replication
    - migration
    - checkpointing
    - debugging
  • Different concurrent OSes
    - e.g.: Linux and Windows
  • Total mediation
→ Would be mostly unnecessary
  • if OSes were doing their job...
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

![Diagram of virtual machines and hypervisor]
Hypervisor

→ Program that runs on real hardware to implement the virtual machine
→ Controls resources
  • Partitions hardware
  • Schedules guests
    • Performs 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

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
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

\[
\text{ld } r0, \text{ curr thrd} \\
\text{ld } r1, (r0, \text{ASID}) \\
\text{mv } \text{CPU ASID}, r1 \\
\text{ld } \text{sp}, (r1, \text{kern stk}) \\
\]

Hypervisor

\[
\text{lda } r1, \text{ vm_reg_ctxt} \\
\text{ld } r2, (r1, \text{ofs r0}) \\
\text{sto } r2, (r1, \text{ofs ASID}) \\
\]
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

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

### Hypervisor

```assembly
lda r1, vm_reg_ctxt
ld r2, (r1,ofs_r0)
sto r2, (r1,ofs_ASID)
```
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 now 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 Denali [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 (x86)

→ 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)
  • guest can update without (expensive) hypervisor invocation
    - 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

```
psid
Trap
```

```
<table>
<thead>
<tr>
<th>VPSR</th>
<th>PSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

```
mov r1,#VPSR
ldr r0,[r1]
orr r0,r0,#VPSR_ID
sto r0,[r1]
```
Virtualization Techniques

→ E.g. lazy update of virtual machine state
  • virtual state is kept inside hypervisor
  • 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: add a mapping:
    - guest enables FPU
    - no need to invoke hypervisor at this point
    - hypervisor syncs state on virtual kernel exit
Virtualization Mechanics: Page Table Access

Hypervisor maintains guest PT

• On guest PT access must translate PTEs
  – store: translate guest “PTE” to real PTE
  – load: translate real PTE to guest “PTE”

• Each guest PT access traps!
  – high overhead
Virtualization Mechanics: Shadow Page Table

Let guest keep its own PT
• Hypervisor maintains shadow

• TLB semantics ⇒ weak consistency
• Batch updates on sync points

User  
\text{ld r0, adr}

(Virtual) guest page table

Guest virtual address

Guest OS

Virt PT ptr
(Software)

Hypervisor

(Physical) guest page table

Guest physical address

Hypervisor's guest memory map

Hypervisor mirrors guest's page table updates

PT ptr
(Hardware)

Physical address

Memory

data
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 MOVTS 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

- **Native syscall**
  - User mode
  - Kernel mode
  - Hyp mode
  - Can configure traps to go directly to guest OS

- **Virtual syscall**
  - User mode
  - Kernel mode
  - Hyp mode

**Trap to guest**
ARM Virtualization Extensions (3)

Emulation

1) Load faulting instruction
   • Compulsory L1-D miss!

2) Decode instruction
   • Complex logic

3) Emulate instruction
   • Usually straightforward

IR

R2

L1 I-Cache

L1 D-Cache

L2 Cache

\[
\begin{align*}
&\text{mv CPU\_ASID,} r1 \\
&\text{ld r1,}(r0,\text{ASID}) \\
&\text{mv CPU\_ASID,} r1 \\
&\text{ld sp,}(r1,\text{kern_stk})
\end{align*}
\]

\[
\begin{align*}
&\text{ld r1,}(r0,\text{ASID}) \\
&\text{mv CPU\_ASID,} r1 \\
&\text{ld sp,}(r1,\text{kern_stk})
\end{align*}
\]
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
```

```
L1 D-Cache
```

```
L2 Cache
```

```
lr l1, (r0, ASID)
mv CPU_ASID, r1
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
```

1\textsuperscript{st} PT ptr (Hardware)

Guest OS

(Virtual) guest page table

Guest virtual address

2\textsuperscript{nd} PT ptr (Hardware)

Hypervisor

Hypervisor's guest memory map

Physical address

Memory

data
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
- Reduces hypervisor entries for interrupt virtualization
## 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>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 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
- 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>
<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>

**Real Difference?**

- Just page tables in disguise
- Just kernel-scheduled activities
- Modelled on HW, Re-uses SW
- Minimal overhead, Custom API
• 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 attempts to improve 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!