UNSW
THE UNIVERSITY OF NEW SOUTH WALES

NICTA

COMP9242
Advanced Operating Systems

S2/2011 Week 4: Virtualization

Australian Government
Department of Broadband, Communications and the Digital Economy
Australian Research Council

NICTA Funding and Supporting Members and Partners

Copyright Notice

NICTA

  UNSW

Virtual Machine (VM)

NICTA

"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

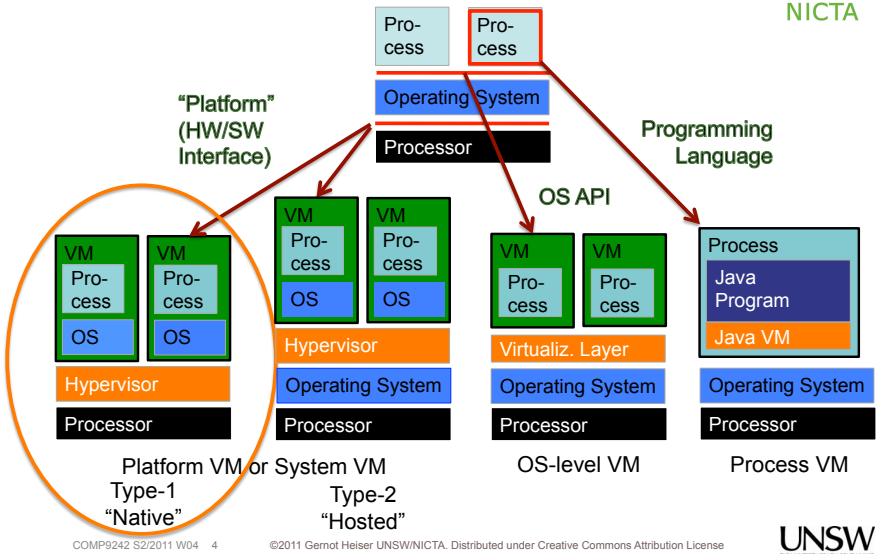
→ Efficient: VM should execute at speed close to that of real hardware
- Requires that most instruction are executed directly by real hardware

Hypervisor aka *virtual-machine monitor*: Software implementing the VM

  UNSW

Types of Virtualization

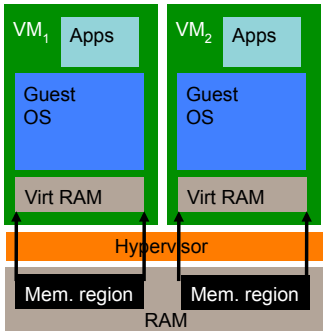
NICTA



  UNSW

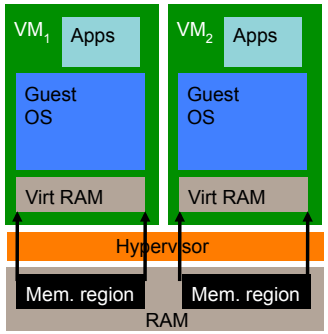
Why Virtual Machines?

- 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...



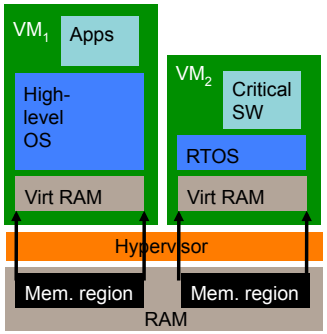
Why Virtual Machines?

- 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...



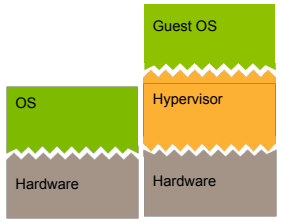
Why Virtual Machines?

- Embedded systems: integration of heterogenous environments
  - RTOS for critical real-time functionality
  - Standard OS for GUIs, networking etc
- Alternative to physical separation
  - low-overhead communication
  - cost reduction



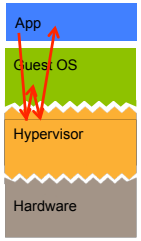
Hypervisor

- Program that runs on real hardware to implement the virtual machine
- Controls resources
  - Partitions hardware
  - Schedules guests
    - 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*

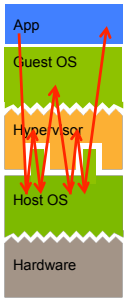


Native vs. Hosted VMM

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

Virtualization Mechanics: Instruction Emulation

- 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

```
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**: 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

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

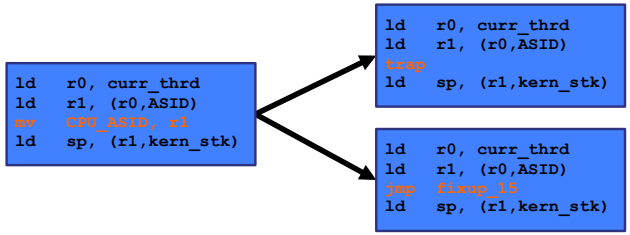
Exception

Hypervisor

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

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

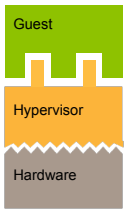


Binary Translation

- 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], L^{4}Linux [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

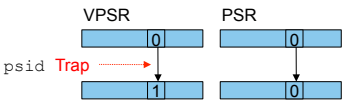


Virtualization Overheads

- 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




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

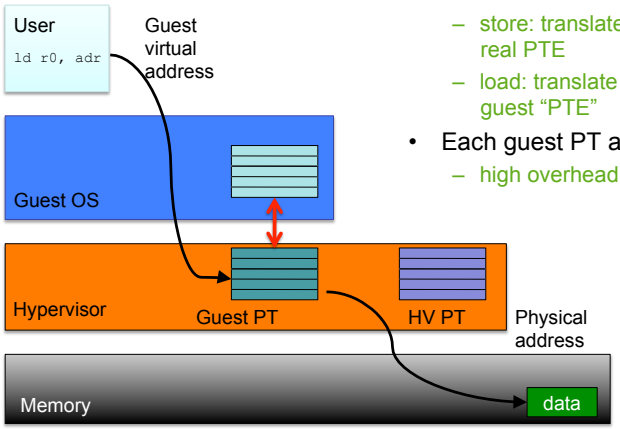

Virtualization Techniques

- 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
    - -hypervsior syncs state on virtual kernel exit

Virtualization Mechanics: Page Table Access

Hypervisor maintains guest PT

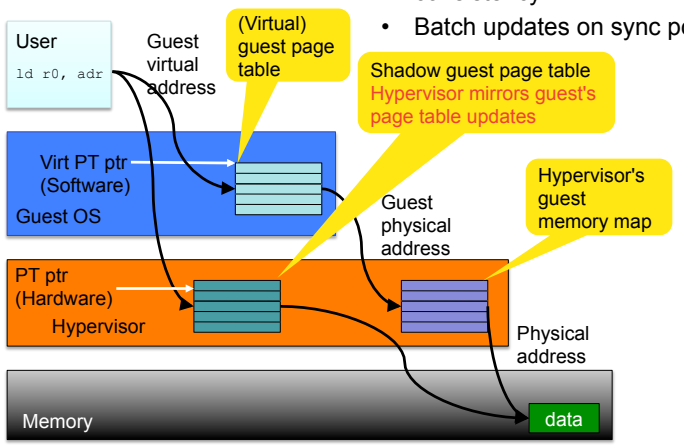
- On guest PT access must translate PTEs
  - store: translate guest "PTE" to real PTE
  - load: translate real PTE to guest "PTE"
- Each guest PT access traps!
  - high overhead



Virtualization Mechanics: Shadow Page Table

Let guest keep its own PT

- Hypervisor maintains shadow
- TLB semantincs ⇒ weak consistency
- Batch updates on sync points

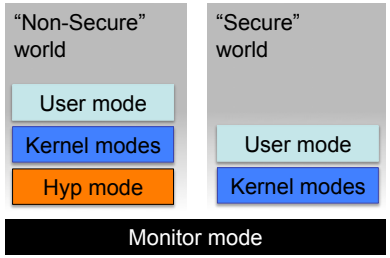


Non-Virtualizable Architectures

- 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 MOVS 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)

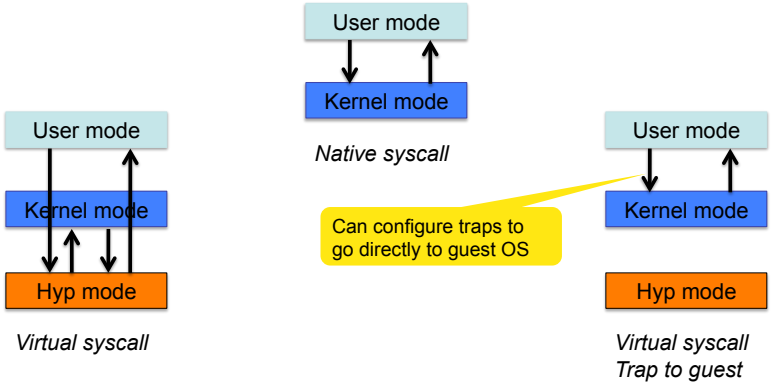
Hyp mode



- 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)

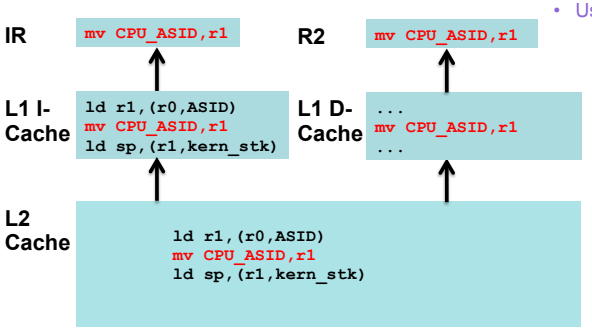
Configurable Traps



ARM Virtualization Extensions (3)

Emulation

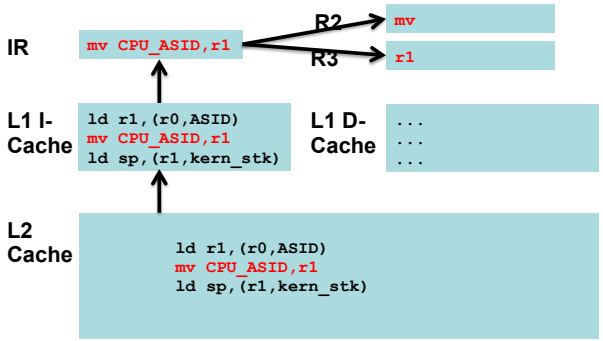
1) Load faulting instruction
   - Compulsory L1-D miss!
2) Decode instruction
   - Complex logic
3) Emulate instruction
   - Usually straightforward



ARM Virtualization Extensions (3)

Emulation Support

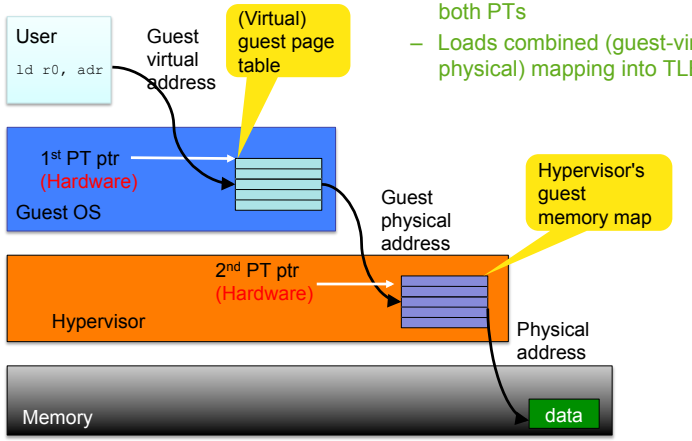
- HW decodes instruction
  - No L1 miss
  - No software decode
- SW emulates instruction
  - Usually straightforward



ARM Virtualization Extensions (4)

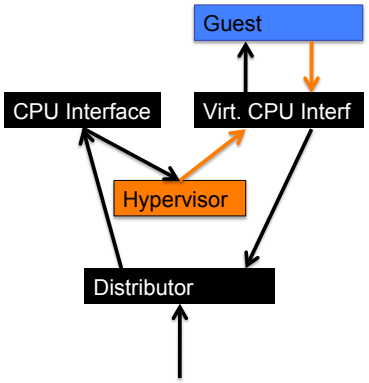
2-stage translation

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



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

Hypervisor	ISA	Type	Kernel	User
OKL4	ARMv7	para-virtualization	9.8 kLOC	0
Prototype	ARMv7	pure virtualization	6 kLOC	0
Nova	x86	pure virtualization	9 kLOC	27 kLOC

- 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)

Operation	Pure virtualization		Para-virtualiz.
	Instruct	Cycles (est)	Cycles (approx)
Guest system call	0	0	300
Hypervisor entry + exit	120	650	150
IRQ entry + exit	270	900	300–400?
Page fault	356	1500	700
Device emul.	249	1040	N/A
Device emul. (accel.)	176	740	N/A
World switch	2824	7555	200

- 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?

Resource	Hypervisor	Microkernel
Memory	Virtual MMU (vMMU)	Address space
CPU	Virtual CPU (vCPU)	Thread or scheduler activation
I/O	• Simplified virtual device • Driver in hypervisor • Virtual IRQ (vIRQ)	• IPC interface to user-mode driver • Interrupt IPC
Communication	Virtual NIC, with driver and network stack	High-performance message-passing IPC

Just page tables in disguise

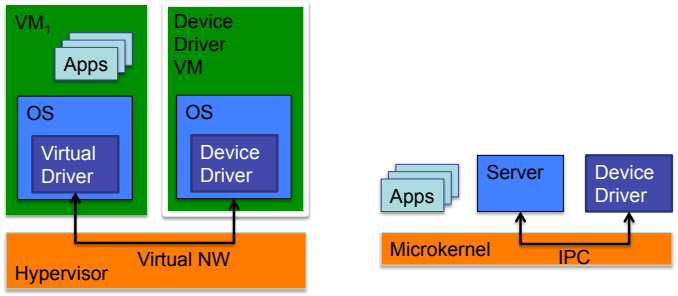
Just kernel-scheduled activities

Real Difference?

Minimal overhead, Custom API

Modelled on HW, Re-uses SW

Closer Look at I/O and Communication



- 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!