Embedded Hypervisor for ARM

Jim Huang (黃敬群)

Developer, 0xlab

jserv@0xlab.org

Dec 6, 2011

Rights to copy

http://0xlab.org/

contact@0xlab.org

Corrections, suggestions, contributions and translations are welcome!

Latest update:Feb 6, 2012

COMMONS DEED

Attribution – ShareAlike 3.0

You are free

- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Under the following conditions

Attribution. You must give the original author credit.

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

License text: http://creativecommons.org/licenses/by-sa/3.0/legalcode

Agenda

(1) Virtualization from The Past

(2) Hypervisor Design

(3) Embedded Hypervisors for ARM

(4) Toward ARM Cortex-A15

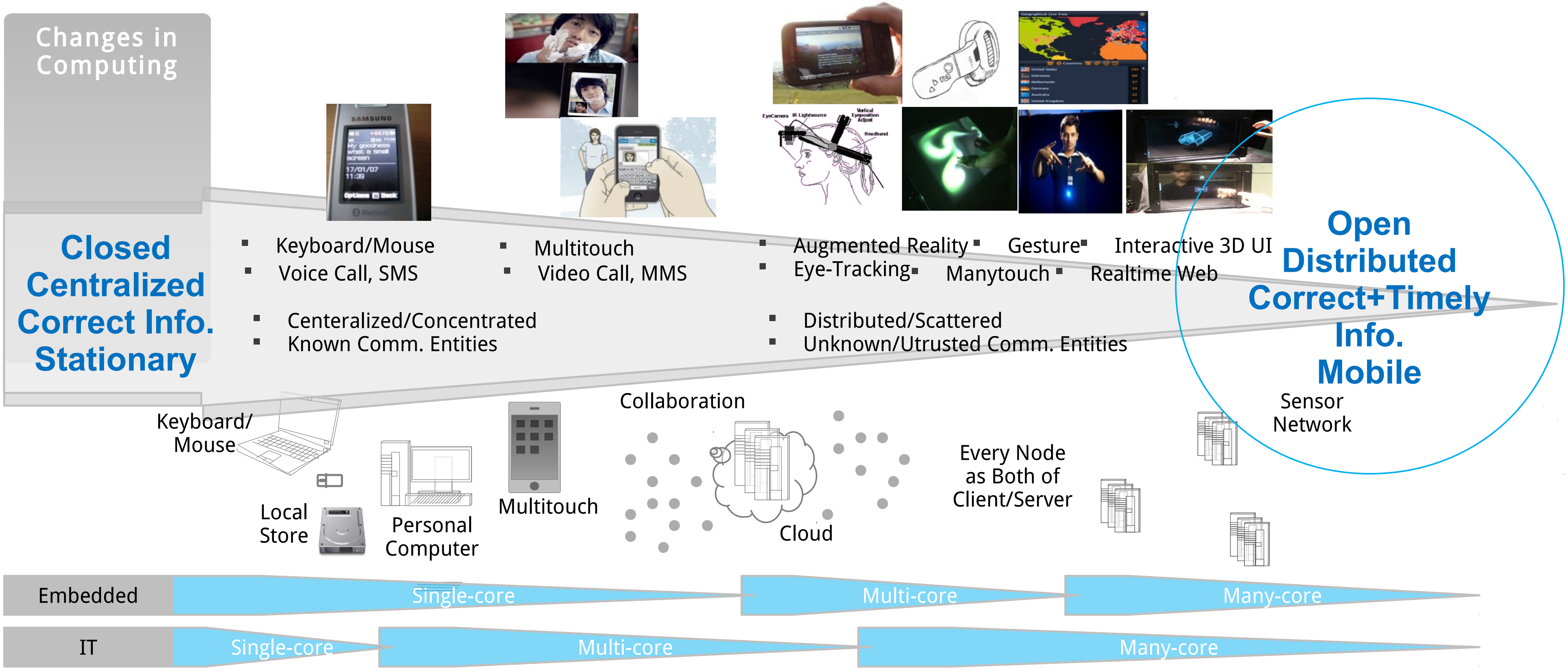
Virtualization from The Past

Definition

"**virtualization** is "a technique for hiding the physical characteristics of computing resources from the way in which other systems, applications, or end users interact with those resources. "

– Wikipedia

Future Computing Trends



Changes in Computing

Closed
Centralized
Correct Info.
Stationary

- Keyboard/Mouse
- Voice Call, SMS

- Multitouch
- Video Call, MMS

- Augmented Reality
- Gesture
- Interactive 3D UI
- Eye-Tracking
- Manytouch
- Realtime Web

- Centeralized/Concentrated
- Known Comm. Entities

- Distributed/Scattered
- Unknown/Utrusted Comm. Entities

Open
Distributed
Correct+Timely Info.
Mobile

Embedded	Single-core	Multi-core	Many-core
IT	Single-core	Multi-core	Many-core

- UC Berkeley Sensornet Chip (TI MSP430 8MHz core, 10KB RAM)

[2009]
- Tiger 1GHz Single-Core
- Dunnington 3GHz 6-core

[2012]
- ARM 2GHz 4-core
- Intel 4GHz 32-core

[2017]
- ARM 3GHz 8-core
- Intel 6GHz 128-core
- SensorNet Chip
(128MHz core, 160KB RAM)

Privacy

Realtime

Source: **Xen ARM Virtualization**, Xen Summit Asia 2011
by Dr. Sang-bum Suh, Samsung

Server Virtualization::Benefits

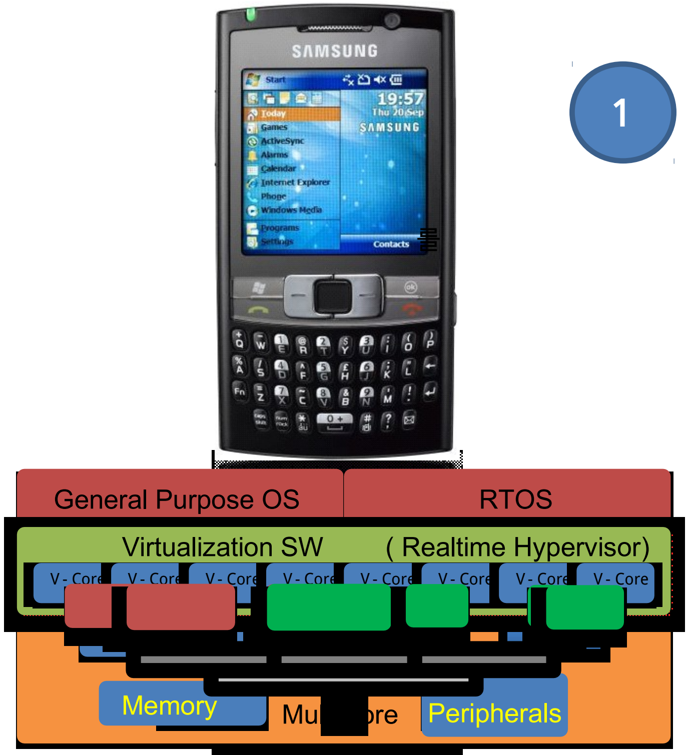
- Workload consolidation
  - Increase server utilization
  - Reduce capital, hardware, power, space, heat costs
- Legacy OS support
  - Especially with large 3rd-party software products
- Instant provisioning
  - Easily create new virtual machines
  - Easily reallocate resources (memory, processor, IO) between running virtual machines
- Migration
  - Predicted hardware downtime
  - Workload balancing

Embedded Virtualization::Benefits

- Workload consolidation
- Flexible resource provisioning
- License barrier
- Legacy software support
  - Especially important with dozens or hundreds of embedded operating systems, commercial and even home-brew
- Reliability
- Security

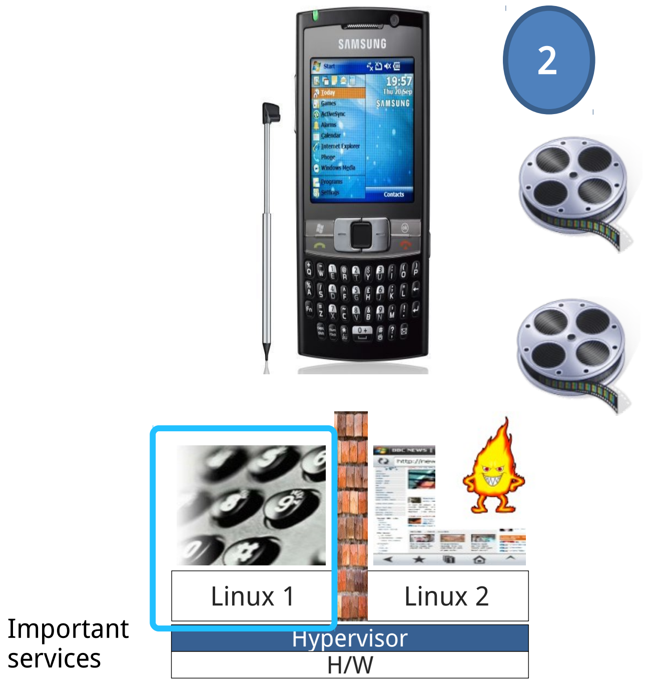
Why?

- (1) **Hardware Consolidation**
  - *Application Processor* and *Baseband Processor* can share multicore ARM CPU SoC to run both Linux and RTOS efficiently.

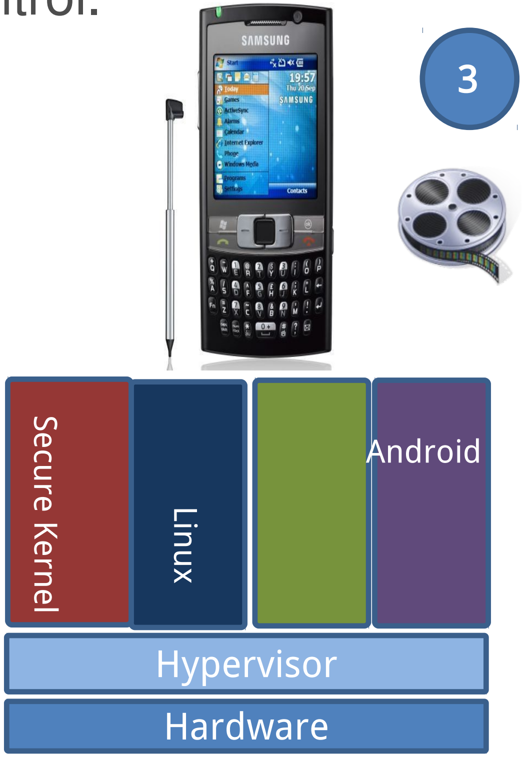


AP SoC + BP [redacted]ted Multicore SoC

- (2) **OS Isolation**
  - important call services can be effectively separated from downloaded third party applications by virtualized ARM combined with access control.



Secure Smartphone



Rich Applications from Multiple OS

- (3) **Rich User Experience**
  - multiple OS domains can run concurrently on a single smartphone.

Source: **Xen ARM Virtualization**, Xen Summit Asia 2011 by Dr. Sang-bum Suh, Samsung

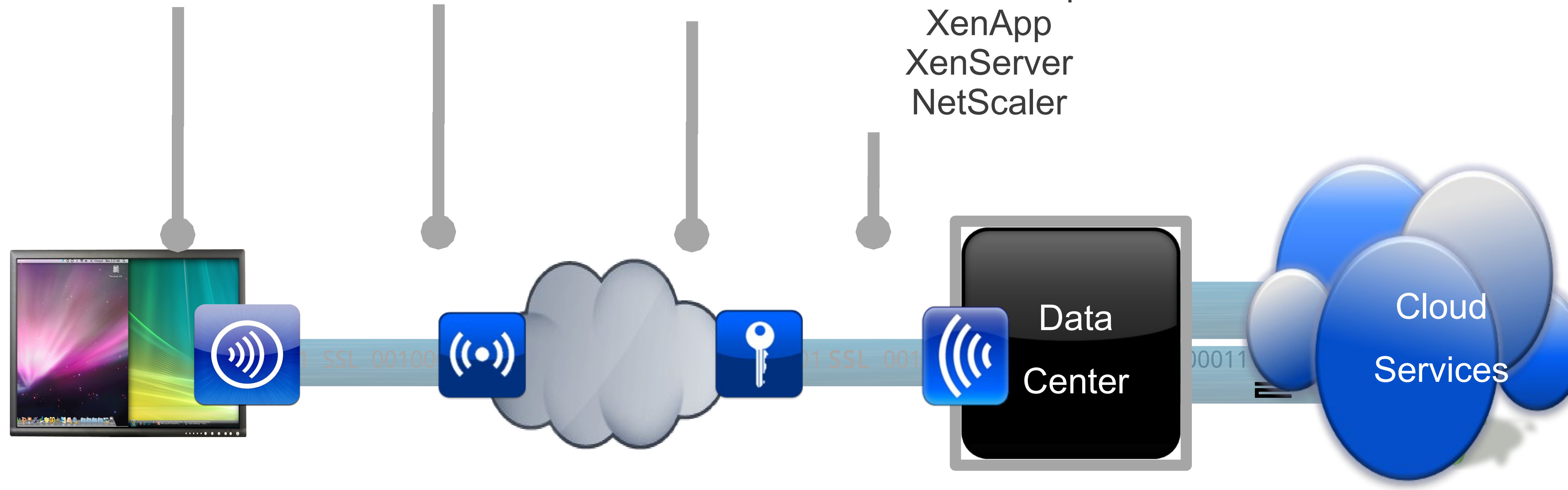
Use Case: **Nirvana:**

The Convergence of Mobile and Desktop Virtualization in One Device by OKLabs + Citrix

Open Kernel Labs™

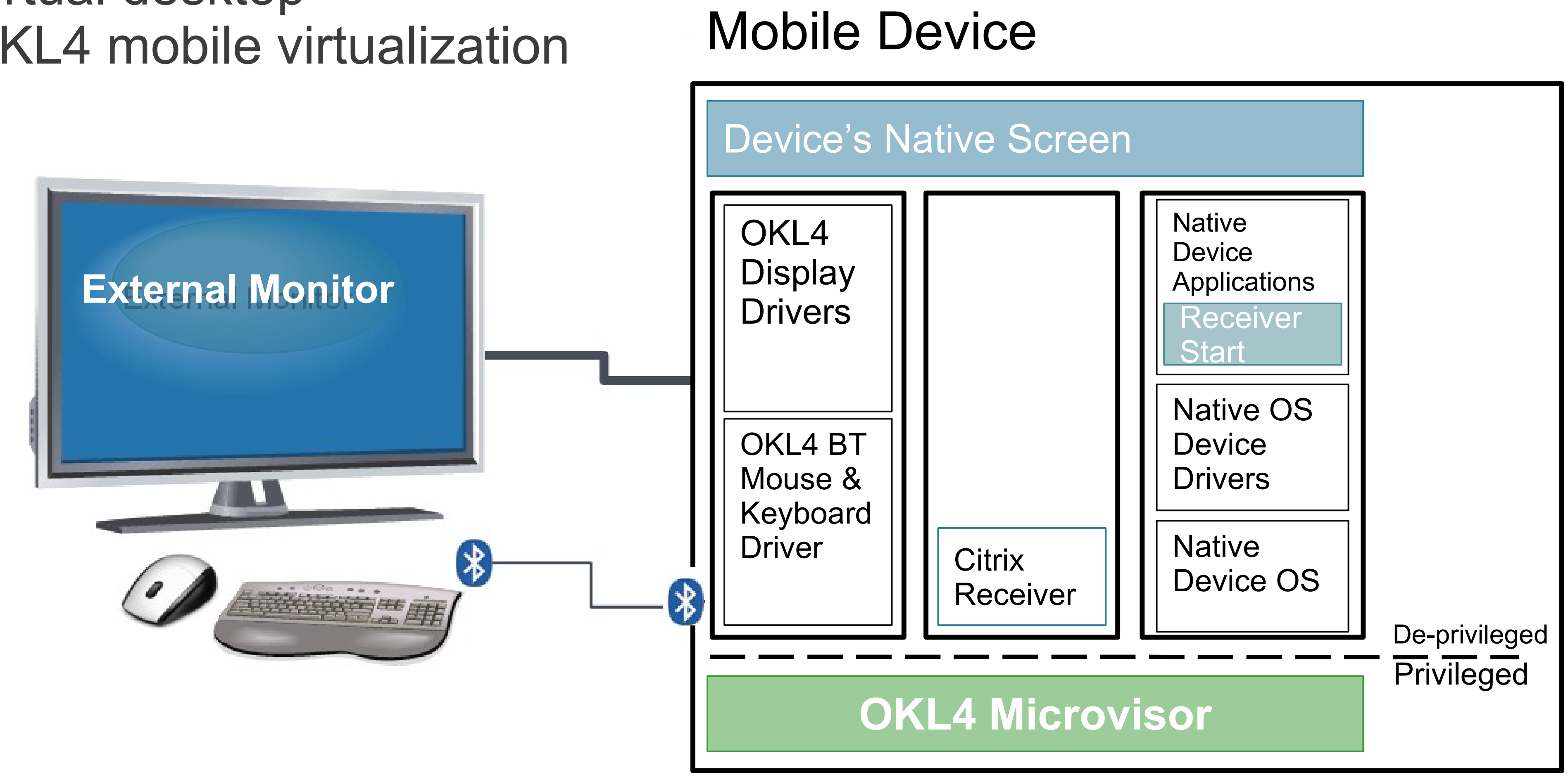
CITRIX®

XenDesktop
XenApp
XenServer
NetScaler



Nirvana Phone

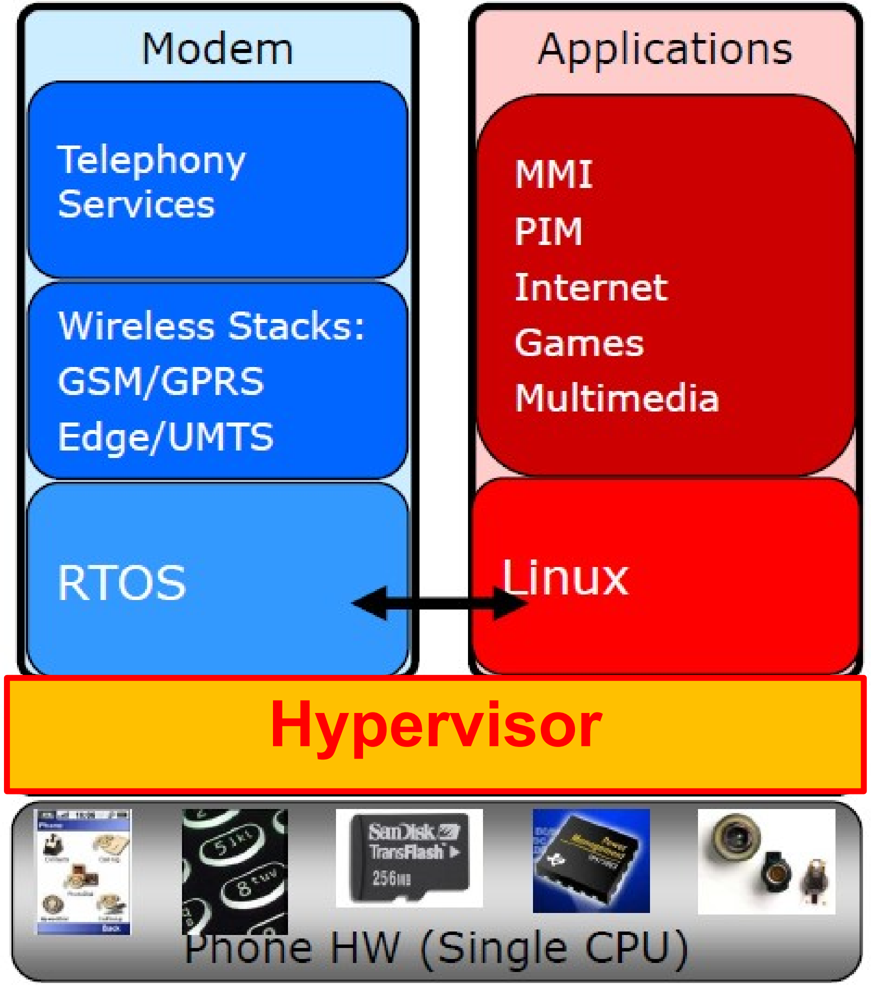
Nirvana phone = Smartphone
+ Full-sized display
+ Keyboard & mouse
+ Virtual desktop
+ OKL4 mobile virtualization



Demo video:
http://www.youtube.com/user/OpenKernelLabs

Use Case: Low-cost 3G Handset

- Mobile Handsets
  - Major applications runs on Linux
  - 3G Modem software stack runs on RTOS domain
- Virtualization in multimedia Devices
  - Reduces BOM (bill of materials)
  - Enables the Reusability of legacy code/applications
  - Reduces the system development time
- Instrumentation, Automation
  - Run RTOS for Measurement and analysis
  - Run a GPOS for Graphical Interface



Modem

Telephony Services

Wireless Stacks: GSM/GPRS Edge/UMTS

RTOS

Applications

MMI

PIM

Internet

Games

Multimedia

Linux

Hypervisor

Phone HW (Single CPU)

Virtualization Tradeoff

- Performance tradeoff
  - Applications that used to own the whole processor must now share
  - Hypervisor adds some runtime overhead as well
  - Full virtualization without hardware support means software emulation
- Increase in management complexity
  - Old scenario: two software stacks + two hardware systems
  - New scenario: two software stacks + one hardware system + one host kernel
- More abstraction, more software layers, more complexity...
  - More bugs
- Increases size of TCB (Trusted Computing Base)
- Increases impact of unpredicted hardware failure

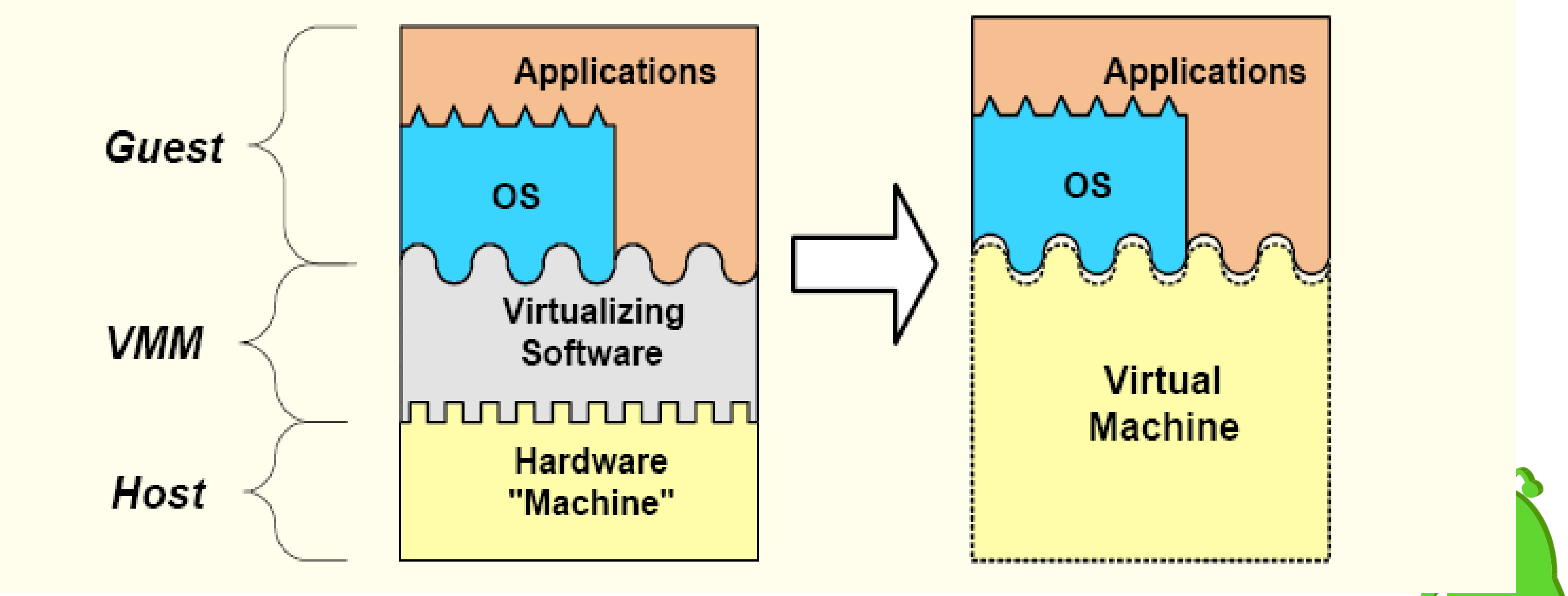
Hypervisor Design

"All problems in computer science can be solved by another level of indirection."

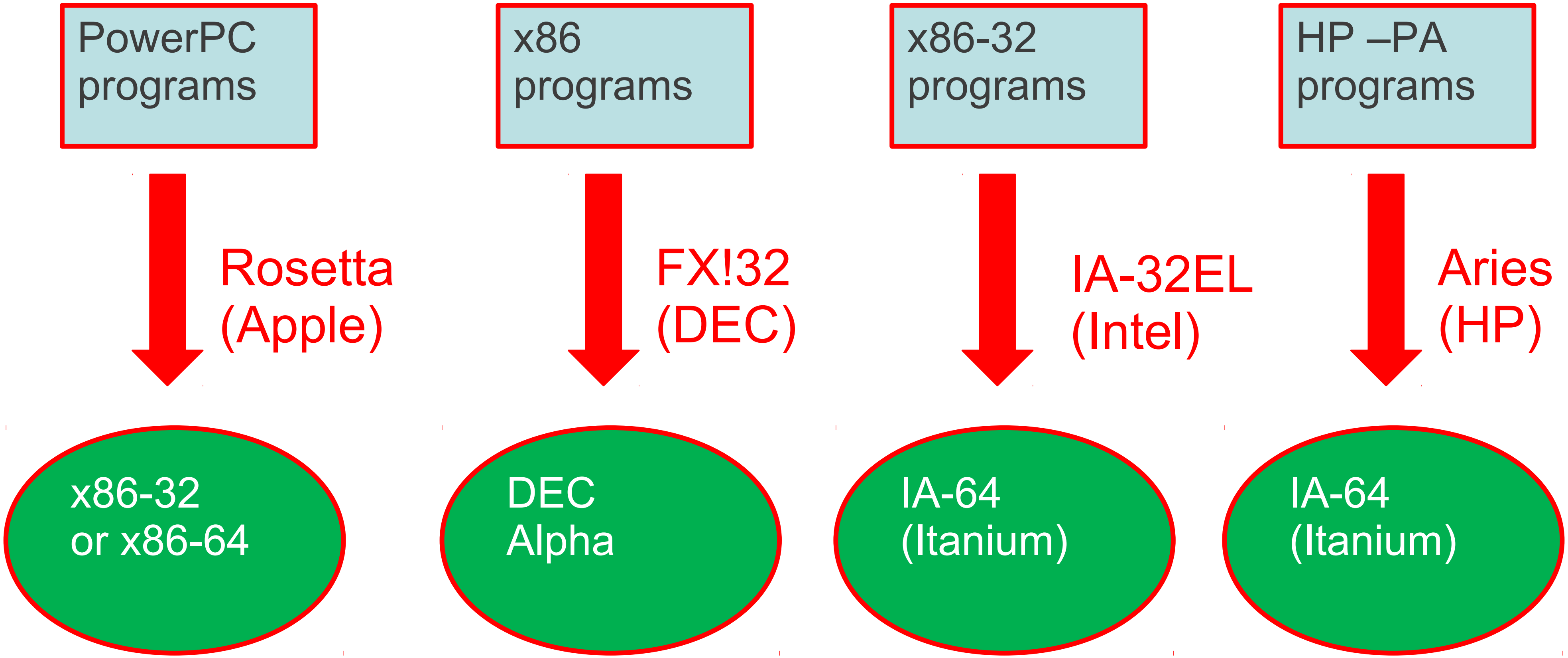
-- David Wheeler --

Virtual Machine

- Add Virtualizing Software to a Host platform and support Guest process or system on a Virtual Machine (VM)



Virtual Machine for Portability (in the past)



PowerPC programs

Rosetta (Apple)

x86-32 or x86-64

x86 programs

FX!32 (DEC)

DEC Alpha

x86-32 programs

IA-32EL (Intel)

IA-64 (Itanium)

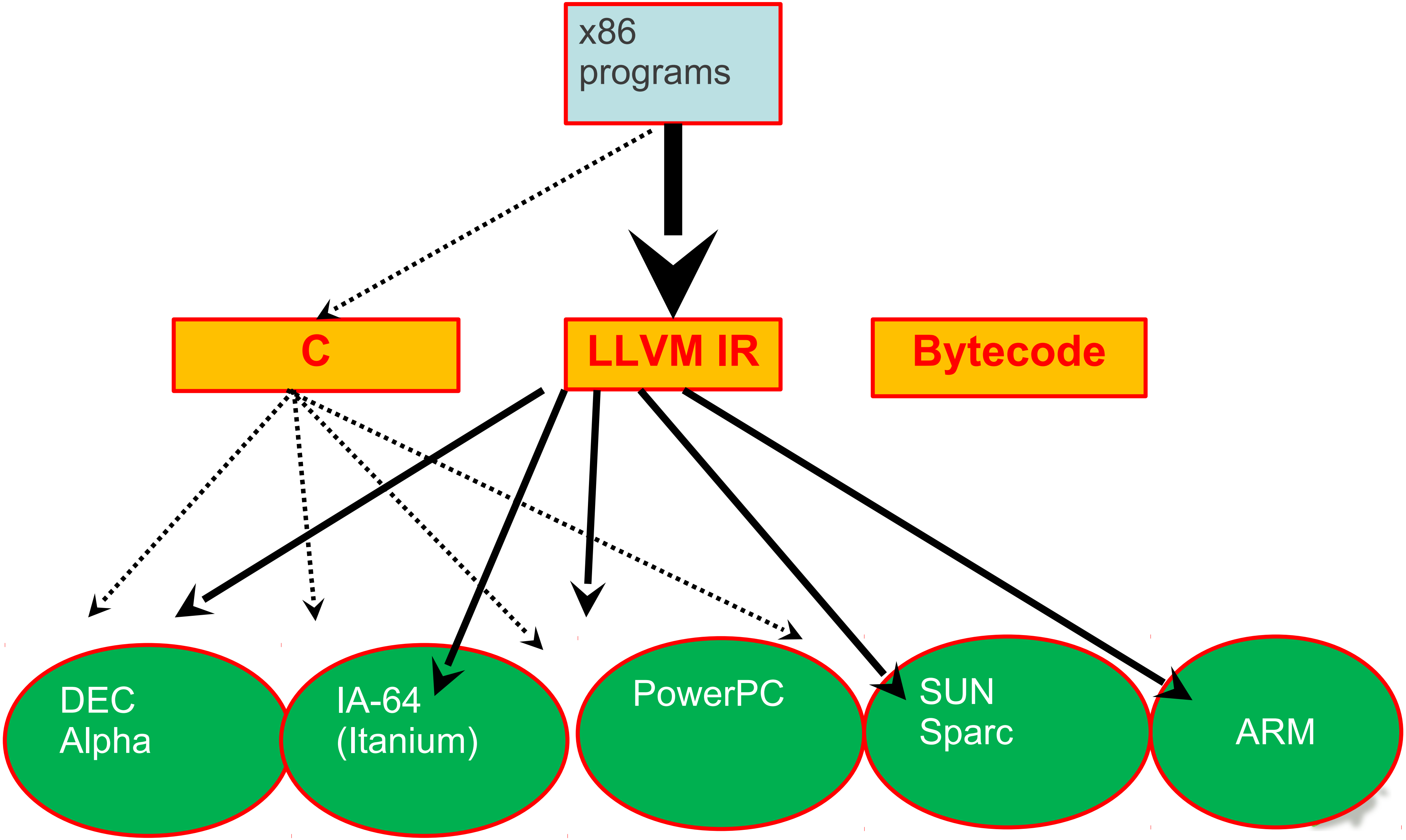
HP –PA programs

Aries (HP)

IA-64 (Itanium)

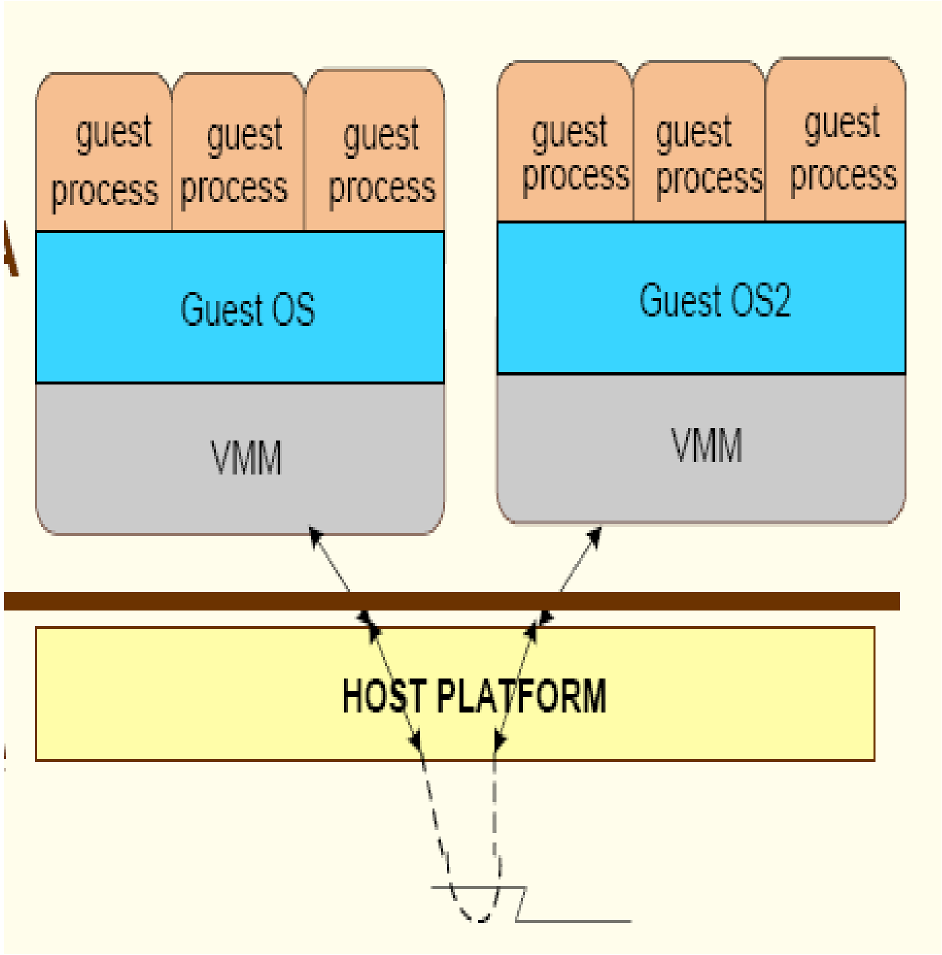
Virtual Machine for Portability

NOTE: we don't discuss such topic in this presentation



System Virtual Machine

- Provide a system environment
- Constructed at ISA level
- Allow multiple OS environments, or support time sharing.
- virtualizing software that implements system VM is called as VMM (virtual machine monitor)
- Examples:
  - IBM VM/360, VMware, VLX, WindRiver Hypervisor, ENEA Hypervisor
  - Xtratum, Lguest, BhyVe (BSD Hypervisor)
  - **Xen**, **KVM**, **OKL4**, **Xvisor**, **Codezero**

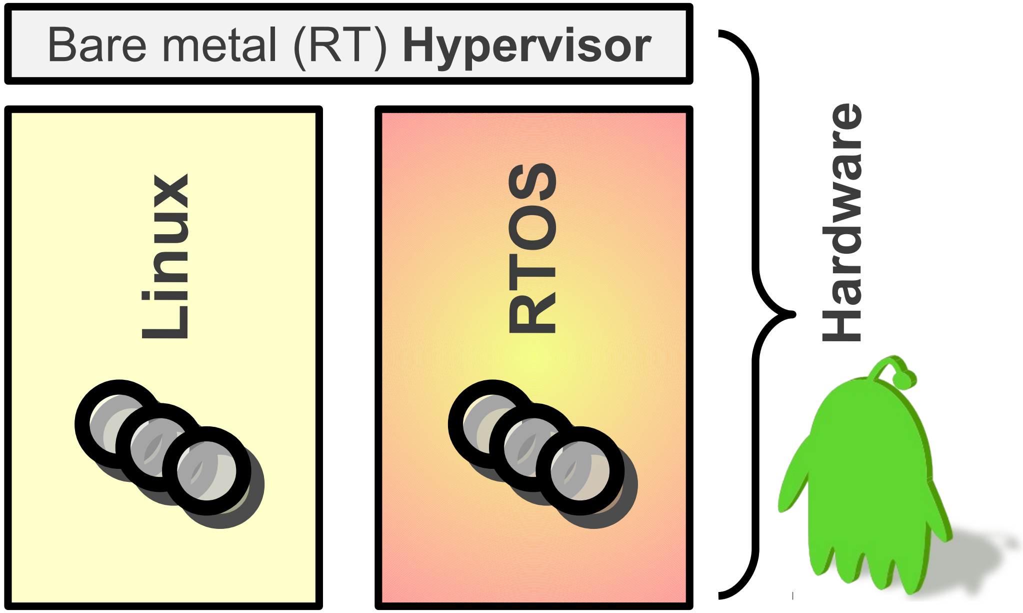


Virtual network communication

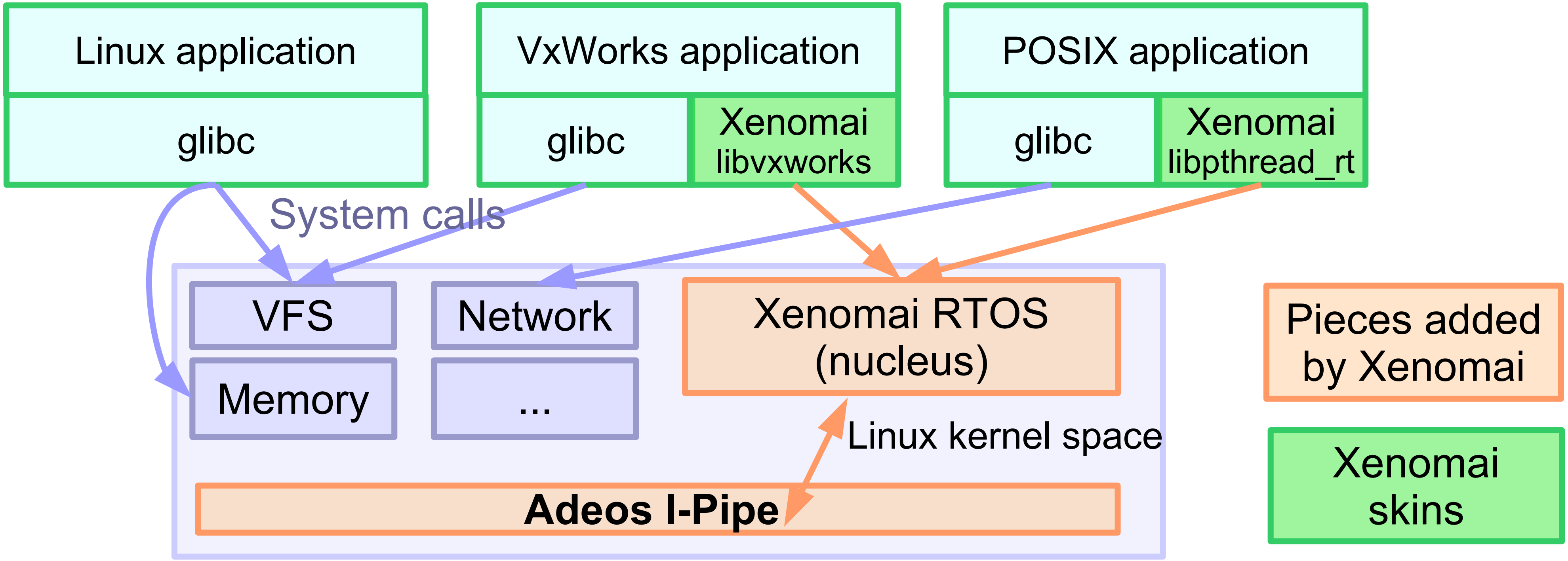
NOTE: We only focus on system virtual machine here.
Therefore, this presentation ignores Linux vserver, FreeBSD jail, etc.

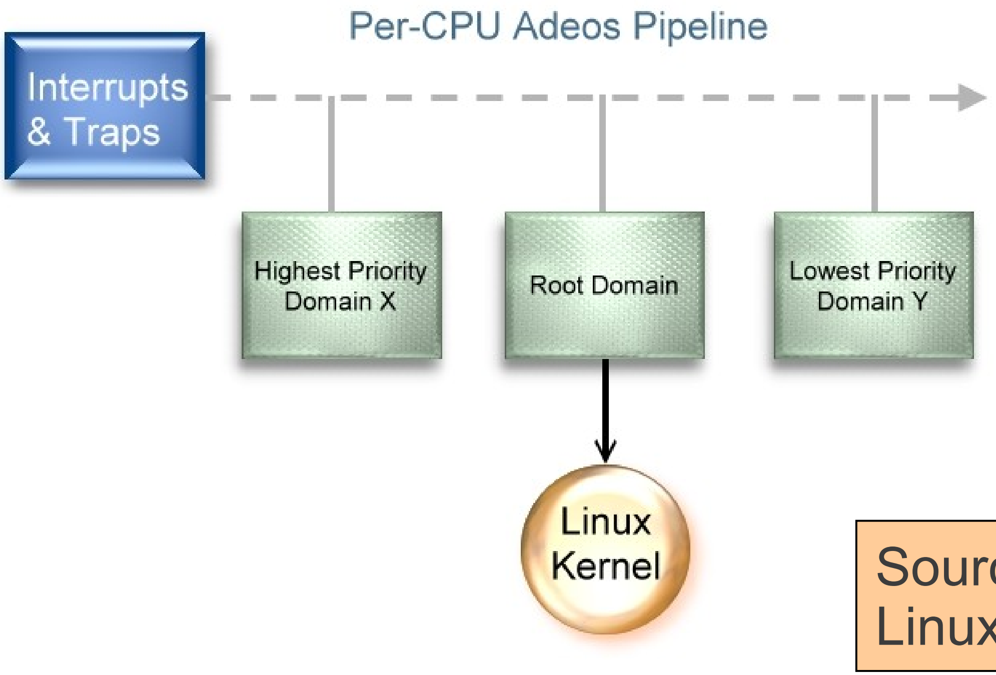
Virtualization is Common Technique

- Example: In the past, Linux is far from being real-time, but RTLinux/RTAI/Xenomai/Xtratum attempted to "improve" Linux by introducing new virtualization layer.
- real-time capable virtualization
- Dual kernel approach



Example: Xenomai (Linux Realtime Extension)


Linux application
glibc
VxWorks application
glibc
Xenomai libvxworks
POSIX application
glibc
Xenomai libpthread_rt
System calls
VFS
Network
Memory
...
Xenomai RTOS (nucleus)
Linux kernel space
Adeos I-Pipe
Pieces added by Xenomai
Xenomai skins

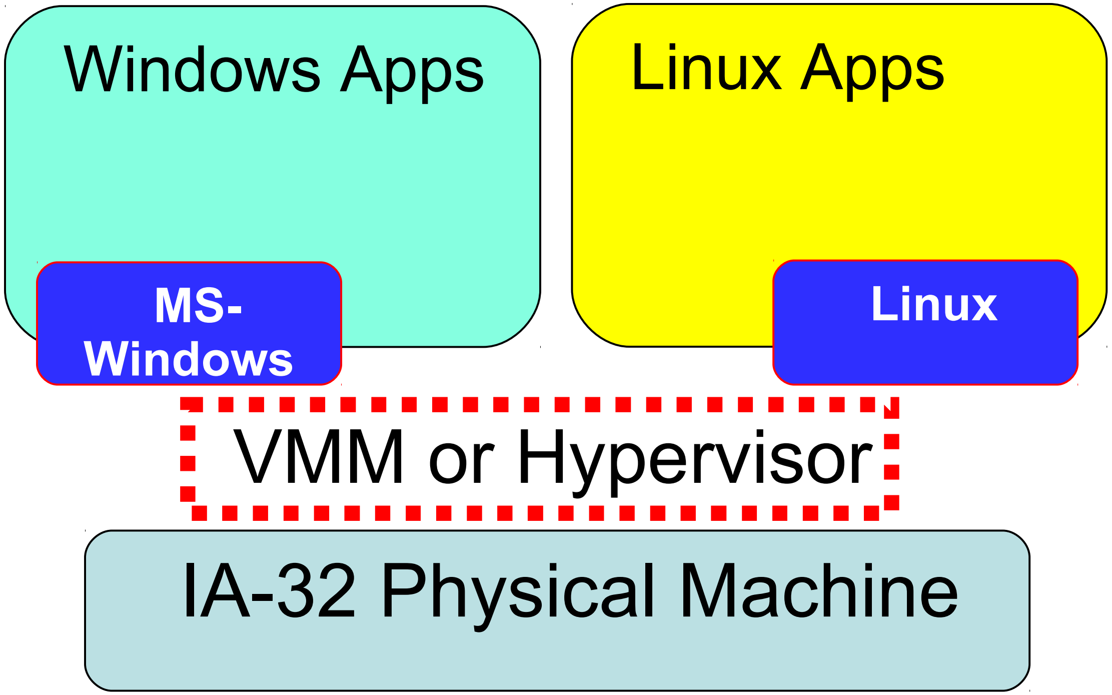


- From Adeos point of view, guest OSes are prioritized domains.
- For each event (interrupts, exceptions, syscalls, etc...), the various domains may handle the event or pass it down the pipeline.

Source: Real-time in embedded Linux systems, Free Electrons (2011)

General Classification of Virtualization technologies

- Type I
- Bare metal system VM



Windows Apps

Linux Apps

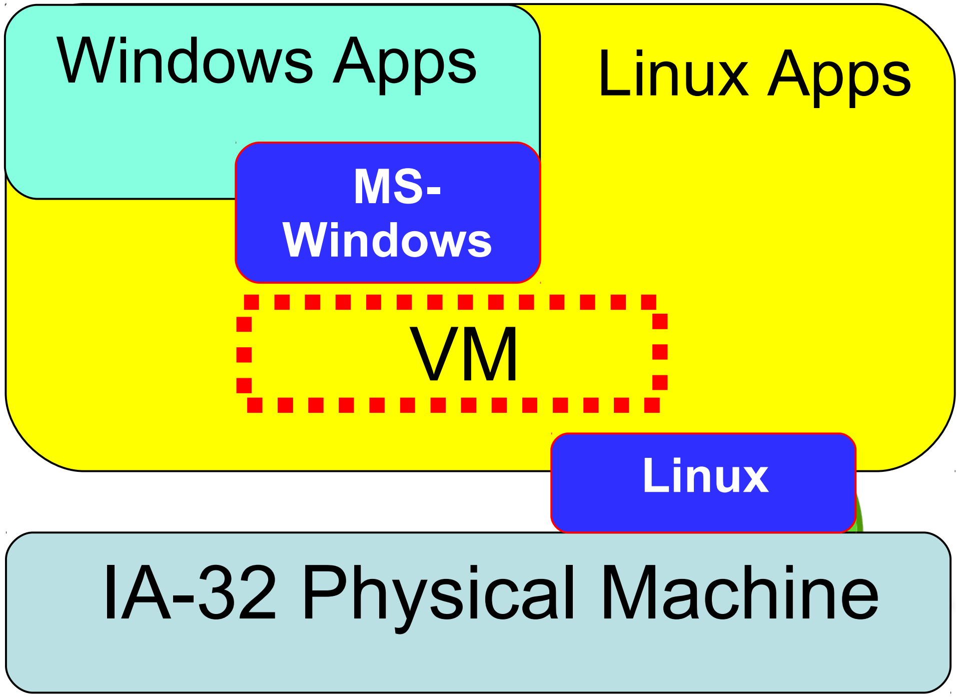
MS-Windows

Linux

VMM or Hypervisor

IA-32 Physical Machine

- Type 2
- Hosted System VM



Windows Apps

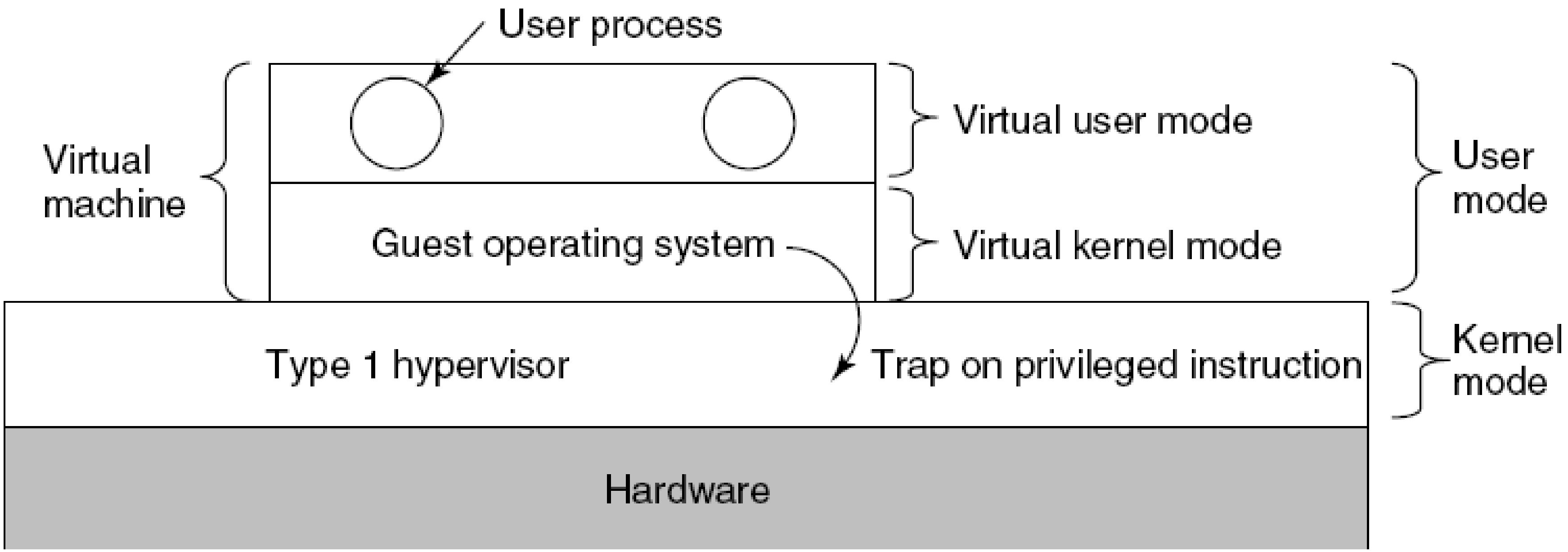
Linux Apps

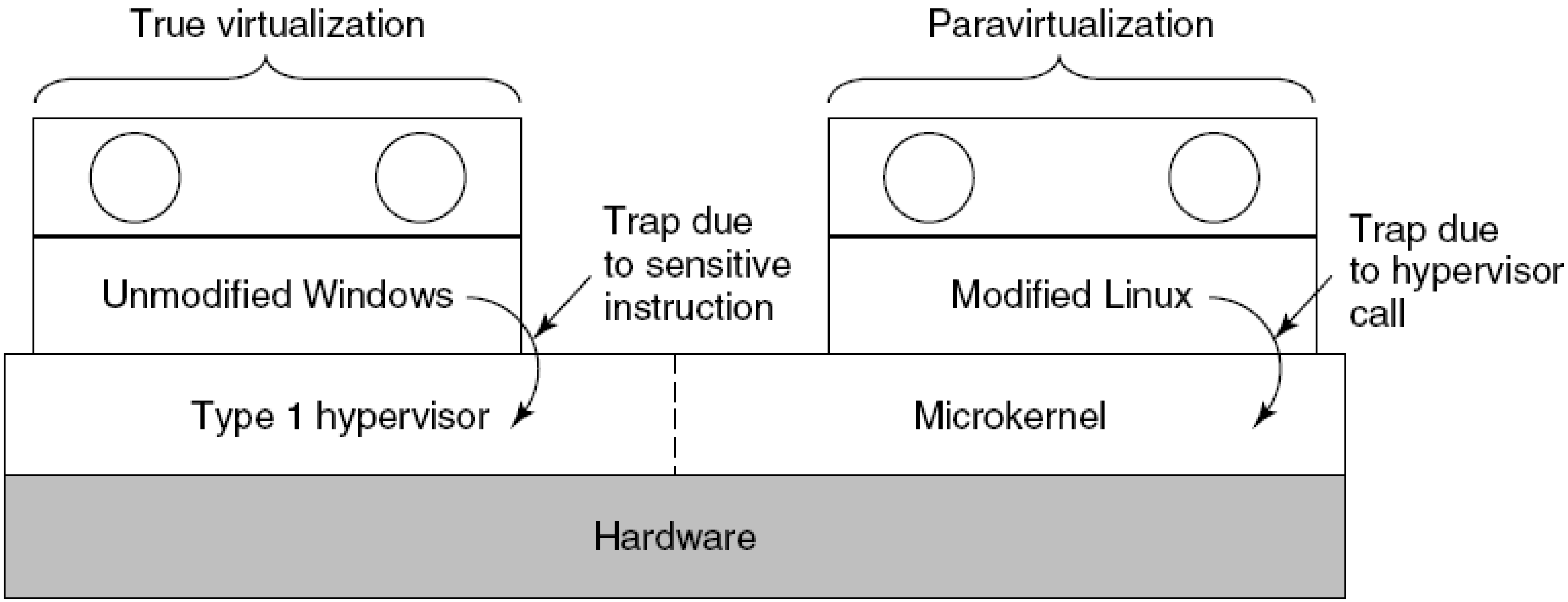
MS-Windows

VM

Linux

IA-32 Physical Machine





Source: Operating Systems Design – Lecture 21 Virtualization, Roy Campbell (Fall 2010)

Myth of Type I and Type II Hypervisor

source: http://gala4th.blogspot.com/2011/10/myth-of-type-i-and-type-ii-hypervisors.html

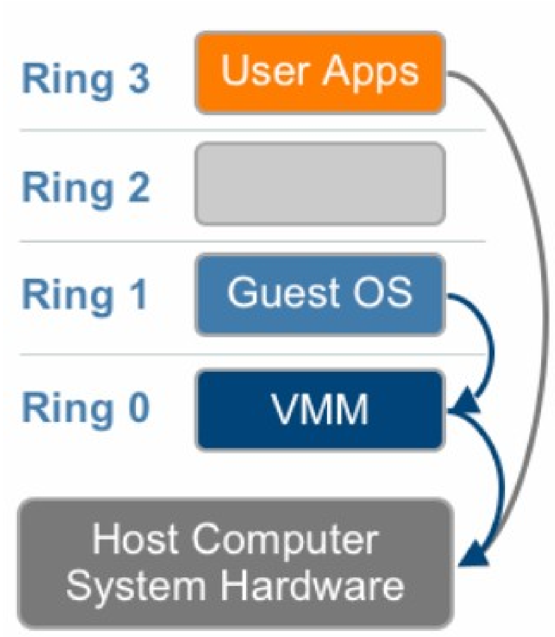
- Myth: *Type-2 is "lesser" than a true "type-1" hypervisor.*
- Virtualization theory really started with a paper from Gerald Popek and Robert Goldberg called *Formal Requirements for Virtualizable Third Generation Architectures*. (1974)
  - **Sensitive Instructions**

    might change the state of system resources
  - **Privileged Instructions**

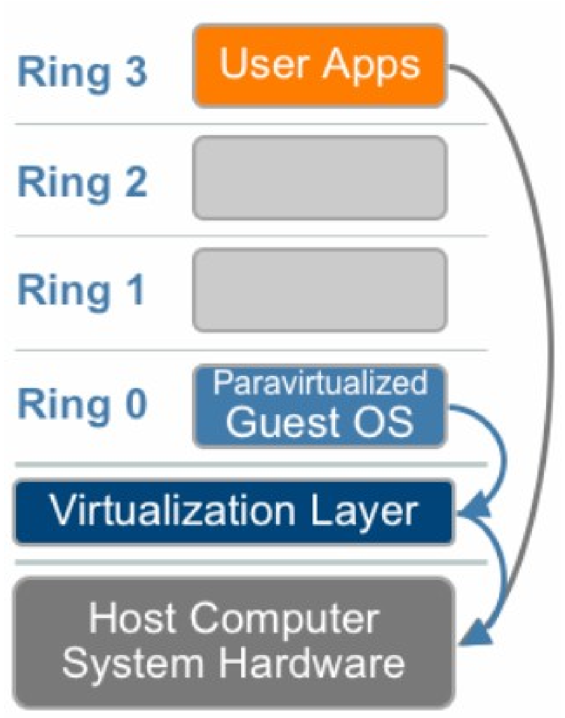
    must be executed with sufficient privilege
- The terms "type-1" and "type-2" originate from a paper by John Robin called *Analyzing the Intel Pentium's Capability to Support a Secure Virtual Machine Monitor*. (USENIX 2000)
  - Popek/Goldberg proof does not eliminate the possibility of using dynamic translation

System Virtualization Implementations

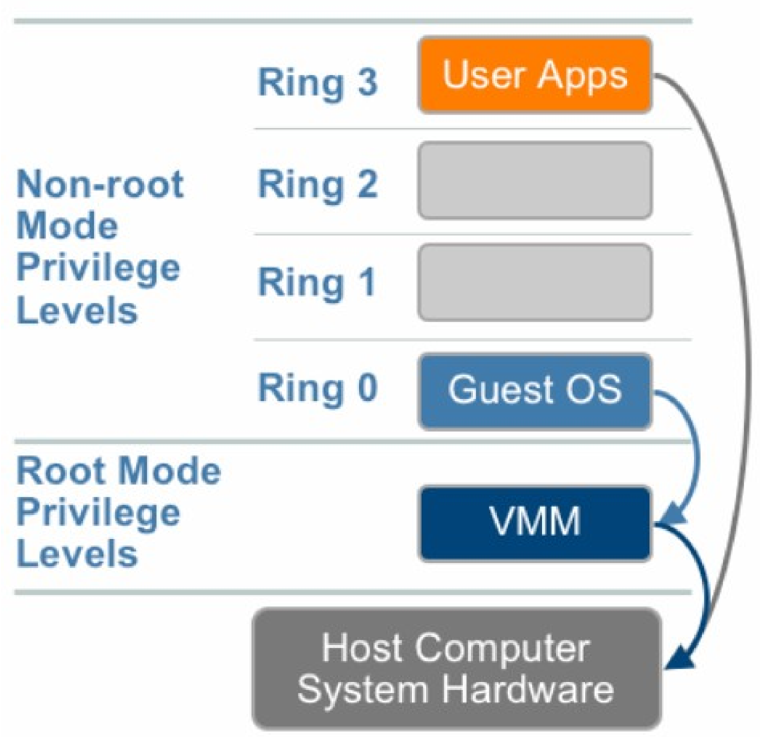
Full Virtualization



Para Virtualization

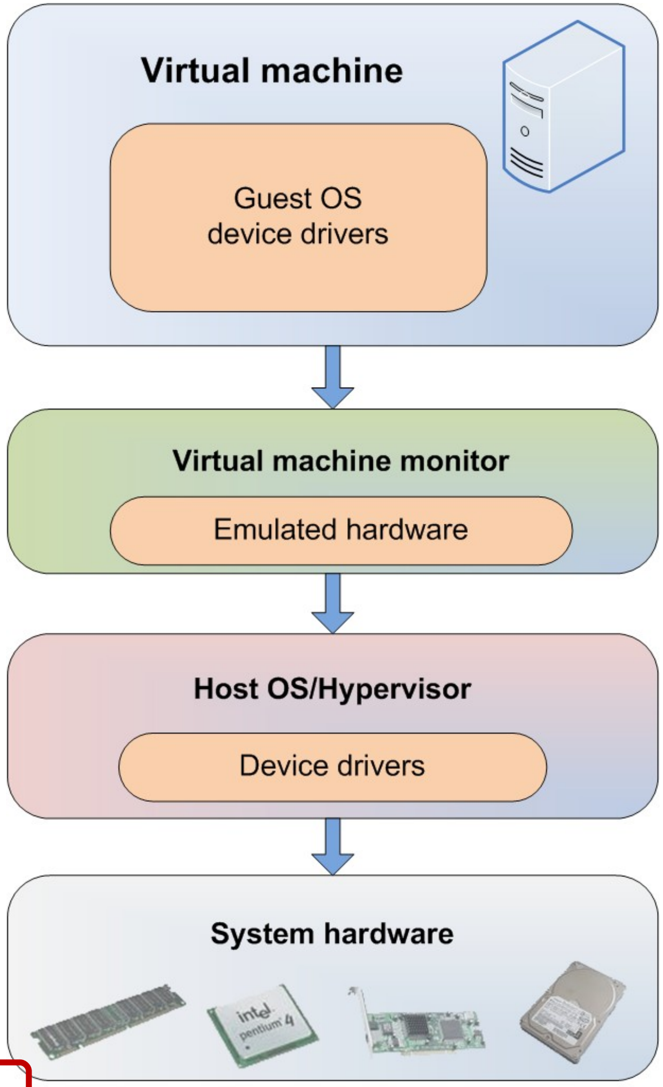


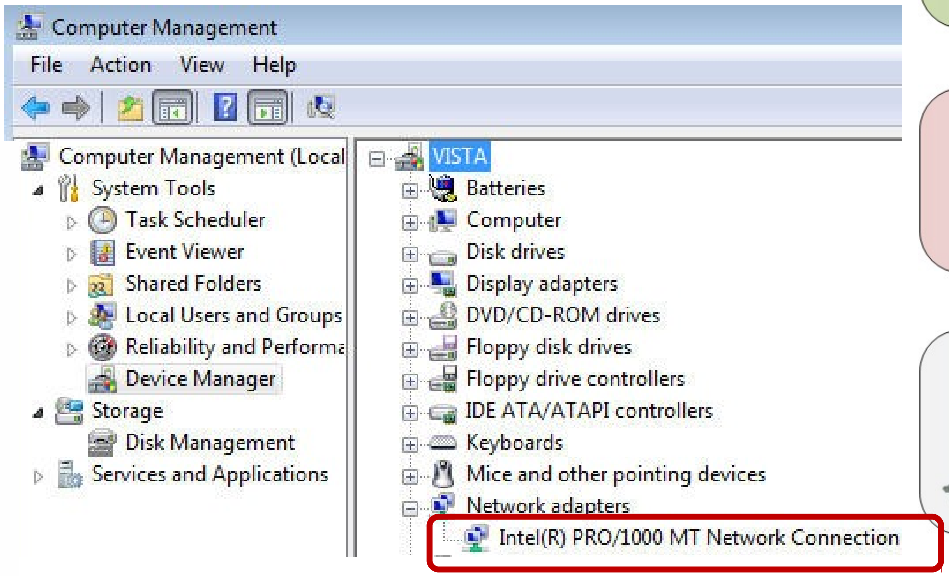
Hardware Assisted Virtualization


Non-root Mode Privilege Levels
Ring 3
User Apps
Ring 2
Ring 1
Ring 0
Guest OS
Root Mode Privilege Levels
VMM
Host Computer System Hardware

Full Virtualization

- Everything is virtualized
- Full hardware emulation
- Emulation = latency

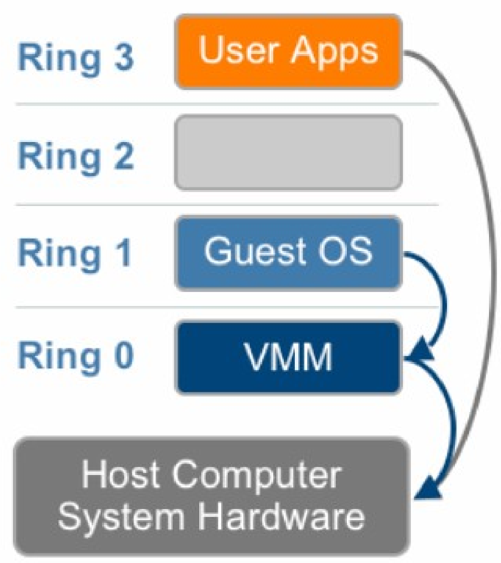



Computer Management

File Action View Help

```
Computer Management (Local
  System Tools
    Task Scheduler
    Event Viewer
    Shared Folders
    Local Users and Groups
    Reliability and Performa
    Device Manager
  Storage
    Disk Management
  Services and Applications
```

```
VISTA
  Batteries
  Computer
  Disk drives
  Display adapters
  DVD/CD-ROM drives
  Floppy disk drives
  Floppy drive controllers
  IDE ATA/ATAPI controllers
  Keyboards
  Mice and other pointing devices
  Network adapters
    Intel(R) PRO/1000 MT Network Connection
```

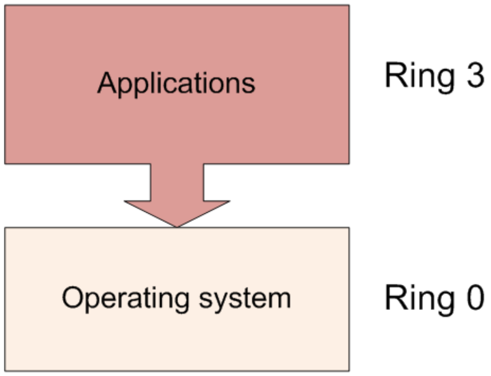


Definitions

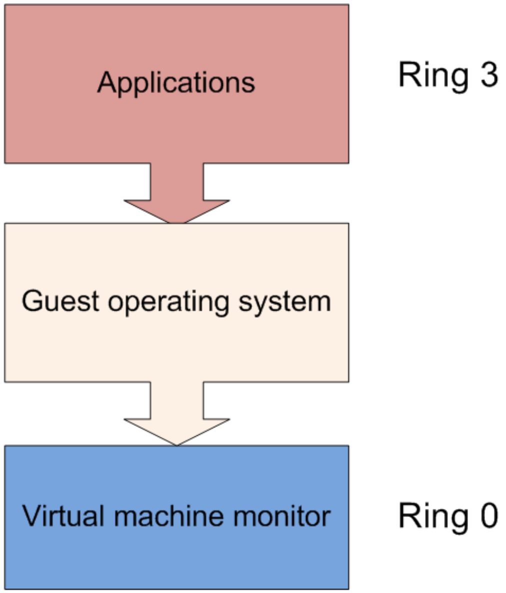
- **Sensitive Instructions** as defined by "*Formal Requirements for Virtualizable Third Generation Architectures*" (1974):
  - Mode Referencing Instructions
  - Sensitive Register/Memory Access Instructions
  - Storage Protection System Referencing Instructions
  - All I/O Instructions
- Theorem about strict **virtualizability** by "*Analyzing the Intel Pentium's Capability to Support a Secure Virtual Machine Monitor*" (2000):
  - For any conventional third generation computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.

Privileged Instructions

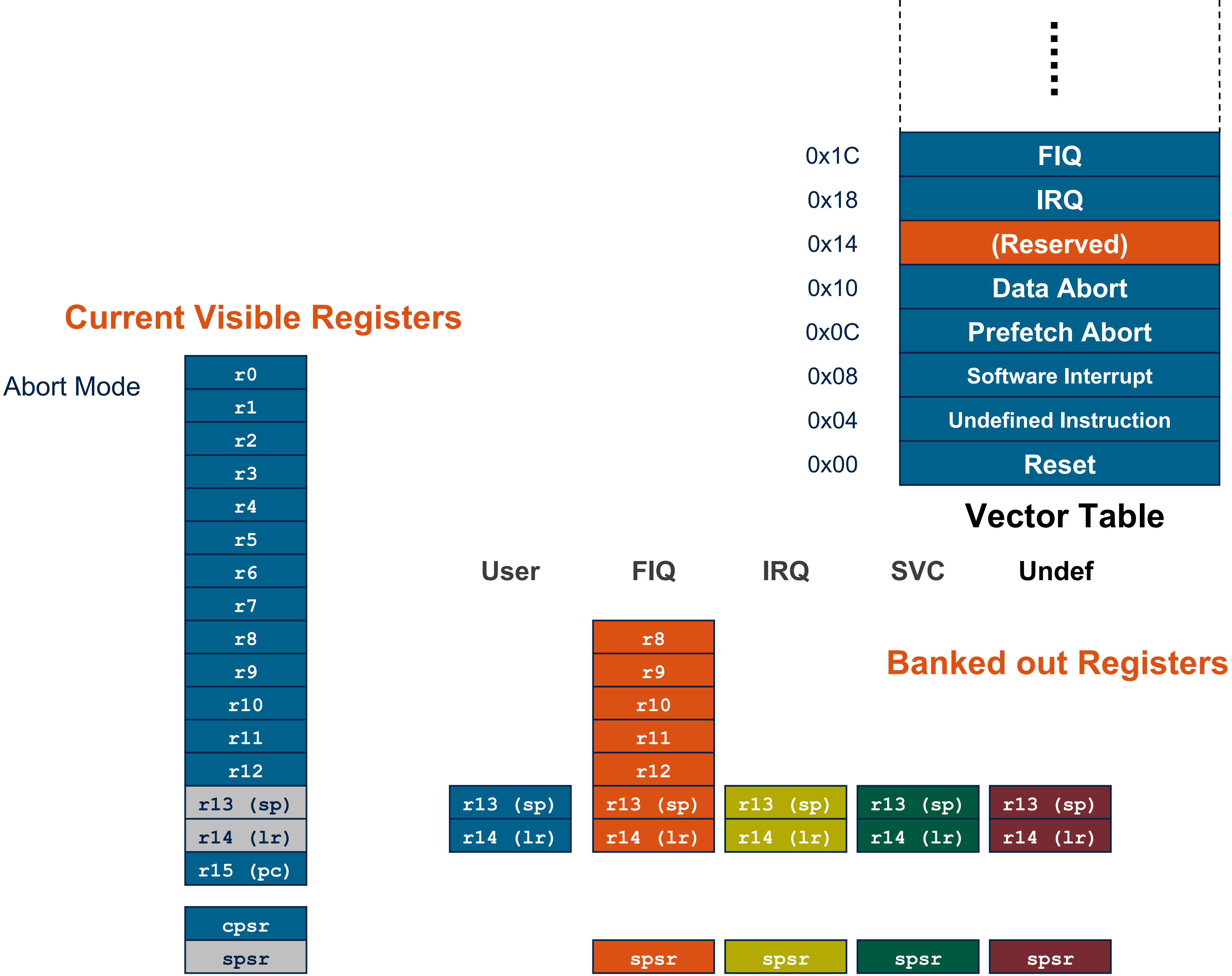
- Privileged instructions: OS kernel and device driver access to system hardware
- Trapped and emulated by VMM
  - Let VM execute most of its instructions directly on hardware
  - Except for some sensitive instructions that trap into the VMM and are emulated
  - Sensitive instructions are those that interfere with:
    - Correct emulation of the VM
    - Correct functioning of the VMM



Traditional x86 architecture



Full virtualization



0x1C	FIQ
0x18	IRQ
0x14	(Reserved)
0x10	Data Abort
0x0C	Prefetch Abort
0x08	Software Interrupt
0x04	Undefined Instruction
0x00	Reset

Vector Table

Current Visible Registers

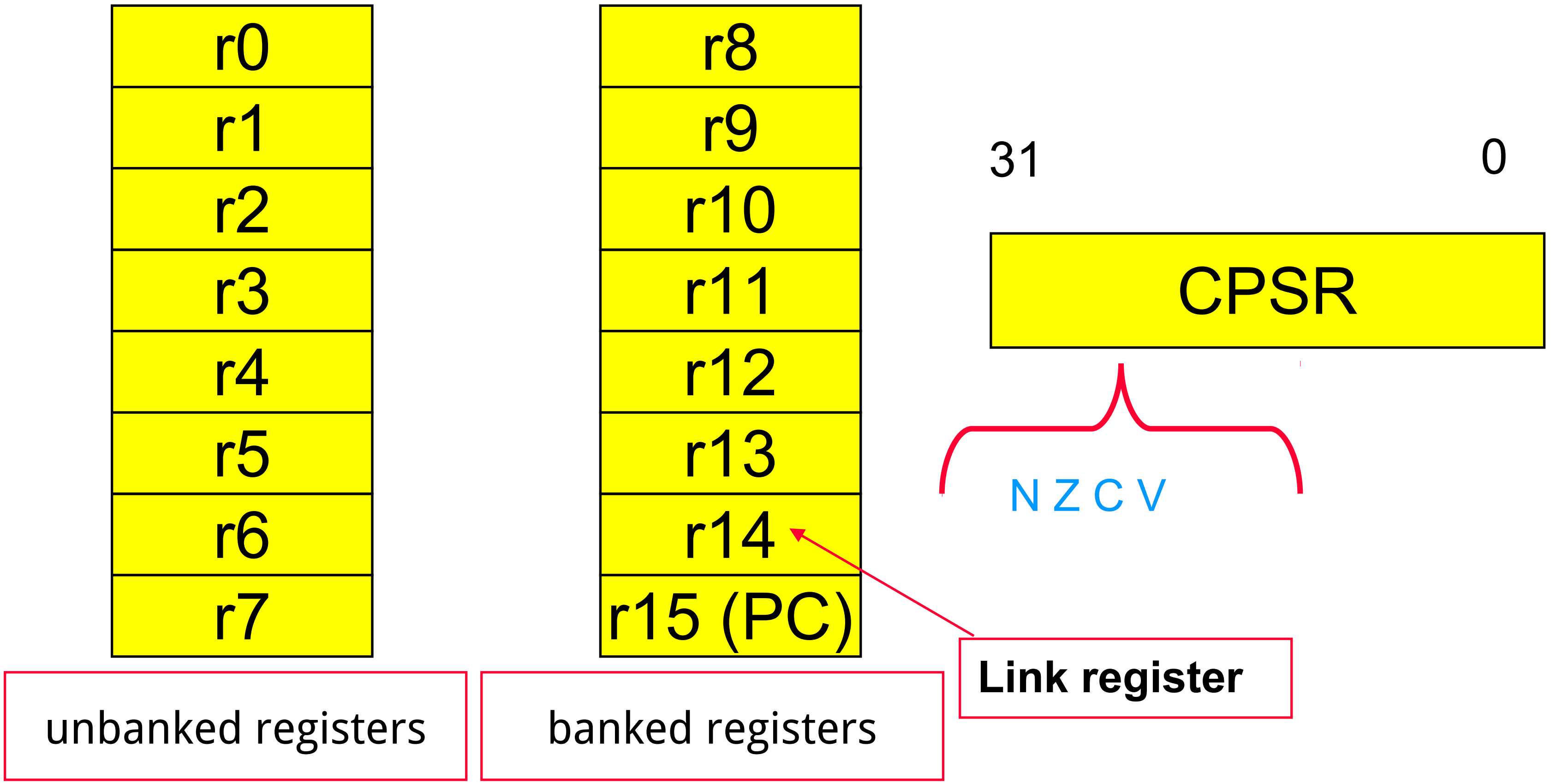
Abort Mode

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)

cpsr
spsr

Banked out Registers

User	FIQ	IRQ	SVC	Undef
	r8			
	r9			
	r10			
	r11			
	r12			
r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)
r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)
	spsr	spsr	spsr	spsr



r0
r1
r2
r3
r4
r5
r6
r7

r8
r9
r10
r11
r12
r13
r14
r15 (PC)

31 0

CPSR

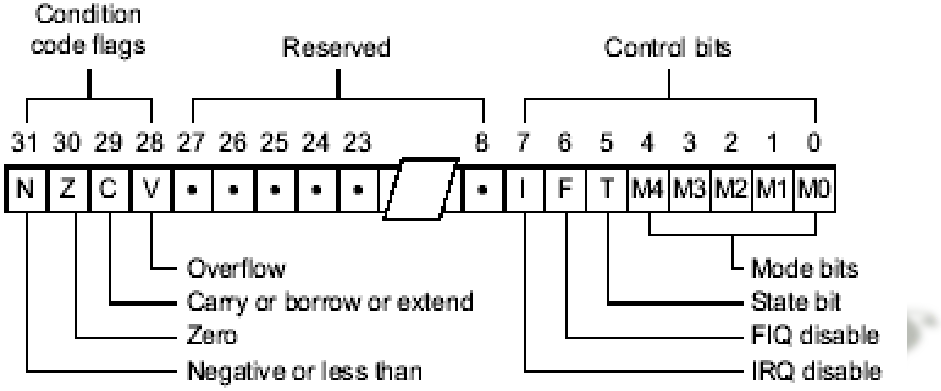
N Z C V

Link register

unbanked registers

banked registers

- Every arithmetic, logical, or shifting operation may set CPSR (*current program statues register*) bits:
  - N (negative), Z (zero), C (carry), V (overflow).
- Examples:
  - -1 + 1 = 0: NZCV = 0110.
  - $2^{31}-1+1 = -2^{31}$: NZCV = 0101.



ARM Architecture (armv4)

- 6 basic operating modes (1 user, 5 privileged)
- 37 registers, all 32 bits wide
  - 1 program counter
  - 5 dedicated saved program status registers
  - 1 Current program status register (PSR)
  - 30 general purpose registers
- Special usage
  - r13 (stack pointer)
  - r14 (link register)
  - r15 (program counter, PC)

Typical ARM instructions (armv4)

- branch and branch with Link (**B**, **BL**)
- data processing instructions (**AND**, **TST**, **MOV**, …)
- shifts: logical (**LSR**), arithmetic (**ASR**), rotate (**ROR**)
- test (**TEQ**, **TST**, **CMP**, **CMN**)
- processor status register transfer (**MSR**, **MRS**)
- memory load/store words (**LDR**, **STR**)
- push/pop Stack Operations (**STM**, **LDM**)
- software Interrupt (**SWI**; operating mode switch)
- co-processor (**CDP**, **LDC**, **STC**, **MRC**, **MCR**)

Problematic Instructions (1)

- Type 1

  Instructions which executed in user mode will cause **undefined instruction** exception
- Example

  MCR `p15, 0, r0, c2, c0, 0`

  Move r0 to c2 and c0 in coprocessor specified by p15 (co-processor) for operation according to option 0 and 0
  - MRC: from coproc to register
  - MCR: from register to coproc
- Problem:
  - Operand-dependent operation

Source: 前瞻資訊科技 – 虛擬化，薛智文，台大資訊所 (2011)

Problematic Instructions (2)

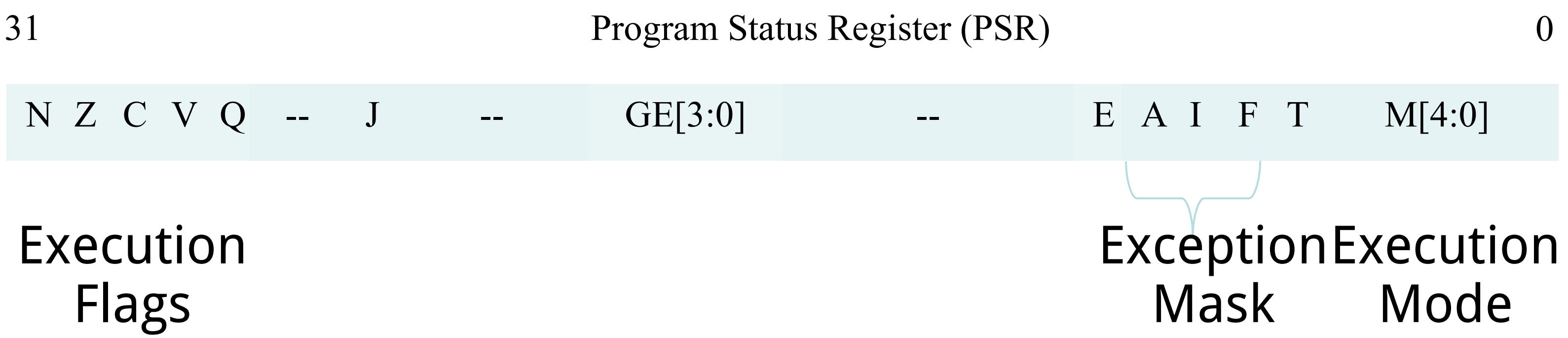
- Type 2

  Instructions which executed in user mode will have **no effect**

- Example

  ```
  MSR cpsr_c, #0xD3
  ```

  Switch to privileged mode and disable interrupt



31 Program Status Register (PSR) 0

N	Z	C	V	Q	--	J	--	GE[3:0]	--	E	A	I	F	T	M[4:0]

Execution Flags

Exception Mask

Execution Mode

Problematic Instructions (3)

- Type 3

  Instructions which executed in user mode will cause **unpredictable behaviors**.

- Example

  ```
  MOVS      PC, LR
  ```

  The return instruction

  changes the **program counter** and switches to **user mode**.

- This instruction causes unpredictable behavior when executed in user mode.

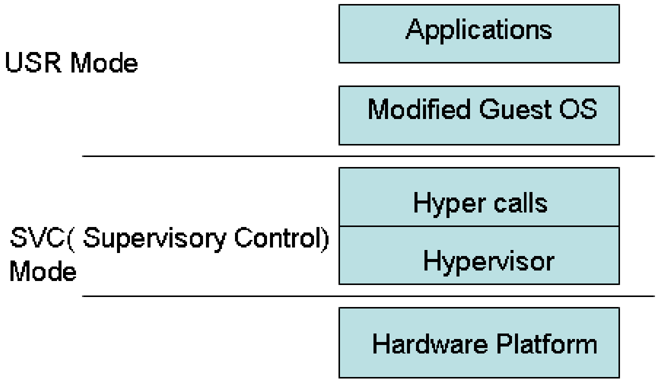
ARM Sensitive Instructions

- Coprocessor Access Instructions
  MRC / **MCR** / **CDP** / **LDC** / **STC**
- SIMD/VFP System Register Access Instructions
  VMRS / **VMSR**
- TrustZone Secure State Entry Instructions
  SMC
- Memory-Mapped I/O Access Instructions
  Load/Store instructions from/into memory-mapped I/O locations
- Direct (Explicit/Implicit) CPSR Access Instructions
  MRS / **MSR** / **CPS** / **SRS** / **RFE** / **LDM** (conditional execution) / **DPSPC**
- Indirect CPSR Access Instructions
  LDRT / **STRT** – Load/Store Unprivileged (“As User”)
- Banked Register Access Instructions
  LDM/**STM** (User mode registers)

Solutions to Problematic Instructions

[**Hardware** Techniques]

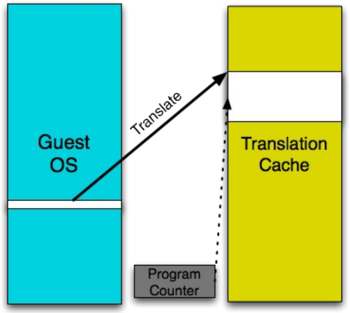
- Privileged Instruction Semantics dictated/translated by instruction set architecture
- MMU-enforced traps
  - Example: page fault
- Tracing/debug support
  - Example: **`bkpt`** (breakpoint)
- Hardware-assisted Virtualization
  - Example: extra privileged mode, HYP, in ARM Cortex-A15



Solutions to Problematic Instructions

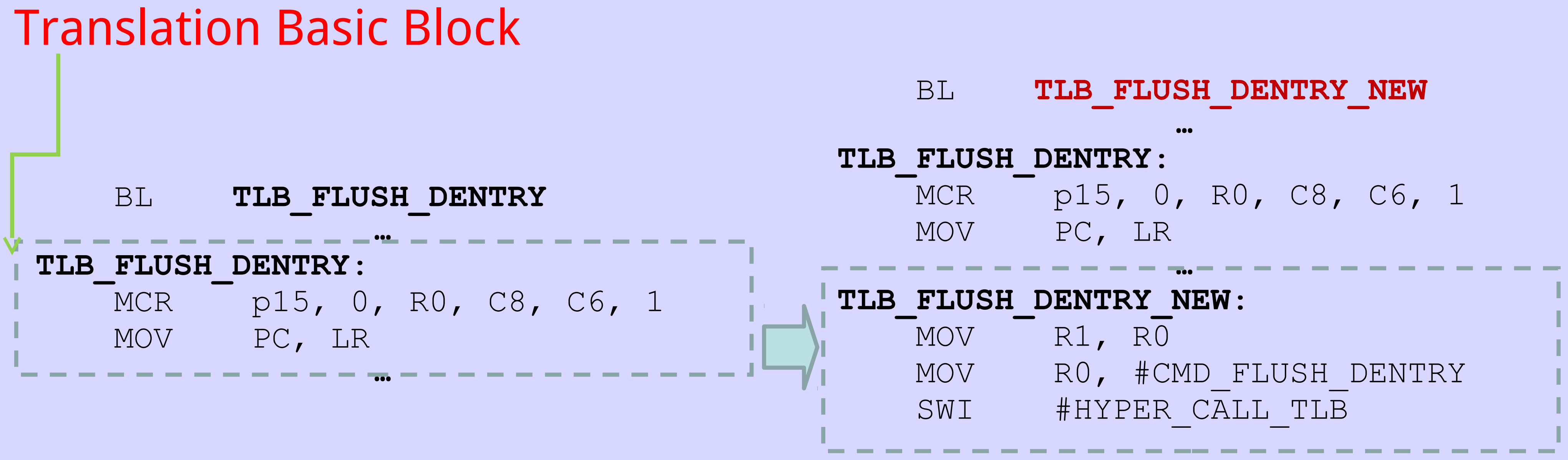
[Software Techniques]

Complexity	Binary translation	Hypercall
Design	**High**	**Low**
Implementation	**Medium**	**High**
Runtime	**High**	**Medium**
Mapped to programming languages	Virtual function	Normal function



Method: trap and emulate

Dynamic Binary Translation



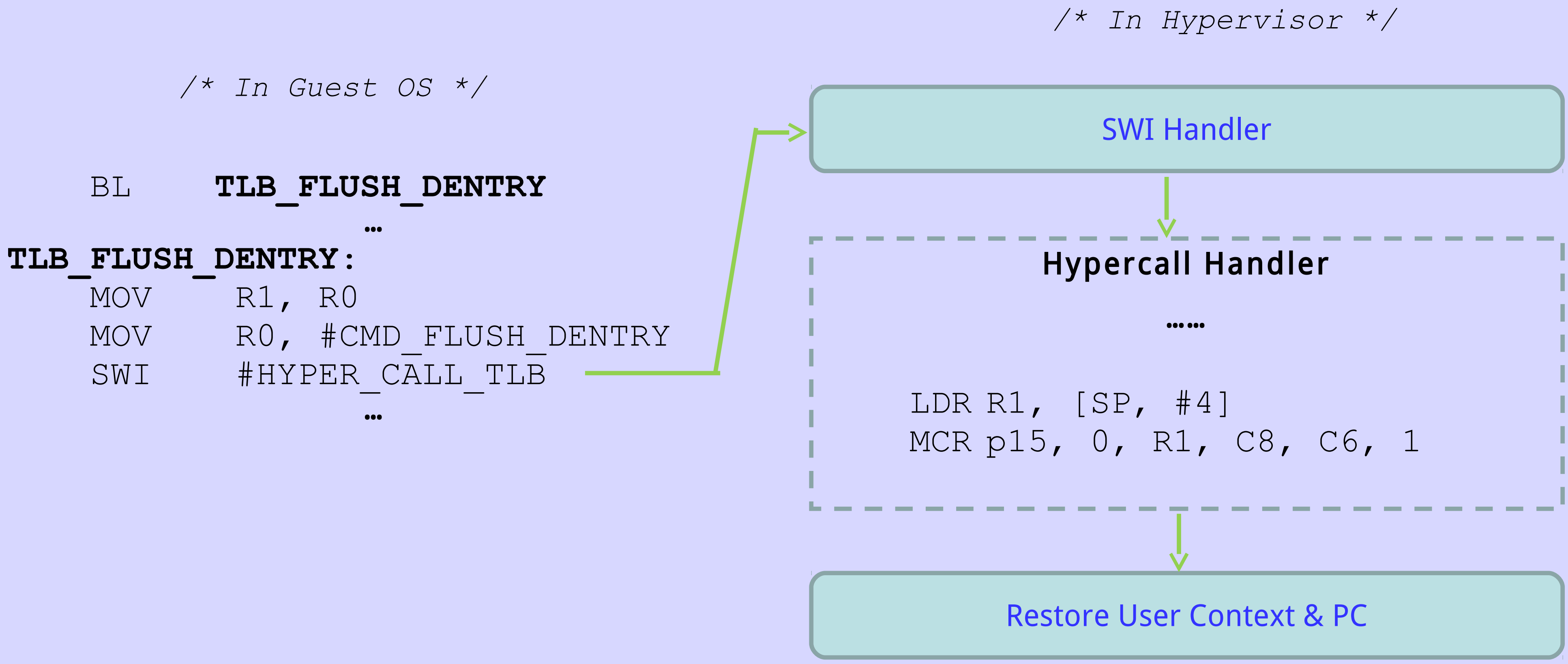
Translation Basic Block

```
    BL      TLB_FLUSH_DENTRY
            …
TLB_FLUSH_DENTRY:
    MCR     p15, 0, R0, C8, C6, 1
    MOV     PC, LR
            …
```

```
    BL      TLB_FLUSH_DENTRY_NEW
            …
TLB_FLUSH_DENTRY:
    MCR     p15, 0, R0, C8, C6, 1
    MOV     PC, LR
            …
TLB_FLUSH_DENTRY_NEW:
    MOV     R1, R0
    MOV     R0, #CMD_FLUSH_DENTRY
    SWI     #HYPER_CALL_TLB
```

- ARM has a fixed instruction size
  - 32-bit in ARM mode and 16-bit in Thumb mode
- Perform binary translation
  - Follow control-flow
  - Translate basic block (if not already translated) at the current PC
  - Ensure interposition at end of translated sequence
  - All writes (but not reads) to PC now become problematic instructions
  - Replace problematic instructions 1-1 with hypercalls to trap and emulate

Virtualization APIs – hypercalls



```
        /* In Guest OS */

    BL      TLB_FLUSH_DENTRY
                    …
TLB_FLUSH_DENTRY:
    MOV     R1, R0
    MOV     R0, #CMD_FLUSH_DENTRY
    SWI     #HYPER_CALL_TLB
                    …
```

```
/* In Hypervisor */
```

SWI Handler

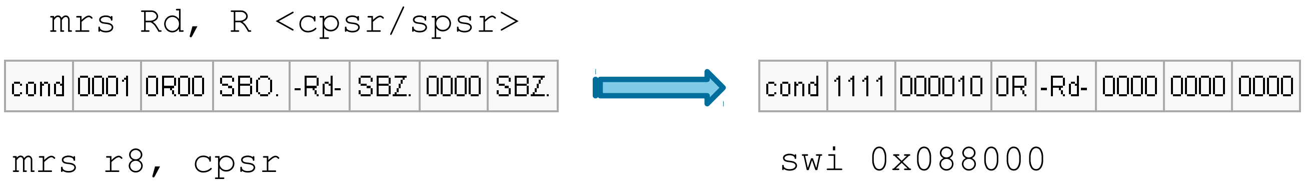
Hypercall Handler

……

```
LDR R1, [SP, #4]
MCR p15, 0, R1, C8, C6, 1
```

Restore User Context & PC

- Use trap instruction to issue hypercall
- Encode hypercall type and original instruction bits in hypercall hint
- Upon trapping into the VMM, decode the hypercall type and the original instruction bits, and emulate instruction semantics



```
mrs Rd, R <cpsr/spsr>
```

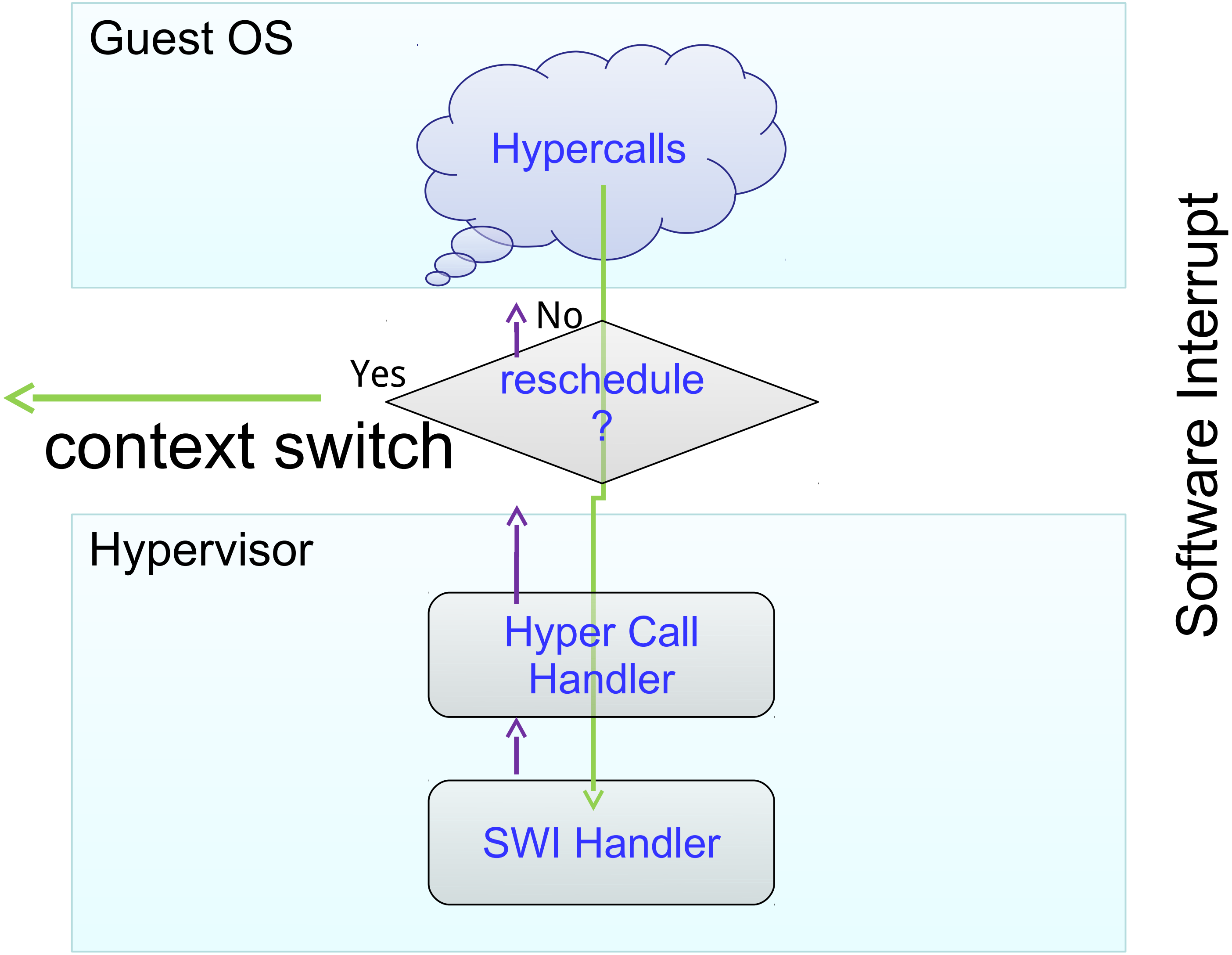
cond	0001	0R00	SBO.	-Rd-	SBZ.	0000	SBZ.

```
mrs r8, cpsr
```

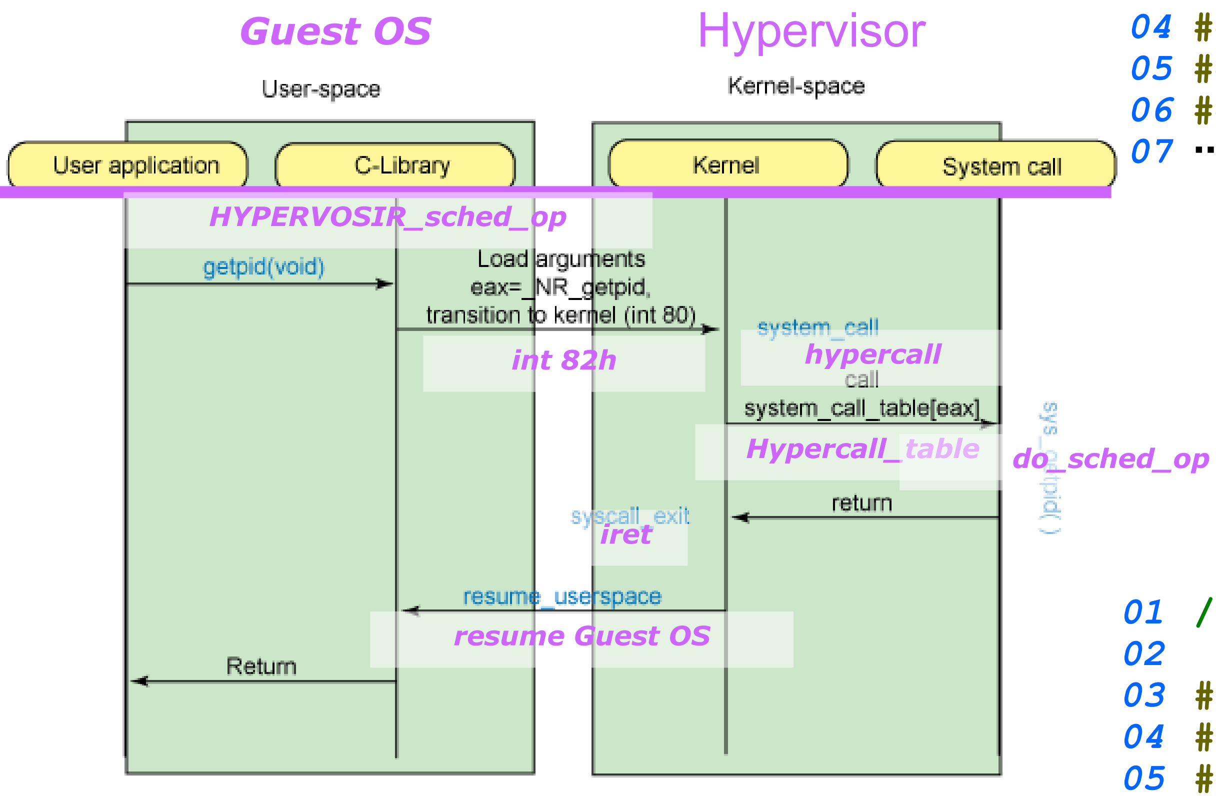
cond	1111	000010	0R	-Rd-	0000	0000	0000

```
swi 0x088000
```

Hypercall


Guest OS
Hypercalls
No
Yes
reschedule
?
context switch
Hypervisor
Hyper Call
Handler
SWI Handler
Software Interrupt

Hypercall in xen-i386



System Call int 0x80

```
// linux/include/asm/unistd.h

#define __NR_restart_syscall        0
#define __NR_exit                   1
#define __NR_fork                   2
#define __NR_read                   3
…
```

Hyper Call int 0x82

```
// xen/include/public/xen.h

#define __HYPERVISOR_set_trap_table 0
#define __HYPERVISOR_mmu_update     1
#define __HYPERVISOR_set_gdt        2
#define __HYPERVISOR_stack_switch   3
…
```

Case study: Xvisor-ARM

https://github.com/xvisor

- File: arch/arm/cpu/arm32/elf2cpatch.py
  - Script to generate cpatch script from guest OS ELF
- Functionality before generating the final ELF image
  - Each sensitive non-priviledged ARM instruction is converted to a hypercall.
  - Hypercall in ARM instruction set is **svc** <imm24> instruction.
  - Encode sensitive non-priviledged instructions in <imm24> operand of **svc** instruction. (software interrupt)
  - Each encoded instruction will have its own unique inst_id.
  - The inst_field for each encoded sensitive non-priviledged instruction will be diffrent.

How does Xvisor handle problematic instructions like MSR?

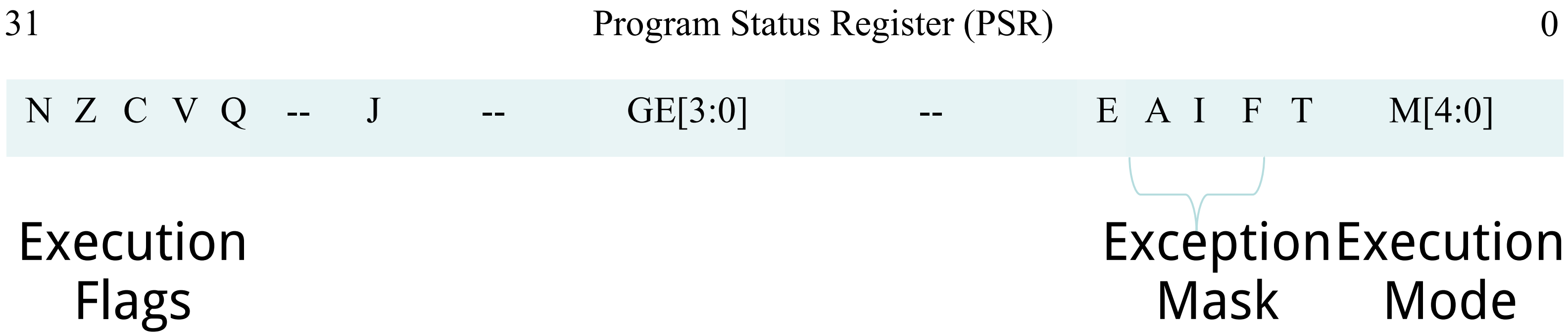
- Type 2

  Instructions which executed in user mode will have **no effect**

- Example

```
MSR cpsr_c, #0xD3
```

Switch to privileged mode and disable interrupt



31 Program Status Register (PSR) 0

N	Z	C	V	Q	--	J	--	GE[3:0]	--	E	A	I	F	T	M[4:0]

Execution Flags

Exception Mask

Execution Mode

First, cpatch (ELF patching tool) looks up the instructions...

```
MSR cpsr_c, #0xD3
```

Switch to privileged mode and disable interrupt

```
# MSR (immediate)
#       Syntax:
#               msr<c> <spec_reg>, #<const>
#       Fields:
#               cond = bits[31:28]
#               R = bits[22:22]
#               mask = bits[19:16]
#               imm12 = bits[11:0]
#       Hypercall Fields:
#               inst_cond[31:28] = cond
#               inst_op[27:24] = 0xf
#               inst_id[23:20] = 0
#               inst_subid[19:17] = 2
#               inst_fields[16:13] = mask
#               inst_fields[12:1] = imm12
#               inst_fields[0:0] = R
```

```
# MSR (immediate)
#       Syntax:
#               msr<c> <spec_reg>, #<const>
#       Fields:
#               cond = bits[31:28]
#               R = bits[22:22]
#               mask = bits[19:16]
#               imm12 = bits[11:0]
#       Hypercall Fields:
#               inst_cond[31:28] = cond
#               inst_op[27:24] = 0xf
#               inst_id[23:20] = 0
#               inst_subid[19:17] = 2
#               inst_fields[16:13] = mask
#               inst_fields[12:1] = imm12
#               inst_fields[0:0] = R
```

```
def convert_msr_i_inst(hxstr):
        hx = int(hxstr, 16)
        inst_id = 0
        inst_subid = 2
        cond = (hx >> 28) & 0xF
        R = (hx >> 22) & 0x1
        mask = (hx >> 16) & 0xF
        imm12 = (hx >> 0) & 0xFFF
        rethx = 0x0F000000
        rethx = rethx | (cond << 28)
        rethx = rethx | (inst_id << 20)
        rethx = rethx | (inst_subid << 17)
        rethx = rethx | (mask << 13)
        rethx = rethx | (imm12 << 1)
        rethx = rethx | (R << 0)
        return rethx
```

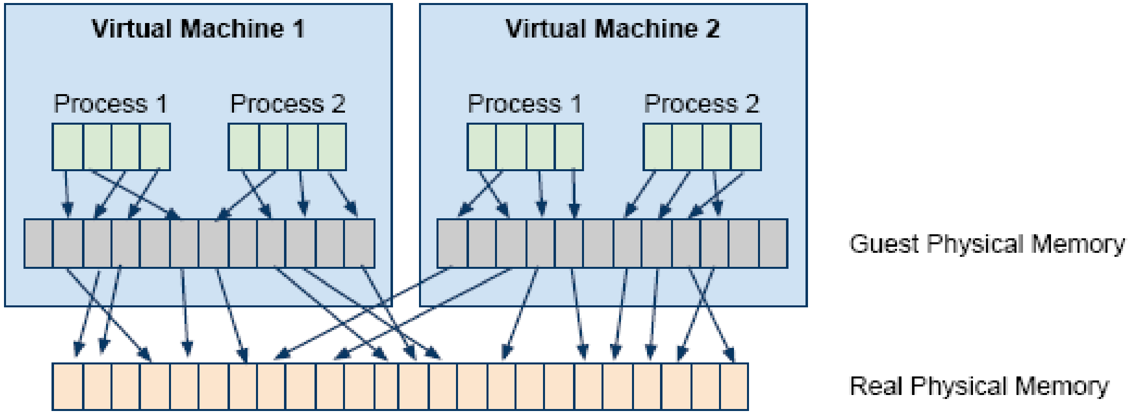
Xvisor utilizes cpatch to convert all problematic instructions for OS image files (ELF format).

Requirements of real Hypervisor

- VMM at higher privilege level than VMs
  - CPU Virtualization
  - Memory Virtualization
  - Device & I/O Virtualization
- User and System modes
- Privileged instructions only available in system mode
  - Trap to system if executed in user mode
- All physical resources only accessible using privileged instructions
  - Including page tables, interrupt controls, I/O registers

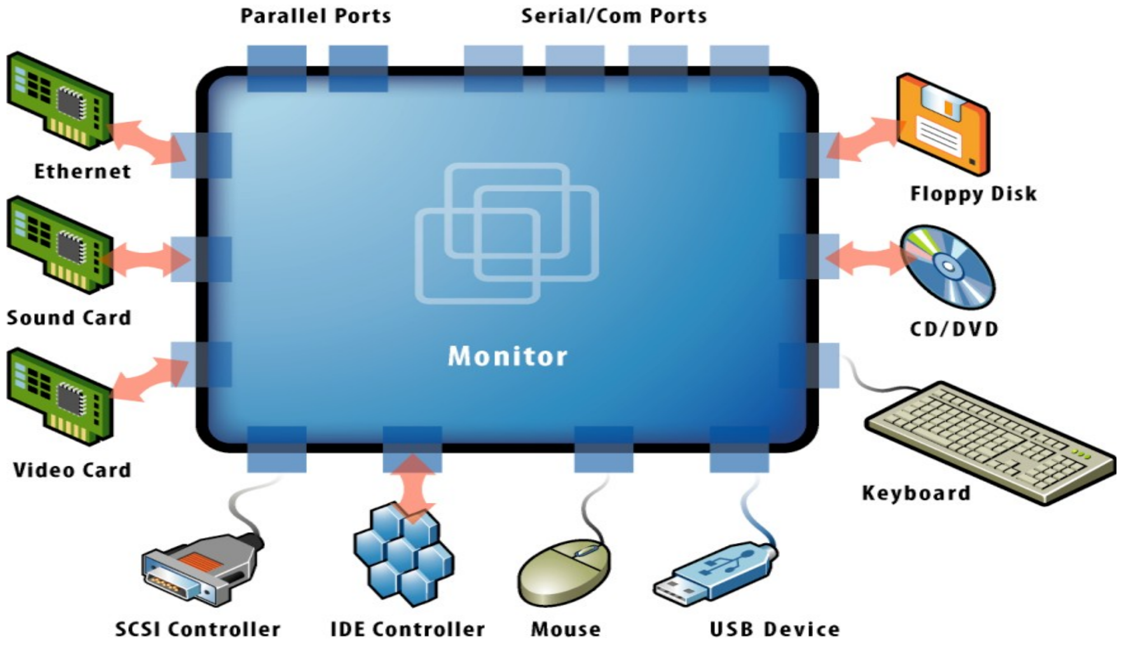
Memory Virtualization

- Deal with allocation of Physical memory among Guest OS
- RAM space shares among Guest OS
- Processors with Memory Virtualization support is expecting in 2nd generation processors (Intel VT and ARM Cortex-A15)



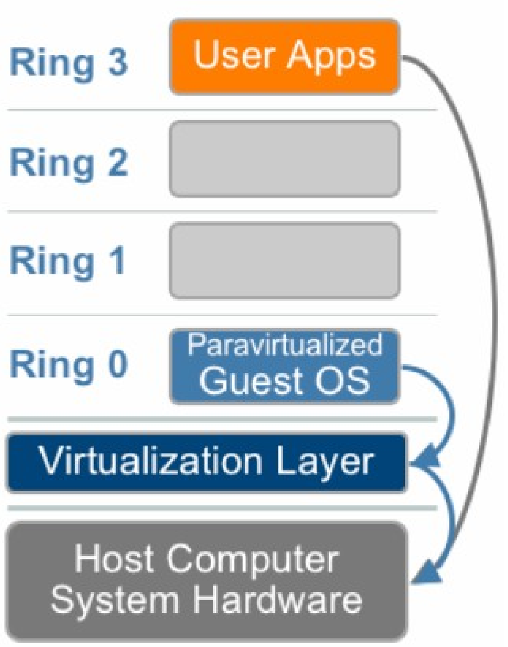
Device and I/O Virtualization

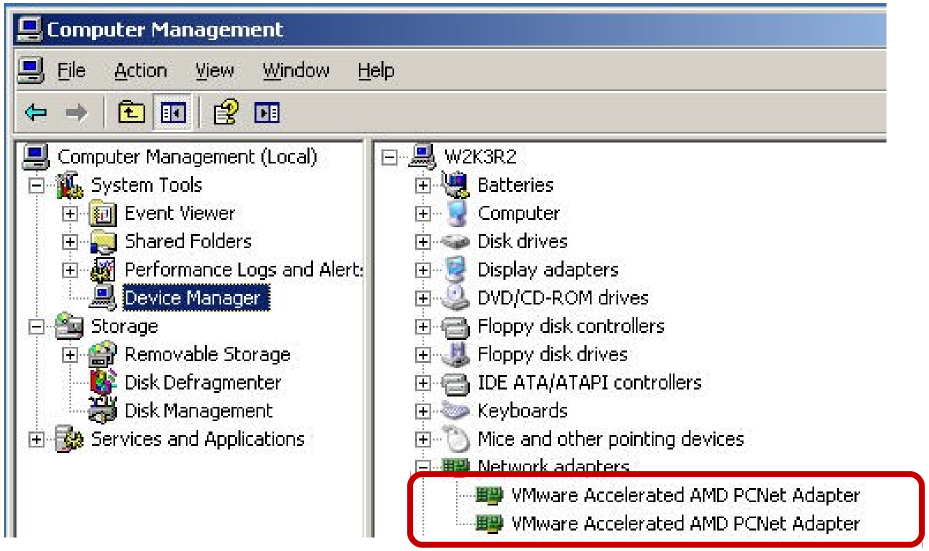
- Deal with routing of I/O requests between virtual devices and the shared physical hardware
- Similar to the single I/O device shared concurrently among different applications.
- Hypervisor virtualizes the physical hardware and present each virtual machine with a standard set of virtual devices



Paravirtualization

- OS or system devices are virtualization aware
- Requirements:
  - OS level – translated/modified kernel
  - Device level – paravirtualized or “enlightened” device drivers



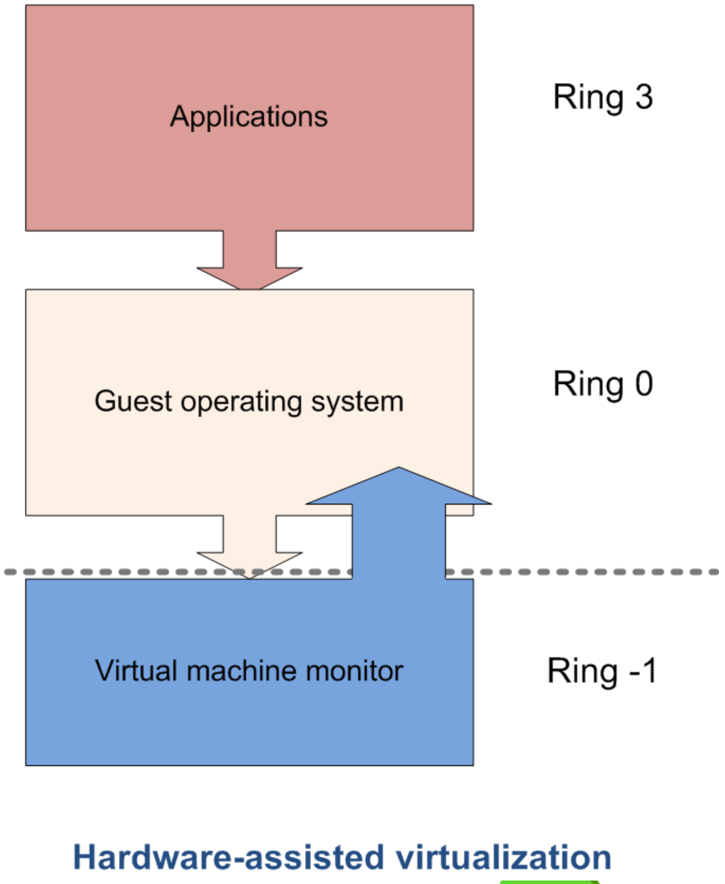


Computer Management (Local)
- System Tools
  - Event Viewer
  - Shared Folders
  - Performance Logs and Alert:
  - Device Manager
- Storage
  - Removable Storage
  - Disk Defragmenter
  - Disk Management
- Services and Applications

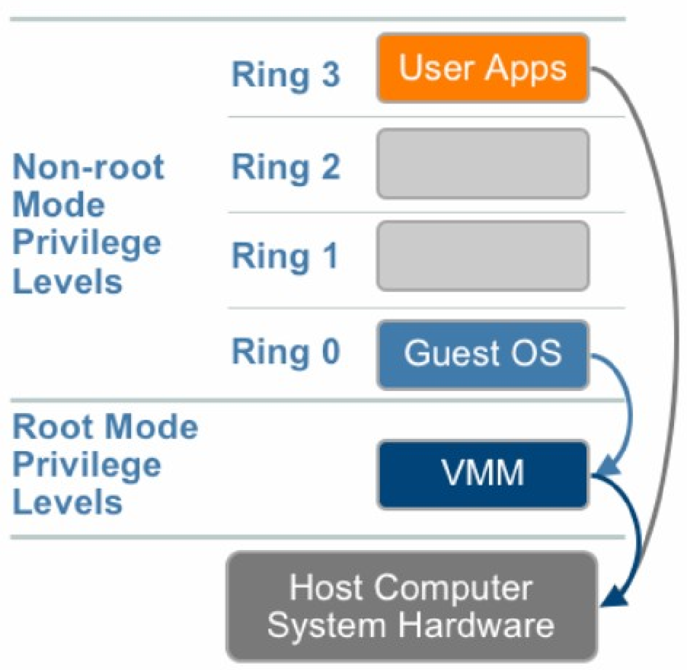
W2K3R2
- Batteries
- Computer
- Disk drives
- Display adapters
- DVD/CD-ROM drives
- Floppy disk controllers
- Floppy disk drives
- IDE ATA/ATAPI controllers
- Keyboards
- Mice and other pointing devices
- Network adapters
  - VMware Accelerated AMD PCNet Adapter
  - VMware Accelerated AMD PCNet Adapter

Hardware-assisted Virtualization

- Hardware is virtualization aware
- Hypervisor and VMM load at Ring -1
- Remove CPU emulation bottleneck
- Provides address bus isolation



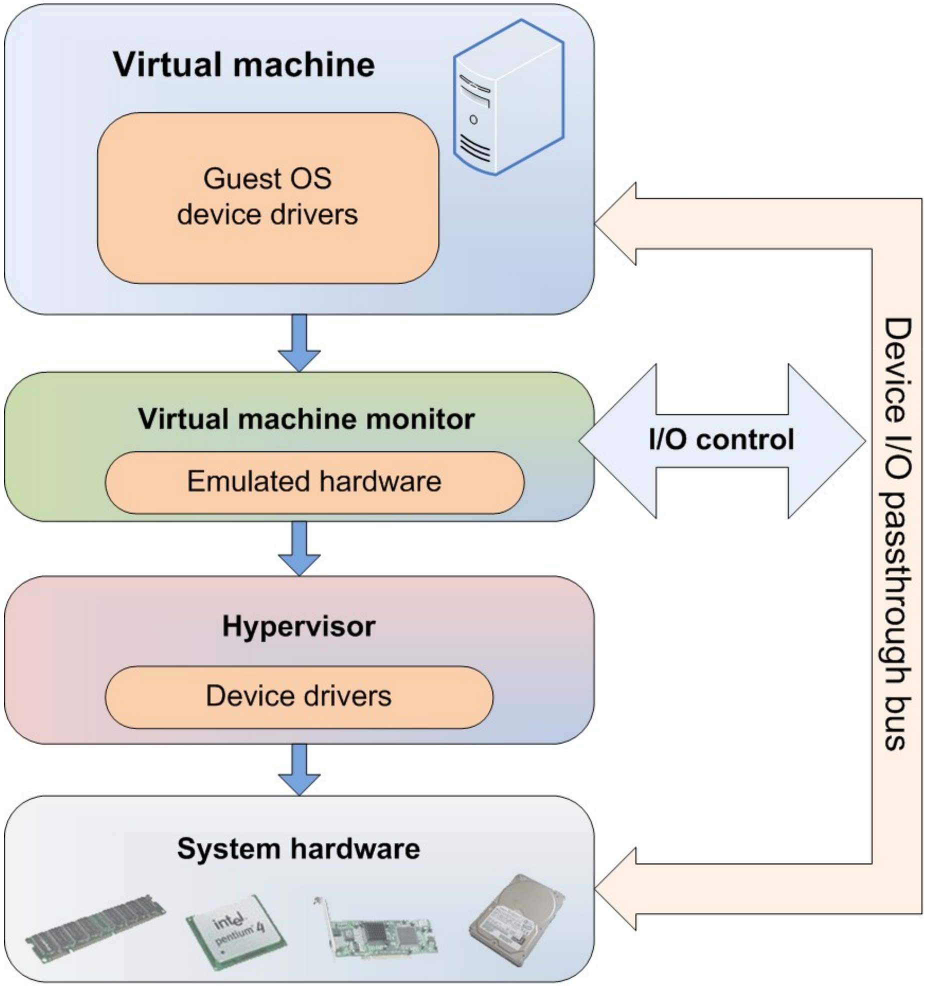
Hardware-assisted virtualization


Ring 3
User Apps
Non-root Mode Privilege Levels
Ring 2
Ring 1
Ring 0
Guest OS
Root Mode Privilege Levels
VMM
Host Computer System Hardware

Hardware-assisted Virtualization

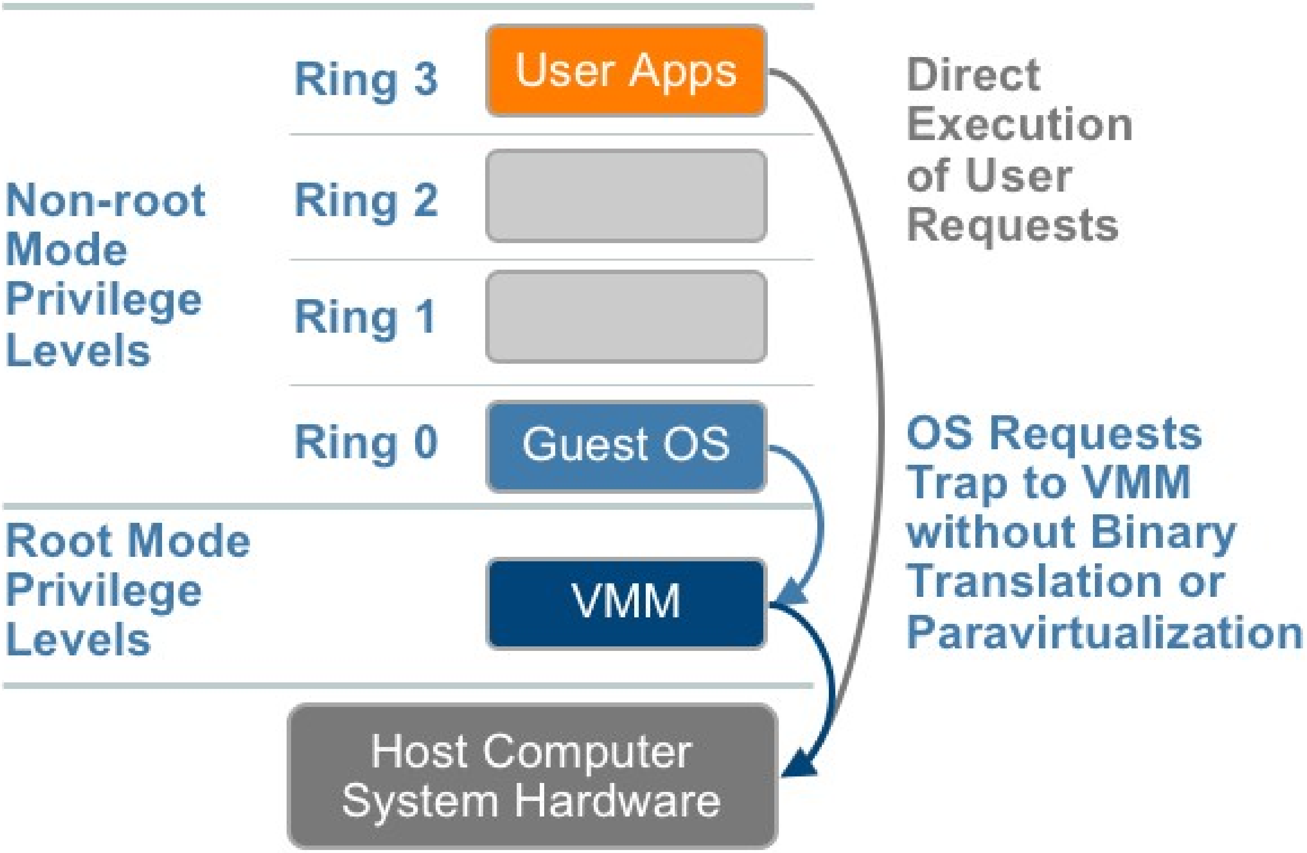
- VMM coordinates direct hardware access
- Memory virtualization solved in 2nd generation hardware assisted platforms
- Passthrough I/O has limited use cases without IOV (I/O Virtualization)

  http://www.pcisig.com/specifications/iov/



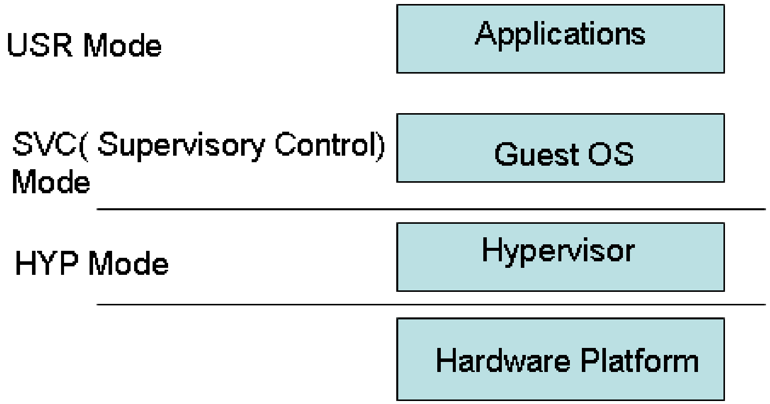
Hardware-assisted Virtualization in x86

- VT technology enables new execution mode (VMX-Root Mode in x86 by Intel) in the processors to support virtualization
- Hypervisor runs in a root mode below Ring0
- OS requests trap VMM without binary translation or PV
- Specialized Hardware support is required
- A special CPU privileged mode is to be selected to support


Ring 3
User Apps
Direct Execution of User Requests
Non-root Mode Privilege Levels
Ring 2
Ring 1
Ring 0
Guest OS
OS Requests Trap to VMM without Binary Translation or Paravirtualization
Root Mode Privilege Levels
VMM
Host Computer System Hardware

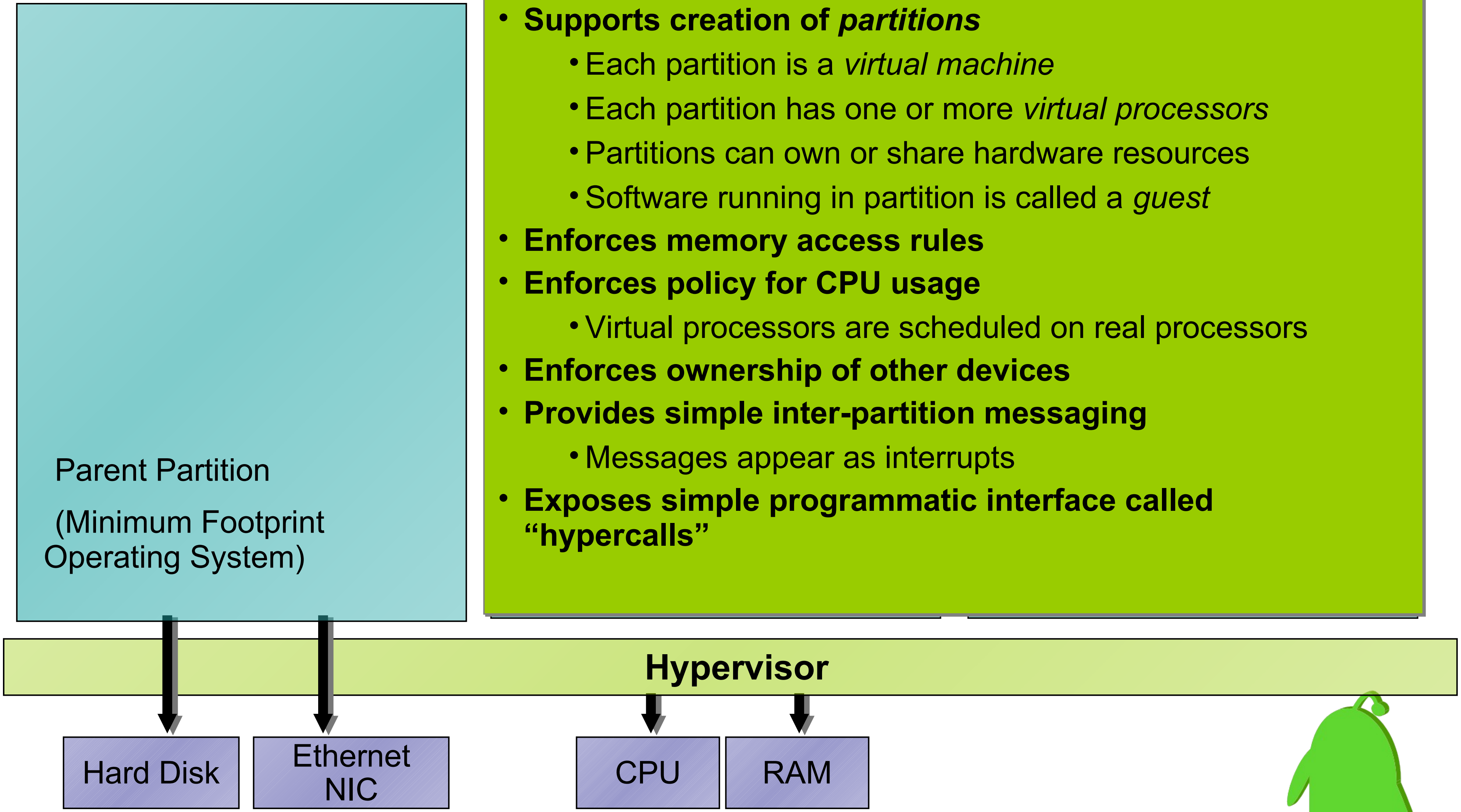
Hardware-assisted Virtualization in ARM

- Enable new execution mode Hypervisor (HYP)
- Hypervisor runs in a Hypervisor (HYP) mode
- Guest OS Runs in Supervisory Control (SVC) mode
- Applications runs in User (USR) mode



Details will be discussed in section "Toward ARM Cortex-A15"

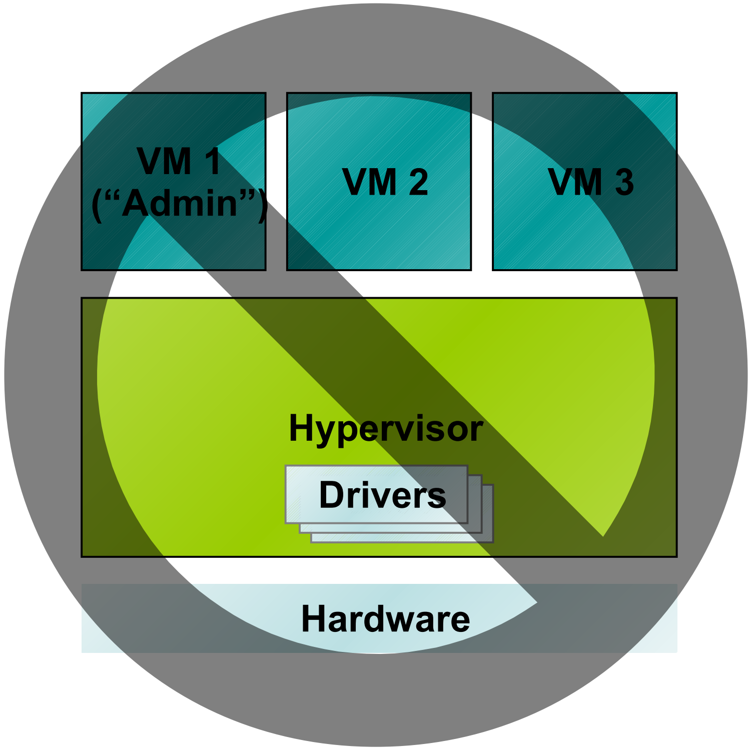
What does Hypervisor looks like



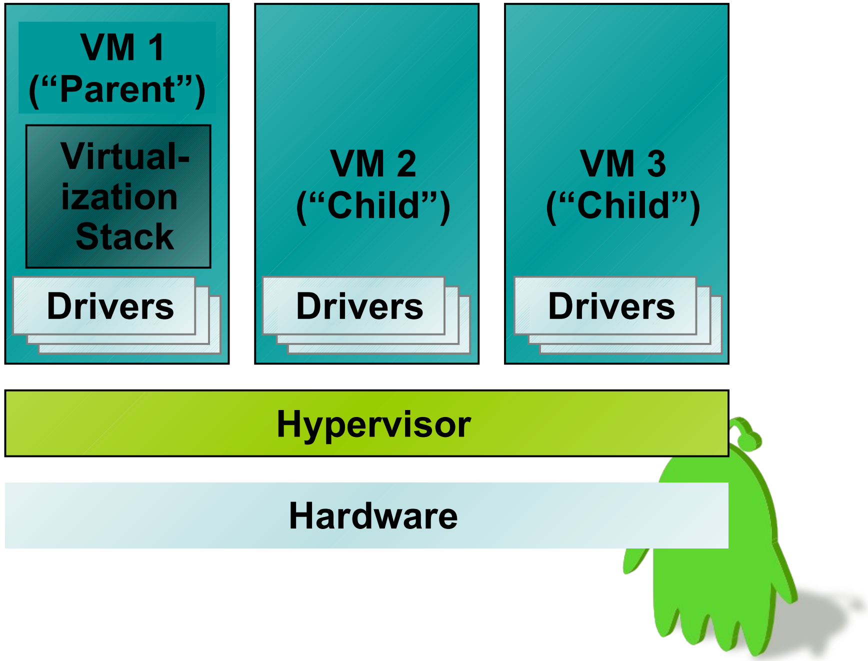
- **Thin layer of software running on the hardware**
- **Supports creation of *partitions***
  - Each partition is a *virtual machine*
  - Each partition has one or more *virtual processors*
  - Partitions can own or share hardware resources
  - Software running in partition is called a *guest*
- **Enforces memory access rules**
- **Enforces policy for CPU usage**
  - Virtual processors are scheduled on real processors
- **Enforces ownership of other devices**
- **Provides simple inter-partition messaging**
  - Messages appear as interrupts
- **Exposes simple programmatic interface called “hypercalls”**

Monolithic vs. Microkernel

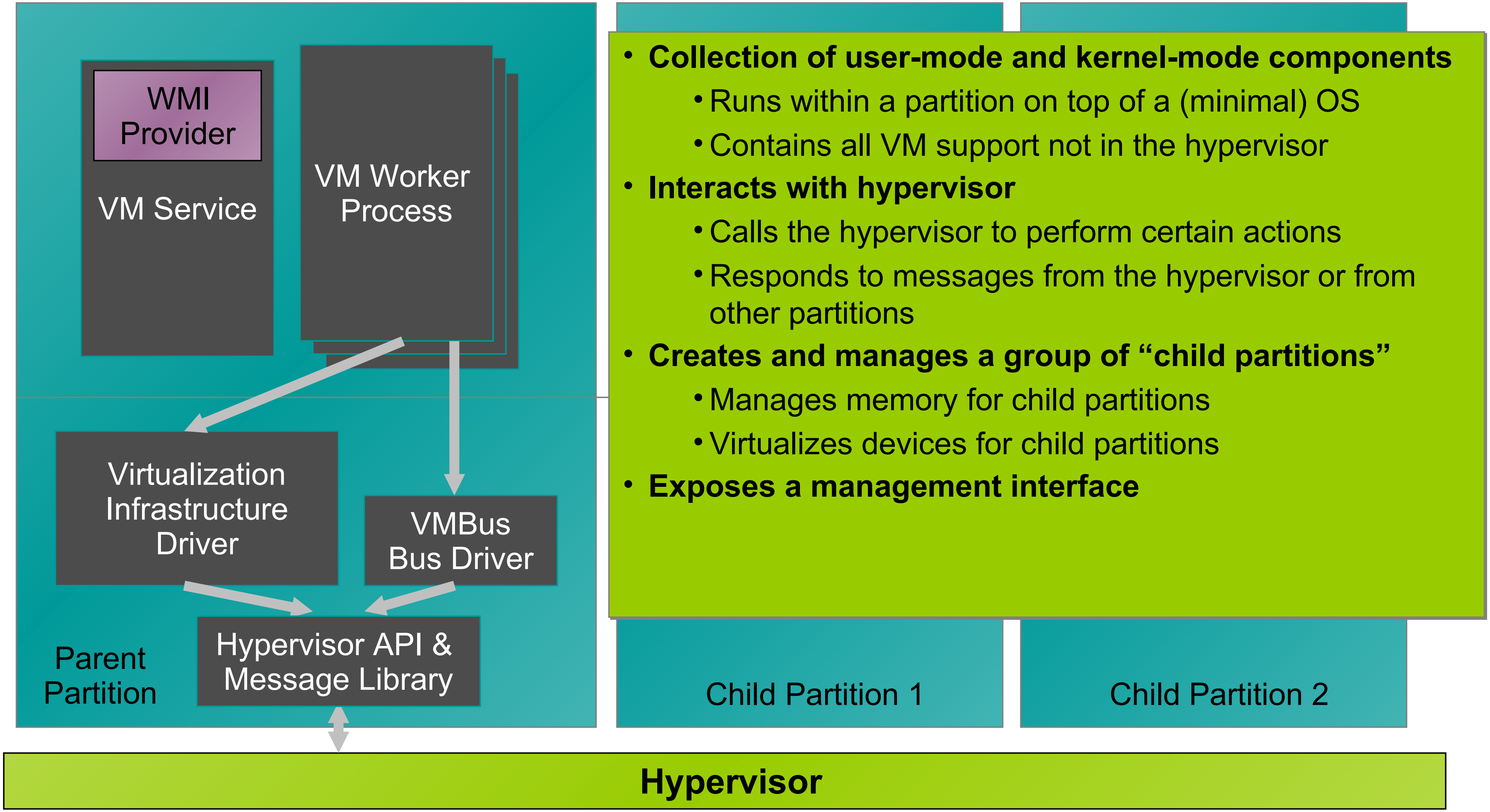
- Monolithic hypervisor
  - Simpler than a modern kernel, but still complex
  - Contains its own drivers model



- Microkernel based hypervisor
  - Simple partitioning functionality
  - Increase reliability and minimize TCB
  - No third-party code
  - Drivers run within guests



Virtualization Stack

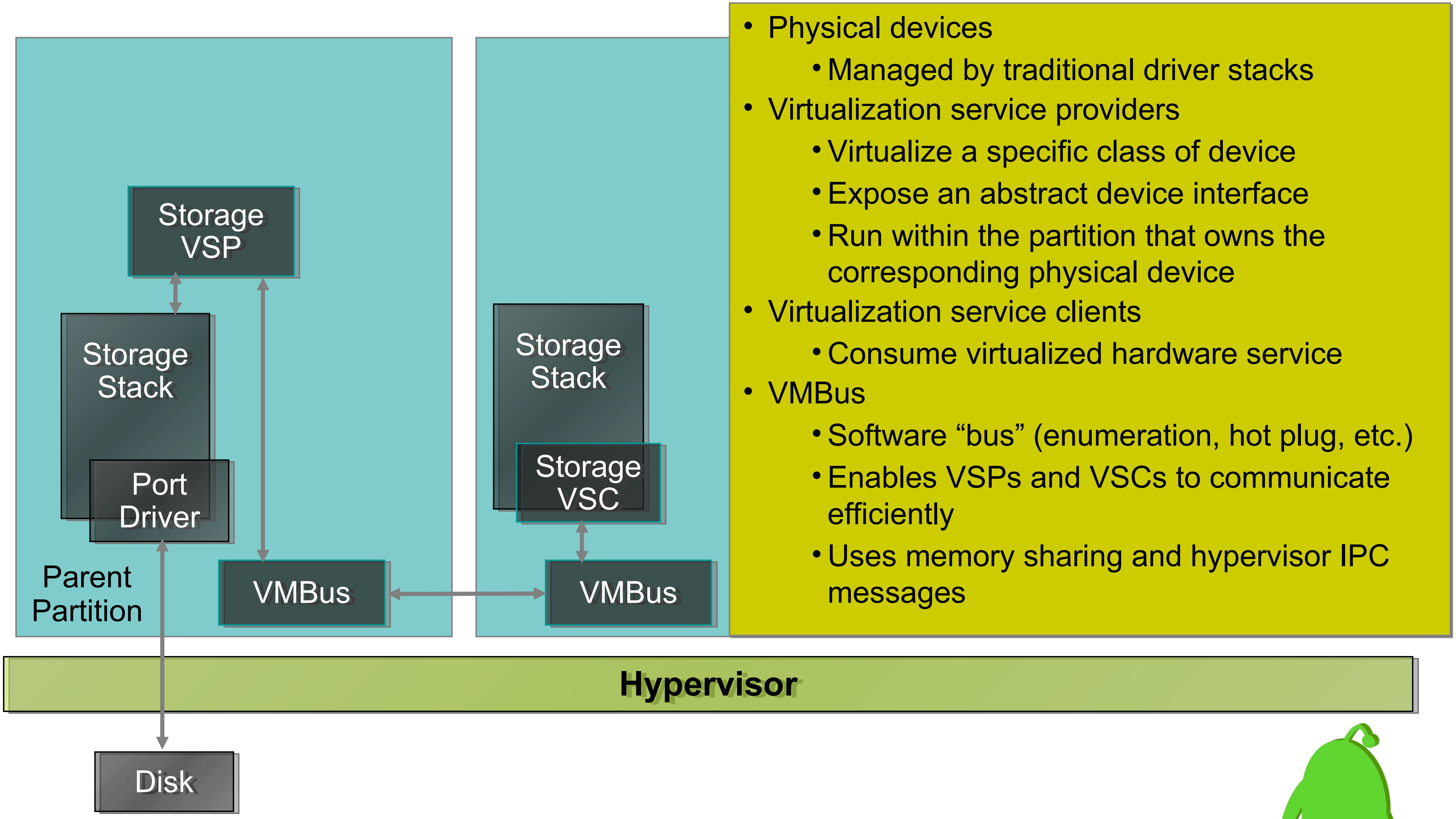


- **Collection of user-mode and kernel-mode components**
  - Runs within a partition on top of a (minimal) OS
  - Contains all VM support not in the hypervisor
- **Interacts with hypervisor**
  - Calls the hypervisor to perform certain actions
  - Responds to messages from the hypervisor or from other partitions
- **Creates and manages a group of “child partitions”**
  - Manages memory for child partitions
  - Virtualizes devices for child partitions
- **Exposes a management interface**

Device Virtualization

- Standard VSP (Virtualization service providers)
  - Storage: support difference drive chains
  - Network: provide virtualized network mechanism
  - Video: 2D/3D graphics w/ or w/o HW acceleration
  - USB: allow a USB device to be assigned to a partition
  - Input: keyboard, mouse, touchscreen
  - Time: virtualization for RTC hardware

Device Virtualization



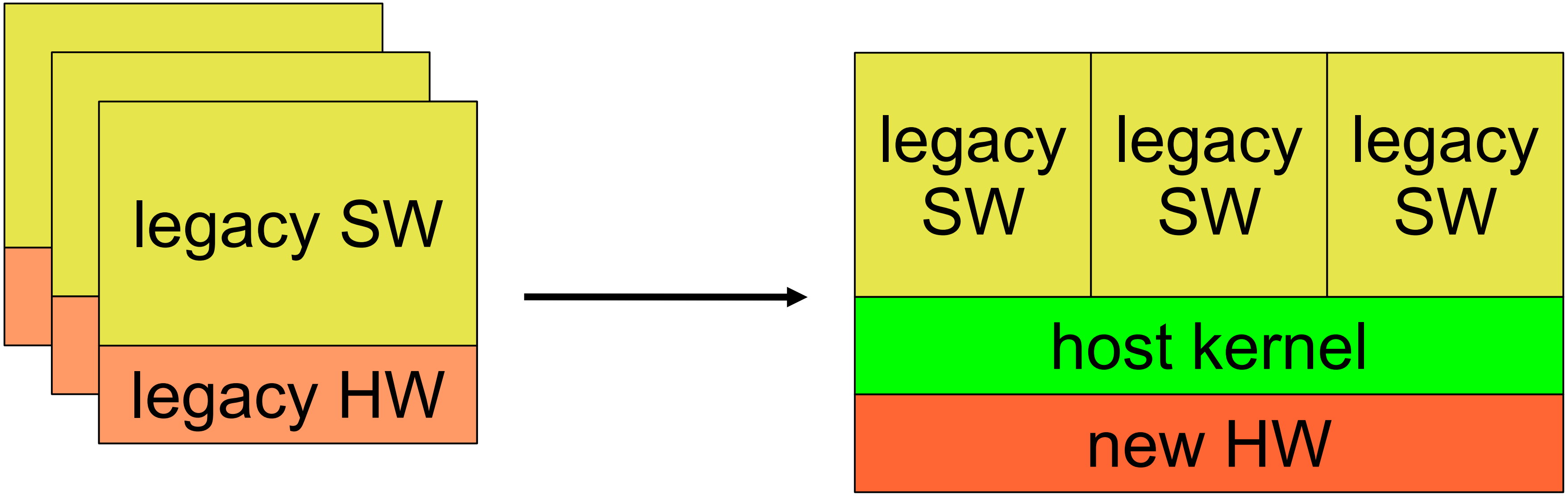
- Physical devices
  - Managed by traditional driver stacks
- Virtualization service providers
  - Virtualize a specific class of device
  - Expose an abstract device interface
  - Run within the partition that owns the corresponding physical device
- Virtualization service clients
  - Consume virtualized hardware service
- VMBus
  - Software "bus" (enumeration, hot plug, etc.)
  - Enables VSPs and VSCs to communicate efficiently
  - Uses memory sharing and hypervisor IPC messages

Embedded Virtualization Use Case

- Workload consolidation
- Legacy software
- Multicore enablement
- Improve reliability
- Secure monitoring

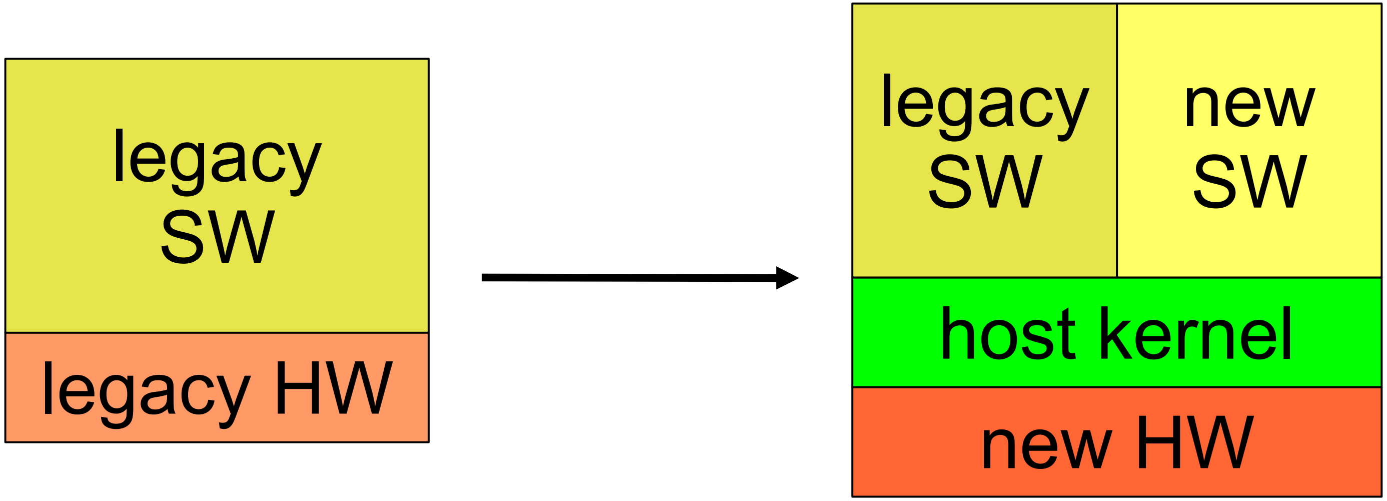
Use Case: Workload Consolidation

- Consolidate legacy systems

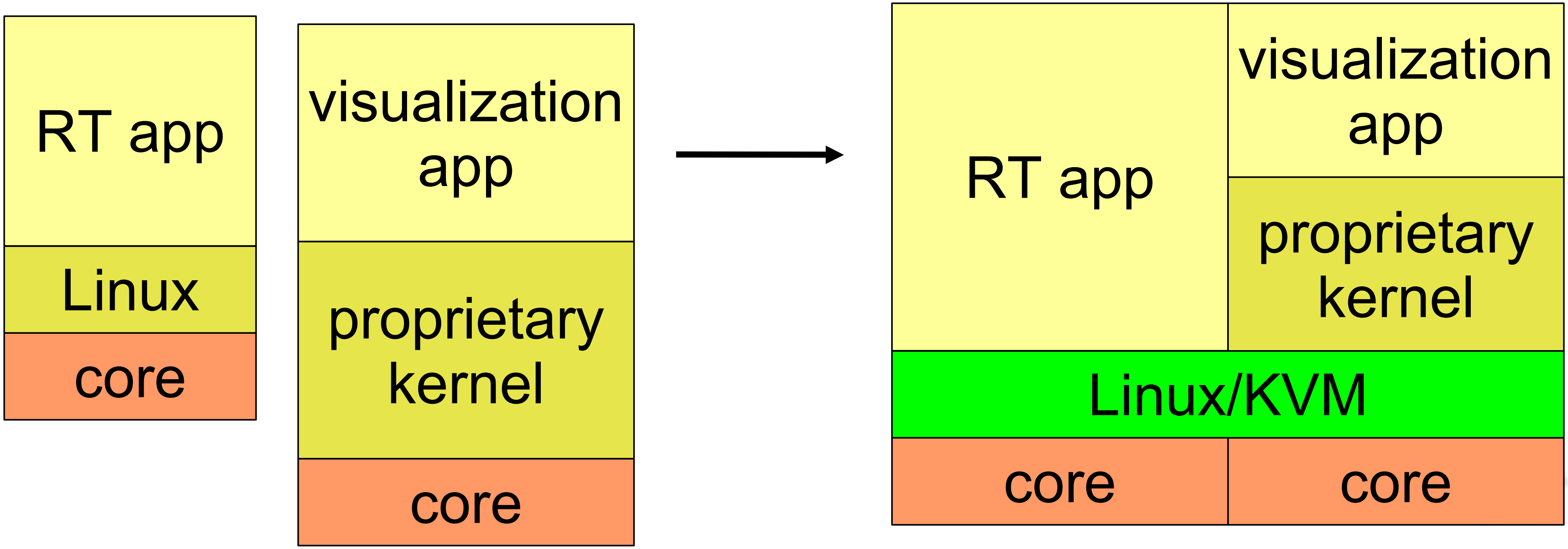


Use Case: Legacy Software

- Run legacy software on new core/chip/board with full virtualization

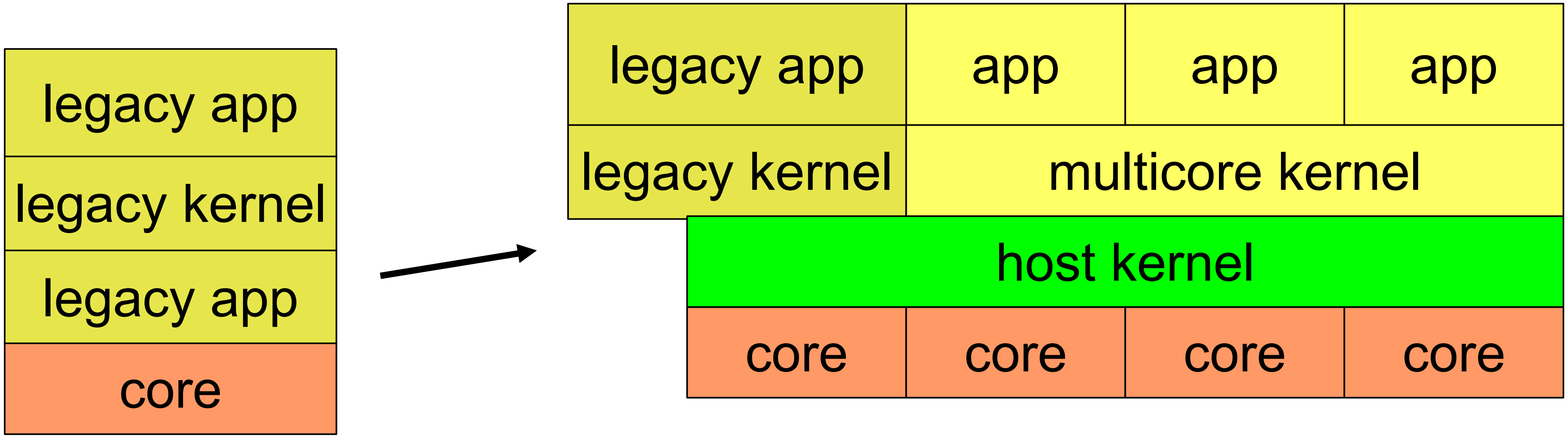


- Consolidate legacy software



Use Case: Multicore Enablement

- Legacy uniprocessor applications



legacy app

legacy kernel

legacy app

core

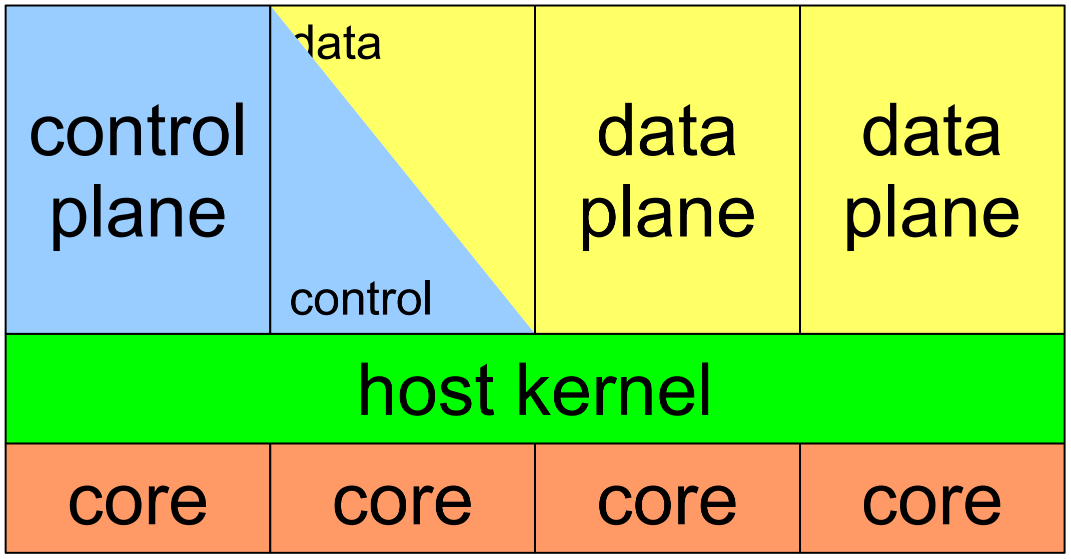
legacy app | app | app | app

legacy kernel | multicore kernel

host kernel

core | core | core | core

- Flexible resource management



control plane

data

control

data plane

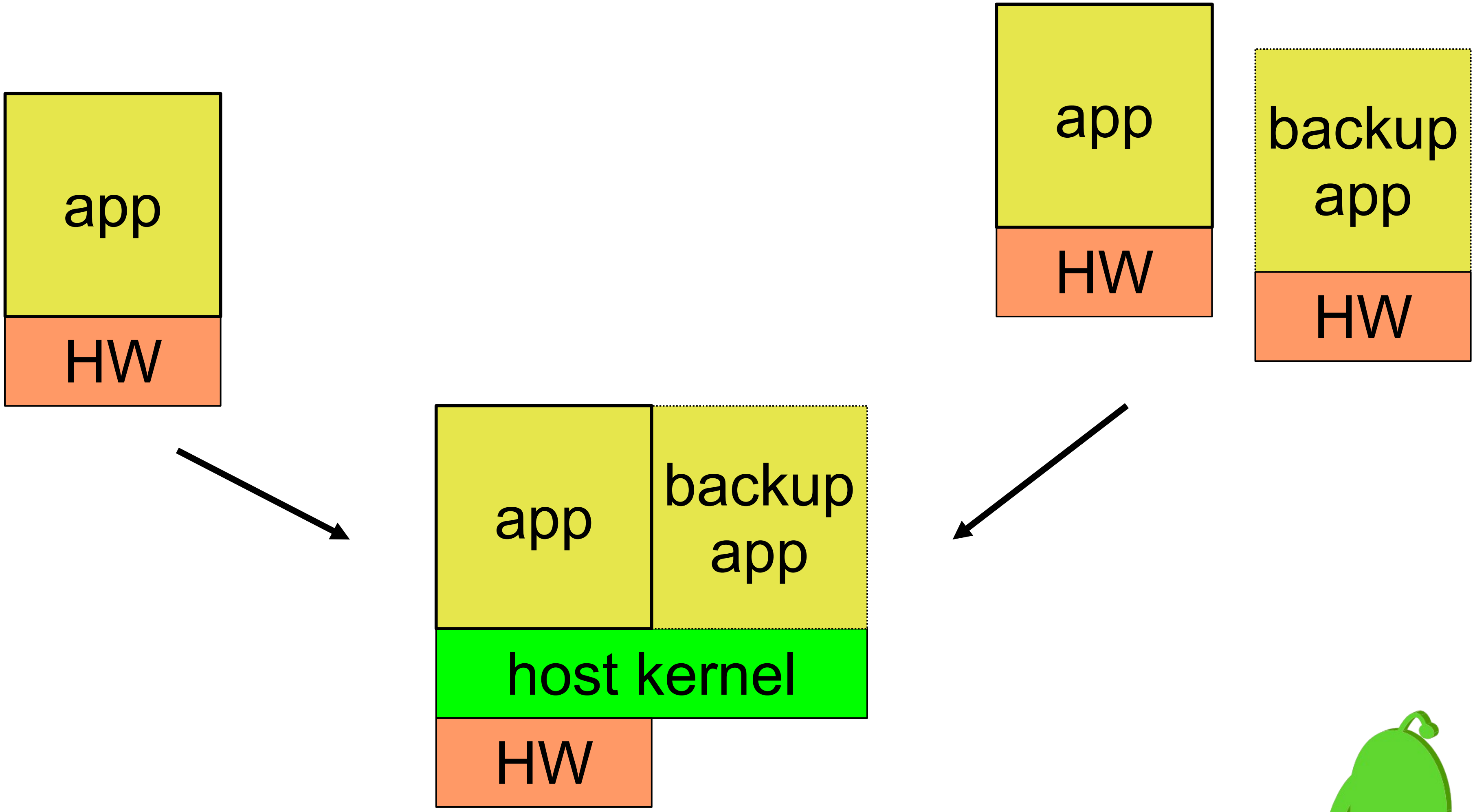
data plane

host kernel

core | core | core | core

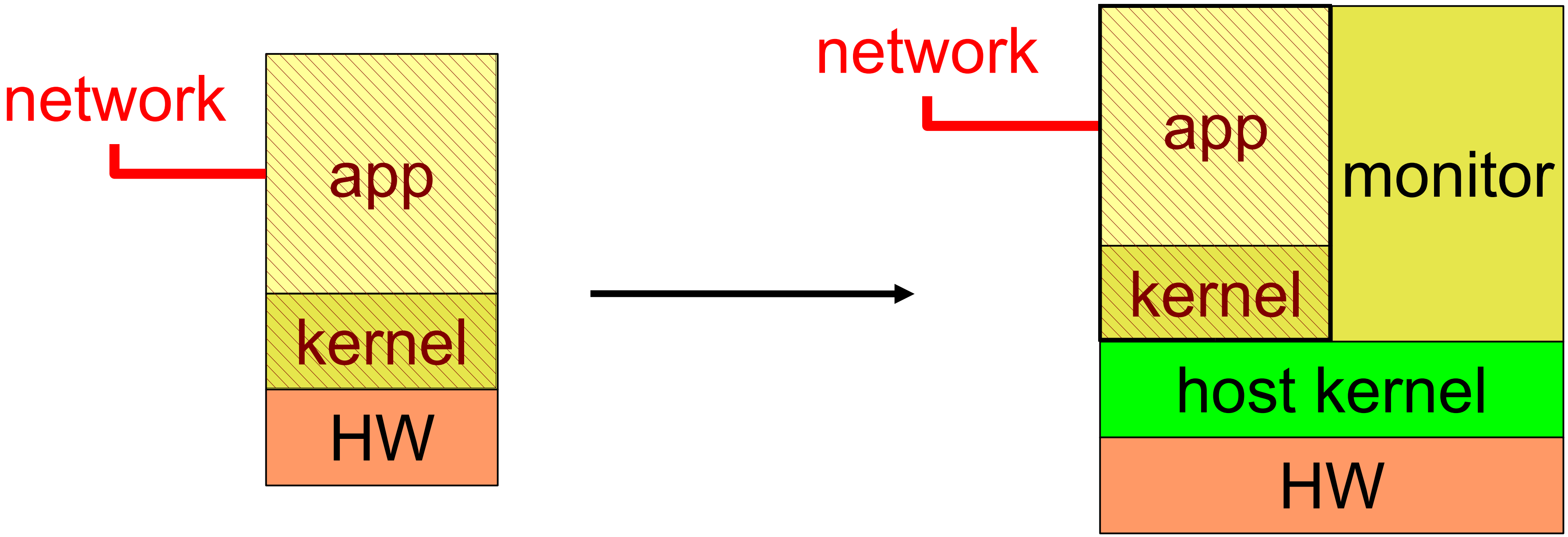
Use Case: Improved Reliability

- Hot standby without additional hardware



Use Case: Secure Monitoring

- Protect monitoring software



Embedded Virtualization Issues

- Memory footprint
- Security
  - Increases size of Trusted Computing Base
- Direct IO Access
- Emulate IO
- Virtual IO
- Real-time support

Direct I/O Access

- Guest can directly access physical IO without host involvement
  - Native speed
- IOMMU provides isolation and physical address translation (DMA)
  - Translation could be done with guest modifications
- Issues:
  - IOMMU **required** for DMA isolation
  - Limited by number of physical IO devices
  - Guests must have device drivers
  - What about legacy guests on new hardware?
  - Breaks migration
  - IRQ delivery and routing

Emulated I/O

- Host software emulates guest IO accesses
- Issues:
  - – Must write software to (perfectly?) emulate hardware
  - – Dramatic increase in IO latency
  - – Host OS must have physical device drivers
    - Device driver availability, licensing concerns

Virtual I/O

- No hardware at all, just inter-guest data transfer
- New guest device drivers co-operate with host
- Issues:
  - Requires guest modification (at least new device drivers)
  - Host OS still needs physical IO drivers

Embedded Hypervisors for ARM

Embedded Hypervisors for ARM

(open source part)

- Xen
  - Xen-arm

    contributed by **Samsung**

    ARM9, ARM11, ARM Cortex-A9 MP
  - Xen-arm-cortext-a15
  - contributed by **Citrix** - https://lkml.org/lkml/2011/11/29/265

    ARM Cortex-A15
- OKL4 (from open to close source), OKLabs
- KVM ARM porting
  - Columbia University
  - NTHU, Taiwan
- Xvisor
- Codezero

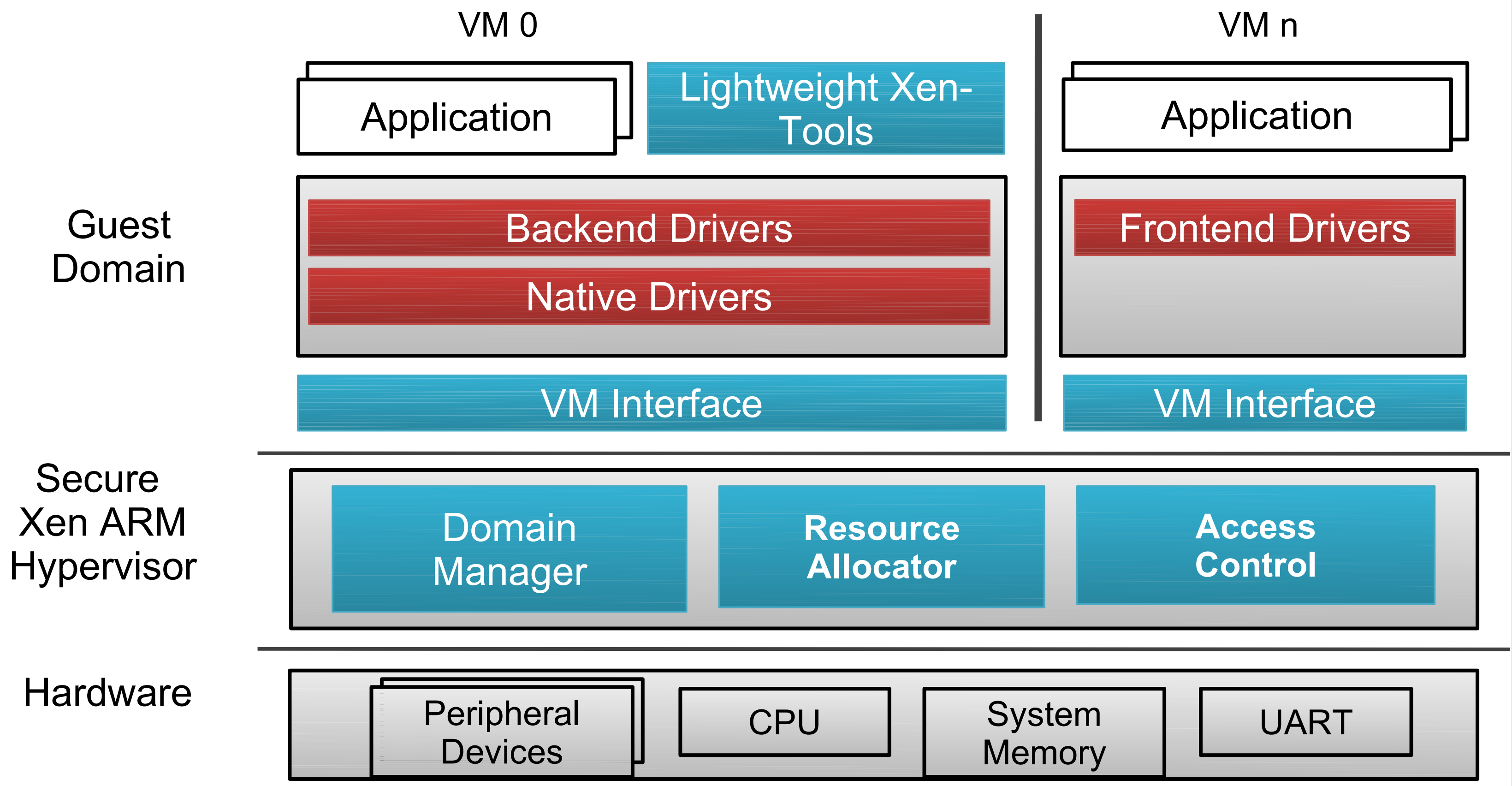
Xen-ARM (Samsung)

Goals

Lightweight virtualization for secure 3G/4G mobile devices

- High performance hypervisor based on ARM processor
- Fine-grained access control fitted to mobile devices

Architecture of Xen ARM



VM 0

Application

Lightweight Xen-Tools

Backend Drivers

Native Drivers

VM Interface

VM n

Application

Frontend Drivers

VM Interface

Guest Domain

Secure Xen ARM Hypervisor

Domain Manager

Resource Allocator

Access Control

Hardware

Peripheral Devices

CPU

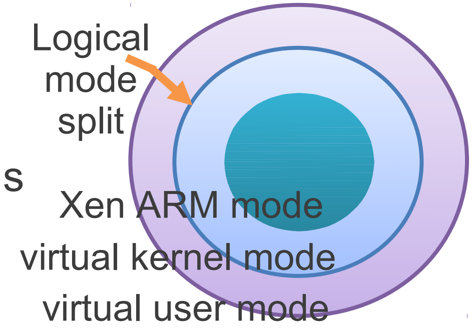
System Memory

UART

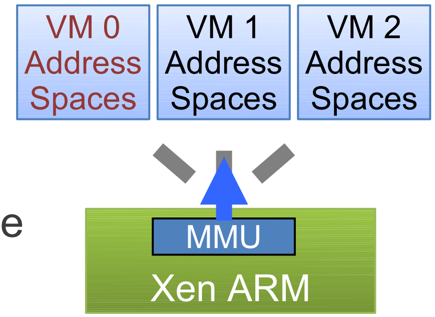
Xen-ARM (Samsung)

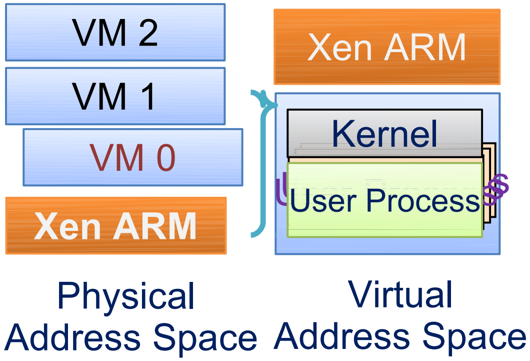
Overview

- CPU virtualization
- Virtualization requires 3 privilege CPU levels, but ARM supports 2 levels
  - Xen ARM mode: supervisor mode (most privileged level)
  - Virtual kernel mode: User mode (least privileged level)
  - Virtual user mode: User mode (least privileged level)

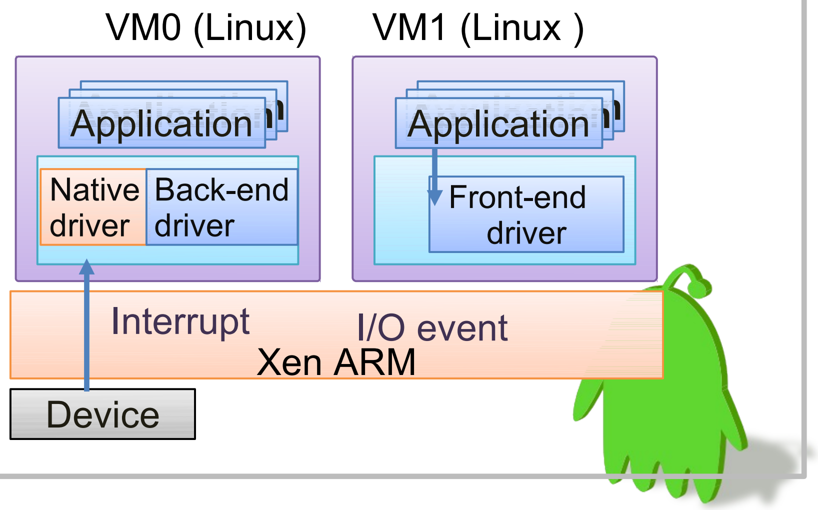


- Memory virtualization
- VM's local memory should be
- protected from other VMs
- Xen ARM switches VM's virtual address space
  - using MMU
  - VM is not allowed to manipulate MMU directly





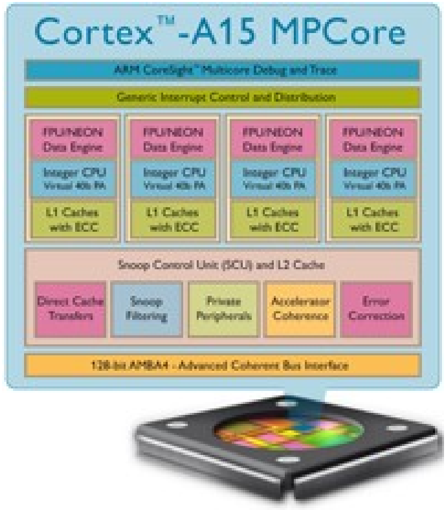
- I/O virtualization
- Split driver model of Xen ARM
  - Client & Server architecture for shared I/O devices
    - Client: frontend driver
    - Server: native/backend driver

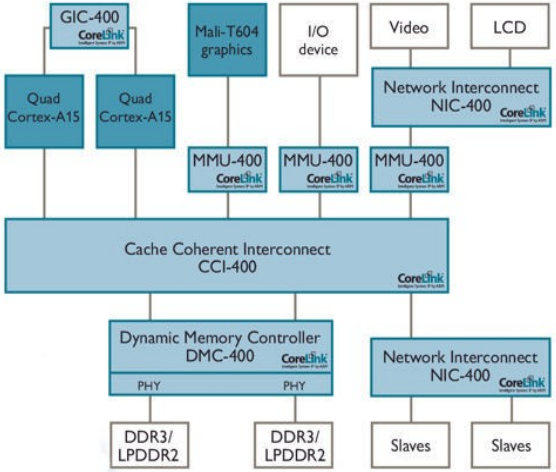


Toward ARM Cortex-A15

New capabilities in ARM Cortex-A15

- Full compatibility with the Cortex-A9
  - Supporting the ARMv7 Architecture
- Virtualization Extension (VE)
  - Run multiple OS binary instances simultaneously
  - Isolates multiple work environments and data
- Supporting Large Physical Addressing Extensions (LPAE)
  - Ability to use up to 1TB of physical memory
- With AMBA 4 System Coherency
  - Other cached devices can be coherent with processor
  - Many core multiprocessor scalability


Cortex™-A15 MPCore
ARM CoreSight™ Multicore Debug and Trace
Generic Interrupt Control and Distribution
FPU/NEON Data Engine
Integer CPU Virtual 40b PA
L1 Caches with ECC
FPU/NEON Data Engine
Integer CPU Virtual 40b PA
L1 Caches with ECC
FPU/NEON Data Engine
Integer CPU Virtual 40b PA
L1 Caches with ECC
FPU/NEON Data Engine
Integer CPU Virtual 40b PA
L1 Caches with ECC
Snoop Control Unit (SCU) and L2 Cache
Direct Cache Transfers
Snoop Filtering
Private Peripherals
Accelerator Coherence
Error Correction
128-bit AMBA4 - Advanced Coherent Bus Interface



Large Physical Addressing

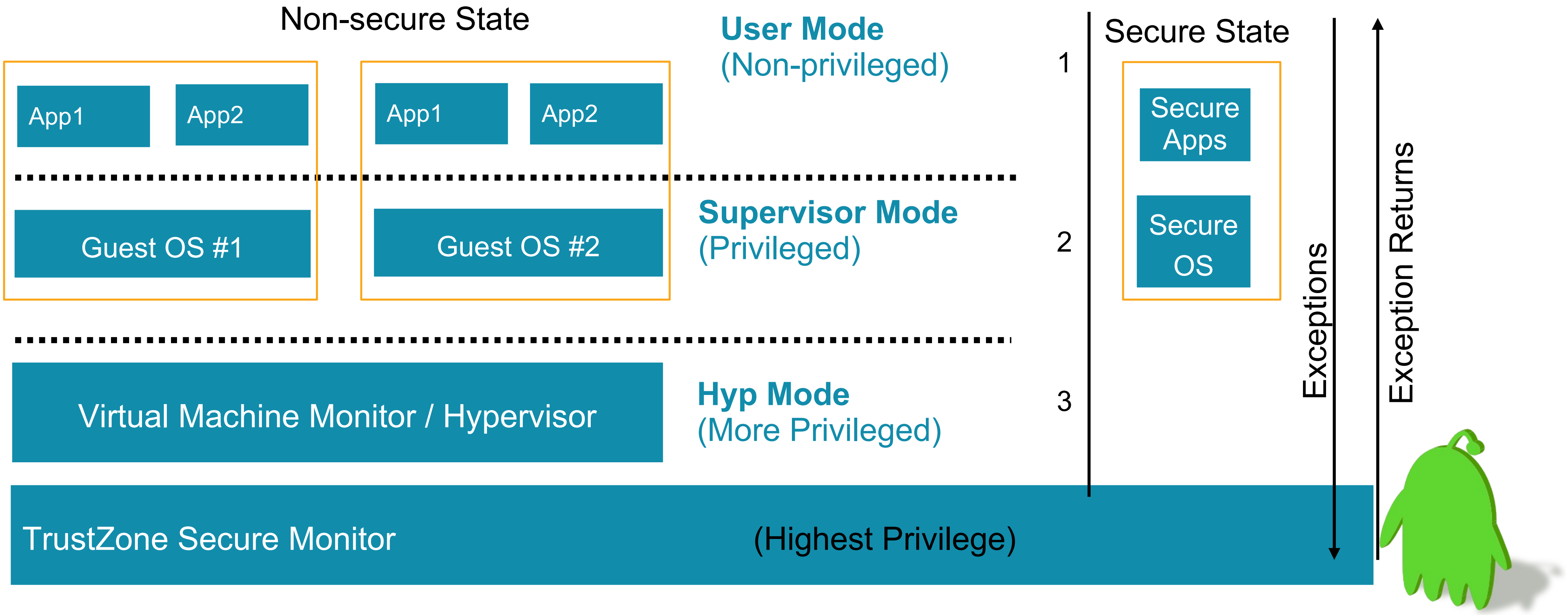
- Cortex-A15 introduces 40-bit physical addressing
  - Virtual memory (applications and OS) still has 32-bit address space
- Offering up to 1 TB of physical address space
  - Traditional 32-bit ARM devices limited to 4GB
- What does this mean for ARM based systems?
  - Reduced address-map congestion
  - More applications at the same time
  - Multiple resident virtualized operating systems
  - Common global physical address in many-core

Virtualization Extensions: The Basics

- New Non-secure level of privilege to hold Hypervisor
  - Hyp mode
- New mechanisms avoid the need Hypervisor intervention for:
  - Guest OS Interrupt masking bits
  - Guest OS page table management
  - Guest OS Device Drivers due to Hypervisor memory relocation
  - Guest OS communication with the interrupt controller (GIC)
- New traps into Hyp mode for:
  - ID register accesses and idling (WFI/WFE)
  - Miscellaneous “difficult” System Control Register cases
- New mechanisms to improve:
  - Guest OS Load/Store emulation by the Hypervisor
  - Emulation of trapped instructions through syndromes

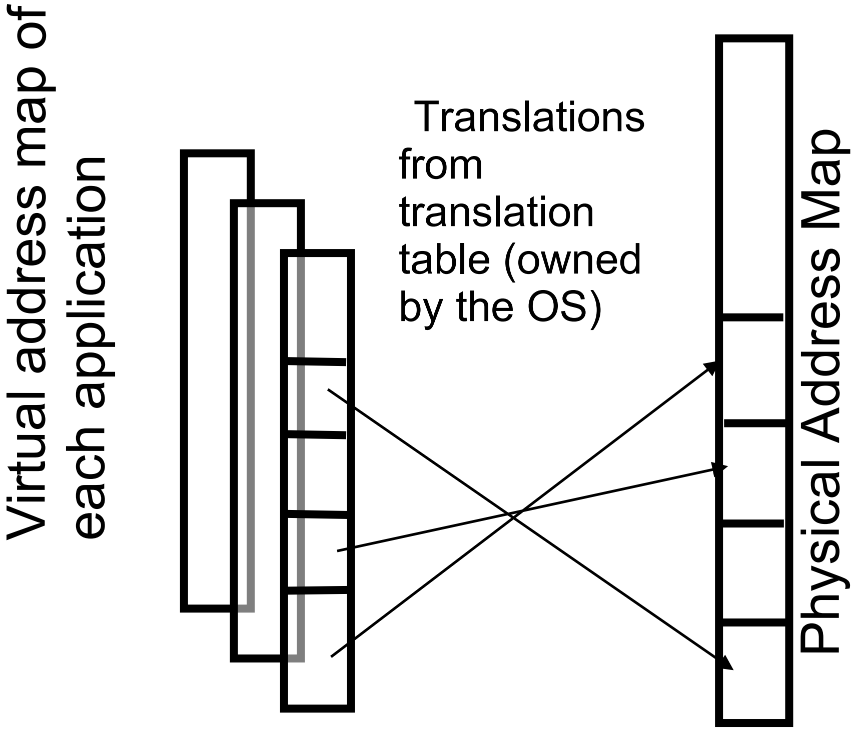
Virtualization: Third Privilege

- Guest OS same kernel/user privilege structure
- HYP mode higher privilege than OS kernel level
- VMM controls wide range of OS accesses
- Hardware maintains TZ security (4th privilege)



Memory - the Classic Resource

- Before virtualization: the OS owns the memory
  - Allocates areas of memory to the different applications
  - Virtual Memory commonly used in “rich” operating systems

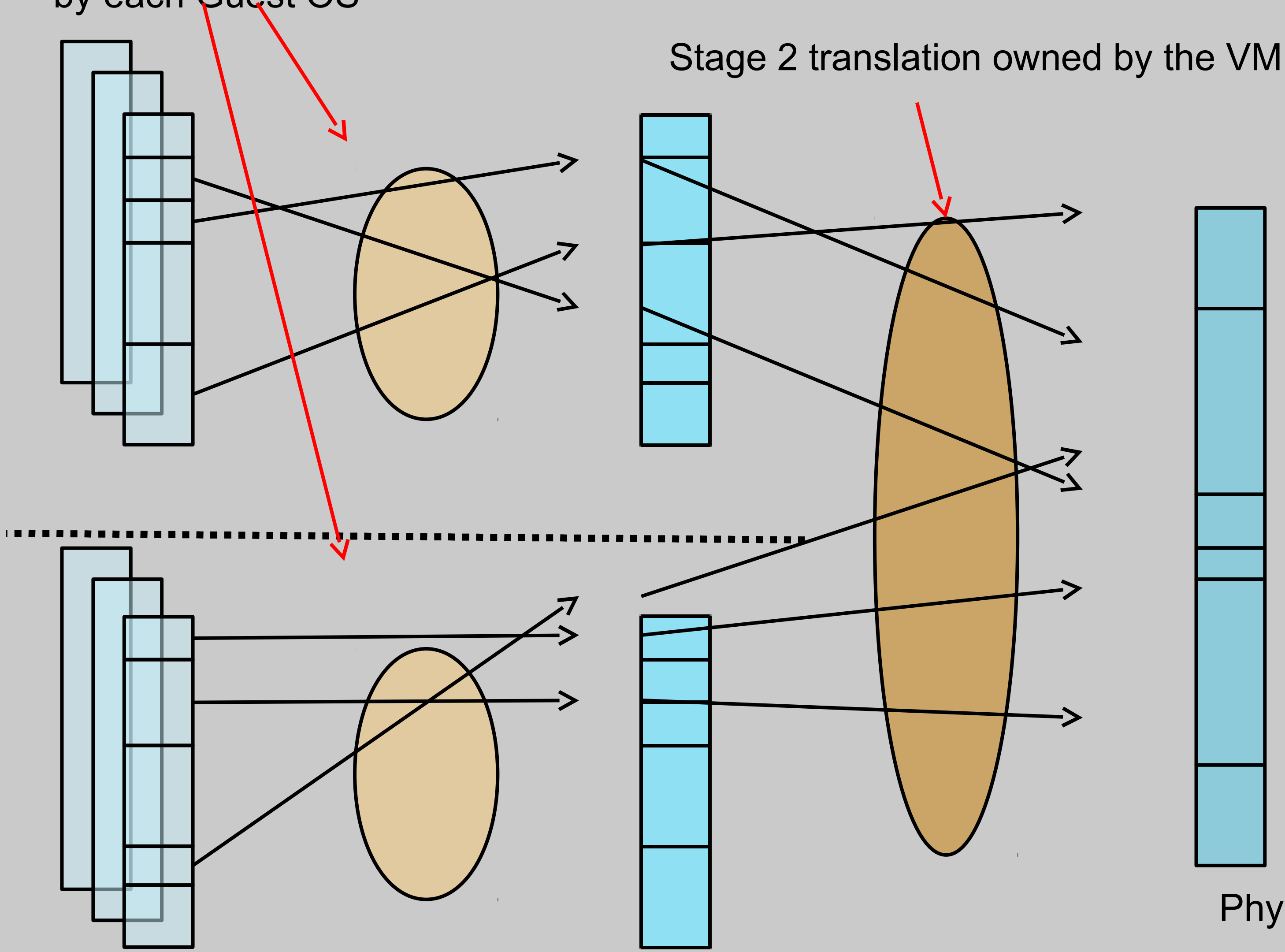


Virtual address map of each application

Translations from translation table (owned by the OS)

Physical Address Map

Virtual Memory in Two Stages



Stage 1 translation owned by each Guest OS

Stage 2 translation owned by the VMM

Virtual address (VA) map of each App on each Guest OS

“Intermediate Physical” address map of each Guest OS (IPA)

Physical Address (PA) Map

Hardware has 2-stage memory translation

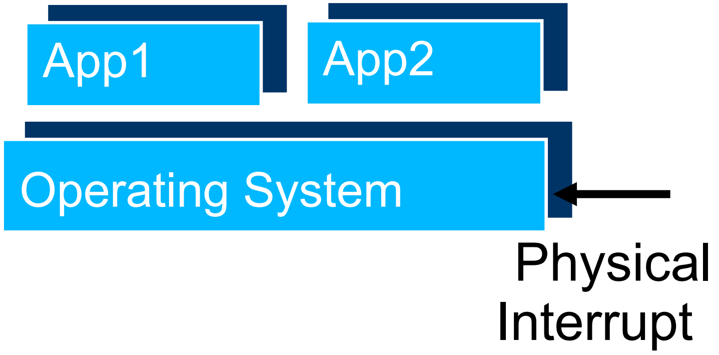
Tables from Guest OS translate VA to IPA

Second set of tables from VMM translate IPA to PA

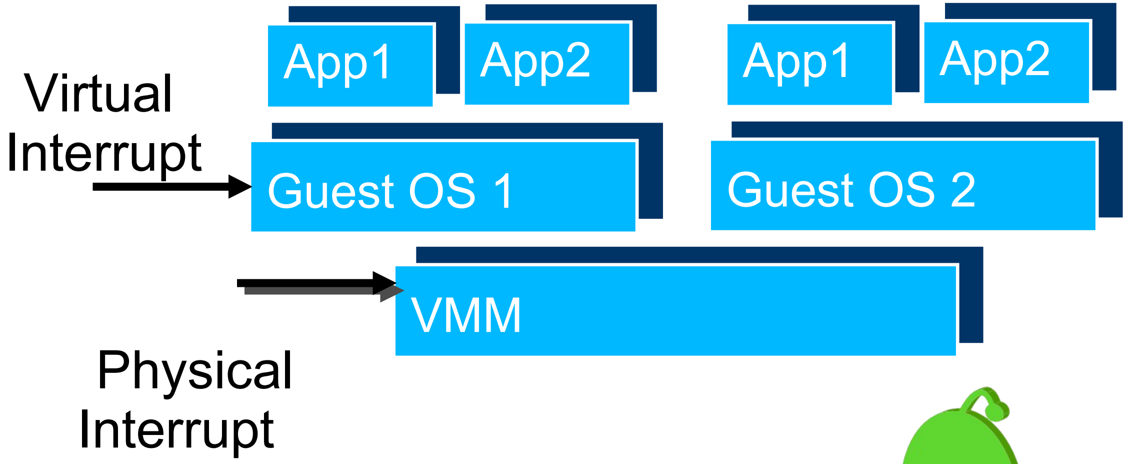
Allows aborts to be routed to appropriate software layer

Classic Issue: Interrupts

- An Interrupt might need to be routed to one of
  - Current or different Guest OS
  - Hypervisor
  - OS/RTOS running in the secure TrustZone environment
- Basic model of the ARM virtualization extensions
  - Physical interrupts are taken initially in the Hypervisor
  - If the Interrupt should go to a Guest OS, Hypervisor maps a "virtual" interrupt for that Guest OS



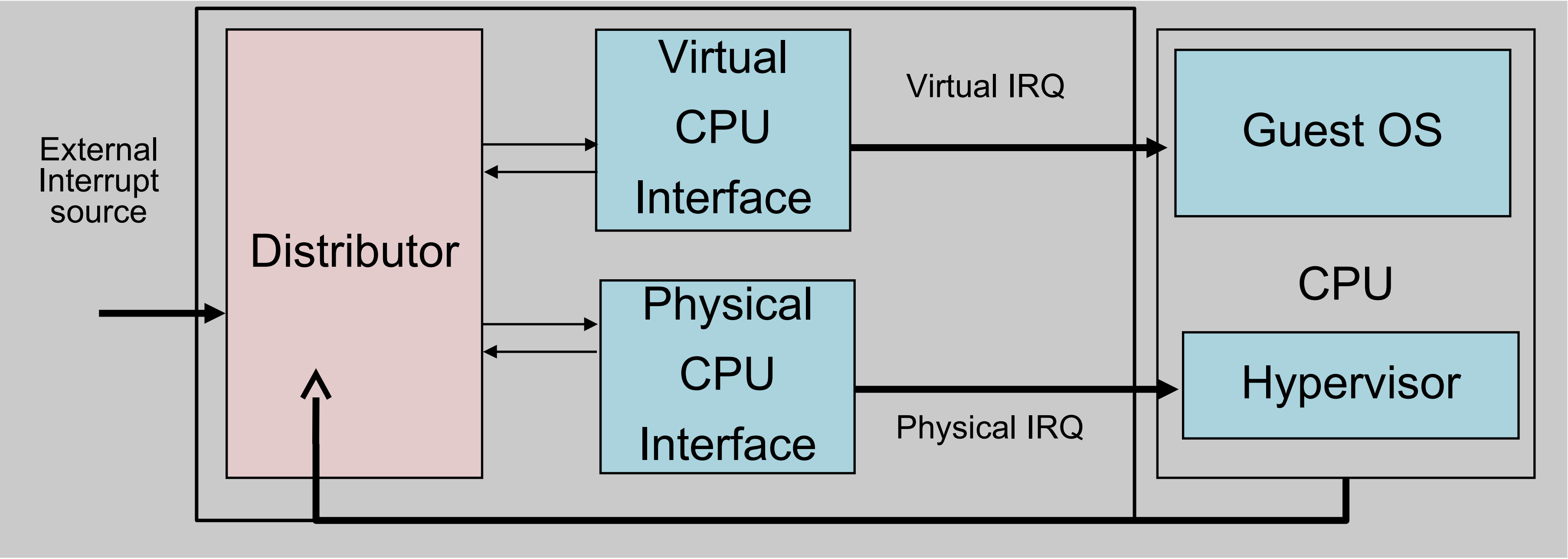
System without virtualization



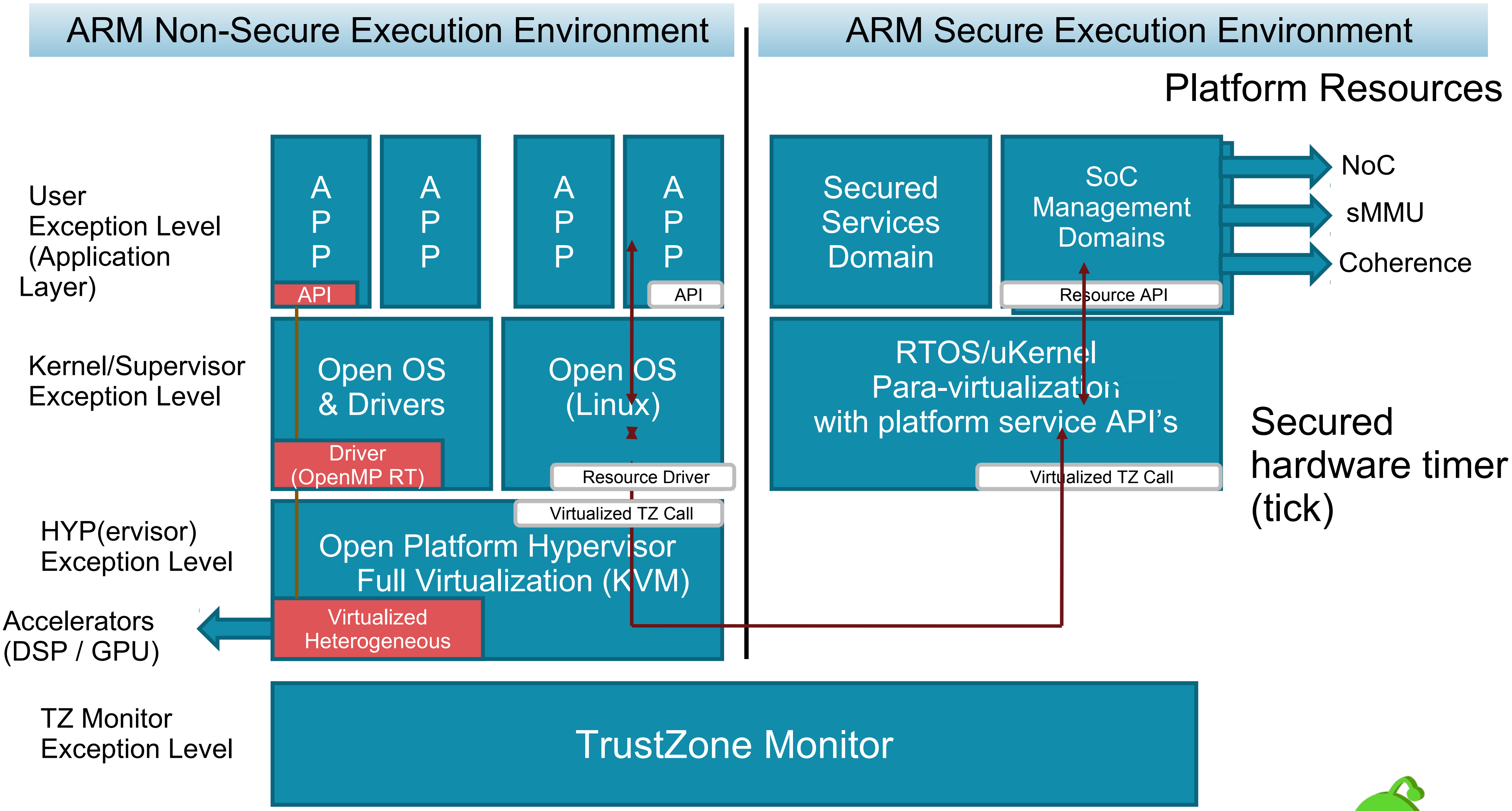
System with virtualization

Virtual interrupt example

- External IRQ (configured as virtual by the hypervisor) arrives at the GIC
- GIC Distributor signals a Physical IRQ to the CPU
- CPU takes HYP trap, and Hypervisor reads the interrupt status from the Physical CPU Interface
- Hypervisor makes an entry in register list in the GIC
- GIC Distributor signals a Virtual IRQ to the CPU
- CPU takes an IRQ exception, and Guest OS running on the virtual machine reads the interrupt status from the Virtual CPU Interface



Spanning Hypervisor Framework



ARM Non-Secure Execution Environment
ARM Secure Execution Environment
Platform Resources
User Exception Level (Application Layer)
Kernel/Supervisor Exception Level
HYP(ervisor) Exception Level
Accelerators (DSP / GPU)
TZ Monitor Exception Level
APP
APP
APP
APP
API
API
Open OS & Drivers
Driver (OpenMP RT)
Open OS (Linux)
Resource Driver
Virtualized TZ Call
Open Platform Hypervisor Full Virtualization (KVM)
Virtualized Heterogeneous
Secured Services Domain
SoC Management Domains
Resource API
NoC
sMMU
Coherence
RTOS/uKernel Para-virtualization with platform service API's
Virtualized TZ Call
Secured hardware timer (tick)
TrustZone Monitor

Source: Hardware accelerated Virtualization in the ARM Cortex™ Processors, John Goodacre, ARM Ltd. (2011)

Reference

- 前瞻資訊科技 – 虛擬化，薛智文，台大資訊所 (2011)
- ARM Virtualization: CPU & MMU Issues, Prashanth Bungale, vmware
- Hardware accelerated Virtualization in the ARM Cortex™ Processors, John Goodacre, ARM Ltd. (2011)
- Hypervisors and the Power Architecture

  http://www.techdesignforums.com/embedded/embedded-topics/embedded-development-platforms/hypervisors-and-the-power-architecture/
- Philippe Gerum, State of Real-Time Linux: Don't Stop Until History Follows, ELC Europe 2009

0xlab

http://0xlab.org