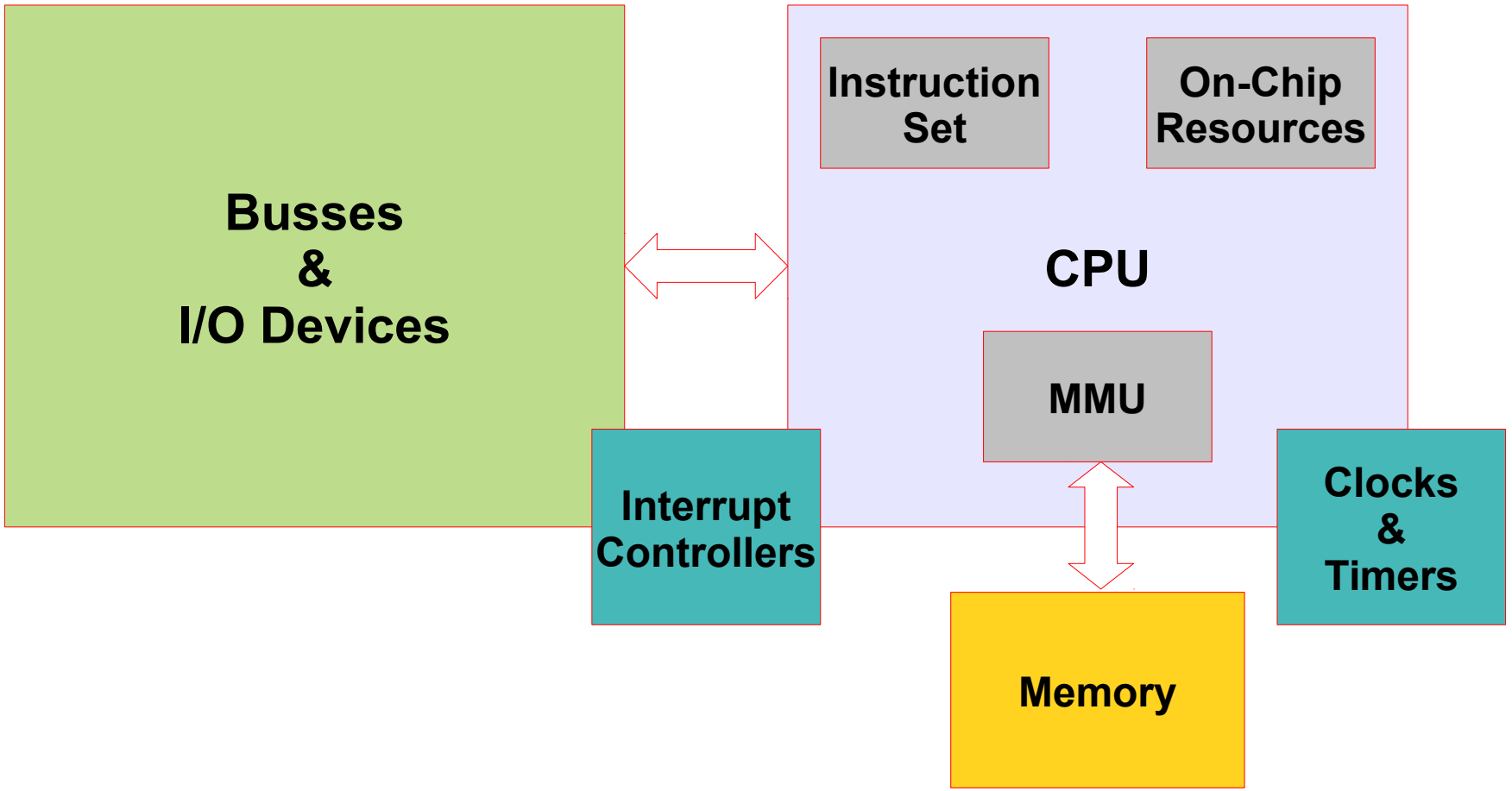
Xvisor: embedded and lightweight hypervisor

Jim Huang (黃敬群) <**jserv**@0xlab.org>

Aug 3, 2013 / COSCUP

Let's review how a computer works



Agenda

(1) Virtualization Concepts

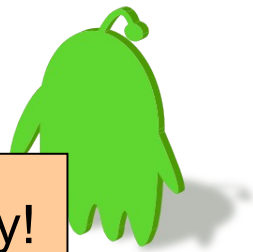
(2) Xvisor for ARM

(3) Device Virtualization

Virtualization Concepts

CPU Performance

- 1965 - IBM S/360 – 0.1 MIPS (133,300 IPS)
  - **Provided full hardware virtualization with the ability to run 14 OS instances.**
- 1972 - IBM S/370 – 1.0 MIPS (1,000,000 IPS)
- 2000 - 1 GHz Intel P3 – 3,000 MIPS (3,000,000,000 IPS)
- 2009 - Qualcomm Snapdragon A8 – 2,000 MIPS
- 2010 - Intel Core i7 – 4 x 147,600 MIPS
- 2010 - Qualcomm Snapdragon MP – 2 x 2,500 MIPS
- 2011 - Qualcomm/Samsung/nVidia A9 MP – 2 x 5,000 MIPS
- 2012 – ARM Cortex A15 MP – 4 x 25,000 MIPS



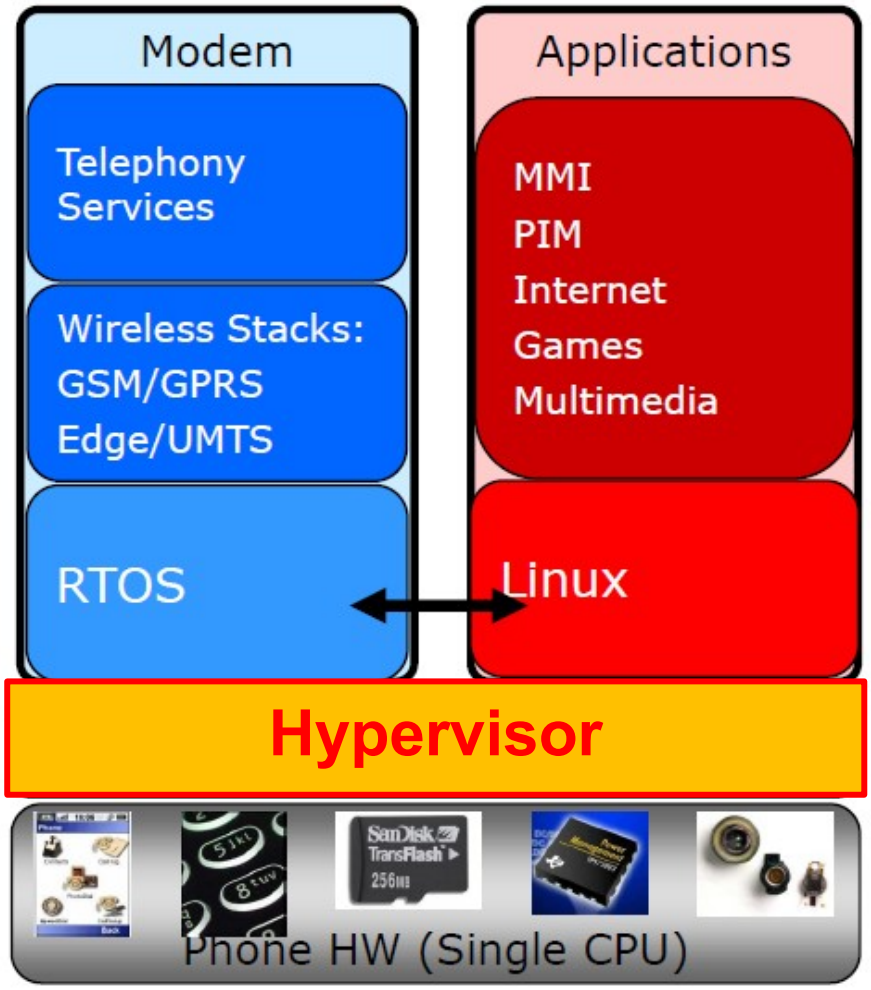
ARM processors are capable to virtualize in much cheaper and powerful way!

Operating System Level Virtualization

- Why to have another layer below existing operating systems?
  - **OS is not perfect: compatibility, stability, security**
- Workload consolidation
  - Increase server utilization
  - Reduce capital, hardware, power, space, heat costs
- Legacy OS support
  - Especially with large 3rd-party software products
- Migration
  - Predicted hardware downtime
  - Workload balancing

Use Case: Low-cost 3G Phone

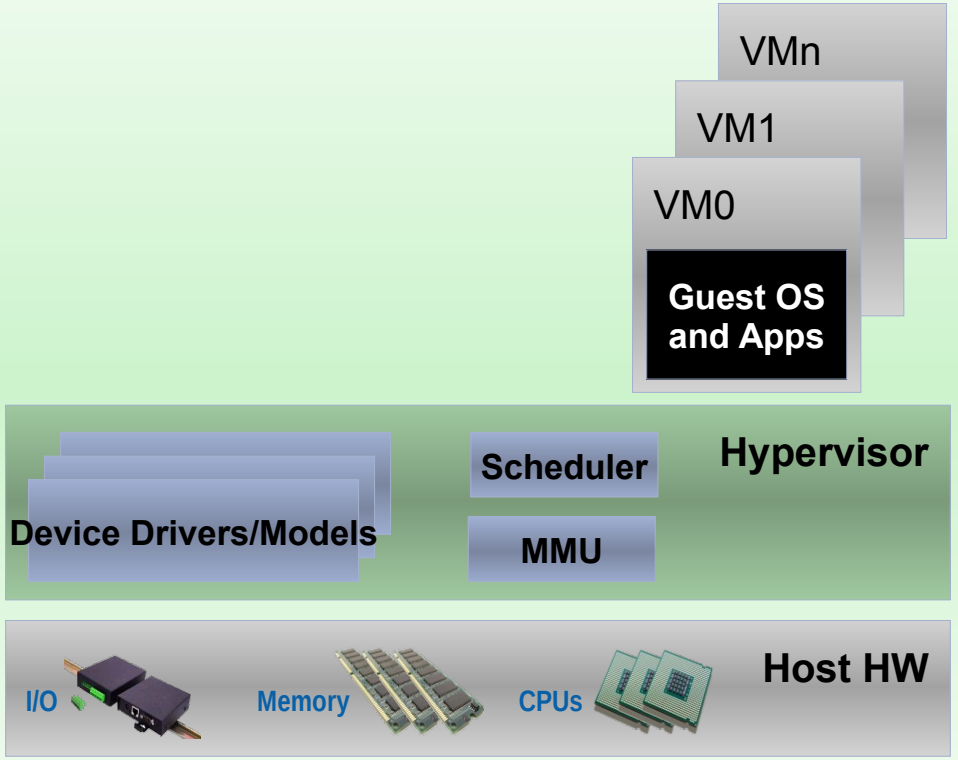
- Mobile Handsets
  - **Major applications runs on Linux**
  - **3G Modem software stack runs on RTOS domain**
- Virtualization in multimedia Devices
  - **Reduces BOM (bill of materials)**
  - **Enables the Reusability of legacy code/applications**
  - **Reduces the system development time**
- Instrumentation, Automation
  - **Run RTOS for Measurement and analysis**
  - **Run a GPOS for Graphical Interface**
- Real cases: Motorola Evoke QA4



General Classification of Virtualization

Type 1: Bare metal Hypervisor

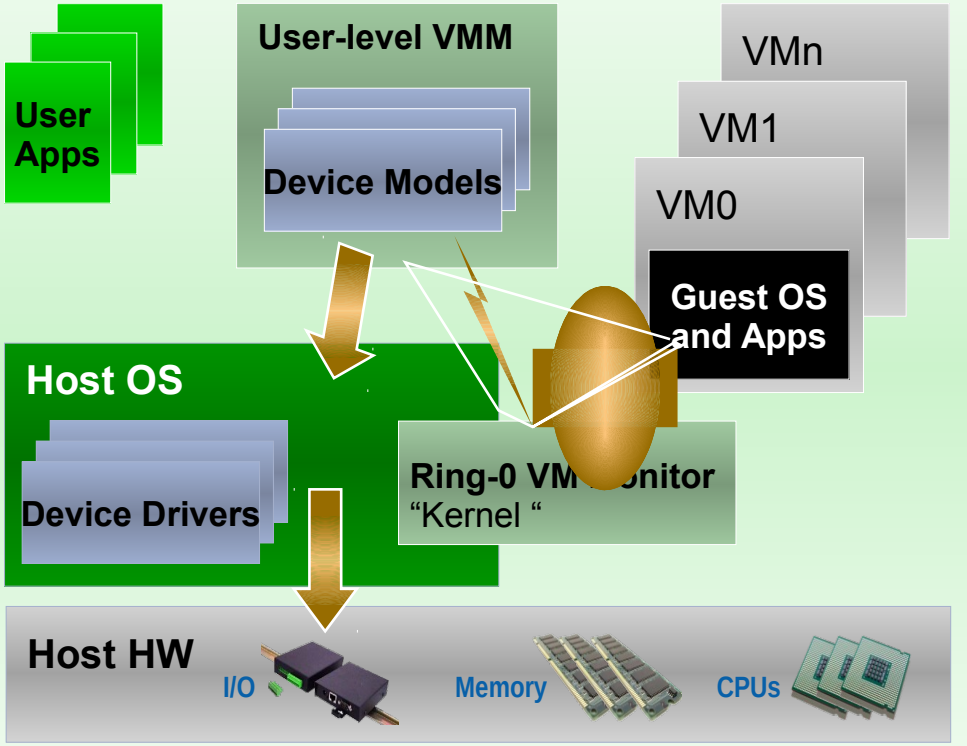
A pure Hypervisor that runs directly on the hardware and hosts Guest OS's.



Provides partition isolation + reliability, higher security

Type 2: OS 'Hosted'

A Hypervisor that runs within a Host OS and hosts Guest OS's inside of it, using the host OS services to provide the virtual environment.



Low cost, no additional drivers
Ease of use & installation

Virtualizable

is a property of the Instruction Set Architecture (ISA)

- A **sensitive instruction**
  - **changes the configuration or mode of the processor,**

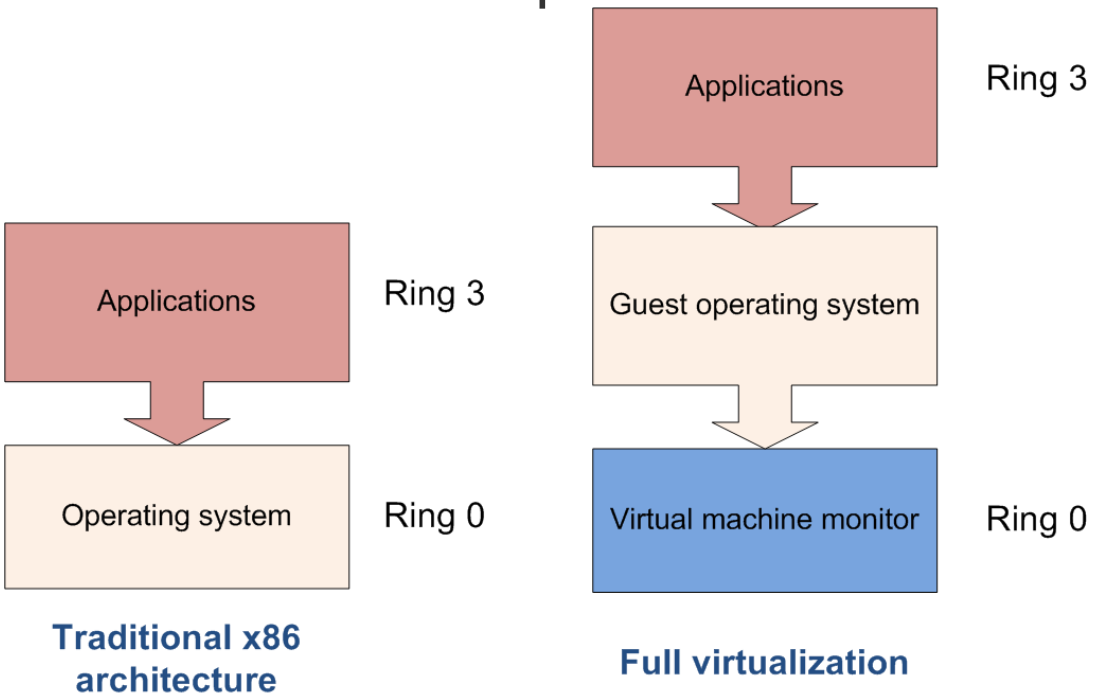
  or
  - **depends in its behavior on the processor's state**

- A **privileged instruction**
  - must be executed with sufficient privilege
  - **causes a trap in user mode**

If all sensitive instructions are privileged, virtual machine monitor can be written.

Privileged Instructions

- Privileged instructions: OS kernel and device driver access to system hardware
- **Trapped and Emulated** by VMM
  - execute guest in separate address space in unprivileged mode
  - emulate all instructions that cause traps



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

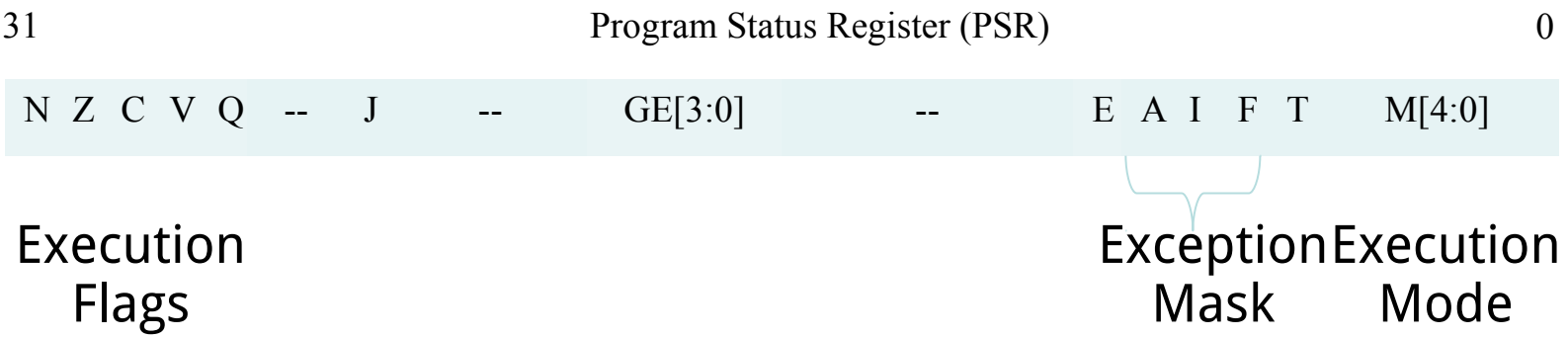
Problematic Instructions (2)

- Type 2

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

  `MSR` `cpsr_c, #0xD3`

  Switch to privileged mode and disable interrupt



Problematic Instructions (3)

- Type 3

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

  ```
  MOVS     PC, LR
  ```

  The return instruction

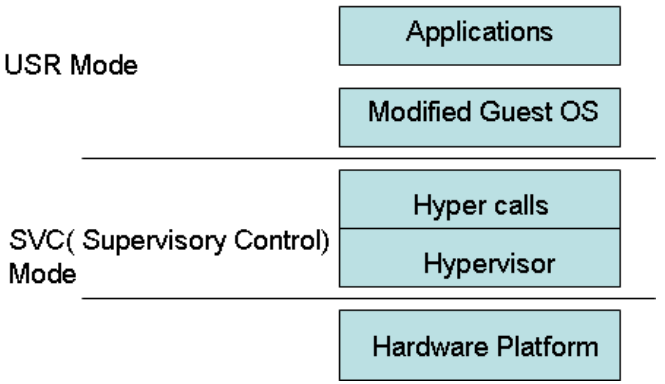
  changes the **program counter** and switches to **user mode**.
- This instruction causes unpredictable behavior when executed in user mode.

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)

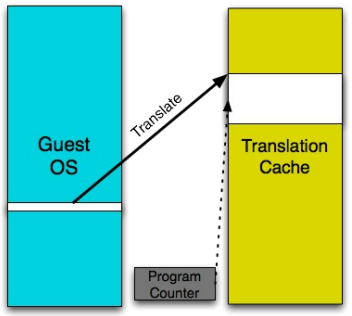
Solutions to Problematic Instructions [Hardware Techniques]

- 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



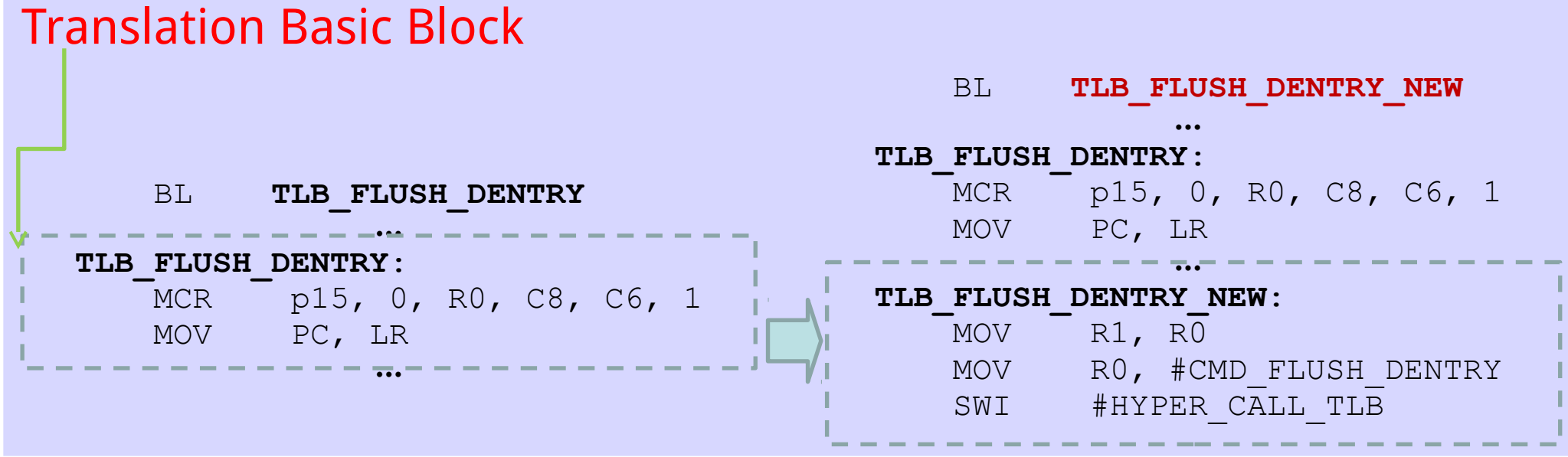
Solutions to Problematic Instructions [Software Techniques]

Complexity	**Binary translation**	**Hypercall**
Design	**High**	**Low**
Implementation	**Medium**	**High**
Runtime	**High**	**Medium**



Method: trap and emulate

Dynamic Binary Translation



- 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
    → self-modifying code

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
    |
    v
Hypercall Handler
.....
LDR R1, [SP, #4]
MCR p15, 0, R1, C8, C6, 1
    |
    v
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.

cond	1111	000010	0R	-Rd-	0000	0000	0000

```
mrs r8, cpsr                    swi 0x088000
```

Xvisor:

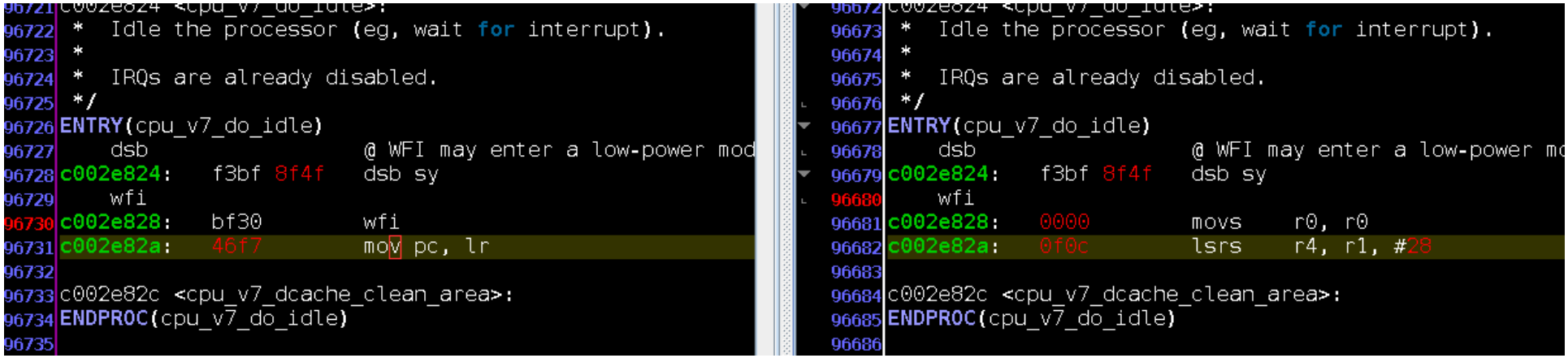
Type 1 Hypervisor; GPL; ARM

Xvisor

- new open source bare-metal hypervisor, which aims towards providing virtualization solution, which is light-weight, portable, and flexible with small memory footprint and less virtualization overhead
- distributed under GNU Public License (GPLv2).
- On real hardware specifically BeagleBoard, it reaches near native CPU performance
  - Native Linux 3.0.4 gives 1120 DMIPS whereas Linux 3.0.4 running as guest on Xvisor ARM gives 1070 DMIPS.
- Major targets: x86 and ARM
  - ARM target supports ARM9, ARM11, Cortex-A8, Cortex-A9, Cortex-A15
  - Virtualization Extension

- File: arch/arm/cpu/arm32/elf2cpatch.py
  - Script to generate cpatch script from guest OS ELF
- Functionality before generating the final ELF image
  - Encode all privileged instructions into SVC instructions (software interrupt)
  - For each privilege instruction, generate a primitive to replace it
  - read the directive from ELF2CPATCH and **mangle the target binary file**
  - **The patched image contains no privilege instructions and could run with in user mode**

```
elf2cpatch.py :
...
if (len(w)==3):
    if (w[2]=="wfi"):
        print "\t#", w[2]
        print "\twrite32,0x%x,0x%08x" % (addr, convert_wfi_inst(w[1]))
    elif (len(w)==4):
        if (w[2]=="cps" or w[2]=="cpsie" or w[2]=="cpsid"):
            print "\t#", w[2], w[3]
            print "\twrite32,0x%x,0x%08x" % (addr, convert_cps_inst(w[1]))
```



How does Xvisor handle problematic instructions like MSR?

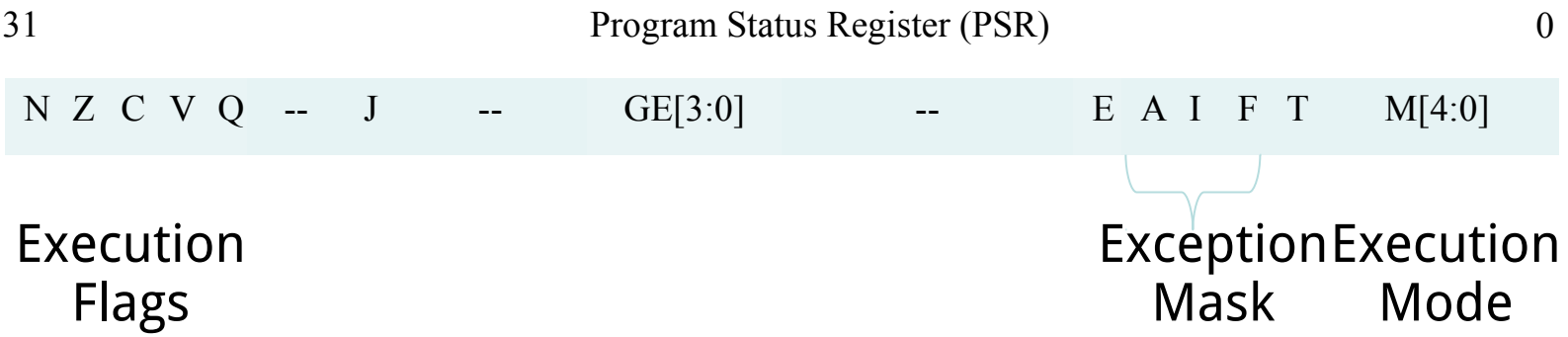
- Type 2

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

- Example

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

  Switch to privileged mode and disable interrupt



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
```

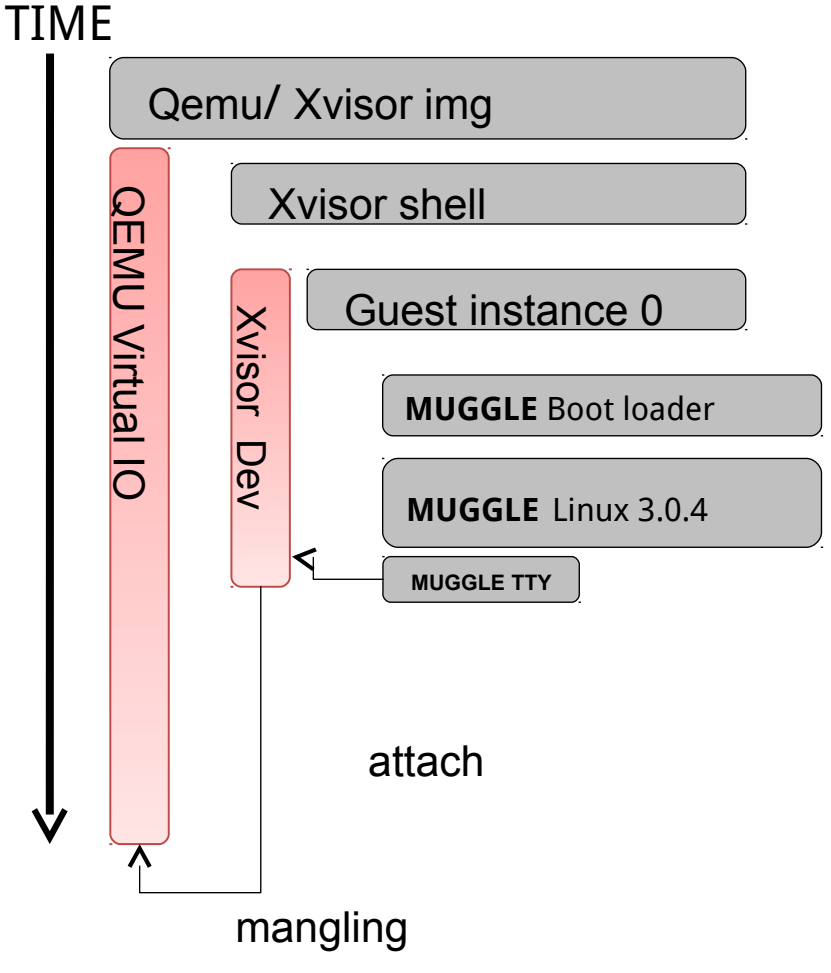
```
# MSR (immediate)
#       Syntax:
#               msr<c> <spec_reg>, #<const>
#       Fields:
#               cond = bits[31:28]
#               R = bits[22:22]
#               mask = bits[19:16]
#               imm12 = bits[11:0]
#       Hypercall Fields:
#               inst_cond[31:28] = cond
#               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).

But it is not enough...

- have to handle virtual CPU, memory, devices.



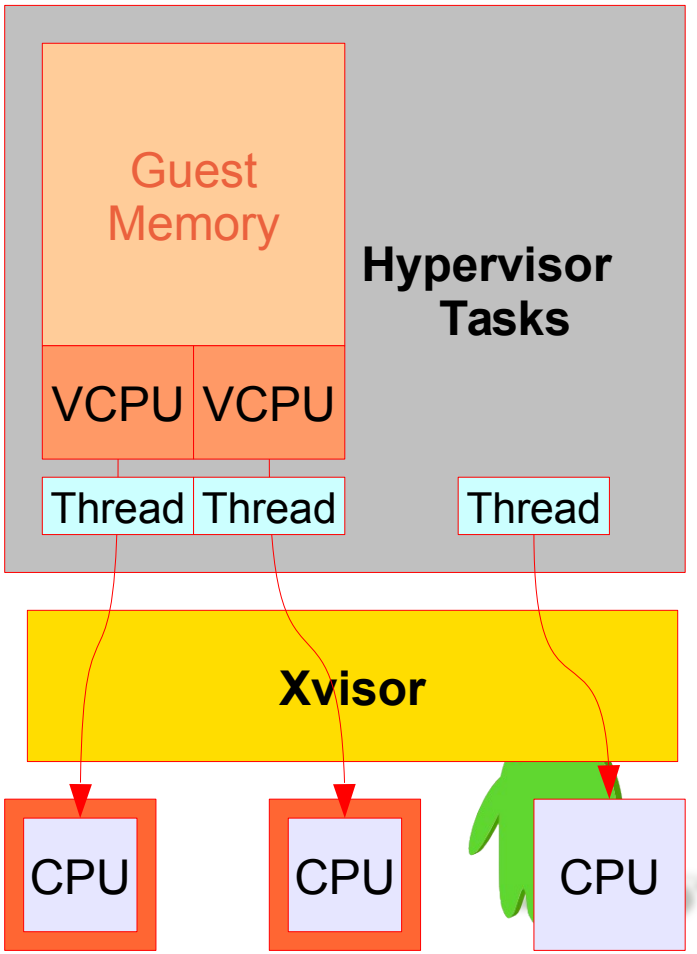
Boot ARM/Linux under Xvisor

```
Freeing init memory: 16K
Creating Pre-Configured Guest Instances
Starting Hypervisor Timer

XVisor# guest kick 0; vserial bind guest0/uart0
guest0: Kicked
[guest0/uart0] ARM Realview PB-A8 Basic Test
[guest0/uart0]
[guest0/uart0] arm-test# or
[guest0/uart0] arm-test# nor_boot
[guest0/uart0] copy 0x70400000 0x40100000 0x300000
[guest0/uart0] copy 0x71000000 0x40400000 0x400000
[guest0/uart0] start_linux 0x70400000 0x71000000 0x400000
[guest0/uart0] Uncompressing Linux... done, booting the kernel.
[guest0/uart0] [    0.000000] Linux version 3.0.4 (anup@anup-desktop) (gcc versi
on 4.6.1 (Sourcery CodeBench Lite 2011.09-70) ) #1 Thu Apr 19 12:08:08 IST 2012
[guest0/uart0] [    0.000000] CPU: ARMv7 Processor [410fc080] revision 0 (ARMv7)
, cr=10c53c7f
[guest0/uart0] [    0.000000] CPU: VIPT nonaliasing data cache, VIPT nonaliasing
 instruction cache
[guest0/uart0] [    0.000000] Machine: ARM-RealView PB-A8
[guest0/uart0] [    0.000000] Ignoring unrecognised tag 0x00000000
[guest0/uart0] [    0.000000] bootconsole [earlycon0] enabled
[guest0/uart0] [    0.000000] Memory policy: ECC disabled, Data cache writeback
[guest0/uart0] [    0.000000] sched_clock: 32 bits at 24MHz, resolution 41ns, wr
aps every 178956ms
[guest0/uart0] [    0.000000] Built 1 zonelists in Zone order, mobility grouping
 on.  Total pages: 24384
[guest0/uart0] [    0.000000] Kernel command line: root=/dev/ram rw ramdisk_size
=0x1000000 earlyprintk console=ttyAMA0 mem=96M
[guest0/uart0] [    0.000000] PID hash table entries: 512 (order: -1, 2048 bytes
)
[guest0/uart0] [    0.000000] Dentry cache hash table entries: 16384 (order: 4,
65536 bytes)
[guest0/uart0] [    0.000000] Inode-cache hash table entries: 8192 (order: 3, 32
768 bytes)
```

Scheduling

- Xvisor is basically a RTOS
- A thread in Xvisor is a “vcpu”
- Xvisor provides a priority-based time slicing scheduler policy
- guest OS know nothing about Xvisor



VCPU Execution

- Guest OS is not expected to release the CPU during its initial state
- While executing WFI instruction...
  - with VE (Virtualization Extension) configuration

    WFI is configured to trap into HYP mode
  - with Non-VE configuration

    WFI is converted into single SVC call during build time

Boot up

- Generate system image

```
tools/scripts/memimg.py -a 0x70010000-o build/qemu.img  \
build/vmm.bin@0x70010000   \
build/tests/arm32/pb-a8/basic/arm_test.bin.patched@0x70800000
```

- Launch QEMU

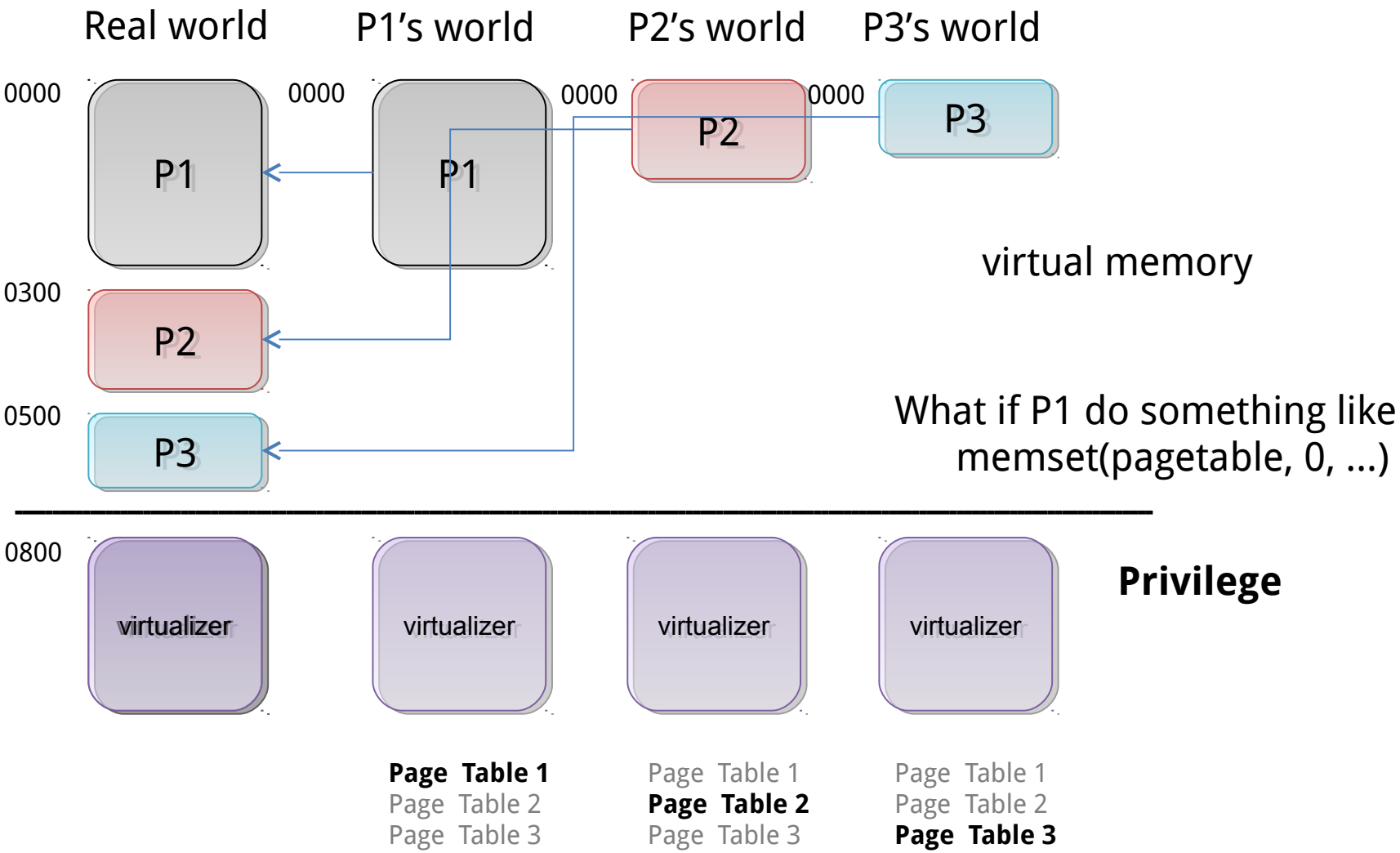
```
qemu-system-arm -M realview-pb-a8 -display none -serial
stdio -kernel build/qemu.img
```

- QEMU starts by executing the image head with vmm.bin. The vmm.bin is linked on VA = 0xff000000 during compiling time and its designed to be able to load with any address. And it relocates itself as follow
  - Copy itself into a ARM-MMU-Section aligned (1M boundary) address

    e.g. 0x80010000-> 0x80000000
  - Enable MMU and mapping its VA to load address

    e.g. 0xFF000000 → 0x80000000

- Then vmm.elf setup data structures to manage all physical page in system
  - The management scheme is buddy allocator.


Real world
P1's world
P2's world
P3's world
0000
0000
0000
0000
P1
P1
P2
P3
0300
P2
0500
P3
virtual memory
What if P1 do something like
memset(pagetable, 0, ...)
0800
virtualizer
virtualizer
virtualizer
virtualizer
Privilege
Page Table 1
Page Table 2
Page Table 3
Page Table 1
Page Table 2
Page Table 3
Page Table 1
Page Table 2
Page Table 3

Memory Virtualization

- Start the guest0/cpu0 vcpu

  `Xvisor>guest kick guest0`
- The guest hardware configure is defined with an standard Device Tree (DT) format http://devicetree.org/Main_Page

Memory in DT

```
vcpu0 {
    device_type = "vcpu";
    compatible = "ARMv7a,cortex-a8";
    start_pc = <0x40000000>;
}
mem1 {
     guest_physical_addr = <0x70000000>;
     host_physical_addr = <0x82000000>;
     physical_size = <0x06000000>; /* 96 MB */
     device_type = "ram";
};
nor_flash {
    manifest_type = "real";
    address_type = "memory";
    guest_physical_addr = <0x40000000>;
    host_physical_addr = <0x80800000>;
    physical_size = <0x00800000>;
}
```

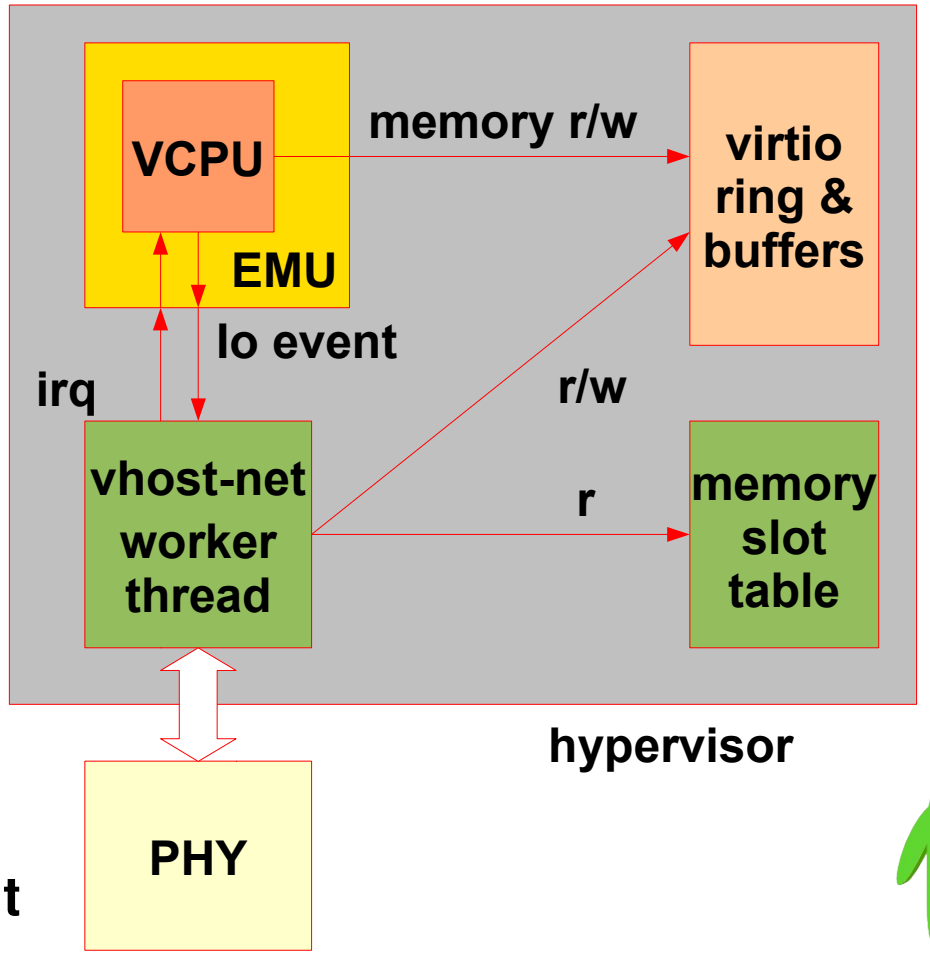
vmm.bin applies a on-demand paging scheme to support space virtualization, In this example, The start address 0x40000000 triggers the prefetch abort which is then catched by vmm.bin

The vmm.bin parse the configure file and find that 0x40000000 is mapping to 0x80800000.
Then Its setup the mapping and resume guest program

Device Virtualization

Device Virtualization Bigpicture

- userspace device emulation
- Paravirtualized device drivers (VirtIO)



The vhost-net model

- Host user space opens and configures kernel helper
- virtio as guest-host interface

Enables multi-gigabit throughput

Device Virtualization in DT

```
gic0 {
    manifest_type = "virtual";
    address_type = "memory";
    guest_physical_addr = <0x1E000000>;
    physical_size = <0x2000>;
    device_type = "pic";
    compatible = "realview,gic";
    parent_irq = <6>;
};
```

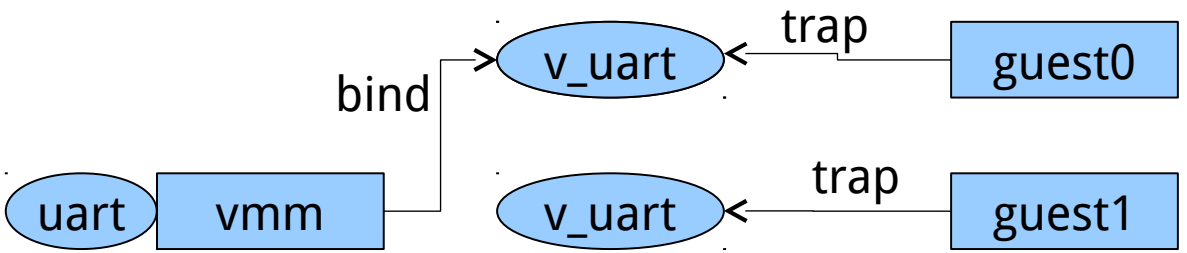
The device virtualization also utilizes the device tree.

For all device region, The Xvisor setups a mmu mapping with a "no r/w permission" attribute in the page table.

Everytime the guest program access the device memory, a page fault is triggered and CPU jump into the data_abort handler.
By decoding the instruction pointed by the fault address, the data-abort handler emulate the device behavior before resume guesst VCPU

Device Emulator

- Since the device is actually a plain memory with its functionalities emulated by software, the multiplex could be easily implemented as following:



- Guest OS runs in pure user-mode, and Xvisor applies the V5 domain field in the page table to emulate the privilege level for the guest OS.

Reference

- ARM Virtualization: CPU & MMU Issues, Prashanth Bungale, vmware
- Hardware accelerated Virtualization in the ARM Cortex™ Processors, John Goodacre, ARM Ltd. (2011)