KVM/ARM

Marc Zyngier
<marc.zyngier@arm.com>

LPC'12

ARM Architecture Virtualization Extensions 1/2

The Virtualization Extensions, introduced with the latest revision of the ARMv7 architecture, is based around a new Hypervisor execution mode (aka HYP, aka PL2).

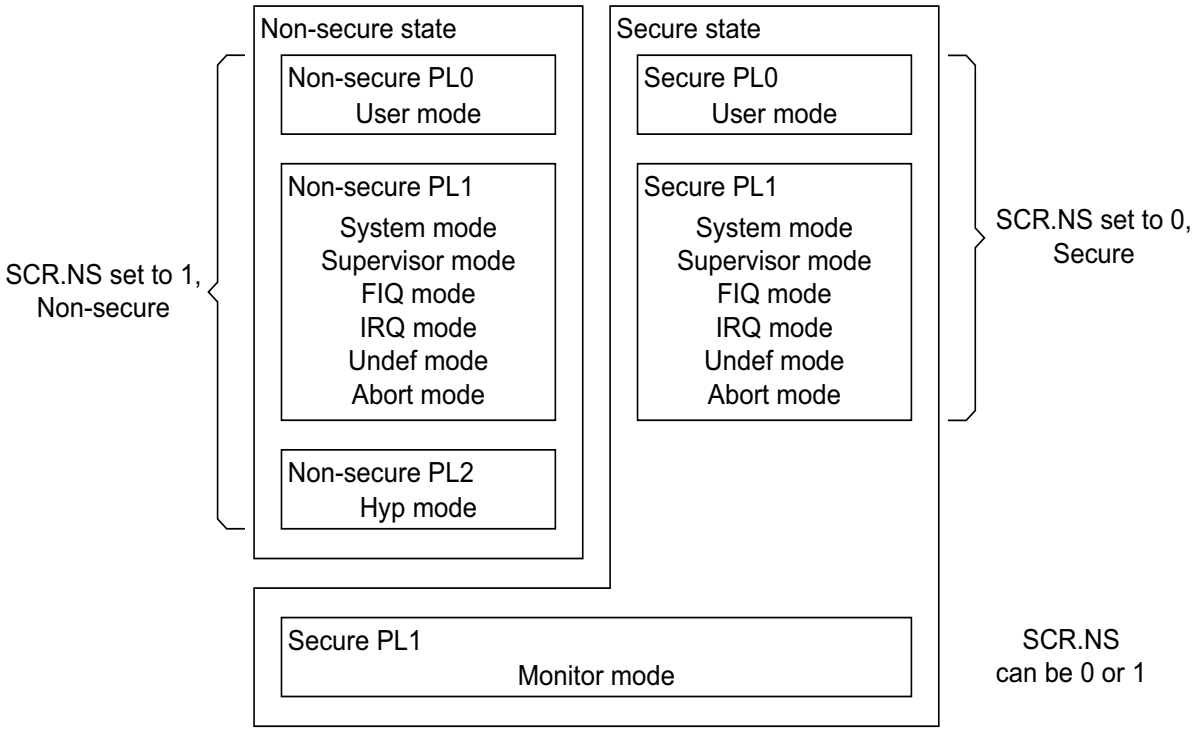

Figure : ARM execution privilege levels

ARM Architecture Virtualization Extensions 2/2

- Non-secure world, higher privilege than SVC
- Second stage translation
  Add an extra level of indirection between guests and physical memory
  TLBs are tagged by VMID
- Ability to trap access to most system registers
  The hypervisor decides what it wants to trap
- Can handle IRQs, FIQs and asynchronous aborts
  The guest doesn't see physical interrupts firing, for example
- Guests and host can call into HYP mode (HVC instruction)
- Standard peripherals HYP aware
  GIC and timers have specific virt-ext features

HYP mode: Not SVC++

Despite HYP mode having a higher privilege level than SVC, it is not a superset of PL1, as its features are quite different.

- Own translation regime:
  Separate stage 1 translation, no stage 2 translation
  Broadly follow the LPAE format (no "classic" page tables)
  Only one Translation Table base Register (HTTBR)
- It would be very difficult to run Linux in HYP mode
  Requires too many changes to be practical
- Instead, the HYP mode can be used as a "world switch" between guests (bare metal) or host and guests (hosted)

KVM/ARM: General architecture

KVM/ARM uses the HYP mode to context switch from host to guest, and back. It performs the following tasks:

- Save and restore host and guest contexts
  Stage-2 Translation Table, trap configuration, GP registers, system control registers, VFP...
- Preempts the guest on:
  - Physical interrupt delivery,
  - Stage-2 translation fault,
  - HVC or SMC,
  - WFI (idle),
  - A few privileged system registers,
  - Some rare cache maintenance operations
- On guest exit, the control is restored to the host
  Handles the exit reason (interrupt, page fault...)
- Hypervisor has no influence at all while running the host
- No nesting. Yet. ;-)

KVM/ARM: Memory management

KVM/ARM: Memory management

- Host in charge of all memory management
  It has no stage-2 translation itself (saves on TLB entries)
- HYP only has access to a few key host structures (kvm, vcpu)
- Guests are in total control of their own page tables
  The host controls their stage-2 translation tables
  No shadow page tables
- Easy to map a real device into the guest "physical" space
  This is how it can access the VGIC CPU interface
- Emulated devices do not have a stage-2 translation
  Access trapped and emulated in kernel or user space
- All cache and TLB accesses are tagged by VMID
- 4k pages only
  Using huge pages, or at least the same granularity as the guest would be a lot better. Requires some QEMU changes.

KVM/ARM: Instruction emulation

- MMIO access triggers a stage-2 translation fault
- Most instructions are described in the HSR
  No need to read back the instruction, everything is already there!
- A handful of instructions must be handled separately
  - Mostly load/store with register writeback
  - This can be racy when combined with TLB invalidations from other cores
  - Pending patch to eliminate these instructions from MMIO accessors in the Linux kernel
- Added complexity due to having to handle multiple ISAs (ARM and Thumb)
- Test suite being written to verify it in a systematic way

KVM/ARM: Host interrupt handling

While running a guest, all physical interrupts are taken in HYP:

- This causes the guest to exit
- We leave the interrupt pending and return to the host
- The pending interrupt will kick in when the host unmasks the interrupts

When the host is running, interrupts are directly handled in SVC. This scheme makes it very simple to handle interrupts, and requires no modification of the kernel.

KVM/ARM: Guest interrupt injection

There are two ways of injecting an interrupt in a guest:

- An architected way, by indirectly manipulating the I, F and A bits in the guest (the whole interrupt controller has to be modelled).
- Using the virtual GIC extensions, which offer a GIC CPU interface that can be mapped into the guest, and controlled by the host.

In both cases, we only inject "virtual" interrupts.
It should be possible to tie a physical interrupt to a guest, but we still lack some of the infrastructure.

KVM/ARM: Booting protocol

Until recently, KVM/ARM used an ad-hoc boot protocol, relying on a secure monitor to be installed by the boot-loader.
During Linaro Connect Q1.12, it was decided that a HYP aware kernel should be entered in HYP mode, and not rely on external services.

- Uses a hypervisor API, not specific to KVM
  - Relies on the kernel being entered directly in HYP mode
  - The kernel installs a HYP stub and drops back to SVC
- KVM uses this stub to install itself, and restores the stub when exiting

The API is simple and universal enough for other hypervisors to use the same! It also matches the Xen requirements.

KVM/ARM: Current status / Upstreaming plans

v11 (aka Spinal Tap) in development, with a lot of new features being merged:

- HYP mode boot
- Virtual GIC
- Thumb2 MMIO emulation

Missing features:

- in-kernel timers (code is ready for merging)
- Stable userspace ABI (MSR, IRQ injection)

Upstreaming plans:

- HYP mode boot has to go in first
  Set of patches sent as RFC to LAKML
- A handful of infrastructure patches in a second round
  mm setup, generic timers, PMU
- The bulk of the KVM code will be able to go in then
  We still need people to review the code!

KVM/ARM: `Signed-off-by:`

- Christoffer Dall (U. of Columbia, Virtual Open Systems)
  Main contributor, glorious leader
- Peter Maydell (Linaro)
  QEMU, walking ARM ARM
- Antonios Motakis (Virtual Open Systems)
  VMID allocator, VFP switching
- Rusty Russell (Linaro)
  CP15 interface, VFP switching, QEMU, awesome hacking
- Marc Zyngier (ARM)
  mm, SMP, VGIC, timers, HYP mode boot, bug chasing, permanent moaning