Modeling Virtual Machines and Interrupts in TLA+ & PlusCal

Valentin Schneider <valentin.schneider@arm.com>

18th of July 2018

Outline

Context

TLA+ at Arm

- Catalin Marinas (Arm Linux Kernel tech lead)
- Linux kernel models
  - ASID allocator - two processes getting the same ASID
  - Ticket spinlocks - two CPUs entering critical section
  - Queued spinlocks - liveness not guaranteed for all processes

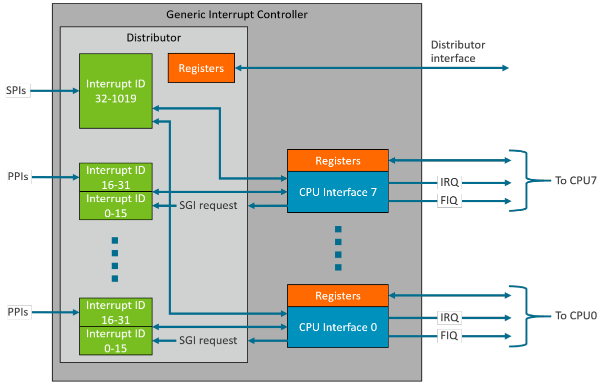
What this presentation is about

- Arm Generic Interrupt Controller (GIC)
  - Programmable interrupt controller
  - Gateway between CPUs and interrupts
  - Model based on GICv2
- Kernel-based Virtual Machine (KVM)
  - Virtualization infrastructure
  - Lets the Linux kernel take the role of a hypervisor
  - Allows us to run VMs (guests)
  - Uses the GIC to present interrupts to guests
  - Model based on Linux v4.16

- What can go wrong?
  - Misbehaving/malicious guests
  - Buggy hypervisor

Technical details

The Generic Interrupt Controller (GIC)

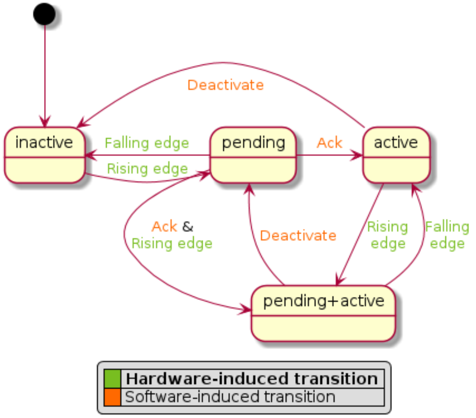


- Interface for handling interrupts
  - Processor has one (or two) interrupt lines
  - GIC can assert that line: CPU is interrupted
  - CPU interacts with the GIC to know more
- Types of interrupts
  - PPI: per-CPU hardware interrupt
  - SPI: global hardware interrupt
  - SGI: per-CPU software-generated interrupt

Internal interrupt state

- Combinations of (pending, active)
  - pending = can be signaled, waiting to be serviced
  - active = cannot be signaled, being serviced
- Transitions caused by
  - Hardware - Interrupt signal changes
    - Affects the pending bit
    - Depends on edge or level transitions (state machine variations)
  - Software - CPU interface interaction
    - Affects the active bit
    - Set with Acknowledge
    - Cleared with Deactivate
- Individual interrupts can be entirely disabled

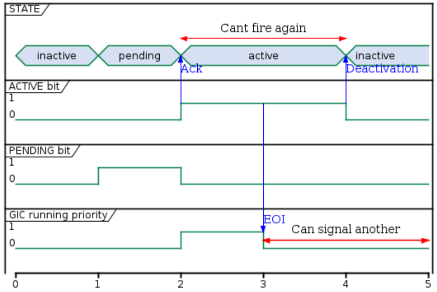
Edge sensitive interrupt state machine



Choosing an interrupt to signal

- CPU interrupt signal raised when at least one interrupt
  - is enabled
  - is pending
  - has a high enough priority
- Priorities
  - Priority mask
  - Running priority
- Acknowledge updates the running priority
  - Interrupts of lesser (or equal) priority can no longer be signaled

- Priority drop - end of interrupt (EOI)
  - Still active so can't be fired again
- Deactivation - clears the "active" state
- Threaded IRQs, Virtual Machines

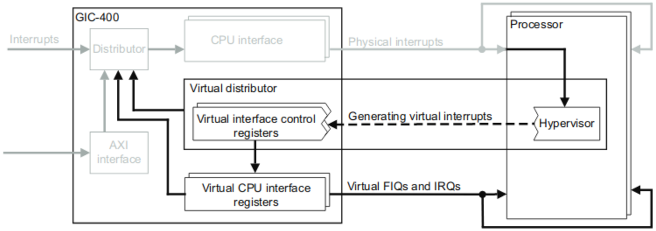


Virtualization extensions

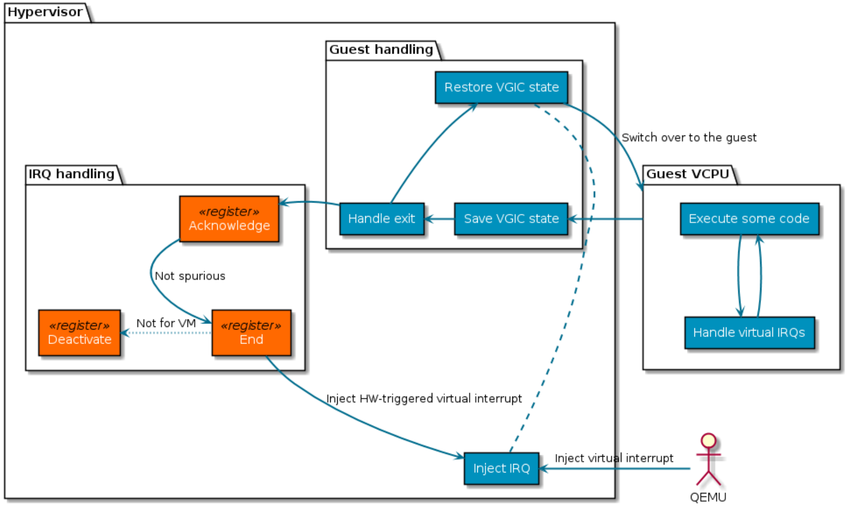
- Virtual Machines need interrupts too!
  - All GIC interactions could be trapped, (interrupt controller emulation)…
  - …But that is slow! (see Raspberry Pi)
  - Extra hardware to help us out

What the vGIC gives us

- GIC virtual interface
  - Similar interface (Ack, End, Deactivate)
  - Isolated from physical Distributor
  - List Registers (LRs)
    - Fills (part of) the role of the Distributor for VMs
    - Describe interrupts
- Sources of virtual interrupts
  - Purely virtual (QEMU, inter-VCPU SGI)
  - Triggered by some hardware
    - Priority drop helps



The big picture



Things that can go wrong

- Misbehaving/malicious guest
  - Interrupts left active
  - The guest must not harm the host!
- Buggy hypervisor
  - Interrupts not being delivered to the guest VCPU
  - Misconfiguration of the GIC
    - Erroneous LR entries

Model

The models

- KVM (Hypervisor + guests)
  - PlusCal
  - Why? Software, collection of sequential steps
- GIC
  - TLA+
  - Why? Hardware, collection of simple, independent steps

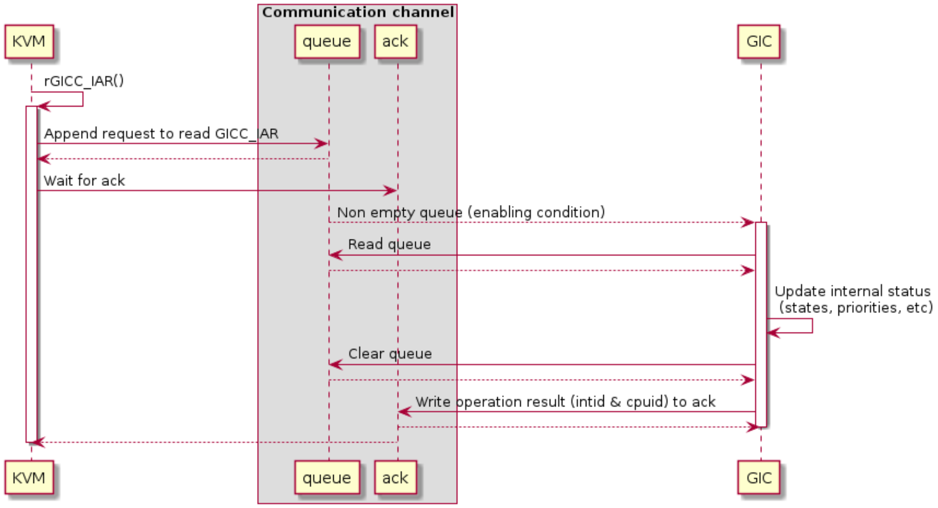
Combining models

- KVM model needs to use the GIC registers

- KVM model would need to modify GIC internal variables
  - UNCHANGED statements managed by PlusCal
  - Only works with operators PlusCal has visibility on
  - Can’t put GIC variables in PlusCal segment, since Inits will conflict

```
CombinedNext ≜
        ∨ KVMNext ∧ UNCHANGED gic_vars
        ∨ GICNext ∧ UNCHANGED kvm_vars
```

Communication channel - diagram


Communication channel
KVM
queue
ack
GIC
rGICC_IAR()
Append request to read GICC_IAR
Wait for ack
Non empty queue (enabling condition)
Read queue
Update internal status (states, priorities, etc)
Clear queue
Write operation result (intid & cpuid) to ack
KVM
queue
ack
GIC

Communication channel - logic

```
(* Single CPU example *)
procedure rGICC_IAR()
{
write:
        await Len(queue) = 0;
        queue := <<"rGICC_IAR">>;
read:
        await Len(ack) > 0;
        intid := ack[1];
        ack := << >>;
        return;
}
```

```
ReadQueue ==
        (* Enabling condition *)
        /\ Len(queue) > 0
        /\ CASE queue[1] = "rGICC_IAR" ->
                /\ ... (* Update internal status *)
                /\ ack' = <<intid, cpuid>>
                /\ queue' = << >>
```

GIC next-state predicate

- ReadQueue
  - Non empty communication channel
  - Emulate requested register access, reply in `ack`
- Randomly assert interrupt ligne
- Pending signal update
  - Read status of interrupt lines
  - Update internal pending signal
- Interrupt pending bit update
  - Read internal pending signal
  - Update relevant interrupts
- Update interrupt signals
  - Interrupt (virt or phys) can be signaled
  - Raise interrupt signal (virt or phys)
- Update maintenance interrupt signals
  - Condition for raising maintenance interrupt met
  - Set maintenance interrupt as pending

Context switching - tasks definition

- These processes must not run in parallel - they have to share the CPU!
- The Hypervisor must explicitely let the Guest run (world switch)
- The Guest can be halted at any moment and let the Hypervisor run instead

```
HYP_TASKS ≜ {HYP_TASK} × CPUS
VCPU_TASKS ≜ VMS × VCPUS

fair process(hypervisor ∈ HYP_TASKS)
fair process(vcpu ∈ VCPU_TASKS)
```

```
context = [cpu ∈ CPUS : ↦ CHOOSE task ∈ HYP_TASKS : task[2] = cpu]
ContextSwitch(cpu, next) ≜ context' = [context EXCEPT ![cpu] = next]

TaskEnabled(self) ≜ ∃ cpu ∈ CPUS : context[cpu] = self
```

Context switching - TaskEnabled

- `TaskEnabled(self)` becomes an enabling condition of every PlusCal step

```
(* PlusCal output *)
out_(self) ≜ ∧ pc[self] = "out_"
             ∧ ...

(* Modified output *)
out_(self) ≜ ∧ pc[self] = "out_" ∧ TaskEnabled(self)
             ∧ ...
```

- Initially, only the Hypervisor threads will be enabled
- It will eventually call `ContextSwitch(cpu, vcpu)`

Context switching - Communication channel tricks

- Processes should not be context switched during a GIC ”transaction”
- Only relevant to the model

```
CanContextSwitch(cpu) ≜
        ∧ Len(queue[cpu]) = 0
        ∧ Len(ack[cpu]) = 0
```

KVM next-state predicate

- Execute an instruction
  - Next-state predicate automatically generated by PlusCal
  - Will run whatever task is currently active on a given CPU
  - Hypervisor: save -> handle exit -> restore -> run guest (&loop)
  - Guest: just virtual interrupt handling
- Exit the guest
  - Guest is running but physical interrupt signal is raised
  - Need to return to the hypervisor to handle it
- Branch to the interrupt handler
  - Hypervisor is running and physical interrupt signal is raised (and irqs allowed)

Results

Property checking

- Liveness properties
  - An interrupt is eventually delivered, IOW the guest sees it
  - SGI (or even PPI) delivery liveness property check runs out of memory after a few hours (32GB)
- Safety properties
  - Sane GIC interaction (e.g. correct LR values)
  - Failure after ~ 150 steps - invalid data regarding virtual SGI
    - Loss of information, issue fixed while model was being written

End product

- GIC model
  - Priority handling, virtualization extensions
- KVM model
  - Context switches
  - Virtual interrupt delivery
  - Instruction traps
- Yet more work to do
  - GICv2 is quite simpler than GICv3+
  - KVM code has progressed since then
- Others
  - Emacs TLA+ mode improved for PlusCal syntax highlighting

Lessons learned

- 1:1 matching between code and model pitfall
  - Take a step back, see what are the actual logical steps
  - No need for a loop to populate the List Registers
- Large tracebacks are not always easy to analyze
  - Regexps can help a bit
- A few PlusCal/TLA+ issues to circumvent
  - Talk about it at the end of the workshop?

Questions

arm

Thank you!

Tweaking the specification

```
∧ Init ∧ GIC!Init
∧ □[CombinedNext]_<<vars, hw_vars>>

(* Eventually run the guest (Forbid interrupt storms) *)
∧ ∀ cpu ∈ CPUS :
∧ context[cpu] ∈ HYP_TASKS ↝ context[cpu] ∈ VCPU_TASKS

(* When running the guest, eventually do a full loop *)
∧ ∀ cpu ∈ CPUS :
        ∧ context[cpu] ∈ VCPU_TASKS ∧ ¬ ENABLED guest_init(context[cpu]) ↝
                ENABLED guest_init(context[cpu])
        ∧ context[cpu] ∈ VCPU_TASKS ∧ ENABLED guest_init(context[cpu]) ↝
                ¬ ENABLED guest_init(context[cpu])

\* BEGIN FAIRNESS
...
\* END FAIRNESS
```

HW-tied interrupts

- Need to forward an e.g. programmed timer
- Hypervisor does a priority drop on it, but it stays active
  - Prevents the host from taking it again
- An equivalent IT is created in the LRs and linked to the real interrupt
- Virtual deactivation automagically triggers a physical deactivation
  - No need to exit the guest on virtual deactivation!

SGI NPIE issue

- SGI issue, NPIE (No Pending Interrupt Enable)
  - Fired when no pending (and only pending) IT in LRs
  - Condition met when interrupt goes from Pending to Active
  - Loss of information when saving the vgic state
  - Fixed while the model was being written, but a good target for fidelity