For Performance and Latency, not for Fun

How to overcome the Big QEMU Lock

Jan Kiszka, Siemens AG, Corporate Technology
Corporate Competence Center Embedded Linux
jan.kiszka@siemens.com
Agenda

- Motivation
- Locking rules
- Memory access dispatching
- Back end / front end interaction
- Exemplary cut-through
- Conclusion
Motivation: Concurrency in QEMU/KVM

- Device models
- I/O back-ends
- GUI
- QMP, HMP
- ...

Examples:
- pulseaudio
- Posix AIO
BQL – One after the other

Einordnen lassen
Pros & Cons

Limitations
- Scalability bottleneck for high-speed I/O
- Causes high latencies, unacceptable for real-time workloads

Benefits
- Simple model, easier to get right
- Well confined, most subsystems do not need to bother
Requirements for New Concurrency Scheme

Improvements for scalability and latency
- Enable decoupled I/O paths with different priorities
- Flexible locking policies, also allowing lock-free schemes

Integration / migration of BQL-dependent components
- Device models
- I/O back ends & timers
- TCG system & user emulation

Compact & comprehensible concept
- Consistent scheme with few or no exceptions
- Low impact on device models
Fine-grained Locking – and all will be good!? 

Source: Glauber Costa
Lock Ordering Rules

Big before small
  - Big = coarse-grained, small = fine-grained

Reasoning
  - Ordering avoids ABBA
  - Risk of priority inversions:
    Waiting on big lock while holding small one
    turns small into big

Implications
  - BQL-dependent services cannot be called while holding finer locks
  - While holding the BQL, any lock can be taken in addition
Lock Ordering Rules (2)

While holding lock A, do not call anything that takes lock B if you can be called back to take A while B is locked
  - Examples:
    Device A triggers access to device B triggers access to Device A
    or
    Context 1: back end A triggers access to device B
    Context 2: device B triggers access to back end A

Reasoning
  - Avoid lock recursion
  - Avoid ABBA deadlock

Implications
  - Managing mutual access of devices and backends will be tricky
Critical BQL Zones (from last year's talk)

CPUState
- Read/write access
- cpu_single_env

PIO/MMIO request-to-device dispatching
Coalesced MMIO flushing

Back-end access
- TX on network layer
- Write to character device
- Timer setup, etc.

Back-end events (iothread jobs)
- Network RX, read from chardev, timer signals, ...

IRQ delivery
- Raising/lowering from device model to IRQ chip
- Injection into VCPU (if user space IRQ chips)
address_space.lock()
region_section = look_up(address)
reference_held = region_section.reference()
address_space.unlock()
if (reference_held) {
    /* means: use fine grained locking */
    region_section.access_handler(…)
    region_section.unreference()
} else {
    /* use BQL */
    bql.lock()
    address_space.lock()
    region_section = look_up(address)
    address_space.unlock()
    region_section.access_handler(…)
    bql.unlock()
}
Memory Region Reconfiguration

add/remove/enable_memory_region()
for_each_address_space()
  address_space.lock()
  address_space.update_topoogy()
  address_space.unlock()

Implications
- May but need not run under BQL
- Access possible after disabling/removing
- Memory region must not vanish after removal

Address space locking alternatives
- RCU
  => accelerates read path
- Stop VM
  => cannot be triggered from region access handlers
Handling Destruction

Referencing section locks down memory region owner
- Object (opaque) addressed via callback must not vanish
- Proposal by Liu Ping Fan
  - New memory region ops for reference/unreference
  - Region owner implements callbacks to reference QOM object (e.g. device)
    => Boilerplate code in device models
- Alternative: pass QOM object (not qdev!)
  - ...replacing opaque
  - ...in addition to opaque, as “owner”

Object destruction on last reference release

Challenge: Race between destruction and callback execution
- Rule: callback must not “re-activate” object
Prevent Dispatch Nesting

Prevention approach
- Reject nested MMIO access, still allowing RAM access
- Uses thread local variable to track nesting

Impact
- Lock recursion
- ABBA deadlock between devices
- May prevent few valid corner cases (still looking for examples...)
Generalization:
Event Dispatching & Callback Management

Reuse these patterns!

Candidates
- Memory regions
- Timers
- File descriptor callbacks
- Event notifiers
- ...

© Siemens AG, Corporate Technology
Jan Kiszka
Locking of Front Ends and Back Ends – Separate Locks

Device A, MMIO write
device.lock()
qemu_mod_timer()

Alarm timer X, modify
timer.lock()
modify_timer_list
set_alarm_timer
...

Alarm timer X, timer thread
wait_expiry()
timer.lock()
callback = lookup_n_ref()
timer.unlock()
callback()

Device A, timer handler
device.lock()
handle_timeout()
...
Locking of Front Ends and Back Ends – Back End as Library

Device A, MMIO write

device.lock()
qemu_mod_timer()

Alarm timer X, modify

modify_timer_list
set_alarm_timer
...

Device A, timer thread

wait_event()
device.lock()
alarm_timer_check()

Alarm timer X, expiry

callback = lookup(timer)
callback()

Device A, timer handler

handle_timeout()
...

...
“Let's use glib's main loop!”

Advantages
- Abstractions for event handling on all supported host platforms
- Can obsolete many lines of code in QEMU

Show-stopper
- Uses internal locks in an uncontrollable way
- Locks are incompatible with RT prioritization

=> OK for main (best effort) I/O thread,
no-go for real-time I/O paths
Managing Legacy

Motivation:

- BQL will be present for a long time, maybe forever

How to create BQL-free services?

- Keep existing interfaces
- Provide BQL-free alternatives

=> Existing code continues to work
    (TCG, device models, ...)

=> No need to convert “uninteresting” subsystems
    (UI, slirp, ...)
Direct IRQ Forwarding (slide from last year)

Typical IRQ path
- Device changes level / generates edge
- IRQ routers (PCI host, bridges, IRQ remapper, etc.) forward to interrupt controller
- Interrupt controller forwards to CPU

=> **Routing involves** multiple device models, i.e. potentially **multiple critical sections**

Cannot take the long road if source & sink are in-kernel
- Hacks exist to explore and monitor routes – on x86

=> Generic mechanism required

**Fast path from device to target CPU**
- No interaction with routing devices
- State changes (reroutes, blockings) reported to subscribers
- Routing device states can be updated on demand

PCI-specific workaround merged for vfio & pci-assign
Scenario: 
(Partially) Decoupled PC RTC Device Model

Use case
- Real-time capable periodic timer

Requirements
- BQL-free periodic timer IRQ
- BQL-free read of register C (IRQ cause)
- BQL-free write of index register

Derived requirements
- BQL-free PIO dispatching
- BQL-free alarm timer backend
- Strategy to avoid complete conversion
Scalable Clock/Timer Subsystem

Clock issues
- CLOCK_REALTIME works without locks
- CLOCK_HOST requires dedicated lock for reset detection
- CLOCK_VIRTUAL requires lock for timers_state – but then stumbles over icount

Timer issues
- Multi-instance support required, binding to separate threads
- Preferred future model yet open
  - timerfd (+ current signal code as fallback)
  - select/poll timeouts
Prototype Results

First cut-through
- Unlocked PIO dispatch
- Flag controls BQL need per memory region
- Multi-instance alarm timer (dynticks only)
- mc146818rtc changes
  => Guest accepted RTC as reliable clock source

Not considered (means: left broken behind)
- PIO hotplug => keep hands off devices!
- HPET control over RTC => -no-hpet
- Lost tick compensation => -global [...].lost_tick_policy=discard
- VM-clock based RTC => -rtc clock=host|rt
- IRQ delivery in TCG mode => -enable-kvm
Summary

Down with the BQL!
- Limits I/O scalability
- Prevents RT use cases

Locking is hard, so let's use more of it!
- Fine grained locking can help
- Strict ordering rules, nesting prevention required

Lots of fun ahead!
- Subsystems require BQL-free interfaces
- Device models need to be converted
- Likely some tricky corner cases remaining...

Work toward cut-through!
- Generic show case needed, e.g. low-latency networking via E1000
- Further suggestions welcome => RT-KVM BoF
Any Questions?

Thank you!