Effective multi-threading in QEMU

Paolo Bonzini
Principal Software Engineer, Red Hat
KVM Forum 2013
Why effective?

- Use well-known idioms and mechanisms
- Can be implemented in tiny steps
- Give some benefits and lay the groundwork for future improvements
Outline

- QEMU architecture
- virtio-blk-dataplane architecture
- Unlocked memory dispatch
- Unlocked MMIO
QEMU architecture (up to 0.15)

- select()
- Bottom halves
- Timers
- cpu_exec (TCG/KVM)

Poll for events

Timer tick (1 ms)
QEMU architecture (1.0)

- select()
- Bottom halves
- Timers
- Event loop
- I/O thread

The big QEMU lock!
QEMU thread structure

```c
for (;;) {
    slirp_pollfds_fill();
    qemu_iohandler_fill();
    g_main_context_prepare();
    g_main_context_query();

    qemu_mutex_unlock_iothread();
    poll(...);
    qemu_mutex_lock_iothread();

    if (g_main_context_check()) {
        g_main_context_dispatch();
    }
    slirp_pollfds_poll();
    qemu_iohandler_poll();
}

for (;;) {
    kvm_arch_put_registers(cpu);
    kvm_arch_pre_run(cpu);

    qemu_mutex_unlock_iothread();
    kvm_vcpu_ioctl(cpu, KVM_RUN);
    qemu_mutex_lock_iothread();

    kvm_arch_post_run(cpu);
    switch(run->exit_reason) {
        case ...
    }
}
```
QEMU architecture (now)

- Paolino Bonzini – KVM Forum 2013

- I/O thread
- select()
- nsec
- Timers
- BH
- fd
- Timers
- AioContext
- GMainLoop
- Event loop
- I/O thread

- Migration
- VNC
- SPICE
- Smartcard

- cpu_exec (KVM)

Paolo Bonzini – KVM Forum 2013
Enter virtio-blk-dataplane

Direct Random I/Os at 4KB Block Size - Single Guest

Host: Intel E7-8870@2.4 GHz, 40 cores, 256GB

- w/data-plane
- KVM virtio-blk
- Hyper-V
- vSphere 5.1

1000 I/O's per Second (kIOPS)
Enter virtio-blk-dataplane

- select()
- Timers
- AioContext
- GMainLoop
- Event loop
- I/O thread
- cpu_exec (KVM)
- AioContext dataplane
- cpu_exec (KVM)
- AioContext dataplane

Magic?
Dataplane architecture

- AioContext
- Memory API
- Block layer
- I/O thread

- AioContext
- Hostmem
- Linux-aio
- Dataplane
Lessons learned from dataplane

- Most of the time, the BQL is not a bottleneck
- Never take the BQL in “thread-centric” code
- Modularize existing code
  - Isolate data structures per thread
  - Avoid locks altogether
- Prototypes are great, but better have a plan
Dataplane architecture

- AioContext
- Memory API
- Block layer
- I/O thread
  
- AioContext
- hostmem
- linux-aio
- dataplane
The plan

- Memory dispatch
- Memory API
- Block layer
- AioContext
- I/O thread

- Memory dispatch
- Block layer
- AioContext
- dataplane
Lessons learned from dataplane

- Most of the time, the BQL is not a bottleneck
- Never take the BQL in “thread-centric” code
- Modularize existing code
- Prototypes are great, but better have a plan
- The BQL is going to stay for a long, long time
Memory API data structures

address_space_memory

- offset=0x0
  MemoryRegion, size=64MB
  RAM

- offset=0x1fc00000
  MemoryRegion, size=8K
  I/O callbacks 0

- offset=2K
  MemoryRegion, size=4K
  I/O callbacks 1

FlatView

- MemoryRegionSection
  mr-relative 0x0...0x3fffffff
  as-base 0x0

- MemoryRegionSection
  mr-relative 0x0...0x7ff
  as-base 0x1fc00800

- MemoryRegionSection
  mr-relative 0x0...0x10
  as-base 0x1fc00000

AddressSpaceDispatch

Paolo Bonzini – KVM Forum 2013
What's behind DMA?

**address_space_map**
- Visit radix tree
- If source is MMIO, call I/O read ops
- Return address of mapped memory

**address_space_unmap**
- Find MemoryRegion
- If source was MMIO, call I/O write ops
- If source was RAM, mark it as dirty

**Dataplane (hostmem):**
- Binary search list of RAM regions
- Return address

**Dataplane (hostmem):**
- Nothing :-)  
  (Migration falls back to non-dataplane)
Memory API data structures

- Dataplane threads can use binary search on FlatView
- AddressSpaceDispatch is faster but not needed in the short term
- Still needed in the longer term for dirty bitmap/live migration
Immutable data structures

- Recreate FlatView from scratch on every update (cost: 1 extra malloc/free)
- Reference count FlatView, take reference while visiting

```c
qemu_mutex_lock(&flat_view_mutex);
old_view = as->current_map;
as->current_map = new_view;
qemu_mutex_unlock(&flat_view_mutex);
flat_view_unref(old_view);
```

```c
qemu_mutex_lock(&flat_view_mutex);
view = as->current_map;
flat_view_ref(view);
qemu_mutex_unlock(&flat_view_mutex);
...;
flat_view_unref(view);
```

- Result: very small critical sections
Extending reference counting

- Current memory API does not work well with hot-unplug
- If a device disappears while I/O is in progress:
  - MemoryRegions go away between `address_space_map` and `address_space_unmap`
  - QEMU can access dangling pointers
- Solution: add an *owner* to the MemoryRegion
Memory data structures lifetimes

FlatView

MemoryRegion

MemoryRegionSection

MemoryRegion

MemoryRegionSection

MemoryRegionSection

Device 1

Device 2

owner

owner

Paolo Bonzini – KVM Forum 2013
Memory data structures lifetimes

Device 1
owner
MemoryRegion

Device 2
owner
MemoryRegion

MemoryRegionSection

address_space_map

Paolo Bonzini – KVM Forum 2013
Another problem: removal vs. reclamation

- Removal: remove references to data items within a data structure
- Reclamation: frees data items that are already removed from the data structure
- With reference counting, these two steps can happen at separate times
- QOM “unrealize” method currently does both!
Separating removal and reclamation

- Removal: make device inaccessible from guest
  - memory_region_del_subregion
  - Corresponds to current “unrealize” time

- Reclamation: free the data items
  - memory_region_destroy
  - When last reference goes away (instance_finalize)

- Not just memory regions (e.g. NIC, block device, etc.)
From reference counting to RCU

- Immutable data structure are the basis of RCU (Read-Copy-Update) technique popular in Linux
- RCU runs removal concurrently with readers
- Reclamation only starts after readers no longer hold references
RCU basics

- RCU is a bulk reference-counting mechanism!

```c
x = global;
global = y;
call_rcu(x, g_free);
```

```
rchu_read_lock()  rcu_read_unlock()
```

---

Paolo Bonzini – KVM Forum 2013
Why RCU?

- RCU avoids the need for fine-grained locking
  - The write side keeps using the BQL
  - Avoid reasoning about lock hierarchies
- RCU makes fast paths really fast
  - Little or no overhead on the read side
  - No need to take locks on hot TCG paths
### Converting FlatView to RCU

<table>
<thead>
<tr>
<th>C-Code</th>
<th>C-Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>qemu_mutex_lock(&amp;flat_view_mutex);</code></td>
<td><code>old_view = as-&gt;current_map;</code></td>
</tr>
<tr>
<td><code>old_view = as-&gt;current_map;</code></td>
<td><code>as-&gt;current_map = new_view;</code></td>
</tr>
<tr>
<td><code>as-&gt;current_map = new_view;</code></td>
<td><code>call_rcu(old_view, flat_view_unref);</code></td>
</tr>
<tr>
<td><code>qemu_mutex_unlock(&amp;flat_view_mutex);</code></td>
<td></td>
</tr>
<tr>
<td><code>flat_view_unref(old_view);</code></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C-Code</th>
<th>C-Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>qemu_mutex_lock(&amp;flat_view_mutex);</code></td>
<td><code>rcu_read_lock();</code></td>
</tr>
<tr>
<td><code>view = as-&gt;current_map;</code></td>
<td><code>view = as-&gt;current_map;</code></td>
</tr>
<tr>
<td><code>flat_view_ref(view);</code></td>
<td><code>flat_view_ref(view);</code></td>
</tr>
<tr>
<td><code>qemu_mutex_unlock(&amp;flat_view_mutex);</code></td>
<td><code>rcu_read_unlock();</code></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td><code>flat_view_unref(view);</code></td>
<td><code>flat_view_unref(view);</code></td>
</tr>
</tbody>
</table>

- The same technique can be applied to AddressSpaceDispatch
Implementation state

● MemoryRegion ownership: done
● Separate locking for FlatView: done
● Removing hostmem: patches posted
● RCU for FlatView: patches ready
● RCU for AddressSpaceDispatch: TCG?
● Lock-free address_space_rw/map/unmap: Missing dirty bitmap handling
A real-world trace

- 64.69%    _raw_spin_lock
- 48.06%    futex_wait_setup
  - 99.32% [qemu-system-x86_64] __lll_lock_wait
- 44.71%    futex_wake
  - 99.33% [qemu-system-x86_64] __lll_unlock_wake
The next step: lock-free MMIO/PIO?

for (;;) {
    kvm_arch_put_registers(cpu);
    kvm_arch_pre_run(cpu);

    kvm_vcpu_ioctl(cpu, KVM_RUN);

    kvm_arch_post_run(cpu);
    switch(run->exit_reason) {
    case KVM_EXIT_IO:
        address_space_rw(...);
        break;

    case KVM_EXIT_SHUTDOWN:
        qemu_mutex_lock_iothread();
        ...
        qemu_mutex_unlock_iothread();
        break;
    }
}
Q: Is it valid?
PCI in a nutshell

- PCI is a bus where you have reads and writes, interrupts are raised, etc.
- PCIe is a packet network that fakes the same
- PCIe packets go from the CPU to the devices and back
- Packets can be reordered only in limited ways
PCIe packets

- Packet types
  - Read
  - Read completion
  - Write to memory, including MSI
  - I/O or configuration write
  - I/O or configuration write completion

- Reordering packets must obey rules in the PCIe spec
Q: What kind of reordering would QEMU apply with unlocked MMIO?

A: For each CPU, everything is serialized

Multiple CPUs will not observe incorrect reordering if accesses are atomic
Atomicity (try 1)

- An operation is *atomic* if it appears to the rest of the system to occur instantaneously.
Atomicity (try 2)

- All operations should have a *linearization point*
- All operations appear to occur instantly at their linearization point
- Linearizability == atomicity!
Observing atomic operations

- Causal relationships ("happens-before") let an observer order the linearization points.

Linearization point must be after VCPU2 writes 1!
Is MMIO linearizable?

```c
mr = address_space_translate(as, addr, &addr, &len, true);
memory_region_dispatch_write(mr, addr, val, 4);

mr = address_space_translate(as, addr, &addr, &len, false);
memory_region_dispatch_read(mr, addr, &val, 4);
```

- No locks taken: assume I/O callbacks are atomic and have their own linearization point
- `address_space_translate`'s linearization point is where it fetches the AddressSpaceDispatch
- If the memory map is static, we can ignore it
- Otherwise, two linearization points are already one too many
Example

- Same register visible from two different BARs

Linearization point must have been before BAR0 was disabled!

Linearization point must be after VCPU2 writes 1!

- Contradiction: access not atomic!
But we already get it wrong

- Concurrency happens even with the BQL!
  - The BQL is released between `address_space_map` and `address_space_unmap`
- A translation returned by `address_space_map` can be used arbitrarily far in the future
- Example
  - `address_space_map` returns RAM address
  - bus-master DMA is disabled before unmap
  - Writes should be forbidden, but they happen!
So, does it matter?

- “Unlocked” MMIO/PIO is opt-in behavior
  - Use it for paravirtual devices
  - Use it for devices with a static memory map
- Double-checked locking
  - Take fine-grained lock, check as->dispatch didn't change; if it did, release lock and retry dispatch
  - Prevents fully BQL-free MMIO/PIO
- OSES quiesce devices before disabling DMA
- Answer: no
Experiments

- Microbenchmark using kvm-unit-tests
- Measure cost of accessing PM timer concurrently from multiple VCPUs
- Ivy Bridge processor with 4 CPUs
- Thanks to Jan Kiszka for the patches
MMIO cost breakdown (ACPI PM timer)

- pm timer
- user exit+dispatch
- emulator
- vm exit

Clock cycles:
- 1 VCPU
- 2 VCPU
- 4 VCPU
- 8 VCPU

BQL
Effect of removing the BQL (ACPI PM timer)

- 1 VCPU: 15700 clock cycles with BQL, 15200 clock cycles without BQL, 30% reduction
- 4 VCPU: 3% performance difference
- 8 VCPU: Performance difference not shown in the diagram

Paolo Bonzini – KVM Forum 2013
What's next?

- Upstream patches
  - Unrealize vs. instance_finalize
  - RCU
  - Unlocked I/O
- Complete switch of virtio-blk-dataplane to block layer
- Make dirty bitmap access atomic
- ... Fine-grained locking for TCG? (2014?)
Questions?
Atomic operations API – C++11 vs. QEMU

● `atomic_read(p) / atomic_set(p,v) → atomic_load(p,relaxed) / atomic_store(p,v,relaxed)`
● `atomic_mb_read(p) / atomic_mb_set(p,v) → atomic_load(p,seq_cst) / atomic_store(p,v,seq_cst)`
● `smp_mb() / smp_rmb() / smp_wmb() → atomic_thread_fence(seq_cst/acquire/release)`
● `atomic_fetch_add/sub/and/or(p,v) → atomic_fetch_add/sub/and(p,v,seq_cst)`
● `atomic_cmpxchg(p, old, new) → atomic_compare_exchange_strong(p,old,new)`
RCU API – Linux vs. QEMU

- Threads need to report quiescent states
  - rcu_quiescent_state()
  - rcu_thread_offline()/rcu_thread_online()
- Not needed for threads that use semaphores or condition variables
  - rcu_dereference(p) ➔ atomic_rcu_read(&p)
  - rcu_assign_pointer(p,v) ➔ atomic_rcu_set(&p,v)