QContext, and Supporting Multiple Event Loop Threads in QEMU

Michael Roth
mdroth@linux.vnet.ibm.com
QEMU Threading Model Overview

- Historically (earlier this year), there were 2 main types of threads in QEMU:
  - **vcpu threads** – handle execution of guest code, and emulation of hardware access (pio/mmio) and other trapped instructions
  - **QEMU main loop (iothread)** – everything else (mostly)
    - GTK/SDL/VNC UIs
    - QMP/HMP management interfaces
    - Clock updates/timer callbacks for devices
    - device I/O on behalf of vcpus
QEMU Threading Model Overview

- All core qemu code protected by global mutex
- vcpu threads in KVM_RUN can run concurrently thanks to address space isolation, but attempt to acquire global mutex immediately after an exit
- lothread requires global mutex whenever it's active
High contention as threads or I/O scale
QEMU Thread Types

- vcpu threads
- iothread
- **virtio-blk-dataplane thread**
  - Drives a per-device AioContext via aio_poll
  - Handles event fd callbacks for virtio-blk virtqueue notifications and linux_aio completions
  - Uses port of vhost's vring code, doesn't (currently) use core QEMU code, doesn't require global mutex
  - Will eventually re-use QEMU block layer code
QEMU Block Layer Features

- Multiple image format support
- Snapshots
- Live Block Copy
- Live Block migration
- Drive-mirroring
- Disk I/O limits
- Etc...
More dataplane in the future

- Scalable, high performance I/O with full feature support is a big win for users
- Likely to see more dataplane implementations in the future (virtio-scsi, virtio-net, NetClients?)
How do we manage these event loops?

- Ad-hoc event loop implementations?
- How to handle event assignment? 1 thread per device? What about multiqueue?
- Multiple devices per thread?
- Standard command-line syntax?
- Re-configurable at runtime?
QContext Overview

- Object that represents an event loop
  - QOM-based object, can be instantiated via -object
  - creates it's own event loop thread
  - unique id that can be passed to any devices that want to offload a set of events
- Each QContext can drive a set of event sources (AioContexts, GSources, etc)
- Can be managed/introspected via QOM properties
QContext basic usage

qemu -object qcontext,id=ctx1,threaded=yes \ 
  -device virtio-blk,x-data-plane=on,context=ctx1,...

qemu -object qcontext,id=ctx1,threaded=yes \ 
  -device virtio-blk,x-data-plane=on,context=ctx1,... \ 
  -object qcontext,id=ctx2,threaded=yes \ 
  -device virtio-blk,x-data-plane=on,context=ctx2,... \ 
  ...

...
QContext Overview

- Object that represents an event loop
  - QOM-based object, can be instantiated via -object
  - creates its own event loop thread
  - unique id that can be passed to any devices that want to offload a set of events

- Each QContext can drive a set of event sources (AioContexts, GSources, etc)
- Can be managed/introspected via QOM properties
Consolidating dataplane threads

```
qemu -object qcontext,id=ctx1,threaded=yes \ 
  -device virtio-blk,x-data-plane=on,context=ctx1,... \ 
  -device virtio-blk,x-data-plane=on,context=ctx1,... \ 
  ...
```
QContext Overview

- Object that represents an event loop
  - QOM-based object, can be instantiated via -object
  - creates its own event loop thread
  - unique id that can be passed to any devices that want to offload a set of events
- Each QContext can drive a set of event sources (AioContexts, GSources, etc)
- Can be managed/introspected via QOM properties
Consolidating dataplane threads

```
mdroth@loki:~$ qom-list /objects/
ctx1/
qcontext-main/
type

mdroth@loki:~$ qom-list /objects/ctx1
thread_id
threaded
id
type

mdroth@loki:~$ qom-get /objects/ctx1.thread_id
6787
```
Main Loop Event Sources

- Main Loop
  - IOHandler list
  - GSources
  - QEMUTimers
  - Slirp
- AioContext
  - AioHandlers
  - Bottom-Halves
  - QEMUTimers
Event Registration – IOHandlers

- qemu_set_fd_handler(fd, fd_read_fn, fd_write_fn, user_data)
- qemu_set_fd_handler2(fd, read_poll_cb, read_cb, write_cb, user_data)
- set_fd_handler2(ctx, fd, read_poll_cb, read_cb, write_cb, user_data)
- Needs to be thread-safe now (or does it?)
Thread-safe Event Registration/Modification

- Just use a simple mutex!
- Recursive mutex? No.
- g_main_context_acquire – still susceptible to ABBA deadlock
- Defer registration via bottom-halves

```c
set_fd_handler(fd, ...):
lock(iohandler_list)
iohandler_list.modify(fd1, ...)
unlock(iohandler_list)
```

```c
iohandler_dispatch:
lock(iohandler)
For iohandler in iohandler_list:
dispatch(iohandler)
→ set_fd_handler(fd, ...)
unlock(iohandler)
```
Thread-safe Event Registration/Modification

- Just use a simple mutex!
- **Recursive mutex? No.**
- `g_main_context_acquire` – still susceptible to ABBA deadlock
- Defer registration via bottom-halves
Thread-safe Event Registration/Modification

- Just use a simple mutex!
- Recursive mutex? No.
- `g_main_context_acquire` – still susceptible to ABBA deadlock, but can drop all locks prior to avoid lock-order reversal. Ugly.
- Defer registration via bottom-halfs

```c
lock(tap_mutex)
set_fd_handler(ctx, fd, ...):
  gmc_acquire(ctx)
  iohandler_list.modify(fd1, ...)
  unlock(iohandler_list)
  gmc_release(ctx)
unlock(tap_mutex)
```

```c
iohandler_dispatch:
  gmc_acquire(ctx)
  For iohandler in iohandler_list:
    dispatch(iohandler)
  \→ lock(tap_mutex)
  unlock(iohandler)
  gmc_release(ctx)
```

Thread-safe Event Registration/Modification

- Just use a simple mutex!
- Recursive mutex? No.
- `g_main_context_acquire` – still susceptible to ABBA deadlock
- **Defer registration via bottom-halfs**
Questions