Multiqueue in the block layer

Making the block layer thread-safe

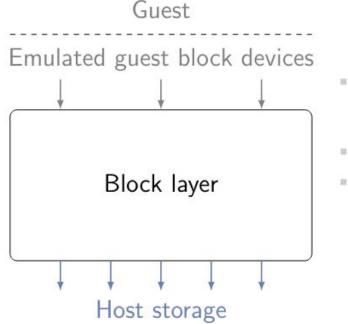
Emanuele Giuseppe Esposito Software Engineer eesposit@redhat.com IRC: esposem Kevin Wolf Principal Software Engineer kwolf@redhat.com IRC: kwolf



Introduction & our goal



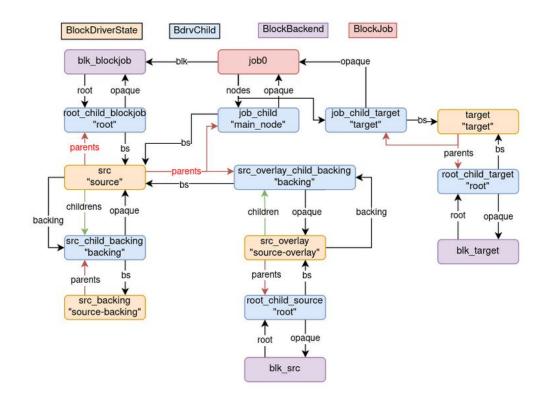
The QEMU Block Layer



- Read/write data from/to host storage (outside of QEMU)
- Interpret image formats
- Manipulate data on the way:
 - Encryption
 - Throttling
 - Duplication

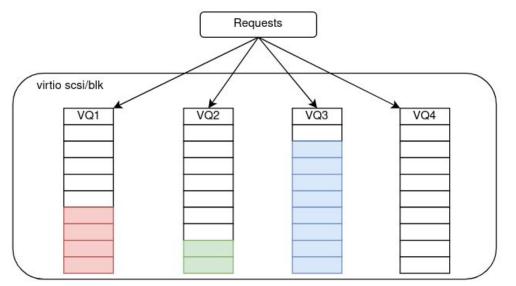


The QEMU Block Layer



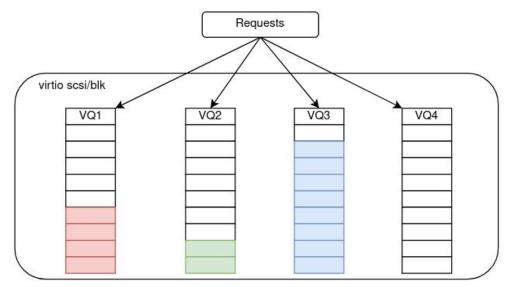


The guest perspective



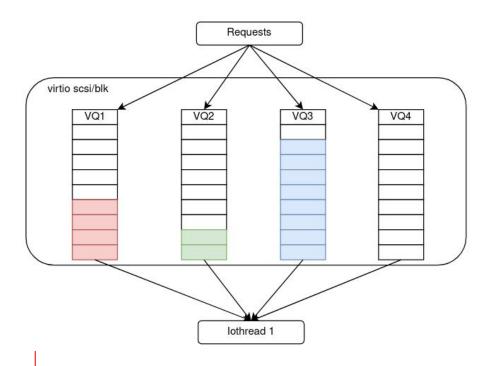


The guest perspective



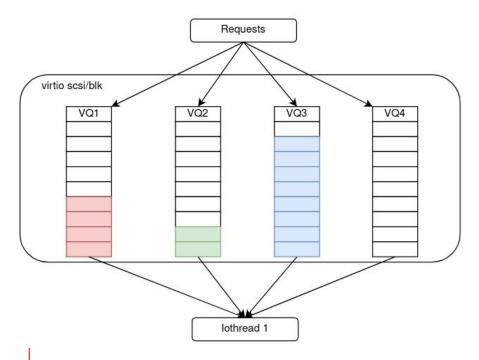


The reality



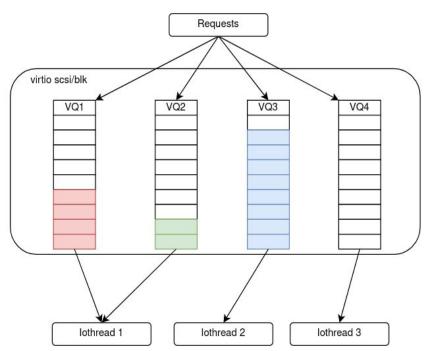


The reality





The goal





Problems to solve



11

AioContext lock

How the block layer achieves thread safety today

- At the very beginning, we had only the main loop and BQL
- Then, iothreads were introduced to improve performance
 - iothreads access the block graph, so synchronization is needed
- Introduce an AioContext per iothread with a lock
 - At first: protect only BlockDriverState
 - Today: Mega lock
- Concept works when each image/device is bound to a single iothread



The AioContext Lock is everywhere

- virtio-blk/scsi API: uses AioContext Lock
- BlockBackend API: uses AioContext Lock
- Block API: uses AioContext Lock
- BlockJob/Job API: uses AioContext Lock
- BlockDriverState callbacks: use AioContext Lock

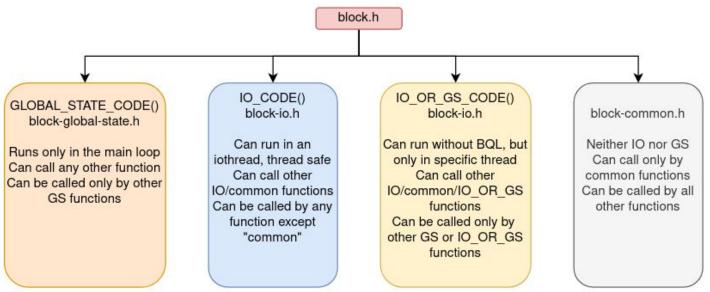
Not good for our goals: we need each component to be thread safe!



13

Step 1: divide APIs in global state and IO

- Verify through assertions
- Kevin's script to recursively checks callers rules



Step 2a: replace AioContext lock with another lock (easy)

- Job -> introduce job_mutex
 - Removing it also introduced AIO_WAIT_WHILE_UNLOCKED
- ThreadPool -> custom lock
- Virtio-blk and scsi (not-so-easy)
- Other minor APIs



Step 2b: replace AioContext lock in graph modifications (HARD)

- A **lot** of BlockDriverState callbacks are recursive and read the graph
- Concurrent modification would cause the reader to read an invalid graph
- Luckily: only very few places where the graph is modified



Possible solutions



Option 1: drain

Politely ask others to leave the graph alone

Use drains, more specifically subtree_drained_{begin/end}

Pros

- Blocks the whole graph
 - No new I/O are submitted and waits that all the running ones are done
 - No graph modification happening after drained_begin()
- It is already present in the block layer code



Option 1: drain

Politely ask others to leave the graph alone

Cons

- It is already by itself recursive, how to stop a concurrent drain?
- Not designed to do this
- Not the right guarantees. Something else can still send IO requests if it doesn't check that a drain is happening
- Covers only I/O, not other operations using the graph



Option 1: drain

Politely ask others to leave the graph alone

Cons

- It is already by itself recursive, how to stop a concurrent drain?
- Not designed to do this
- Not the right guarantees. Something else can still send IO requests if it doesn't
 - check that a drain is happening
- Covers only I/O, not other operations using the graph

Conclusion

- Kevin dropped subtree_drained_* API from QEMU
- block layer



Option 2: rwlock

A simple global read-write lock

- We have many readers, few writers
- Reader is a coroutine that will block when a write is in progress
- New writer waits for readers, doesn't allow new readers to enter the section

Pros

- Locks are explicit and mandatory, not like drains that require cooperation (if not called or not implemented, there is no enforcing)
- Locks are added locally in the Block API, no need to add/implement them in the caller (monitor, devices, etc)



Option 2: rwlock

...not so simple to add

Cons

- Requires to convert all BlockDriverState callbacks to coroutines
- Stricter locking rules
 - Must (at least for now) co-exist with AioContext lock. So it must be even more carefully placed.
 - More info later



Making sure it's correct



The graph lock

Protecting the block graph structure

Chose option 2: An RwLock-like lock for the parent-child links

- rdlock() when following an edge in the graph (almost everywhere)
 - This is a coroutine_fn, callers need to run in coroutines
- wrlock() when modifying an edge in the graph (main loop)
 - Must not be called from iothreads or coroutine contexts
 - Must not hold AioContext locks while polling
- Calling aio_poll() is forbidden while holding the lock (deadlocks)
- Minor problem: Who on earth is supposed to be able to review that the correct lock is taken and the conditions are met?!



clang TSA

Thread safety analysis

With clang, we can turn many locking inconsistencies into compiler errors

- Macros to annotate functions and structs with locking requirements
- -Wthread-safety performs checks at compile time
 - Enabled by default in QEMU if available
- Be aware of some serious limitations with C:
 - Only global locks for struct fields, can't refer to struct itself
 - Function pointers go unchecked
 - TSA doesn't understand g_auto (\rightarrow mutex guards)



clang TSA in QEMU

What it looks like in QEMU code

Annotate functions with GRAPH_RDLOCK or GRAPH_WRLOCK

static void GRAPH_WRLOCK bdrv_child_cb_attach(BdrvChild *child);

Annotate (function) pointers with GRAPH_RDLOCK/WRLOCK_PTR

void GRAPH_WRLOCK_PTR (*attach)(BdrvChild *child); QLIST_HEAD(, BdrvChild GRAPH_RDLOCK_PTR) parents;

...which both expand to TSA macros from qemu/clang-tsa.h

#define GRAPH_WRLOCK TSA_REQUIRES(graph_lock)
#define GRAPH_RDLOCK_PTR TSA_GUARDED_BY(graph_lock)

Users outside the block layer will use these directly



Vrc

Analysing the call graph statically

- Tool to find paths in the call graph
- Functions can be matched by name, regex, labels
 - Labels defined in the tool or in source code
 - coroutine_fn, coroutine_mixed_fn, no_coroutine_fn are defined in source code
- This can be used to find bugs, for example:
 - Show non-coroutine_fns calling a coroutine_fn
 - Show all paths from a coroutine_fn to AIO_WAIT_WHILE()

https://github.com/bonzini/vrc



Implementing graph locking



The plan is simple enough

...as long as we ignore the details

For each function that reads the children or parents list of a node:

- Take the graph reader lock locally around the access
- Push the lock to the callers: Mark the function as GRAPH_RDLOCK
 - If all callers are coroutine_fn, take the lock there, done
 - If all callers run in the main thread, pretend to take the lock there, done
 - Otherwise, move it to a coroutine and add a generated co_wrapper Take the graph reader lock when entering the coroutine

Finally declare bs->children and bs->parents to be TSA_GUARDED_BY(graph_lock) Remove the AioContext lock



Coroutine related problems

It changed more than we thought

Moving functions to coroutines + co_wrapper made some assumptions invalid:

- co_wrappers spawn a coroutine and wait for it in a nested event loop
- AIO_WAIT_WHILE() drops the AioContext lock temporarily
 - Surprise: Some callers never locked it
 - Before the change, they got away with it, now it crashes
- coroutine_fn correctness has become more important
 - Changing the graph requires the writer lock now
 - This can't run in coroutines

Not hard to fix, but changes in many places



AioContext locking

What did you think would happen if nobody checks its correctness?

Taking the write lock runs a nested event loop to wait for readers

- Can't hold any AioContext locks
 - Readers in other threads need the lock to make progress
- Many function didn't even specify which lock (if any) a caller should hold
 - Different callers do different things
- Other functions do, but callers don't care

The goal is to remove the AioContext lock, but we actually need to fix it first







Next steps

Remaining work for basic multiqueue support

- Block drivers are ready
- Let TSA check that children/parent links are protected by the graph lock
 - Rebase existing patches and fix new test case failures
 - A few more places only assert that the lock is held instead of having it checked
- Check locking for remaining BlockDriverState fields
- Change devices to send requests from multiple threads
 - Includes qdev work to configure the queue/iothread mapping
 - Rely on graph locking instead of AioContext lock
 - Preliminary patches exist



Some numbers



But is it really worth it?

Running a benchmark to estimate the performance gains

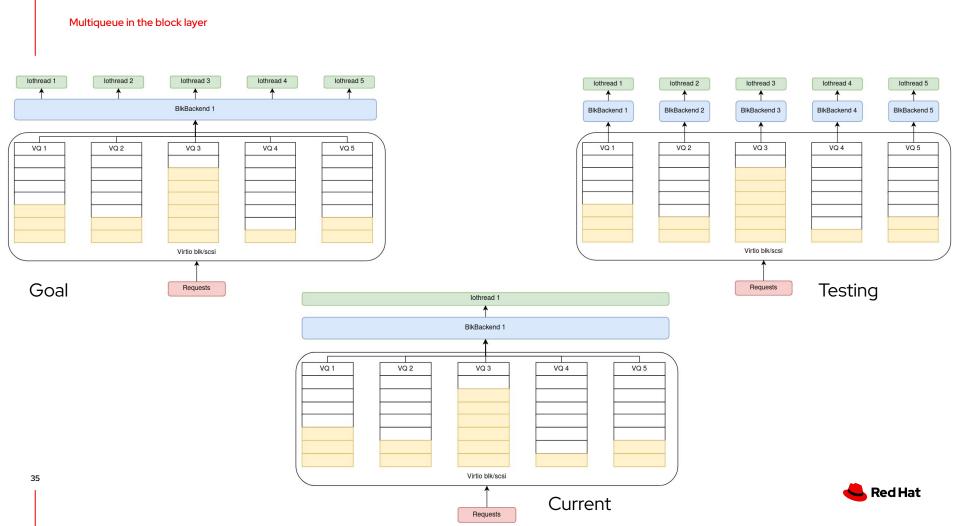
- Simulate the multithreaded environment before it's implemented:
 - Stefan Hajnoczi hacked the virtio-blk device to create an IOThread and BlockBackend for each vq
 - Open the image twice instead of solving thread safety, okay for raw images
- 8 CPUs SMP guests that submit a lot of I/O (no CPU pinning)
- fio configuration:

rw=randread numjobs=8 iodepth=16 cpus_allowed=0-7 cpus_allowed_policy=split

QEMU command-line:

qemu-system-x86_64 --name qemu-test,debug-threads=on -M accel=kvm -m 1G -smp 8 -cpu host --blockdev file,filename=test.img,locking=off,node-name=drive0,aio=native,cache.direct=on --device virtio-blk-pci,drive=drive0





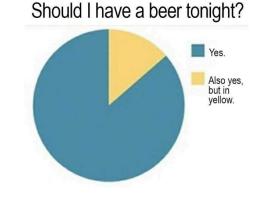
But is it really worth it?

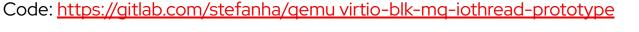
Benchmark results

Results:

- No IOThread 141k
- 1 IOThread 156k
- Per-vq IOThreads 280k

Very promising results :)







Questions?

