ublk
virtual block devices in user space

DevConf.CZ 2023

Stefano Garzarella
<sgarzare@redhat.com>

German Maglione
<gmaglione@redhat.com>

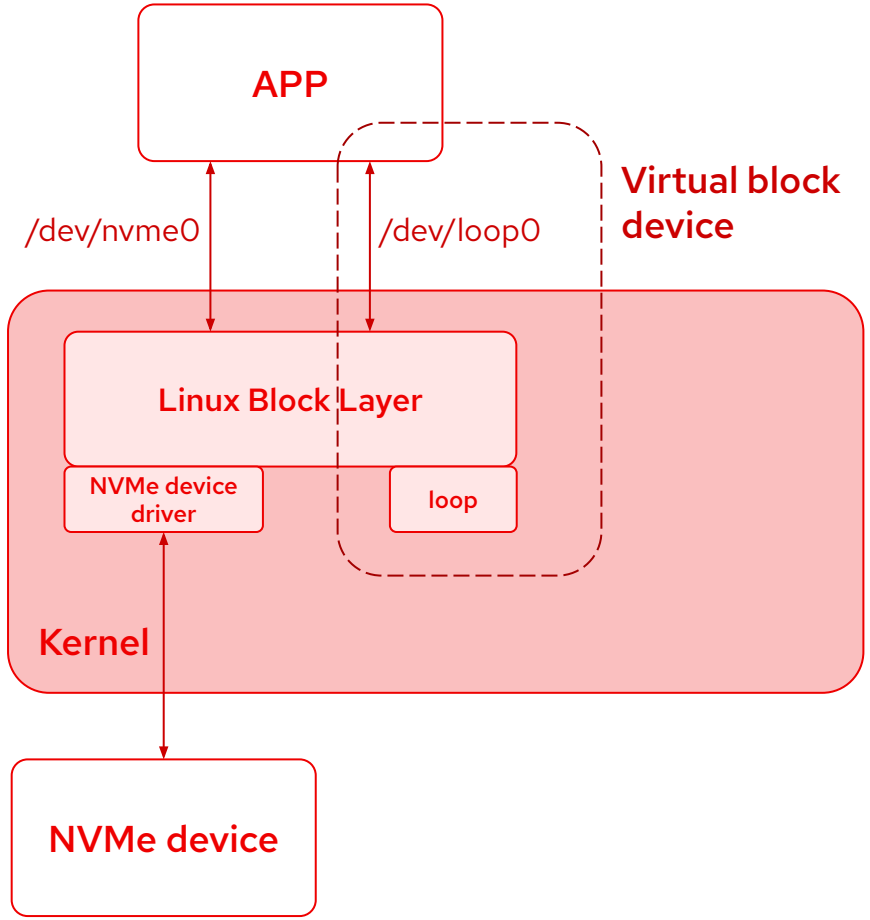
Red Hat

Agenda

- Virtual Block Devices
- ublk overview
- io_uring overview
  - io_uring passthrough command
- ublk
  - control and data paths
  - user space libraries
  - use cases
- QCOW2
- QEMU Storage Daemon (QSD)
  - ublk export
- Rust Storage Daemon

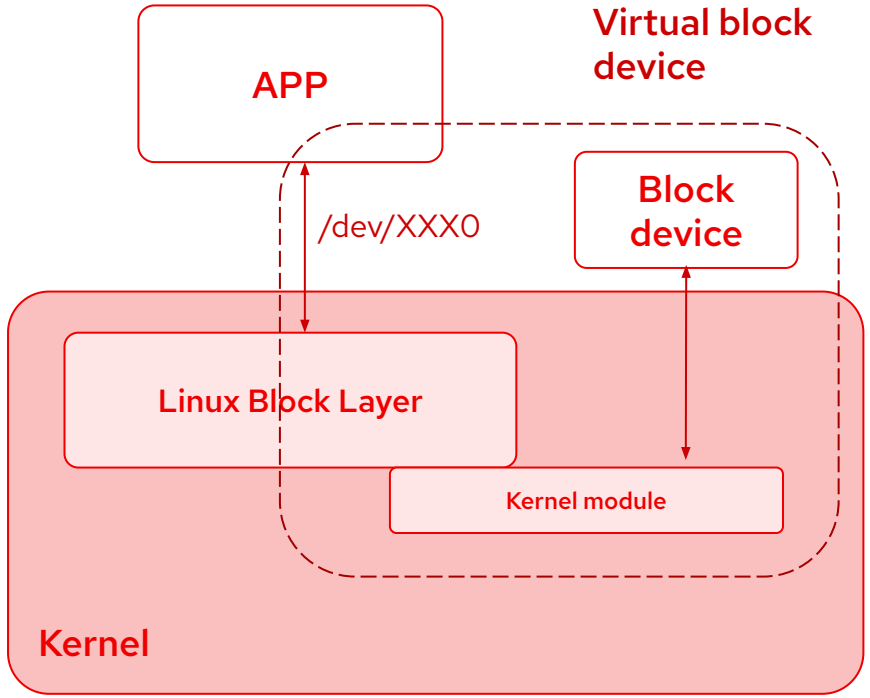
Virtual Block Devices

- "Virtual"
  - **emulate a block device in software**
- **Why ?**
  - Virtual Machines
  - Disk image format (e.g. QCOW2) features
    - growing images, snapshots, backing files, compression
  - Isolation (e.g. containers, VMs, etc.)
  - Disk image managing
    - editing, manipulation
  - Network storage (e.g. NBD, iSCSI, Ceph RBD, etc.)
- Linux provides several **in-kernel virtual block devices**
  - e.g. loop, NBD, RBD, etc.

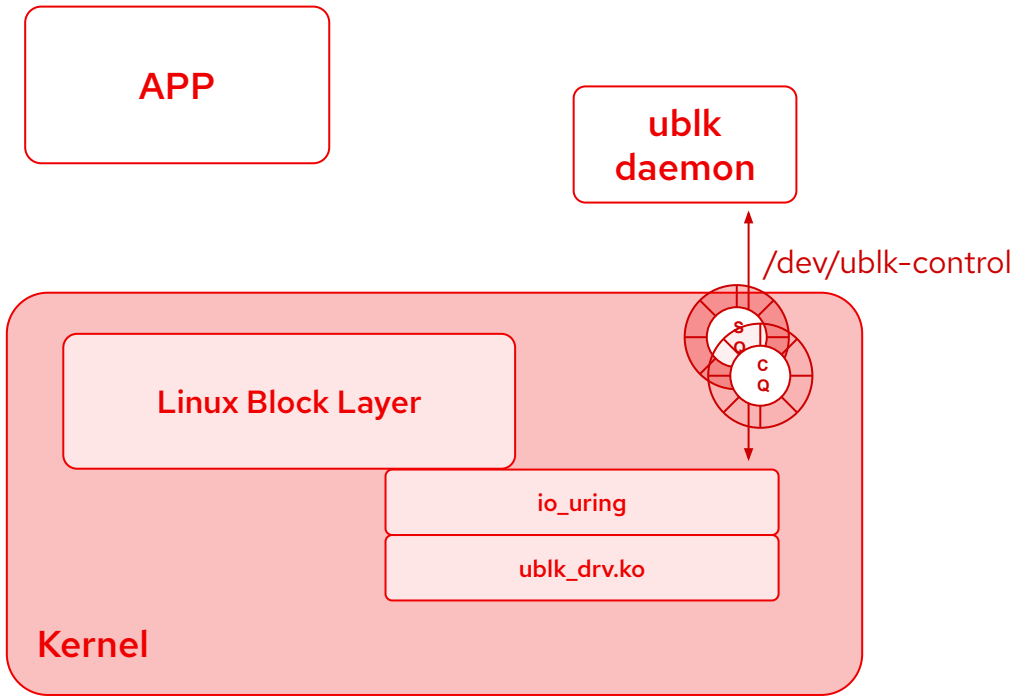


Virtual Block Devices in user space

- Small piece in the kernel, just to handle the users pace interface
  - **Request handling in user space**
- **Why in user space?**
  - Safety
  - Isolation
  - Maintainability
  - Portability
  - Testing and debugging
- Tradeoff
  - Performance



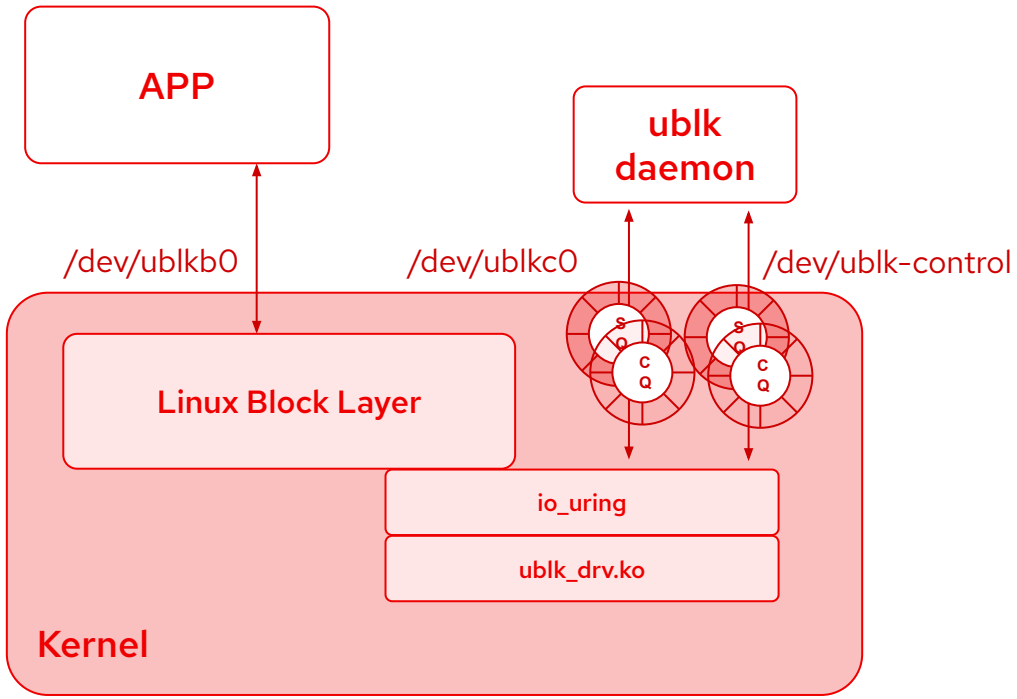
ublk: user space block device/driver framework



- ublk
  - io_uring to overcome the performance limitation of block devices in user space
  - introduced by Ming Lei in Linux v6.0
    - https://lwn.net/Articles/903855/
- **ublk_drv.ko** kernel module
  - forward block requests coming from the Linux block layer to the ublk daemon
  - **io_uring passthrough command** (similar to IOCTL)
  - **/dev/ublk-control**
    - used to setup a device

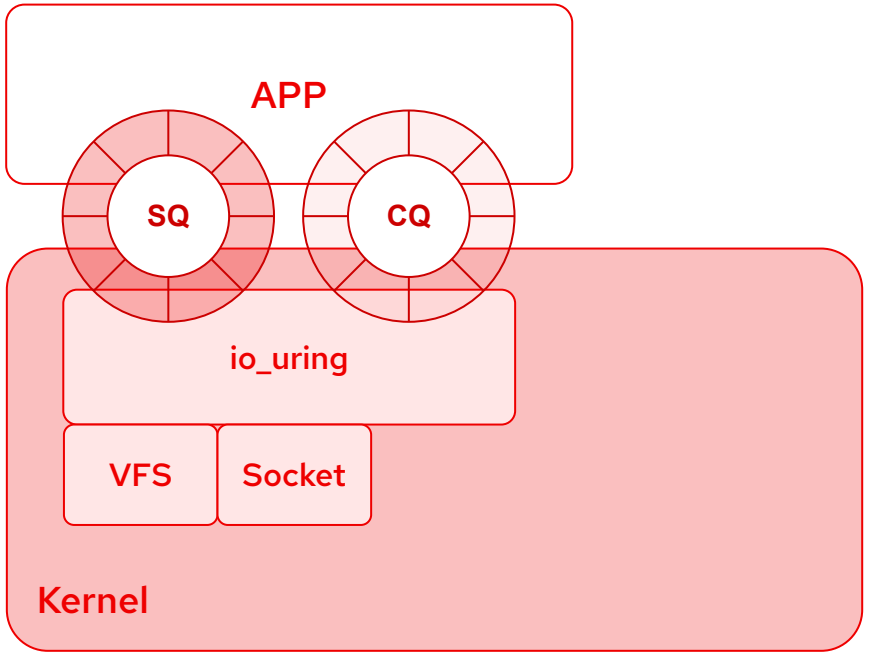
Red Hat

ublk: user space block device/driver framework



- **/dev/ublkbN**
  - regular Linux block device
- **/dev/ublkcN**
  - ublk character device used as kernel/daemon main interface
    - shared memory
      - I/O requests info (sectors, offset, etc.)
    - io_uring queues
      - io_uring passthrough command (similar to IOCTL)
      - kernel notifies of new requests to be handled
      - ublk daemon commit requests handled

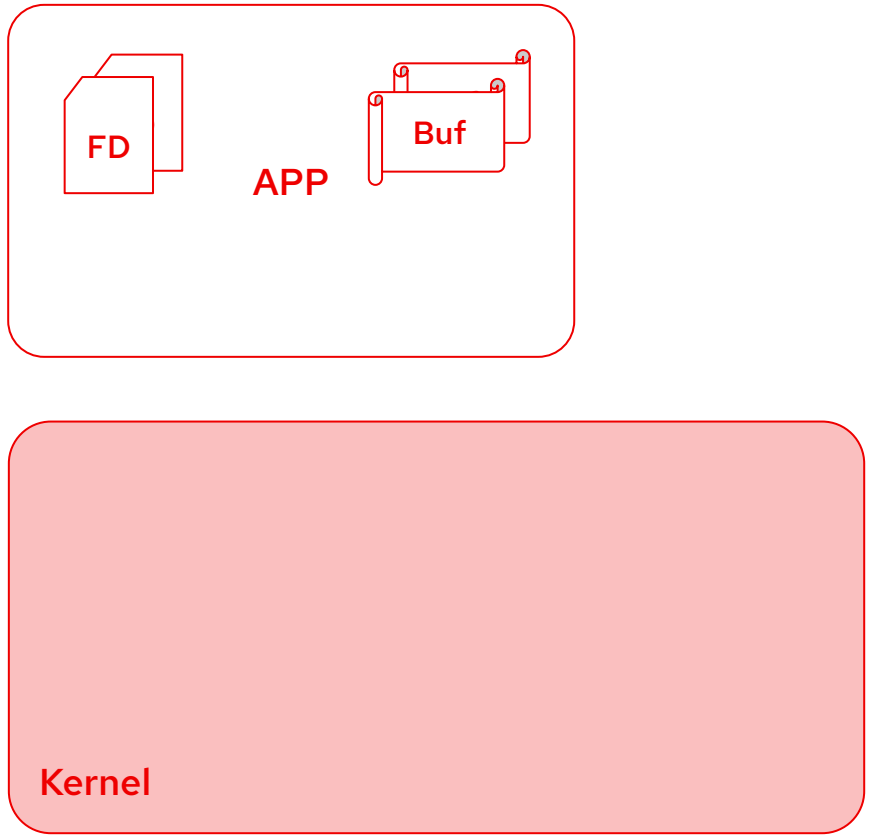
io_uring overview



KVM Forum 2020: Speeding Up VM's I/O Sharing Host's io_uring Queues With Guests
Recording: https://www.youtube.com/watch?v=aPyDk7I5_8Y

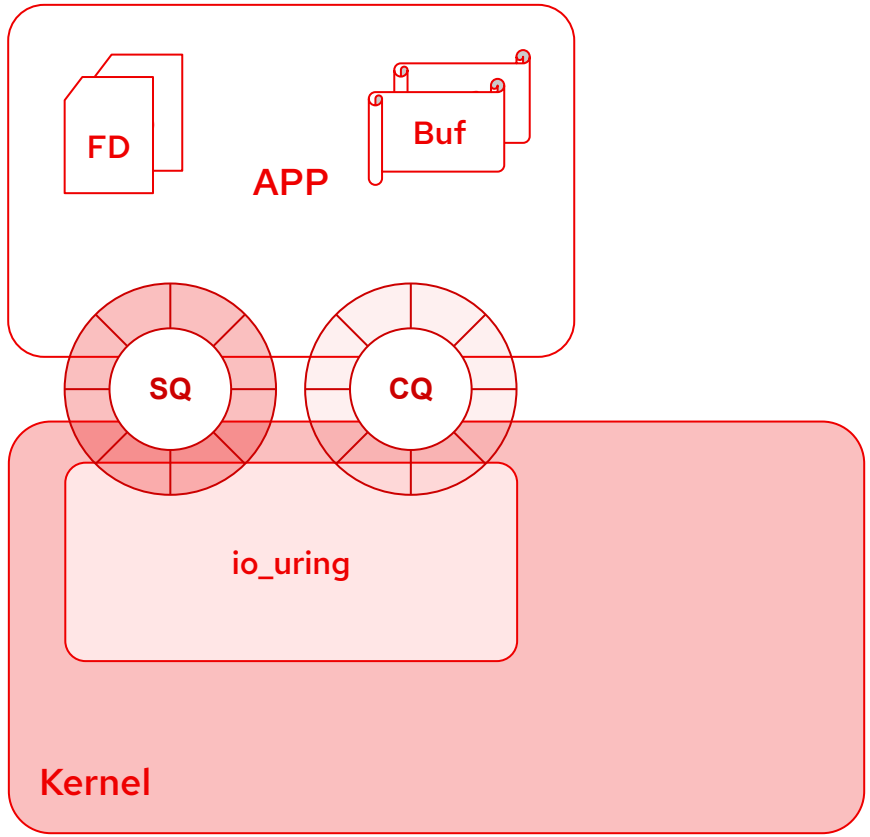
- A **Linux** interface for **asynchronous I/O**
  - Introduced by Jens Axboe in Linux v5.1
  - Not only for block oriented I/O
- A **pair of rings** shared between kernel and application
  - Submission Queue (**SQ**)
    - Producer: application, Consumer: kernel
  - Completion Queue (**CQ**)
    - Producer: kernel, Consumer: application
- Three system calls
  - **`io_uring_setup(2)`**
  - **`io_uring_register(2)`**
  - **`io_uring_enter(2)`**

io_uring: system calls



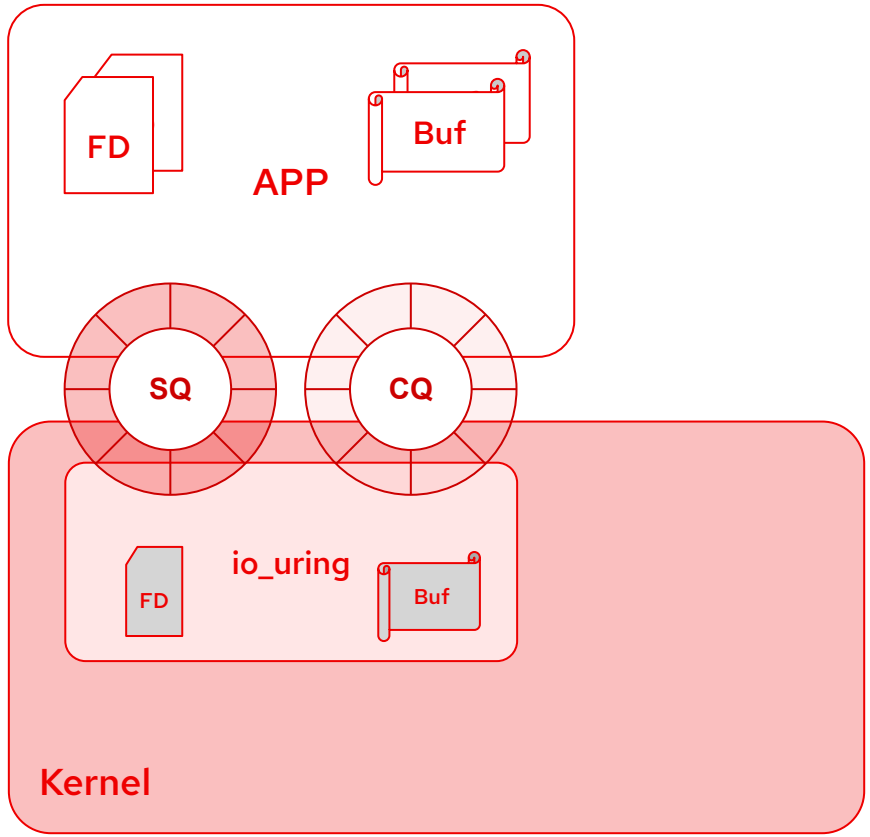
io_uring: system calls

- **`io_uring_setup(2)`**
  - setup a context for performing asynchronous I/O



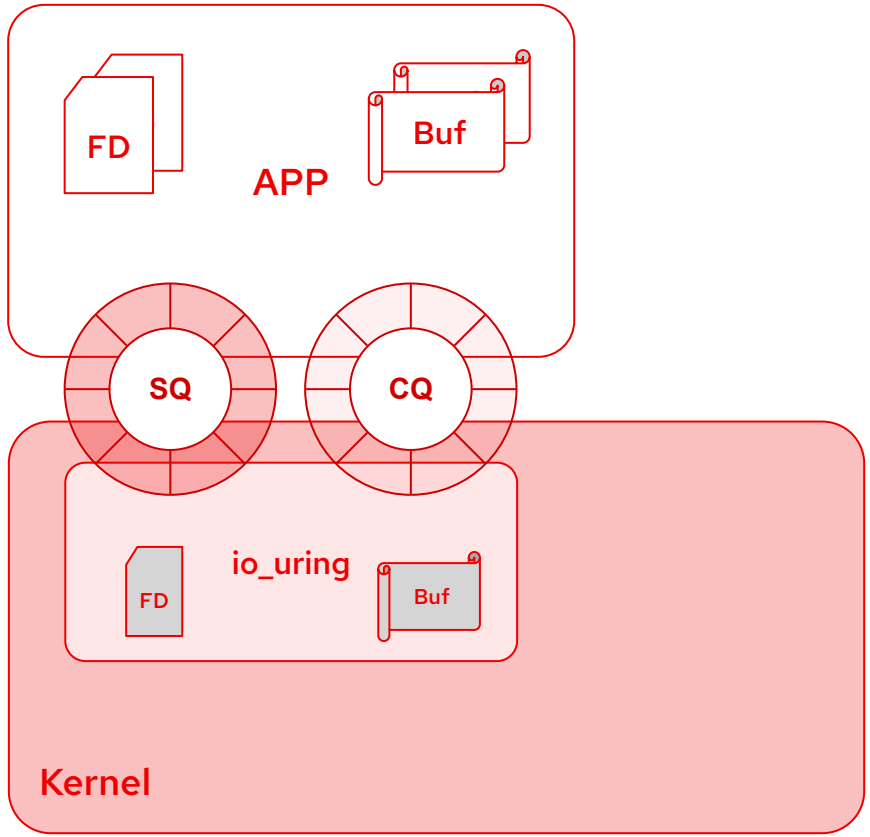
io_uring: system calls

- **`io_uring_setup(2)`**
  - setup a context for performing asynchronous I/O
- **`io_uring_register(2)`**
  - registers resources (e.g. user buffers, files, eventfd, personality, restrictions) in the context



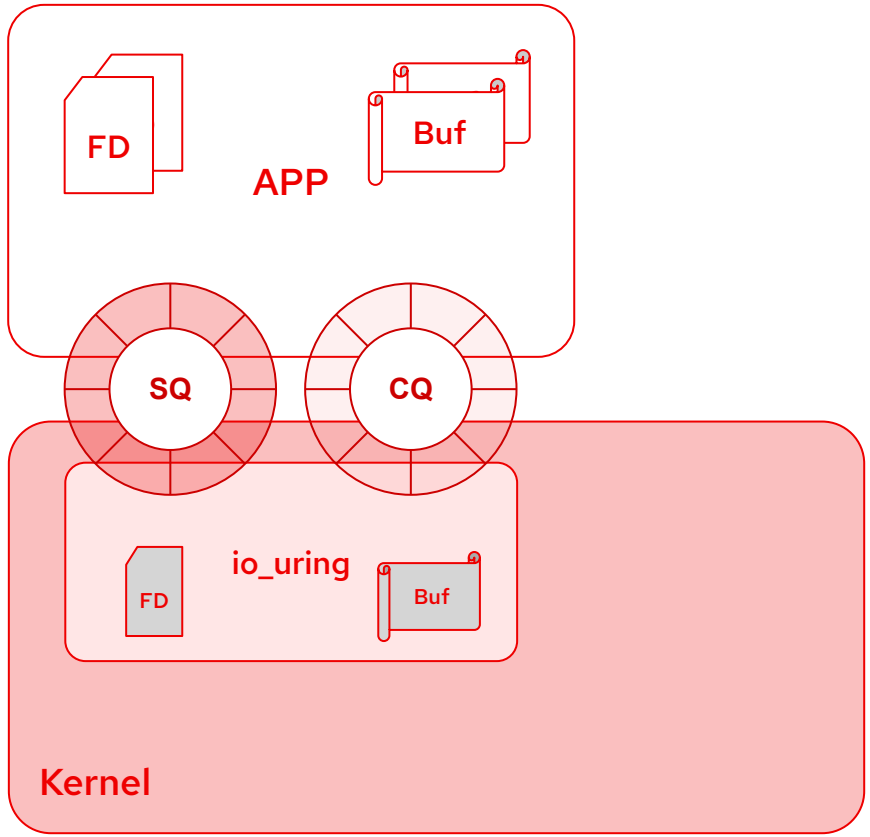
io_uring: system calls

- **`io_uring_setup(2)`**
  - setup a context for performing asynchronous I/O
- **`io_uring_register(2)`**
  - registers resources (e.g. user buffers, files, eventfd, personality, restrictions) in the context
- **`io_uring_enter(2)`**
  - initiate and/or complete asynchronous I/O
  - single system call
    - submit new operations to do
    - reap operations result



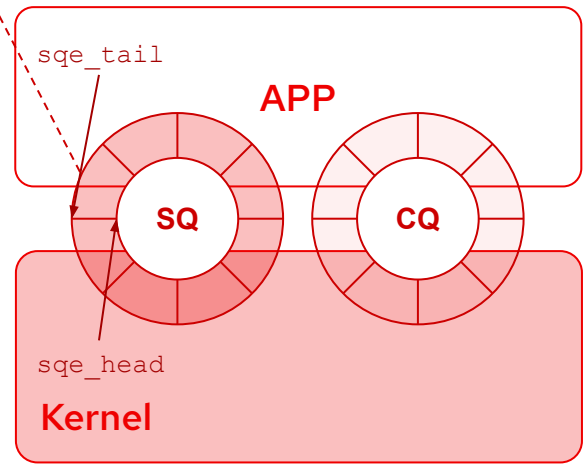
io_uring: system calls

- **`io_uring_setup(2)`**
  - setup a context for performing asynchronous I/O
- **`io_uring_register(2)`**
  - registers resources (e.g. user buffers, files, eventfd, personality, restrictions) in the context
- **`io_uring_enter(2)`**
  - initiate and/or complete asynchronous I/O
  - single system call
    - submit new operations to do
    - reap operations result
- liburing
  - https://github.com/axboe/liburing
  - provide a convenient C API



io_uring: submission and completion queues

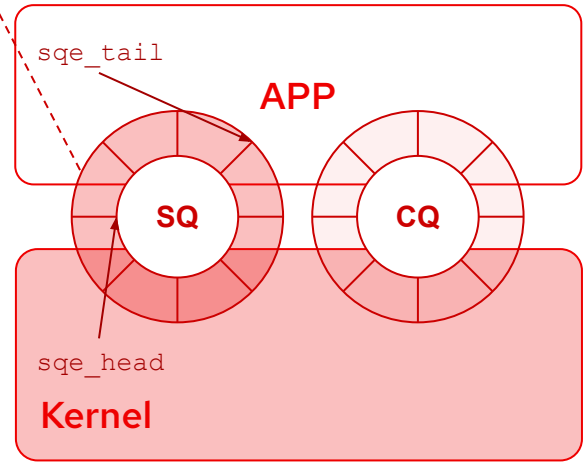
SQE
opcode
flags
fd
addr
off
...
user_data



- **Submission Queue (SQ)**
  - Application
    - produces SQEs (SQ Entry)
      - operation to do (opcode)

io_uring: submission and completion queues

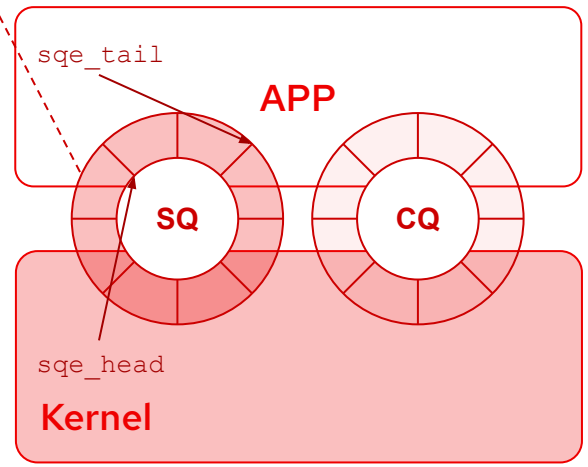
SQE
opcode
flags
fd
addr
off
...
user_data



- **Submission Queue (SQ)**
  - Application
    - produces SQEs (SQ Entry)
      - operation to do (opcode)
    - updates `sqe_tail`
    - invokes `io_uring_enter(2)`

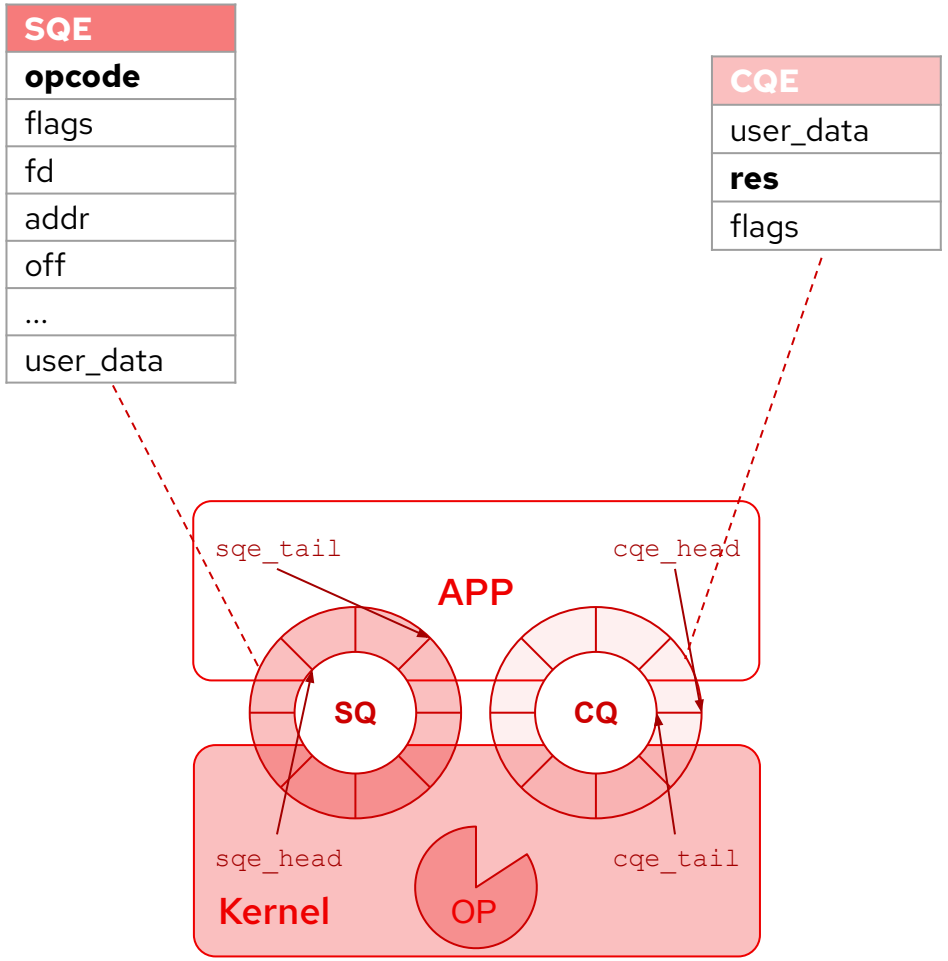
io_uring: submission and completion queues

SQE
opcode
flags
fd
addr
off
...
user_data



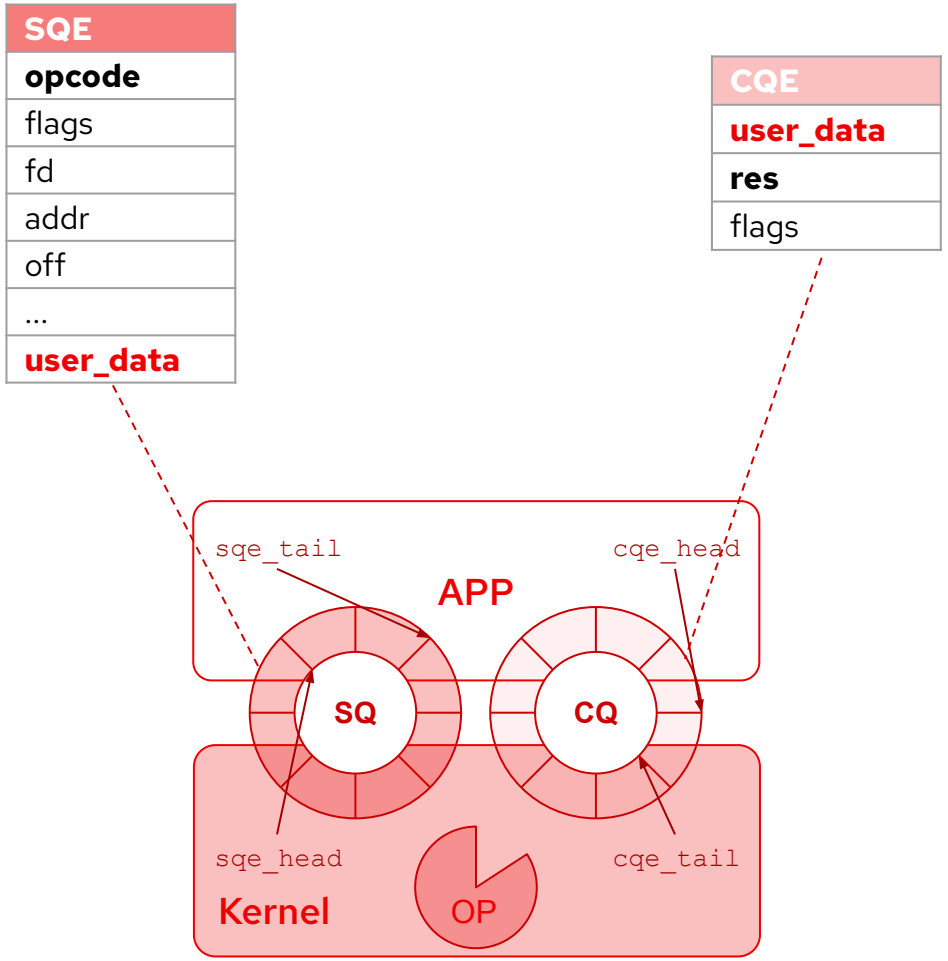
- **Submission Queue (SQ)**
  - Application
    - produces SQEs (SQ Entry)
      - operation to do (opcode)
    - updates `sqe_tail`
    - invokes `io_uring_enter(2)`
  - Kernel
    - consumes SQEs
    - updates `sqe_head`

io_uring: submission and completion queues



- **Submission Queue (SQ)**
  - Application
    - produces SQEs (SQ Entry)
      - operation to do (opcode)
    - updates `sqe_tail`
    - invokes `io_uring_enter(2)`
  - Kernel
    - consumes SQEs
    - updates `sqe_head`
- Kernel processes the operation

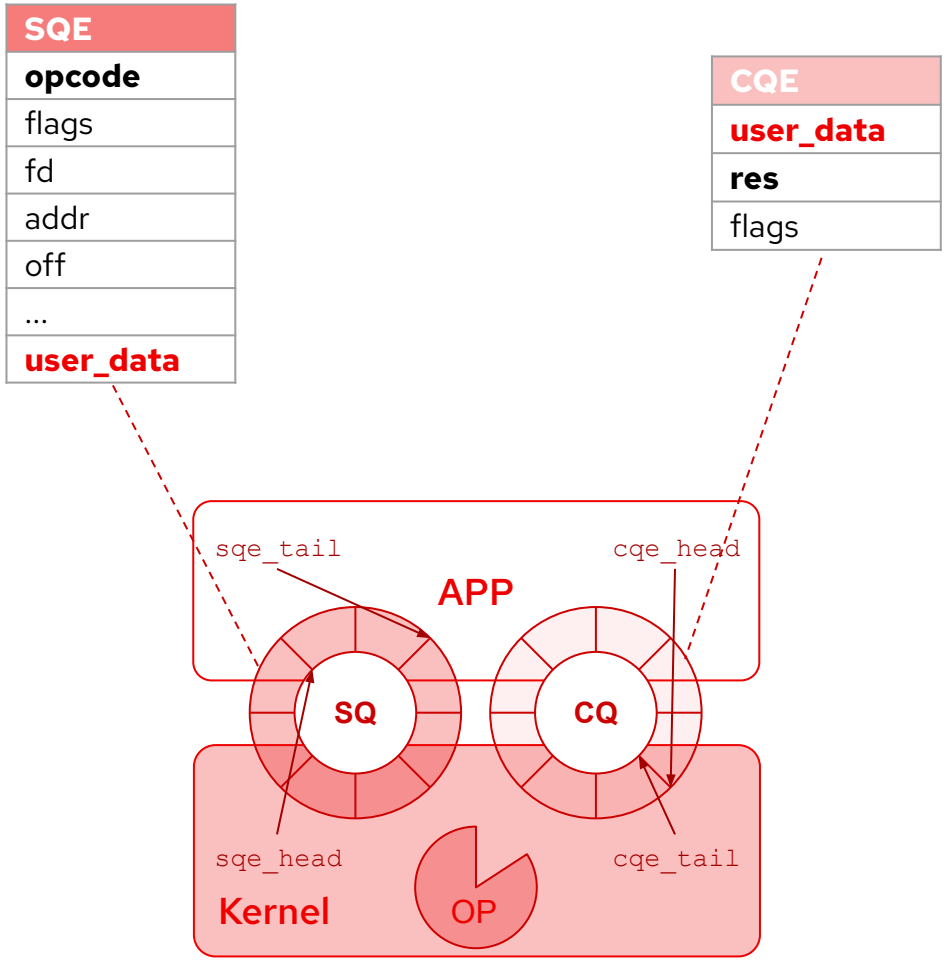
io_uring: submission and completion queues



- **Submission Queue (SQ)**
  - Application
    - produces SQEs (SQ Entry)
      - operation to do (opcode)
    - updates `sqe_tail`
    - invokes `io_uring_enter(2)`
  - Kernel
    - consumes SQEs
    - updates `sqe_head`
- Kernel processes the operation
- **Completion Queue (CQ)**
  - Kernel
    - produces CQEs (CQ entry)
      - operation result (res) and user_data
    - updates `cqe_tail`

Red Hat

io_uring: submission and completion queues

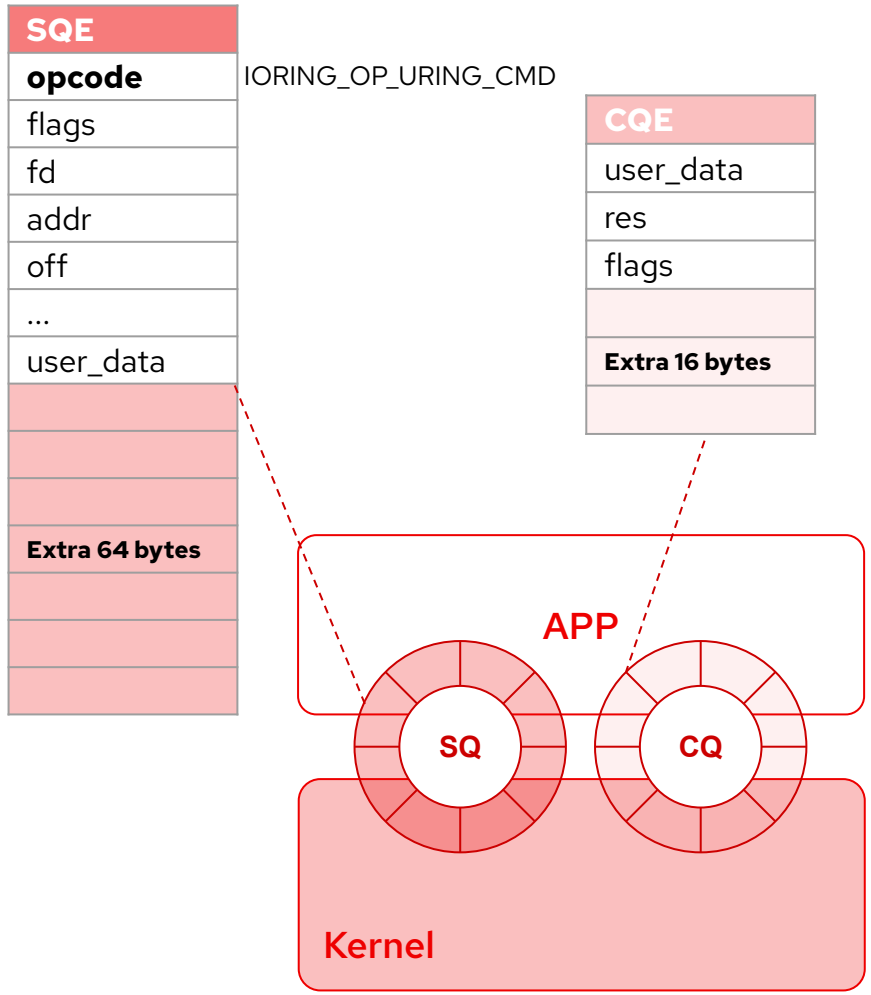


- **Submission Queue (SQ)**
  - Application
    - produces SQEs (SQ Entry)
      - operation to do (opcode)
    - updates `sqe_tail`
    - invokes `io_uring_enter(2)`
  - Kernel
    - consumes SQEs
    - updates `sqe_head`
- Kernel processes the operation
- **Completion Queue (CQ)**
  - Kernel
    - produces CQEs (CQ entry)
      - operation result (res) and user_data
    - updates `cqe_tail`
  - Application
    - consumes CQEs
    - updates `cqe_head`

Red Hat

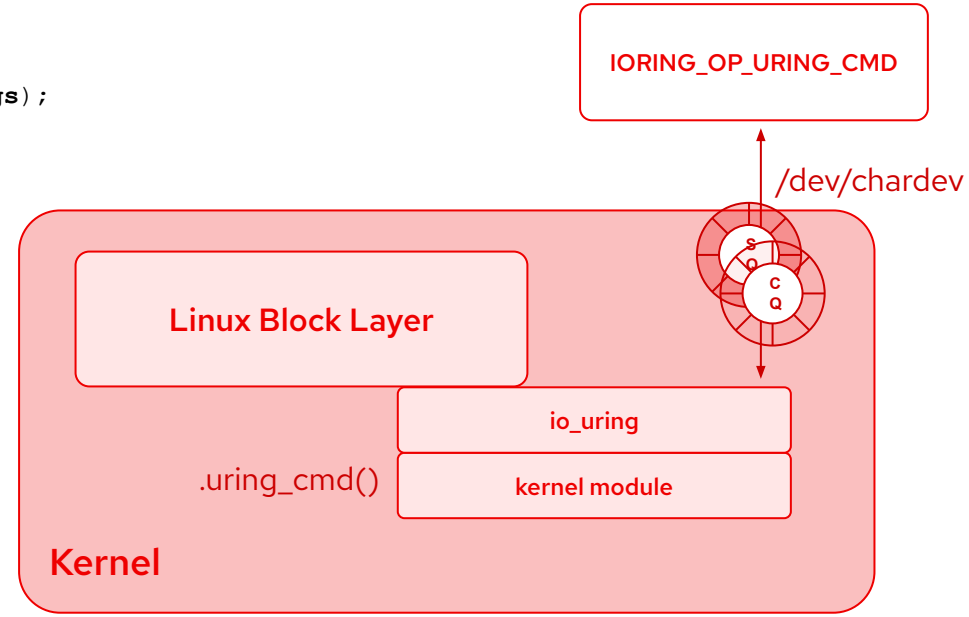
io_uring: passthrough command

- Supported since Linux v5.19
  - https://lore.kernel.org/io-uring/6f712c75-c849-ae89-d763-b2a18da52844@kernel.dk/
- New **SQE opcode**:
  - IORING_OP_URING_CMD
- New **io_uring_setup(2) flags**:
  - IORING_SETUP_SQE128
    - double the SQE size (64 -> 128 bytes)
  - IORING_SETUP_CQE32
    - double the CQE size (16 -> 32 bytes)



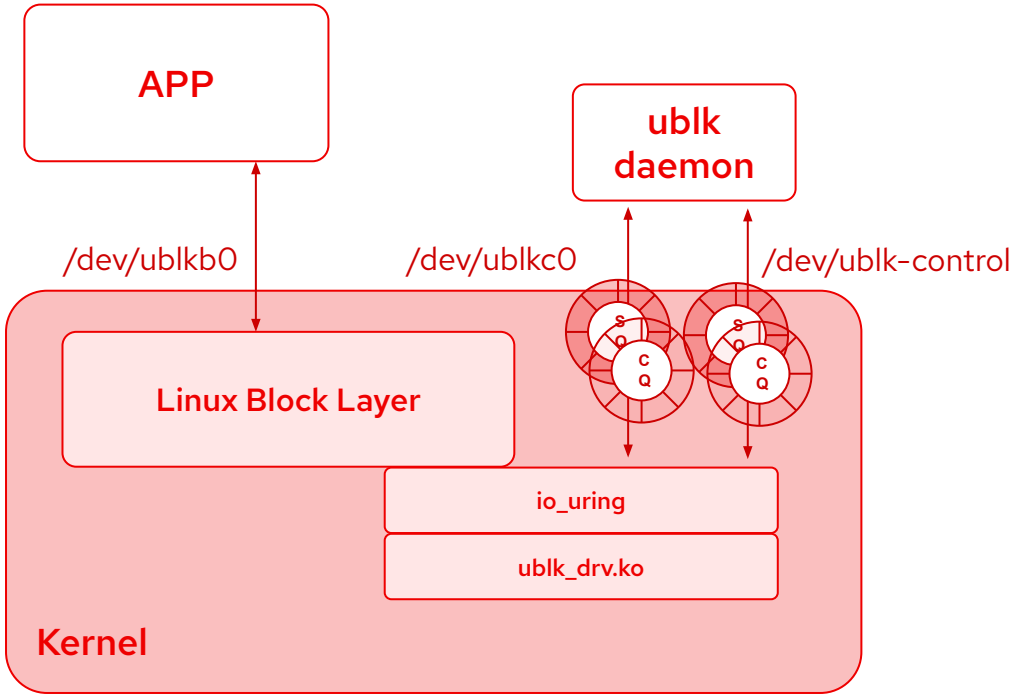
io_uring: passthrough command

- **Async command for special files** (e.g. chardev) exposed by drivers, filesystems, etc.
  - an async alternative to IOCTLs
  - new callback in `struct file_operations`
    - `int (*uring_cmd)(struct io_uring_cmd *ioucmd, unsigned int issue_flags);`
  - currently (Linux v6.4) used by
    - NVMe command passthrough
    - ublk
- Submission:
  - **user space** application sends **arbitrary command** to kernel code
  - IORING_SETUP_SQE128 allows 80 bytes for the command data
- Completion:
  - **kernel** code responds with the **result** asynchronously
  - IORING_SETUP_CQE32 allows 16 extra bytes for the return value



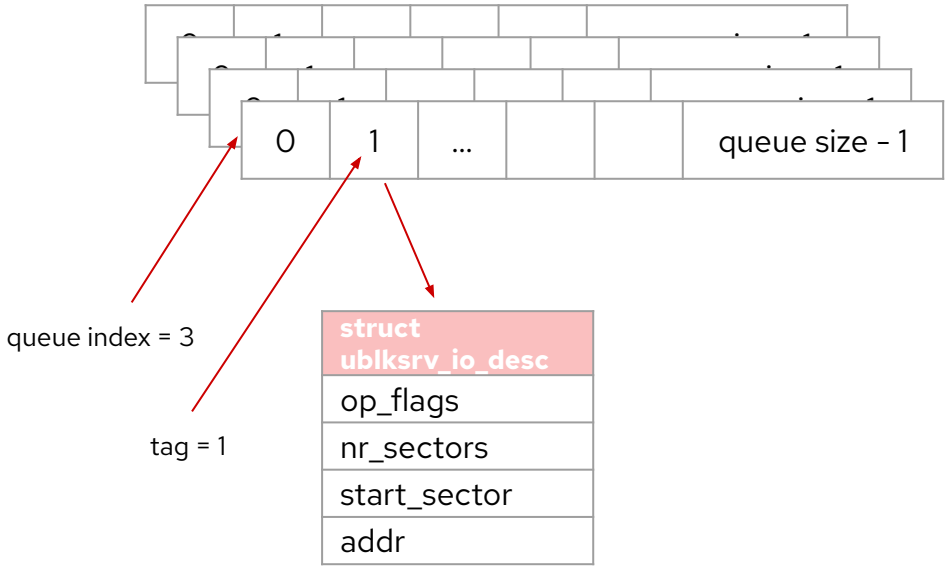
ublk: control path

- Create a new device using `/dev/ublk-control` chardev
  - **UBLK_CMD_ADD_DEV**
    - device id (`/dev/ublkb`**N**), number of queues, queue size, features (e.g. support zero-copy, recovery, etc.)
    - After this command `/dev/ublkcN` will be created
  - **UBLK_CMD_SET_PARAMS**
    - sectors, block size, attributes (FUA, read-only, etc.), discard and write-zeroes parameters, etc.
  - **UBLK_CMD_START_DEV**
    - After this command, `/dev/ublkbN` will be created
  - Other commands available
    - UBLK_CMD_STOP_DEV
    - UBLK_CMD_DEL_DEV
    - UBLK_CMD_GET_QUEUE_AFFINITY
    - etc.



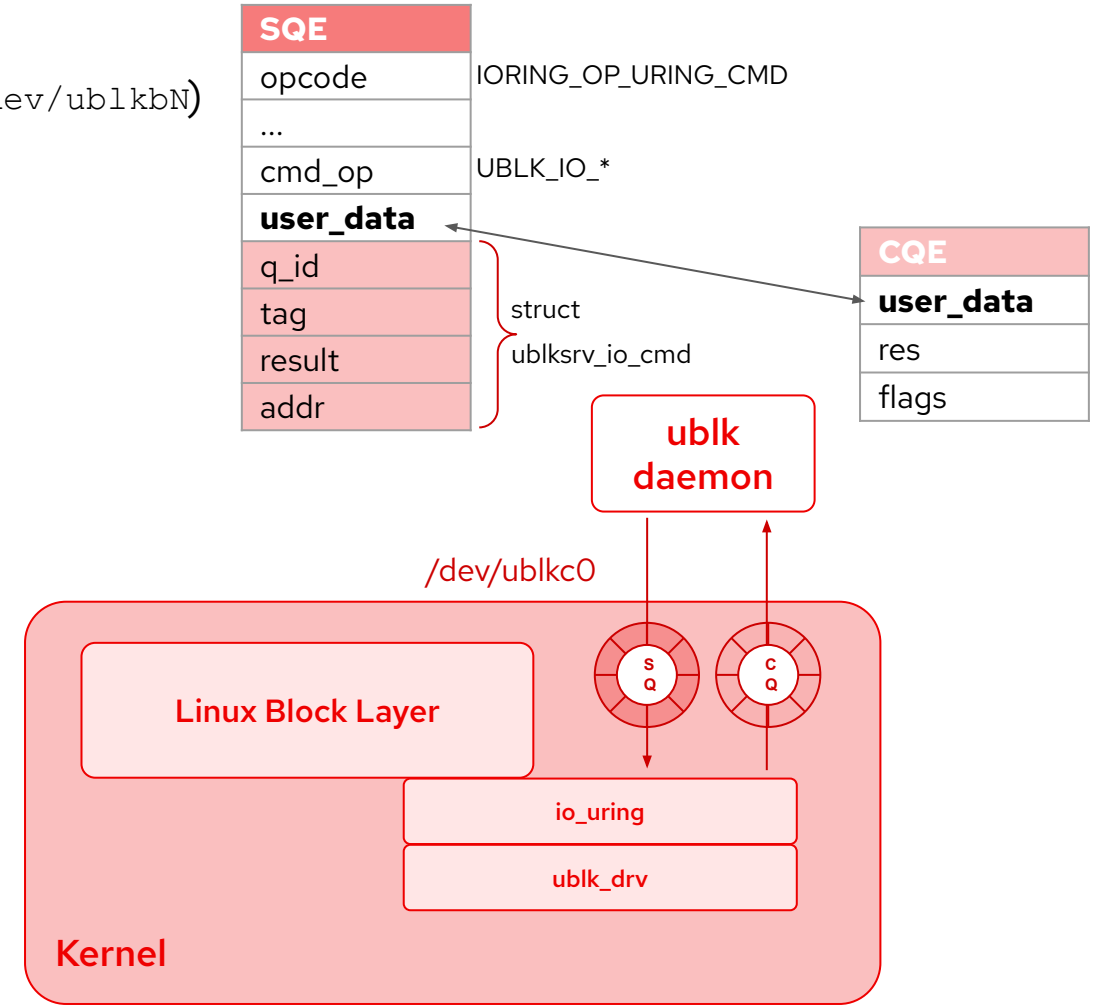
ublk: control path

- Memory map **shared memory** using `/dev/ublkcN` chardev
  - contains a structure for each element in each queue
    - queue index
    - tag (slot index in the queue)
  - written by the kernel before notify the daemon of a new request to handle
  - **I/O request info**
    - op_flags: operation (READ, WRITE, DISCARD, etc.) and flags
    - nr_sectors: request length
    - start_sector: request offset
    - addr: I/O buffer set by the daemon during “fetch” operation



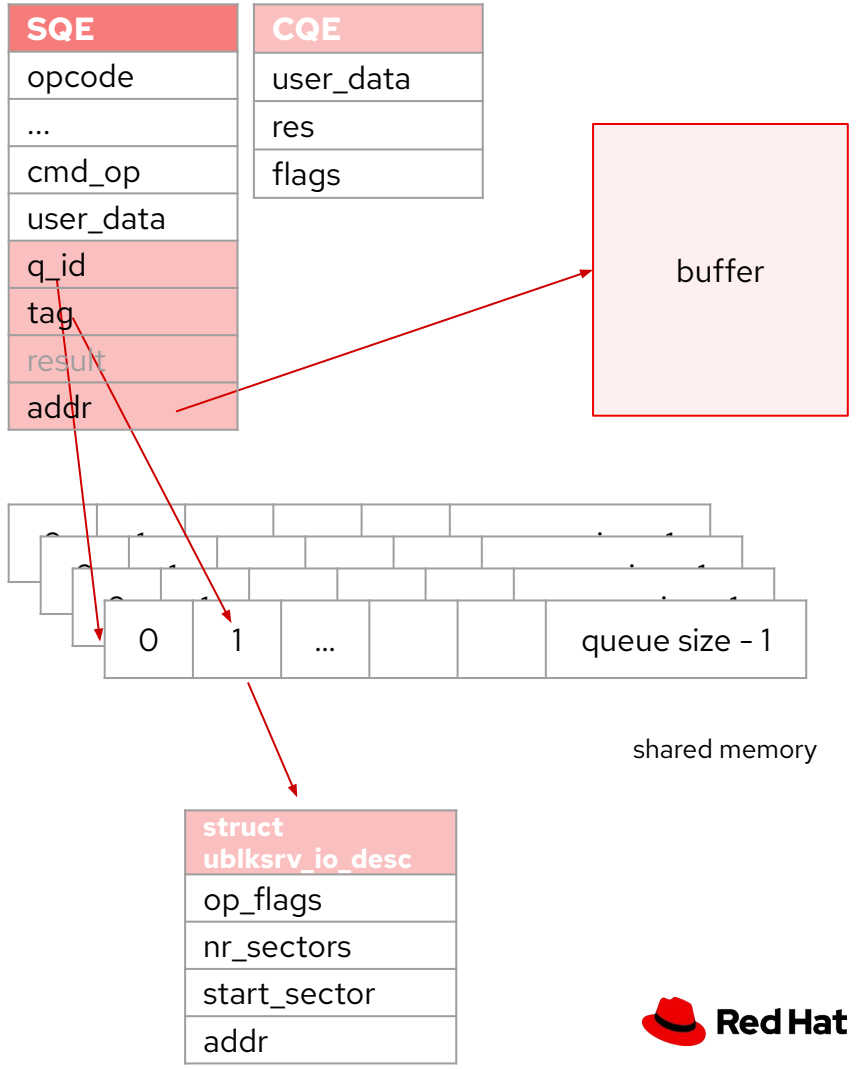
ublk: data path

- **Fetch and commit requests** coming from the block device (`/dev/ublkbN`)
  - io_uring passthrough commands on `/dev/ublkcN` chardev
  - SQE
    - submitted by the daemon to notify the kernel
    - cmd_op: UBLK_IO_*
  - **user_data**
    - used by the daemon to identify the completion
    - can contain tag, cmd_op, private data, etc.
  - CQE
    - completed by the kernel to notify the user space daemon



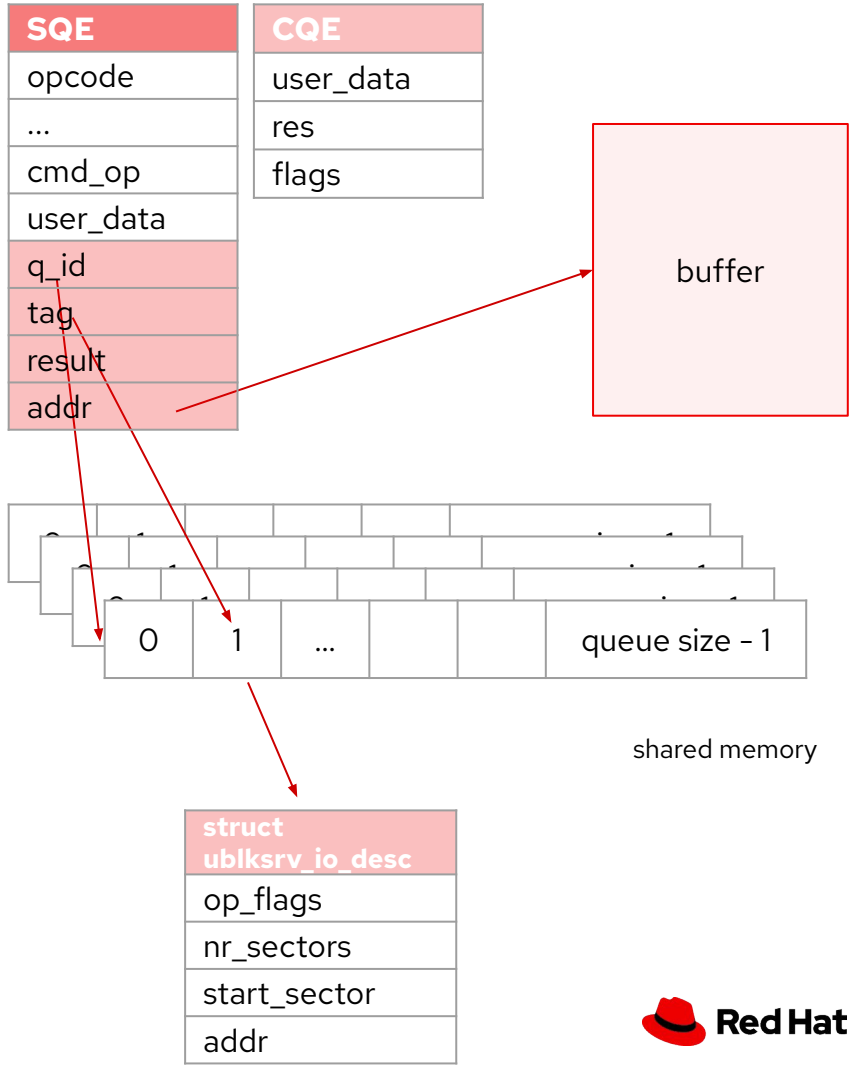
ublk data path: fetch requests

- **UBLK_IO_FETCH_REQ**
  - Issued only once initially to fetch the first requests
  - SQE
    - submitted by the daemon to inform the kernel that it is **ready to process a new request** for that slot
    - q_id and tag
      - identifies the request and also the slot in the shared memory
        - q_id: queue index
        - tag: slot index in the queue
    - result
      - not used
    - addr
      - pointer to the user space buffer that will be used to handle the request
  - CQE
    - completed by the kernel when there is a new request to handle
      - request details stored by the kernel in the shared memory (q_id, tag)



ublk data path: commit and fetch requests

- **UBLK_IO_COMMIT_AND_FETCH_REQ**
  - Issued at runtime to complete previous requests and fetch new requests
  - SQE
    - submitted by the daemon to inform the kernel that
      - **previous operation** in the slot was **completed**
      - it is **ready to process a new request** for that slot
    - q_id and tag
      - identifies the request and also the slot in the shared memory
        - q_id: queue index
        - tag: slot index in the queue
    - **result**
      - result of the completed operation propagated to the block device request
    - addr
      - pointer to the user space buffer that will be used to handle the request
  - CQE
    - completed by the kernel when there is a new request to handle
      - request details stored by the kernel in the shared memory (q_id, tag)



ublk: user space libraries

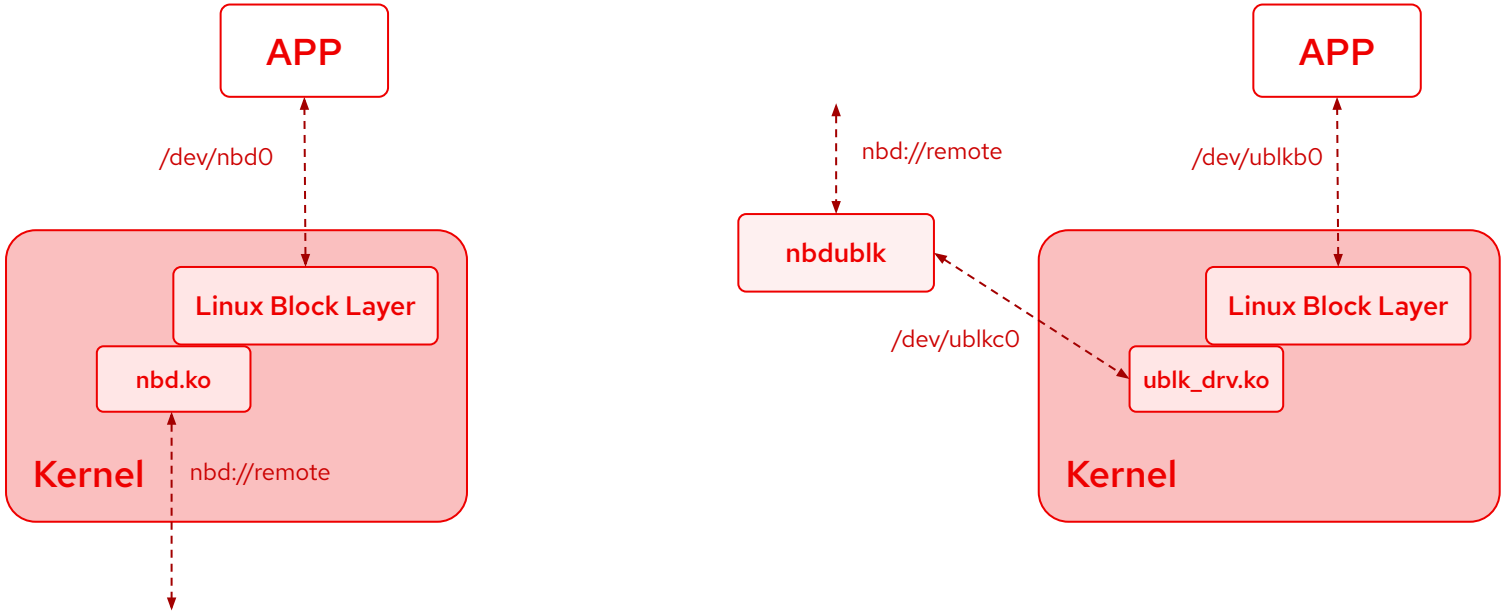
- ubdsrv's ublk library
  - https://github.com/ming1/ubdsrv
  - Developed by Ming together with the ublk-drv.ko kernel module
  - libublksrv: https://github.com/ming1/ubdsrv/tree/master/lib
    - It is planned to make it as one standalone repo and name it as libublk
  - Several examples available: null, loop, qcow2
- Rust ublk library
  - https://github.com/germag/ublk
  - Developed by German Maglione
- SPDK's ublk library
  - https://github.com/spdk/spdk/tree/master/lib/ublk
  - "SPDK ublk target acts as a ublk server. It can handle ublk I/O requests within the whole SPDK userspace storage software stack."
    - https://spdk.io/doc/ublk.html

ublk: use cases

- nbdublk
  - ublk-based NBD client
  - equivalent of the existing `nbd.ko` kernel module, but in user space

```
$ modprobe ublk_drv
$ nbdublk /dev/ublkb0 nbd://remote
```

  - Developed by Richard W.M. Jones
    - https://rwmj.wordpress.com/2022/08/25/an-nbd-block-device-written-using-linux-ublk-user-block-device/



ublk: use cases

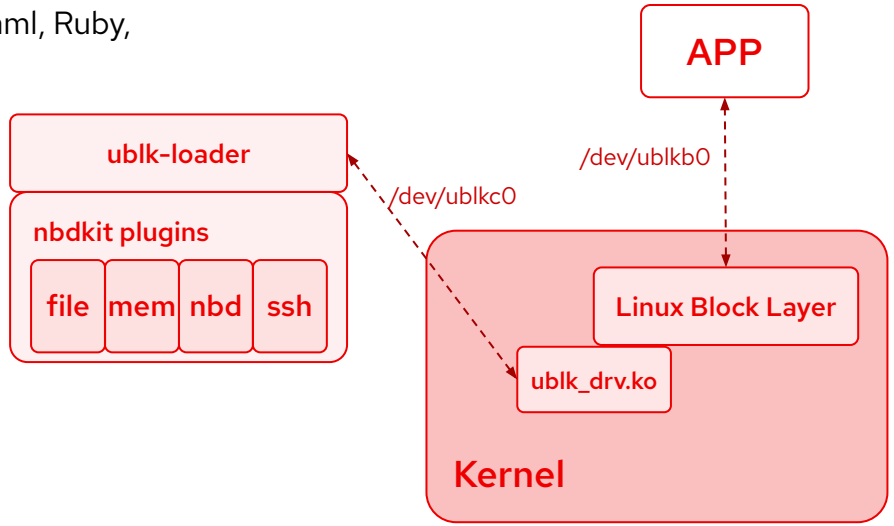
- ublk-loader
  - load nbdkit plugins as ublk device
    - nbdkit is a plugin-based NBD server
      - https://gitlab.com/nbdkit/nbdkit
      - create Linux block devices and export them over the NBD network protocol
      - several plugins already available: blkio, file, memory, nbd, ssh, etc.
        - https://gitlab.com/nbdkit/nbdkit/-/tree/master/plugins
      - new plugins can be easily written in C/C++, Go, Lua, Perl, Python, OCaml, Ruby, Rust, shell script or Tcl

```
$ modprobe ublk_drv

# loop mounting of local file
$ ublk-loader /dev/ublkb0 file disk.img

# sparse RAM disk
$ ublk-loader /dev/ublkb0 memory 1G
```

  - Developed by Richard W.M. Jones
    - https://gitlab.com/rwmjones/ublk-loader

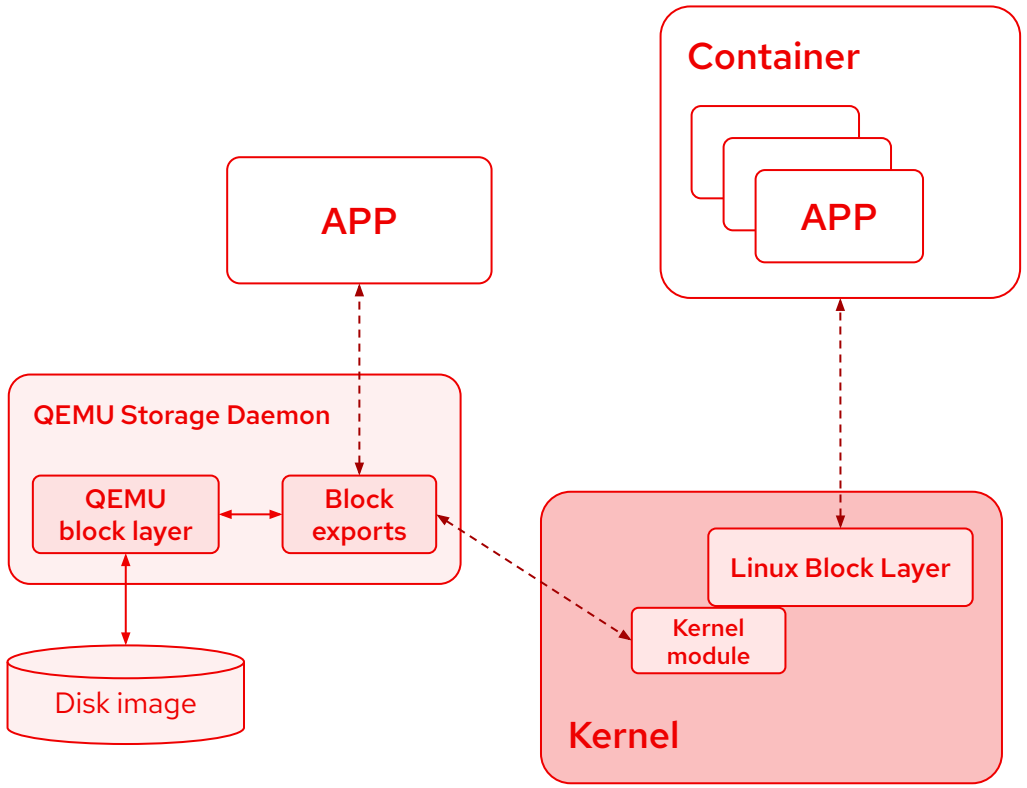


Red Hat

ublk: use cases

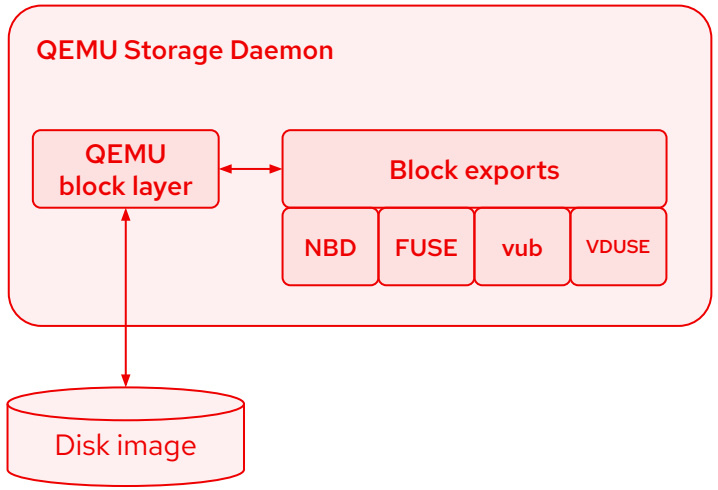
- ublk-qcow2
  - https://github.com/ming1/ubdsrv/tree/master/qcow2
  - mount qcow2 images on the host
    - host applications
    - containers use cases
  - only support basic read/write function on qcow2 image
- QEMU Copy On Write v2
  - QEMU native disk image file format
- Why QCOW2?
  - Versatile disk image format
  - Main features
    - Growing images
    - Backing files
    - Snapshots
    - Compression

QSD: QEMU Storage Daemon



- Move QEMU block layer features in a separate process
  - Export block devices to QEMU or other clients
- Why?
  - Isolation
    - separate process, less privileges, etc.
  - Usable with non-VMs workloads
    - taking advantage of the QEMU block layer features without VMs
      - image formats (e.g. QCOW2), block jobs, I/O throttling, etc.
    - attach images to
      - host OS block layer
      - directly to applications
      - containers
  - Serve multiple clients (VMs, containers, etc.)
    - Sharing hardware device (e.g. split a single disk for multiple clients)

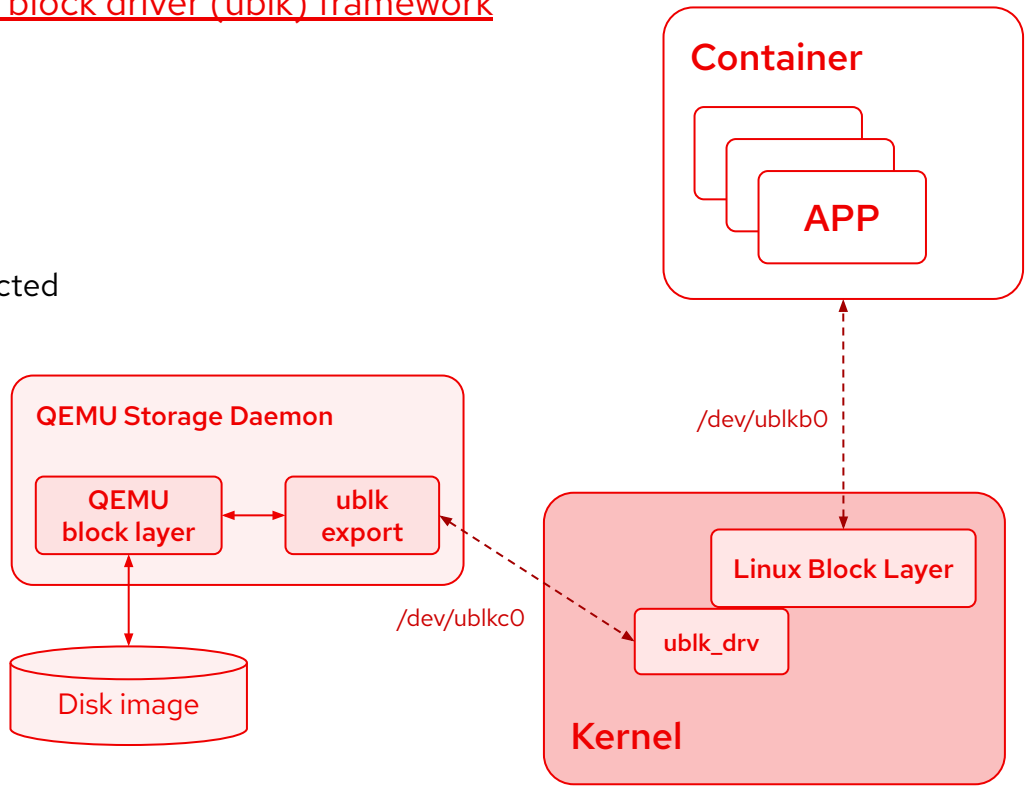
QSD: supported exports



- NBD
  - export disk images over a socket (TCP, unix, vsock, etc.)
  - usually involved in live storage migration
- FUSE
  - allow to mount disk images on the host
- vhost-user-block
  - expose disk images as a virtio-blk device
    - using the vhost-user protocol
      - unix socket + shared memory
- VDUSE
  - expose disk images as a virtio-blk device
    - using the vDPA framework
      - attachable in both host and guest using the standard virtio-blk driver

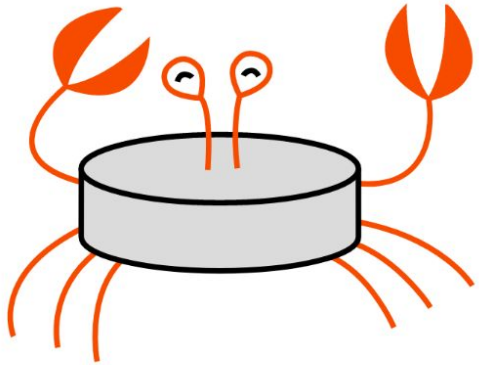
QSD: ublk export

- Developed during a QEMU internship (Outreachy Dec 2022 round)
  - Extend QEMU Storage Daemon, supporting the new Linux's userspace block driver (ublk) framework
    - Intern: Amarjargal Gundjalam
    - Mentors: German Maglione, Ming Lei, and Stefano Garzarella
  - Outreachy
    - Outreachy provides internships in open source and open science.
      Outreachy provides internships to people subject to systemic bias and impacted by underrepresentation in the technical industry where they are living.
  - Amarjargal's blog posts
    - https://dev.to/amarjargal
- Still in development to better integrate QSD and ublk



RSD: Rust Storage Daemon

- Reimplementation of the QEMU block layer, as provided by the qemu-storage-daemon (QSD), in Rust
  - It allows management of VM images, providing virtio-blk devices that can be connected to QEMU instances via the vhost-user protocol.
- Developed by Hanna Czenczek
  - https://gitlab.com/hreitz/rsd
- First version (v0.1) just released (June 2023)
  - https://lists.nongnu.org/archive/html/qemu-block/2023-06/msg00182.html
- Blog posts
  - Part 1 (Overview): https://czenczek.de/blog/rsd-overview.html
  - Part 2 (Performance): https://czenczek.de/blog/rsd-performance.html



Thank you!

Stefano Garzarella
<sgarzare@redhat.com>

German Maglione
<gmaglione@redhat.com>

linkedin.com/company/red-hat

facebook.com/redhatinc

youtube.com/user/RedHatVideos

twitter.com/RedHat

Red Hat