io_uring meets network

Kernel Recipes 2023

Pavel Begunkov

Meta

- **IORING_OP_SENDMSG**
- **IORING_OP_RECVMSG**

submission

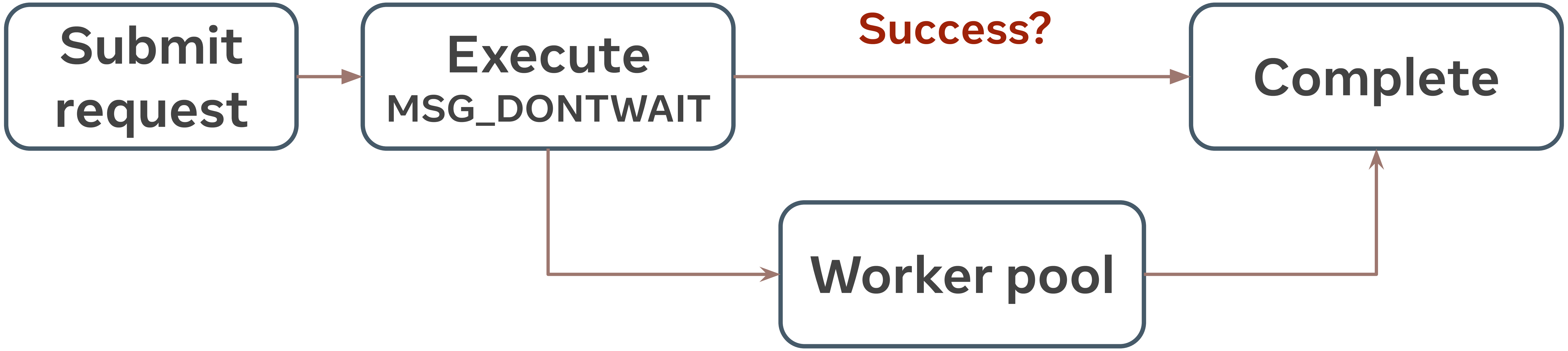
```
struct msghdr msg = { … };
msg_flags = MSG_WAITALL;

sqe = io_uring_get_sqe(&ring);
io_uring_prep_sendmsg(sqe, sockfd,
                              &msg, msg_flags);
sqe->user_data = tag;
io_uring_submit(ring);
```

completion / waiting

```
ret = io_uring_wait_cqe(&ring, &cqe);
assert(cqe->user_data == tag);
result = cqe->res;
```

Early days execution



Submit request

Execute
MSG_DONTWAIT

Success?

Complete

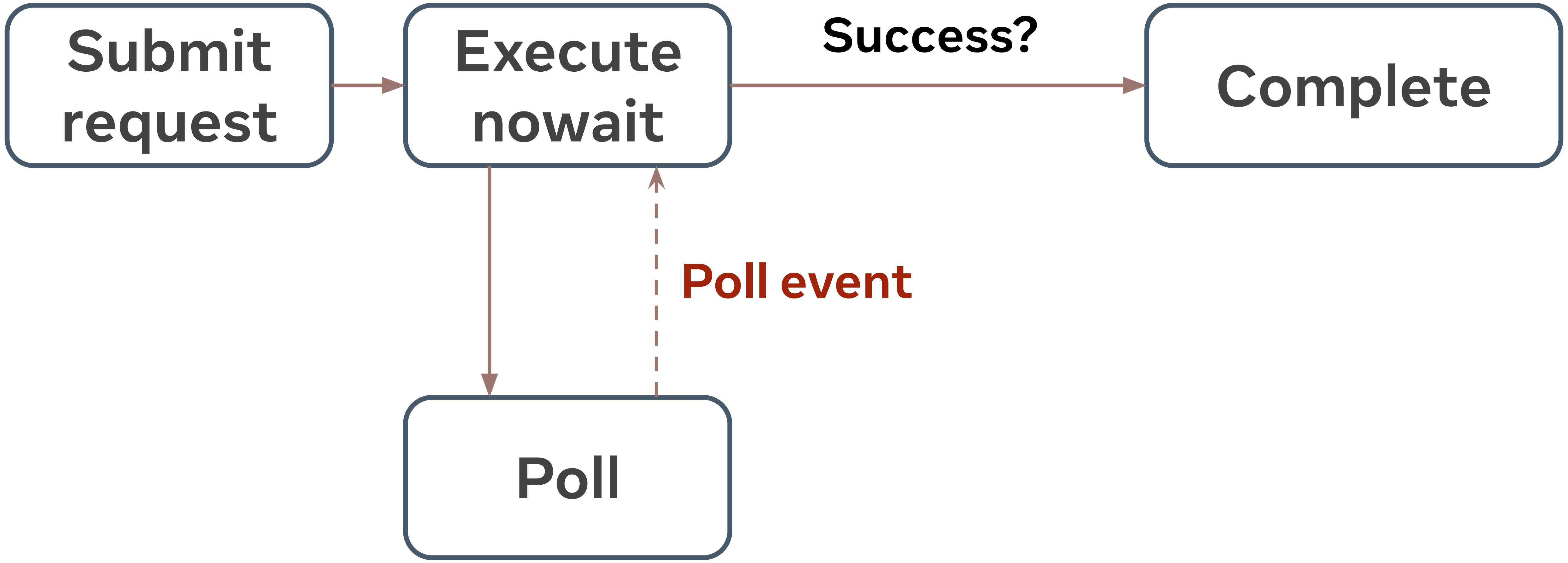
Worker pool

Polling

`IORING_OP_POLL_ADD`

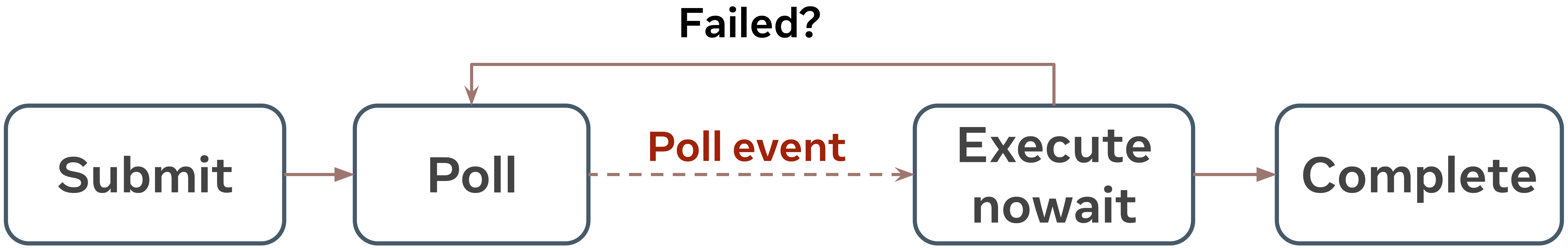
- Asynchronous, as it should be
- Polling a single file
- Terminates after the first desired event
  - User has to send another request to continue polling
- Can be cancelled by **IORING_OP_POLL_REMOVE**
  or **IORING_OP_ASYNC_CANCEL**

- What if we combine IO with polling?
- Kernel internally polls when **MSG_DONTWAIT** failed
- Transparent, uapi stays the same
- Check support with **`IORING_FEAT_FAST_POLL`**



Tip 1: use `IORING_RECVSEND_POLL_FIRST` with receive requests

- Starts with polling, skips the first nowait attempt
- Useful when it’s likely have to wait
- Usually not useful for sends



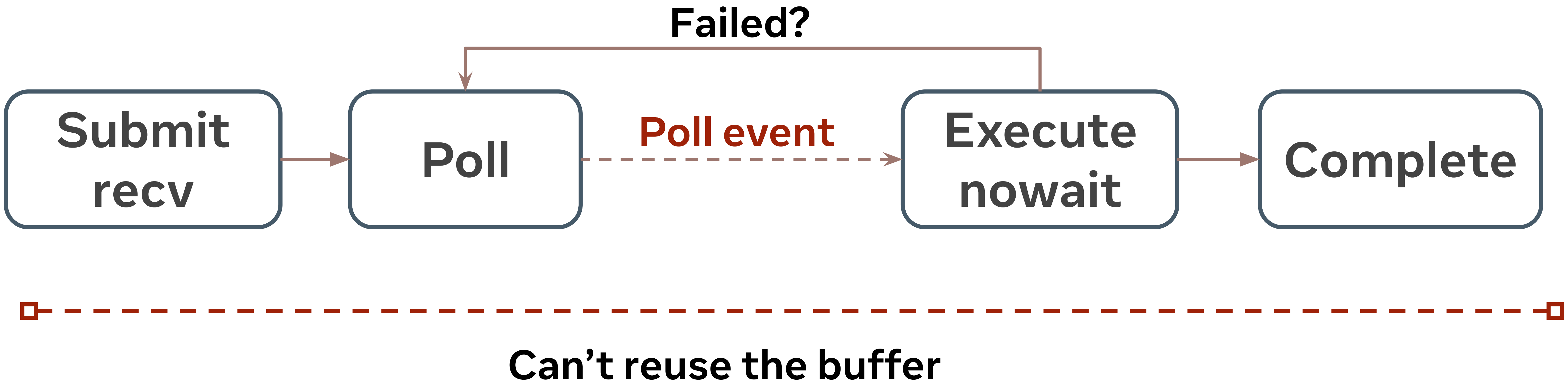
Tip 2: io_uring supports `MSG_WAITALL`, retries short IO

- Works with recv as well as sends
- Ignored by io_uring unless it’s a streaming socket like **TCP**

```
do {
    left = total_len - done;
    ret = do_io(buf + done, left);
    done += ret;
    // poll_wait();
} while (done < total_len && (msg_flags & MSG_WAITALL))
```

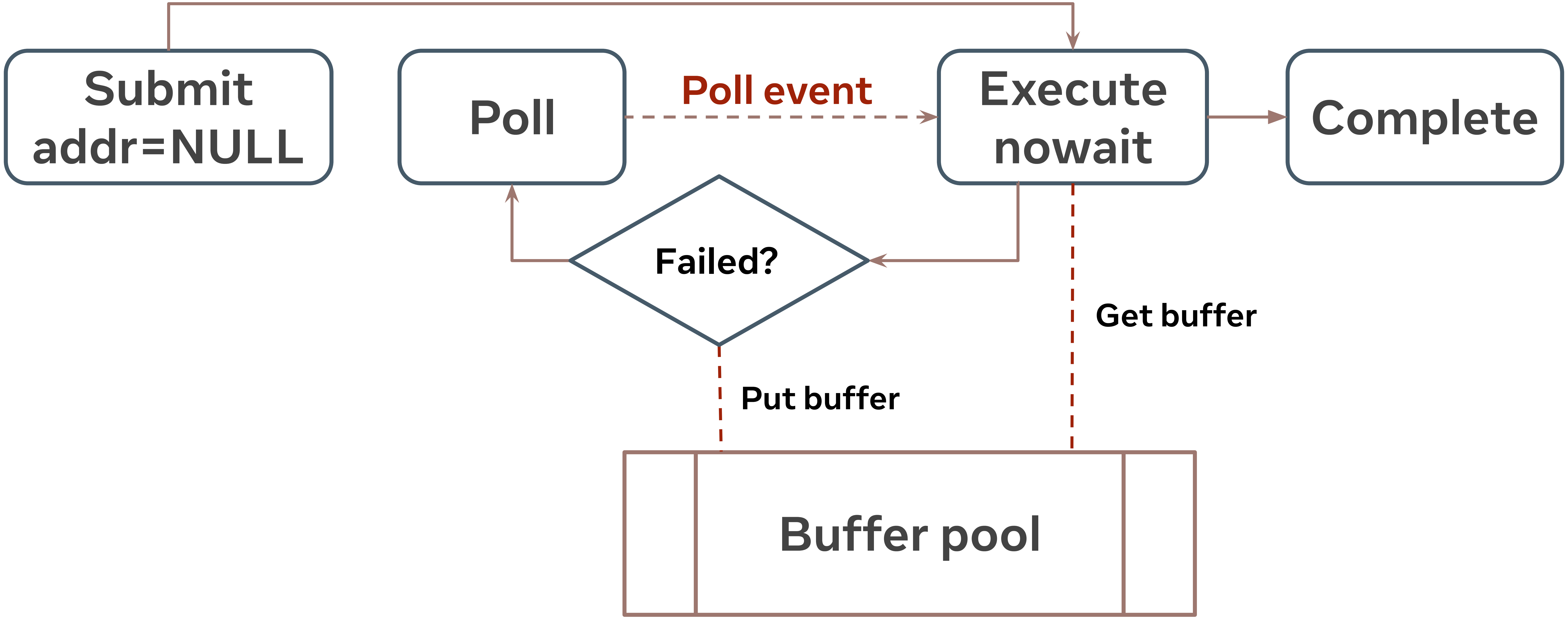
Memory consumption

- Each recv takes and holds a buffer
- Buffers can’t be reused before recv completes
- Many (slow) connections may lock up too much memory



Provided buffers

Let’s the kernel have a buffer pool!



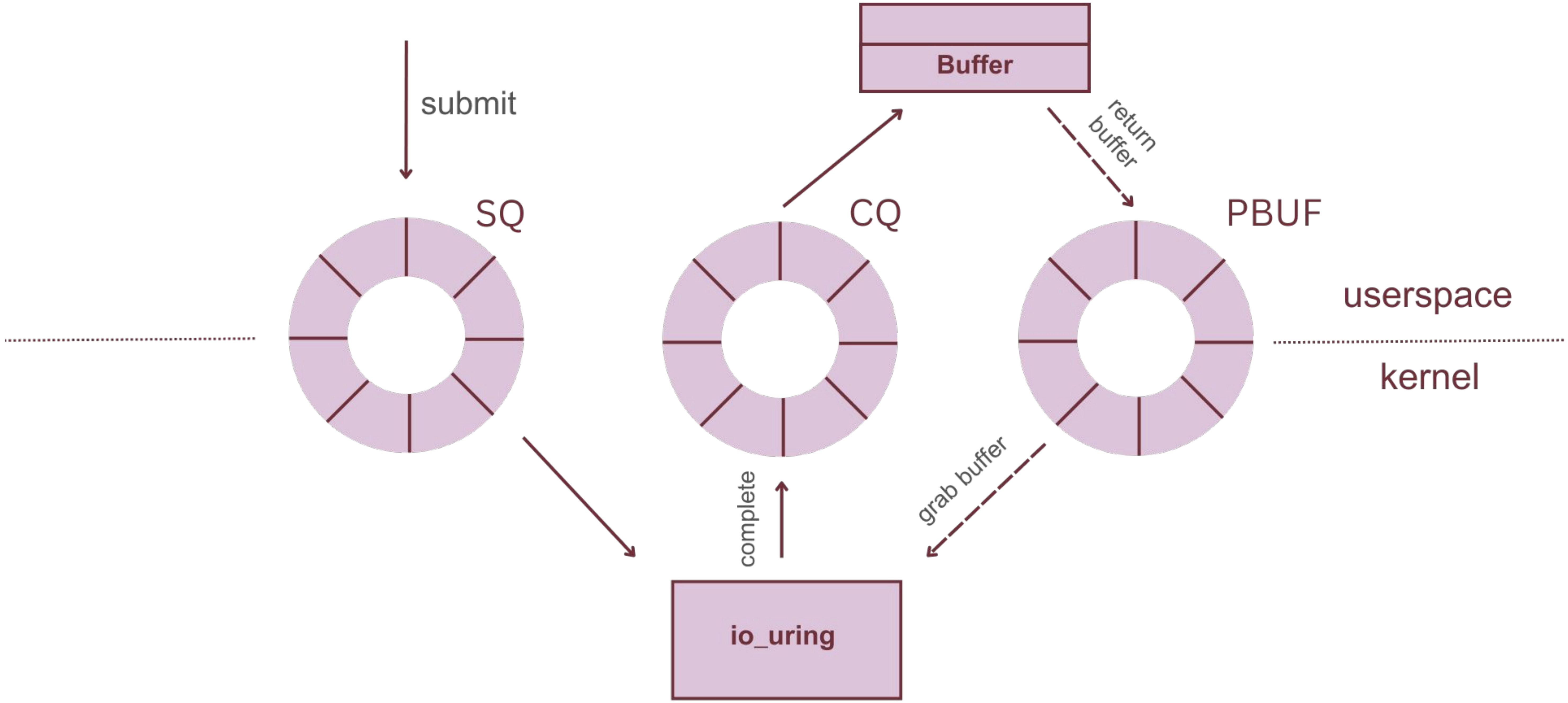
Provided buffers: overview

- In-kernel buffer pool
  - User can register multiple pools
  - Each pool has an ID to refer to
  - Usually, buffers in a pool are same sized
- Don’t set buffer at submission, e.g. sqe->addr = NULL;
  - sqe->flags |= IOSQE_BUFFER_SELECT
  - And specify the buffer pool ID to use
- Request grabs a buffer on demand
  - Requests don’t hold a buffer while polling
  - It’ll grab it right before attempting to execute
- The buffer ID will be returned in cqe->flags
- The user should keep refilling the pool

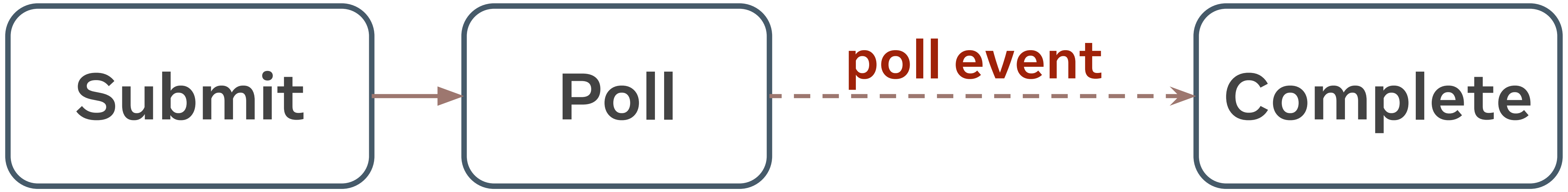
Provided buffers: returning buffers

- **V1: IORING_OP_PROVIDE_BUFFERS**
  - Buffers are returned by sending a special request
  - Slow and inefficient
- **V2: IORING_REGISTER_PBUF_RING**
  - Another kernel-user shared ring
  - User returns buffers by putting them in the ring
  - Nicely wrapped in liburing

Provided buffers v2

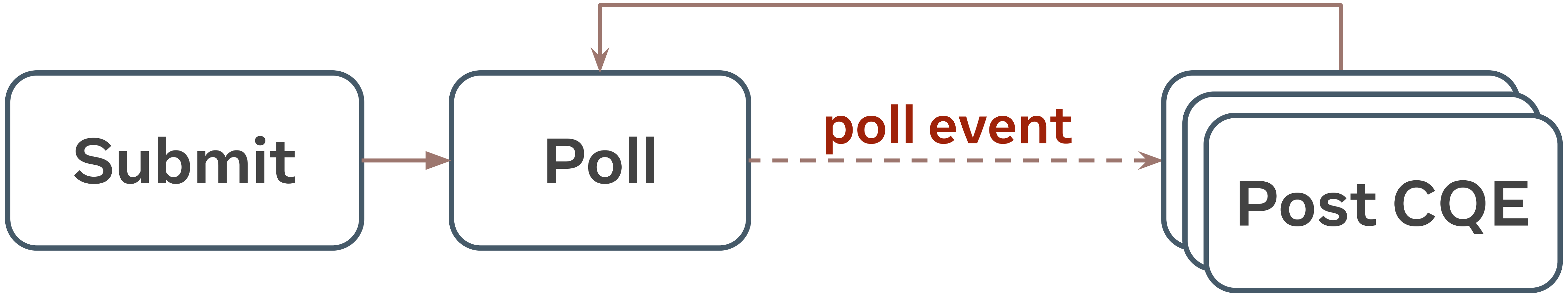


Back to polling

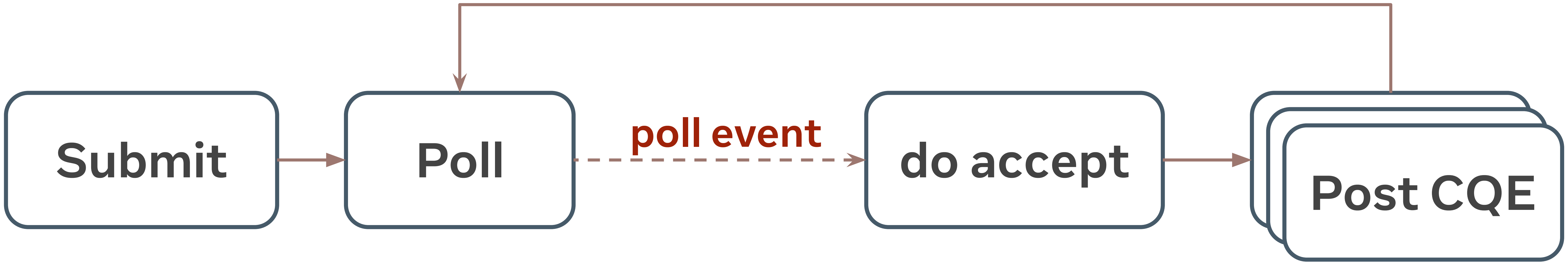


Why poll requests terminate after the first event?

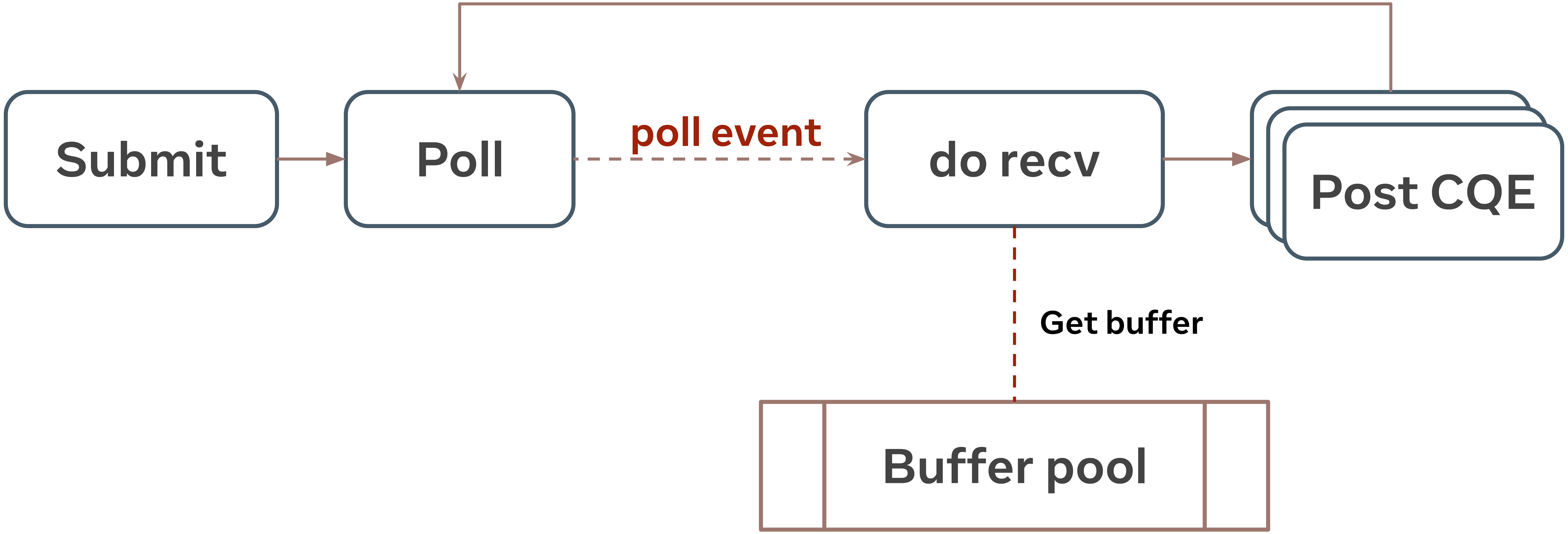
Multishot poll



Multishot accept



Multishot recv



Notes on multishot…

- Requests can be cancelled via **IORING_OP_ASYNC_CANCEL**
  - Or by shutting down the socket

- Requests can fail…
  - Resend if recoverable: out of buffers, CQ is full, **-ENOMEM**, etc.

- **C**ompletion **Q**ueue is finite
  - io_uring will save overflow CQEs, but it's slow
    - User has to enter the kernel to flush overflown CQE
  - Multishot requests will be terminated

- Linked requests don't work well with multishots

Fixed files

IOSQE_FIXED_FILE optimises per request file refcounting

- Makes much sense with send requests
- But not recommended with potentially time unbound requests
    - May cause problems
- Doesn’t benefit multishots, cost is already amortised

Connection management

IORING_OP_CLOSE - closes a file descriptor.

- Interoperable with **close(2)** for regular (non-**IOSQE_FIXED_FILE**) files

Close doesn’t kill a connection with in-flight requests

- Either cancel requests
- Or **IORING_OP_SHUTDOWN / shutdown(2)** it first

There are **IORING_OP_ACCEPT, IORING_OP_CONNECT** and **IORING_OP_SOCKET**

Zerocopy

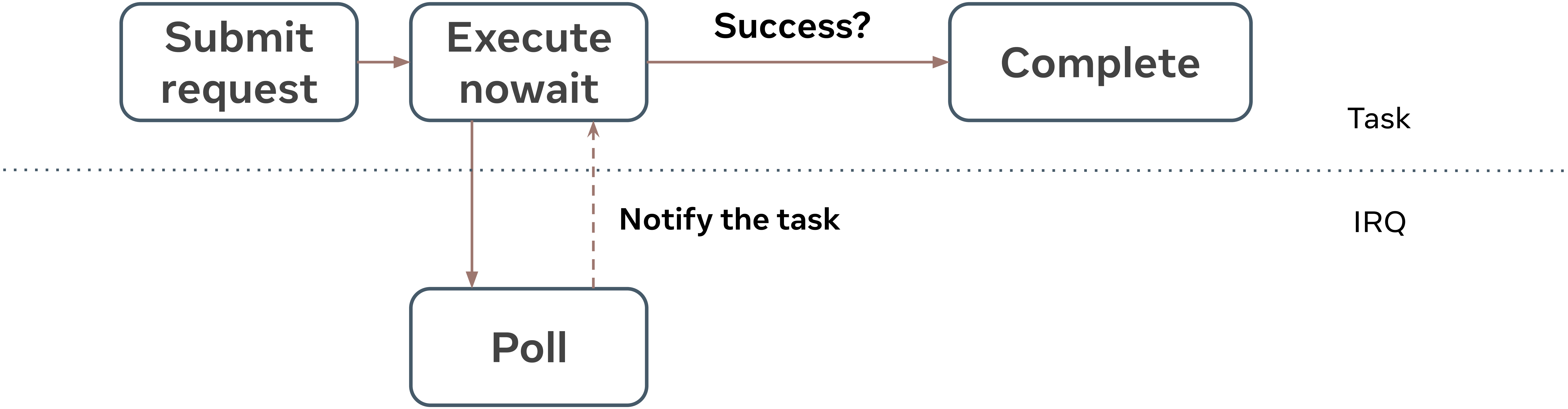
Zerocopy send

- **IORING_OP_SEND_ZC:** 2 CQEs, “queued” and “completed”
- Need to add vectored IO support

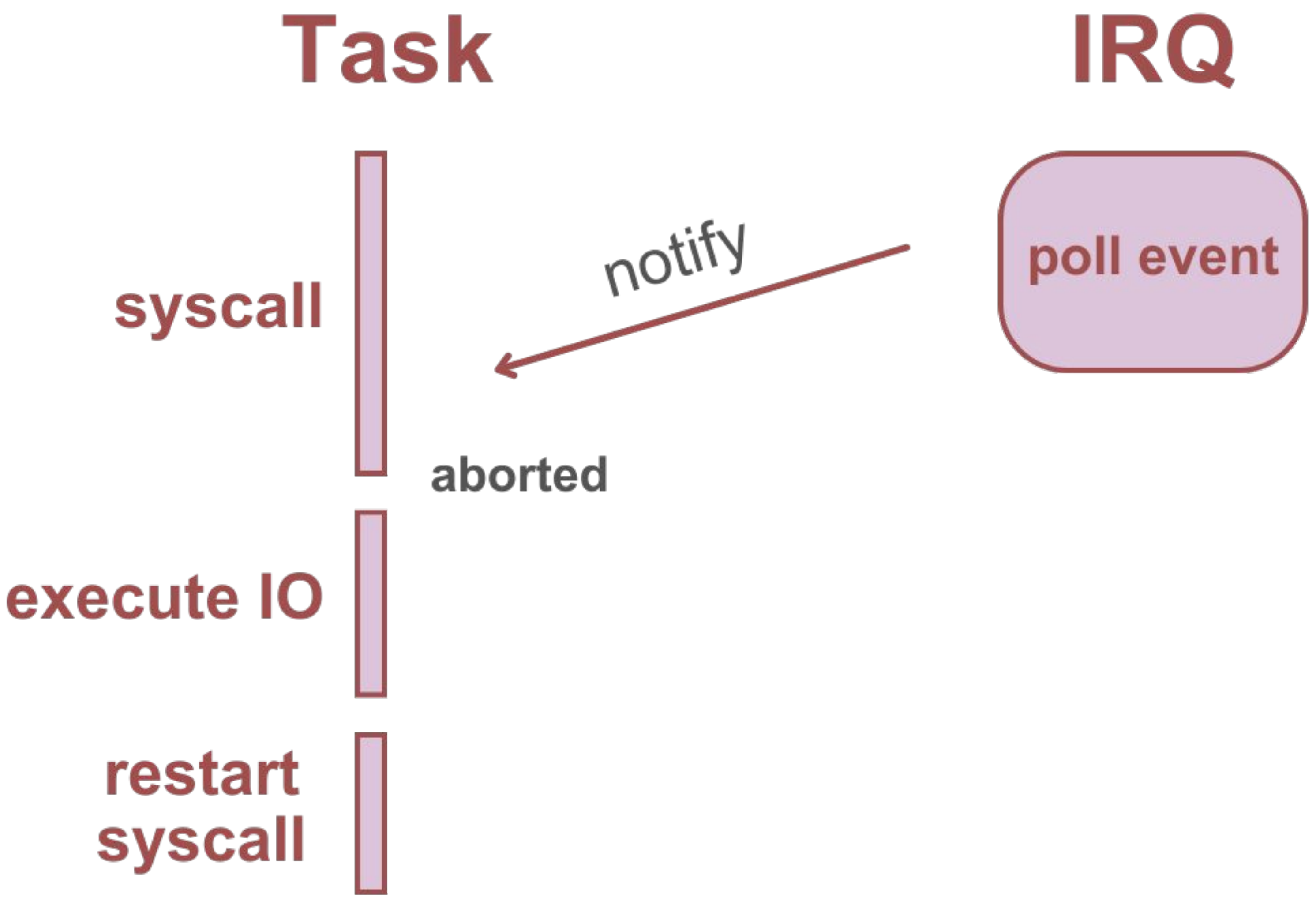
Zerocopy receive

- RFC is out, look for updates
- Multishot recv applications are already half prepared
- https://lore.kernel.org/io-uring/20230826011954.1801099-1-dw@davidwei.uk/

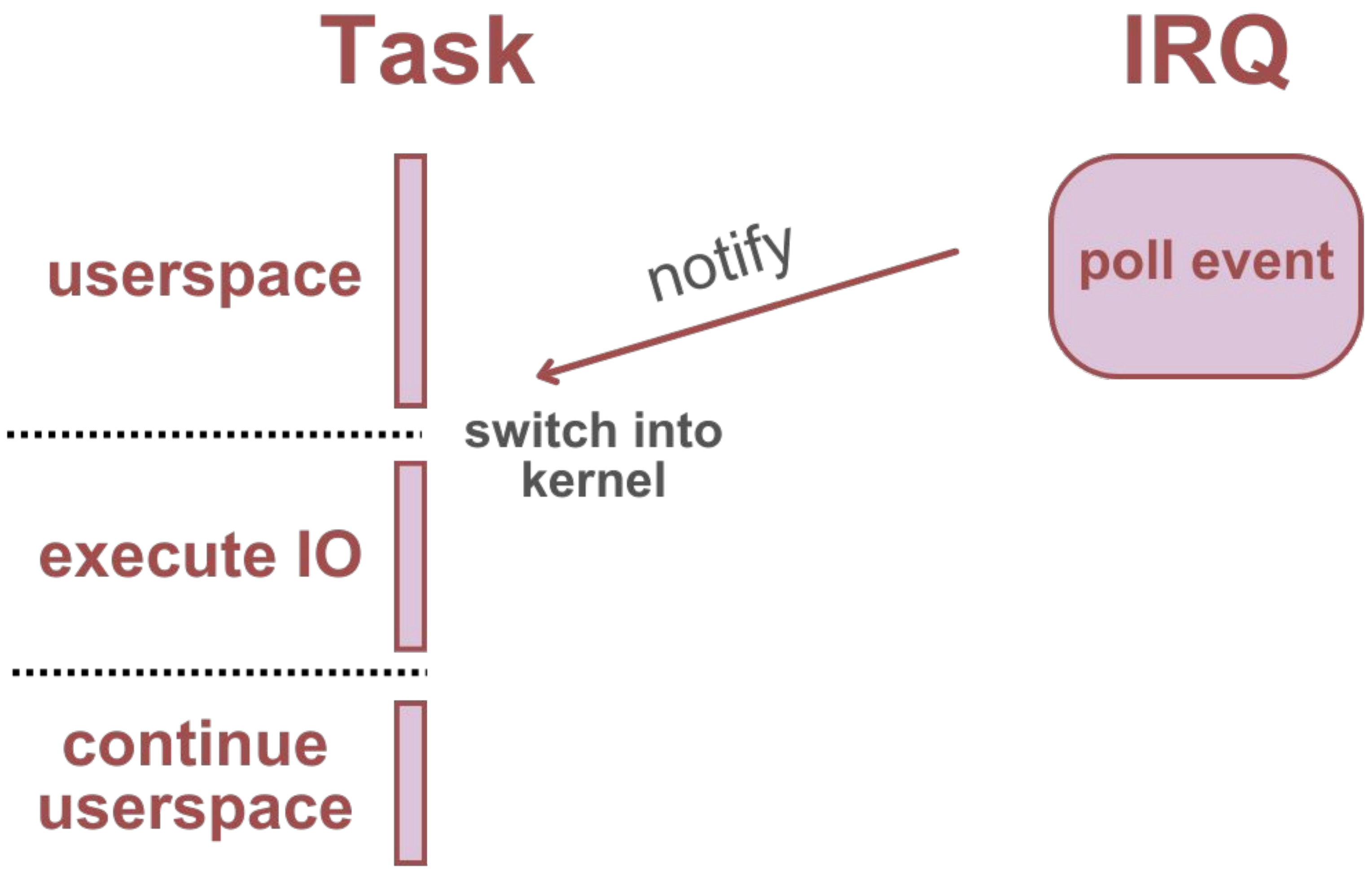
Task execution



Task work


Task
IRQ
syscall
notify
poll event
aborted
execute IO
restart
syscall

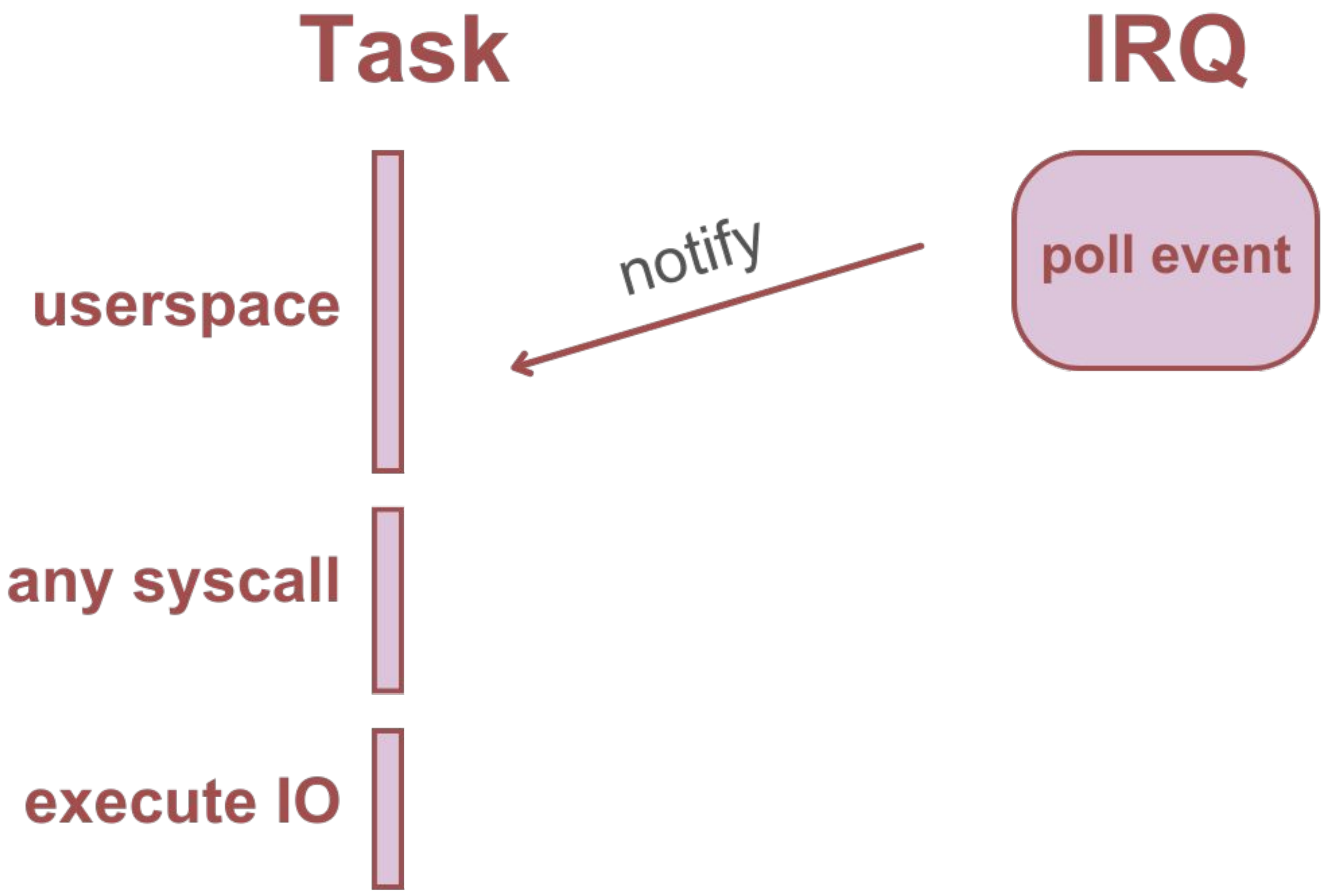
Task work


Task
IRQ
userspace
notify
poll event
switch into kernel
execute IO
continue userspace

Task work overview

- Poll event arrives in an IRQ* context
- We wake up the submitter task to execute the IO
- **task_work** similar to signals but in-kernel
  - Wakes the task if sleeping
  - Interrupts any syscall
  - Forces userspace into the kernel
- Hot path is generally executed by the submitter task

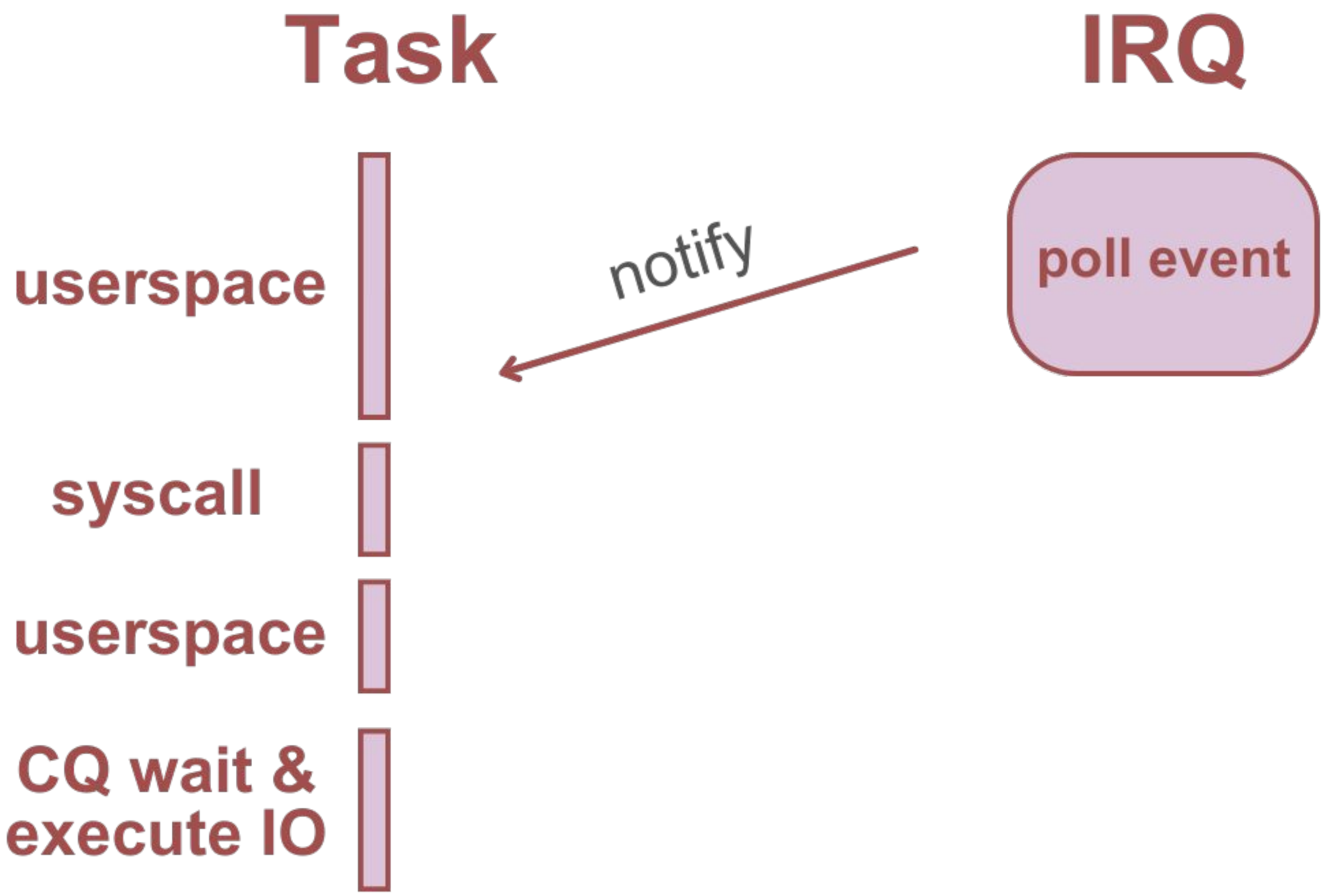
IORING_SETUP_COOP_TASKRUN



IORING_SETUP_COOP_TASKRUN

- Doesn’t interrupt running userspace
- Still aborts running syscalls
- Will be executed with the next syscall
  - Hence the app has to eventually make a syscall
- The user should not busy poll CQ
  - It’s almost never a good idea regardless

IORING_SETUP_DEFER_TASKRUN



IORING_SETUP_DEFER_TASKRUN

- Executed only in **io_uring_enter(2)** syscall
- User has to enter the kernel to wait for events
- Requires **IORING_SETUP_SINGLE_ISSUER**

Performance

Performance highly depends on batching

- submission batching
- as well as completion batching

Be prepared for tradeoffs

- Wait for longer until there is more to submit
- Wait for multiple completions, possibly with a timeout
- Throughput vs latency

Gluing together

- One io_uring instance per process
  - No need to share, no synchronisation around queues
  - Add **IORING_SETUP_SINGLE_ISSUER** and **IORING_SETUP_DEFER_TASKRUN**
- Processes communicate via **IORING_OP_MSG_RING**
- Each process serves multiple sockets
  - The more sockets per process the better, improves batching
- Simple **IORING_OP_SEND[MSG]** requests are usually fine
  - Often complete by the time the submission syscall returns
- One recv request for each socket
  - Needs a provided buffer pool

Timeouts

- CQ waiting with a timeout, see **io_uring_wait_cqe_timeout()**, etc.
- **IORING_OP_TIMEOUT -** timeout request, supports multishot
- **IORING_OP_LINK_TIMEOUT -** per request timeout
    - There is a cost, app might want to implement it in userspace via **IORING_OP_TIMEOUT + IORING_OP_ASYNC_CANCEL**

References

- Liburing - io_uring userspace library
  github.com/axboe/liburing/
  git://git.kernel.dk/liburing.git
- Write up about networking
  https://github.com/axboe/liburing/wiki/io_uring-and-networking-in-2023
- Benchmarking
  https://github.com/dylanZA/netbench
- io_uring mailing list
  io-uring@vger.kernel.org
- Zerocopy receive
  https://lore.kernel.org/io-uring/20230826011954.1801099-1-dw@davidwei.uk/
- Folly library: supports io_uring with all modern features
  https://github.com/facebook/folly.git