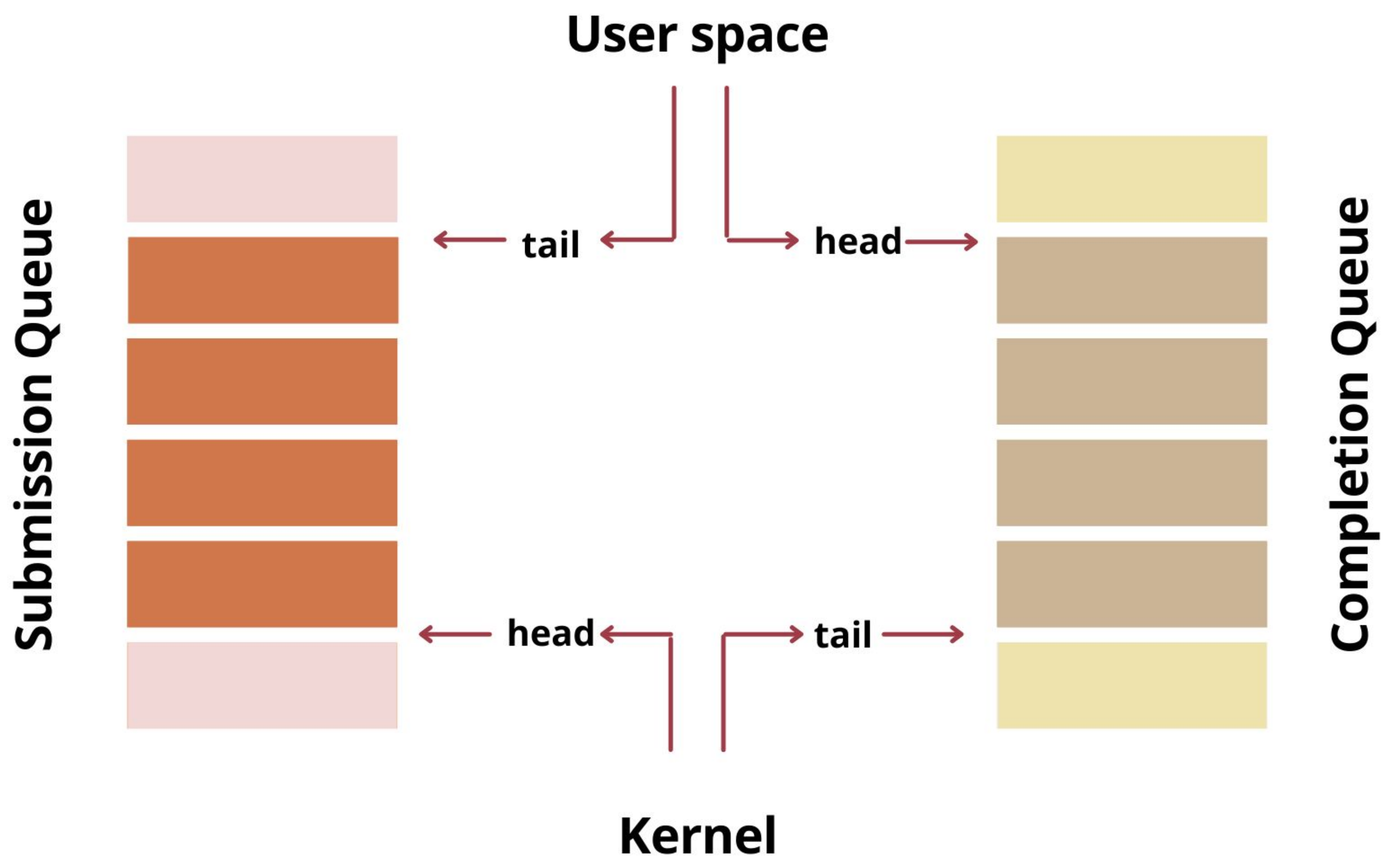
io_uring: BPF controlled I/O

Linux Plumbers 2021

Pavel Begunkov
asml.silence at gmail.com

FACEBOOK

io_uring: introduction



Lots of operations

```
enum {
    IORING_OP_NOP,
    IORING_OP_READV,
    IORING_OP_WRITEV,
    IORING_OP_FSYNC,
    IORING_OP_READ_FIXED,
    IORING_OP_WRITE_FIXED,
    IORING_OP_POLL_ADD,
    IORING_OP_POLL_REMOVE,
    IORING_OP_SYNC_FILE_RANGE,
    IORING_OP_SENDMSG,
    IORING_OP_RECVMSG,
    IORING_OP_TIMEOUT,
    IORING_OP_TIMEOUT_REMOVE,
    IORING_OP_ACCEPT,
    IORING_OP_ASYNC_CANCEL,
    IORING_OP_LINK_TIMEOUT,
    IORING_OP_CONNECT,
    IORING_OP_FALLOCATE,
    ...
```

```
    ...
    IORING_OP_OPENAT,
    IORING_OP_CLOSE,
    IORING_OP_FILES_UPDATE,
    IORING_OP_STATX,
    IORING_OP_READ,
    IORING_OP_WRITE,
    IORING_OP_FADVISE,
    IORING_OP_MADVISE,
    IORING_OP_SEND,
    IORING_OP_RECV,
    IORING_OP_OPENAT2,
    IORING_OP_EPOLL_CTL,
    IORING_OP_SPLICE,
    IORING_OP_PROVIDE_BUFFERS,
    IORING_OP_REMOVE_BUFFERS,
    IORING_OP_TEE,
    IORING_OP_SHUTDOWN,
    IORING_OP_RENAMEAT,
    IORING_OP_UNLINKAT,
    IORING_OP_MKDIRAT,
    IORING_OP_SYMLINKAT,
    IORING_OP_LINKAT,
};
```

Features

- **`SQPOLL`** for syscall-less submission
- **`IOPOLL`** for beating performance records
- Registered resources with fast updates
  - **`IORING_REGISTER_FILES`**: optimised file refcounting
  - **`IORING_REGISTER_BUFFERS`**: eliminates page refcounting, no page table walking, etc.
  - dynamic fast updates: no more full io_uring quiesce
- **`IOSQE_IO_LINK`**: request links for execution ordering
- **`IORING_FEAT_FAST_POLL`**: automatic poll fallback, no need for epoll
- **`IO-WQ`**: internal thread pool, when nothing else works
- multi-shot requests, e.g. poll generating multiple CQEs
- executors (**IO-WQ**, **SQPOLL**) sharing
- *and more …*

Execution flow

First try nowait: IOCB_NOWAIT, LOOKUP_CACHED, etc.

- might just complete, e.g. if data is already there
- O_DIRECT goes async, -EIOCBQUEUED
- added to a waitqueue, e.g. poll requests

Try async buffered read, see FMODE_BUF_RASYNC

Internally try polling if supported, see IORING_FEAT_FAST_POLL

- once fires, goto nowait attempts again

Any other way to go genuinely async; will be more in the future

Fall back to a **thread pool**, slower but often necessary

Misconception debunking

io_uring is not "just a worker pool"

- worker threads is a slower path

io_uring is not I/O Completion Ports (IOCP)

- ... Microsoft is now developing a io_uring for Windows

io_uring is not only about syscall elimination/reduction

- provides asynchrony
- easy parallelism
- provides a state to base optimisations on, e.g. registerested files

The problem

By Natascha Eibl - https://meltdownattack.com/, CC0,
https://commons.wikimedia.org/w/index.php?curid=65233480
https://commons.wikimedia.org/w/index.php?curid=65235937

syscall overhead

Vulnerability mitigations are **expensive**, and so are syscalls

- cost varies with CPU and enabled mitigations

Overhead for syscalls in a tight loop with little work can take **20-50%** (apparently, tested CPU is the worst case)

```
# copy by 4KB at a time
# cp_4kb ./file /dev/zero

  29.47%  busybox  [kernel.vmlinux]  [k] syscall_exit_to_user_mode
  12.68%  busybox  [kernel.vmlinux]  [k] entry_SYSCALL_64
  12.49%  busybox  [kernel.vmlinux]  [k] syscall_return_via_sysret
   0.52%  busybox  [kernel.vmlinux]  [k] do_syscall_64
   ...
```

```
# mitigations enabled
# nop requests, batch 32
# fio/t/io_uring -d32 -s32 -c32 -N1

  16.41%  io_uring  [kernel.vmlinux]  [k] io_submit_sqe
  14.78%  io_uring  [kernel.vmlinux]  [k] syscall_exit_to_user_mode
  10.70%  io_uring  [kernel.vmlinux]  [k] __io_submit_flush_completions
  10.17%  io_uring  [kernel.vmlinux]  [k] io_submit_sqes
   9.78%  io_uring  [kernel.vmlinux]  [k] io_issue_sqe
   7.61%  io_uring  [kernel.vmlinux]  [k] __io_queue_sqe
   7.28%  io_uring  [kernel.vmlinux]  [k] io_req_free_batch
   5.07%  io_uring  [kernel.vmlinux]  [k] entry_SYSCALL_64
   4.79%  io_uring  [kernel.vmlinux]  [k] syscall_return_via_sysret
   4.29%  io_uring  io_uring          [.] submitter_fn
   2.75%  io_uring  [kernel.vmlinux]  [k] io_alloc_req
   ...
```

```
# mitigations enabled
# Null block device, “realistic batching” 4 requests at a time
# modprobe null_blk no_sched=1 irqmode=1 completion_nsec=0 submit_queues=16
# fio/t/io_uring -d4 -s4 -c4 -p1 -B1 -F1 -b512 /dev/nullb0

   9.01%  io_uring  [kernel.vmlinux]  [k] syscall_exit_to_user_mode
   4.87%  io_uring  [kernel.vmlinux]  [k] blkdev_direct_IO
   3.27%  io_uring  [kernel.vmlinux]  [k] entry_SYSCALL_64
   2.92%  io_uring  [kernel.vmlinux]  [k] syscall_return_via_sysret
   2.89%  io_uring  [kernel.vmlinux]  [k] kmem_cache_free
   2.74%  io_uring  [null_blk]        [k] null_queue_rq
   2.68%  io_uring  io_uring          [.] submitter_fn
   2.31%  io_uring  [kernel.vmlinux]  [k] blkdev_bio_end_io
   2.27%  io_uring  [kernel.vmlinux]  [k] io_issue_sqe
   2.19%  io_uring  [kernel.vmlinux]  [k] io_do_iopoll
   2.12%  io_uring  [kernel.vmlinux]  [k] kmem_cache_alloc
   ...
```

Sweet spot for optimisation. How about **SQPOLL**?

- still needs userspace to process completions
- takes a CPU core; high CPU consumption
- cache bouncing

BPF is there to help! Can also help latency

Requirements

Flexibility: what capabilities BPF has to have?

- submitting new requests
- accessing CQEs, multiple if needed
- poking into userspace memory

Low overhead

- Traditionally we've optimised batched submission more
- BPF is expected to have a lower batch ratio

Idea 1: let's add a callback to each request and run it on completion

- needs hooks in generic paths, non-zero cost
- limits control over execution context
- can't do waiting and other async stuff
- BPF needs context, would need allocation
- looks horrible ...

```
struct io_uring_sqe {
    ...
    u32 callback_id;
};

int io_init_req(struct io_uring_sqe *sqe)
{
    if (sqe->callback_id)
        req->bpf_cb = get_bpf(sqe->callback_id);
    ...
}

void io_req_complete(struct io_kiocb *req, long res)
{a
    if (req->callback)
        req->bpf_cb(req, res);
    ...
}
```

New io_uring request type: `IORING_OP_BPF`

No extra per request overhead, everything is enclosed in opcode handlers.

And we can use generic io_uring infrastructure:

- locking and better control of execution context
- completion and other batching
- space in the internal request struct, i.e. struct io_kiocb
- can be linked to other requests
- possible to execute multiple times, i.e. keeping a BPF request alive

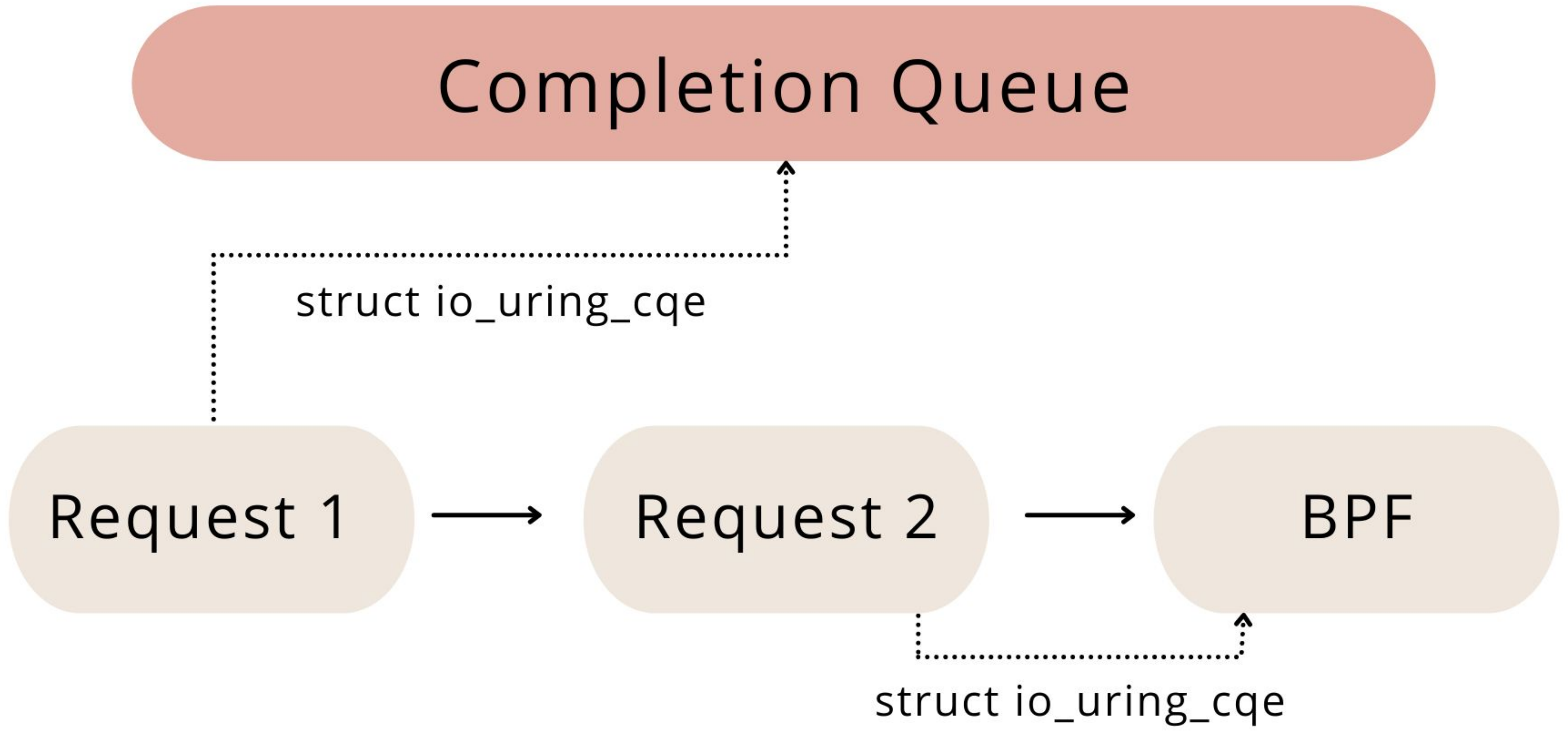
The downside is that extra requests are not free, there is a cost to that, but we can work with it.

Feeding BPF completions

BPF needs feedback from other requests.

The first idea: just use links and pass a CQE of the previous request to BPF!

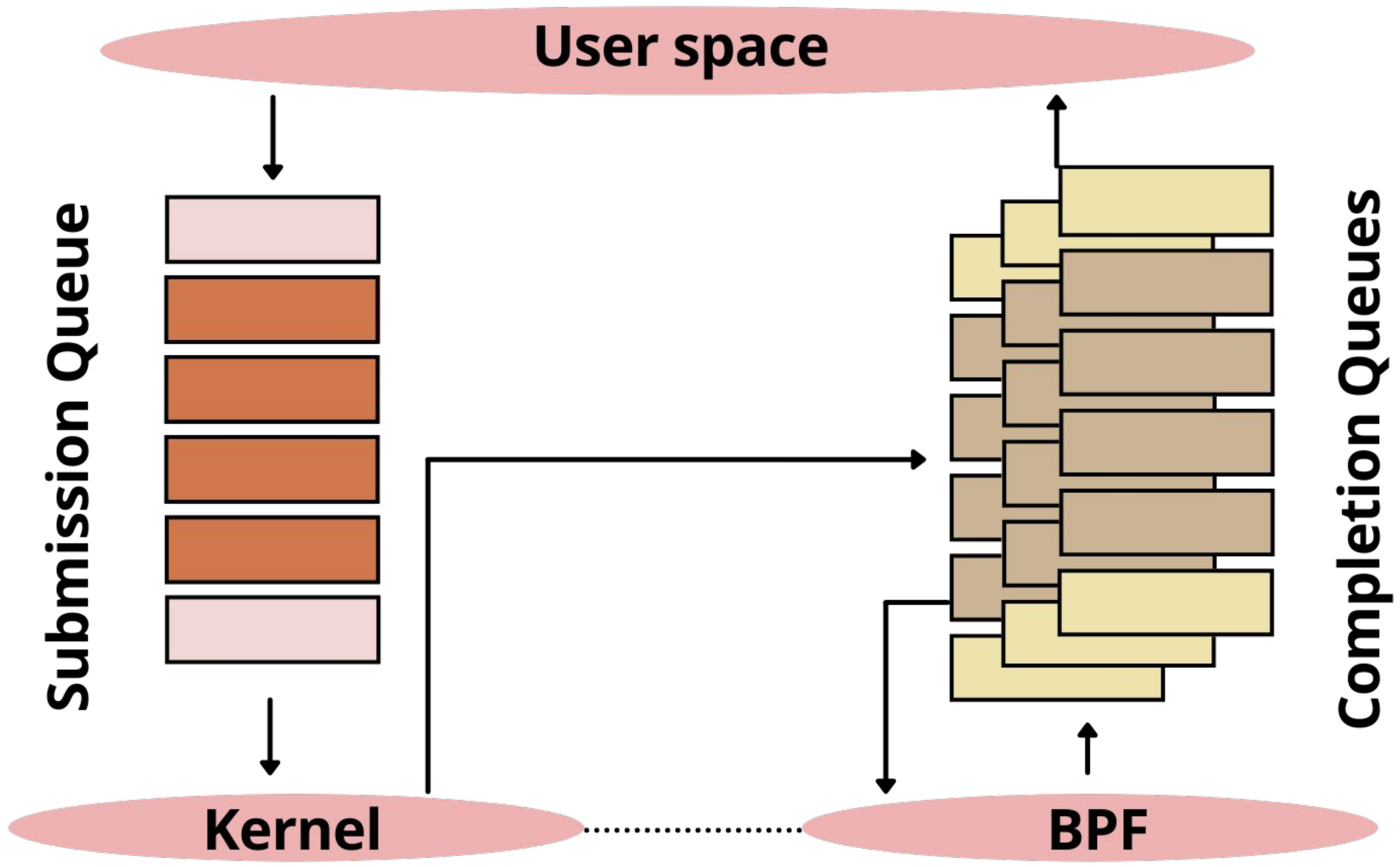
- ugly again
- bound to linking by design
- no way to pass multiple CQEs
- extra overhead for non-BPF code



Multiple CQs

Introduce multiple CQs:

- sqe->cq_idx, each request specifies to which CQ its completion goes
- BPF can emit and consume CQEs to / from any CQ
- Can wait
- Synchronisation is up to the userspace / BPF

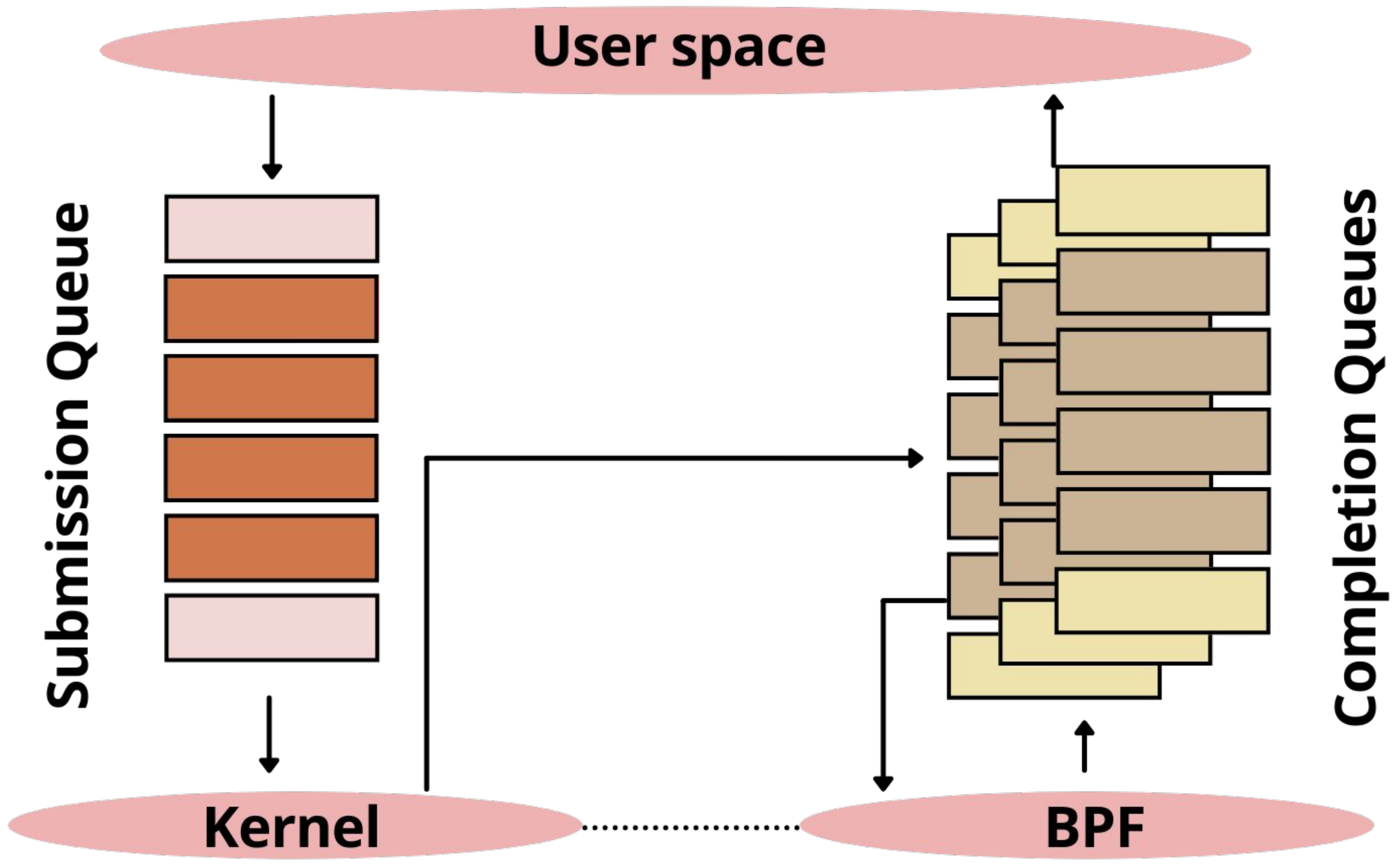


Pros:

- Can pass multiple CQEs
- CQs can be waited on (including by BPF)
- Extra way of communication: posting to a CQ

Example:

Each BPF request has its own CQ. It keeps a number of operations in-flight and posts to the main CQ when it's done with the job.



What about poking into the normal userspace memory?
BPF subsystem already has an answer: **sleepable BPF programs**

It does what it sounds like, allows BPF programs to sleep.

- reading userspace memory is already there
- writing is trivial to add
- a big deal for io_uring as submission might need to sleep
- `bpf_copy_[from,to]_user()` + io_uring performance is yet to be measured

There are also BPF maps / arrays and other infrastructure provided by BPF

- not everything is supported with sleepable programs, may get lifted (if not already)

Overhead

There can be *O(N)* BPF requests, important to keep overhead low

A lot of work has been done! Highlights:

- persistent submission state, request caching
- infrastructure around task_work and execution batching
- task_struct referencing and other overhead amortisation
- removing request refcounting
- completion batching
- native io-wq workers (planned to use)
- upcoming IOSQE_CQE_SKIP_SUCCESS
- just cutting the number of instructions required per request …

QD1 should be in a good shape as well …

… apart from syscalling and __do_sys_io_uring_enter

API: program registration

API is not set in stone yet, can and will change

```
enum {
    ...
    IORING_REGISTER_BPF,
    IORING_UNREGISTER_BPF,
};

int bpf_prog_fds[NR_PROGS] = {...};
// BPF registration can be made optional
ret = __sys_io_uring_register(ring->ring_fd, IORING_REGISTER_BPF, bpf_prog_fds, NR_PROGS);

// unregister programs, inflicts full quiesce
ret = __sys_io_uring_register(ring->ring_fd, IORING_UNREGISTER_BPF, 0, 0);
// or cleaned up automatically on ring exit
```

API: BPF request

```
enum {
    ...
    IORING_OP_BPF,
};

struct io_uring_sqe *sqe = ...;
memset(sqe, 0, sizeof(sqe));
sqe->opcode = IORING_OP_BPF;
sqe->off = bpf_program_idx;
// generic, for all request types
sqe->user_data = (u64)data_ptr; // returned back in CQE. Also, BPF has access to its user_data
sqe->cq_idx = completion_queue_idx; // CQ index to post CQE to
sqe->flags = sqe_flags; // combination of IOSQE_*, as usual
```

API: BPF definitions

```
enum { // Return values for io_uring BPF programs
    IORING_BPF_OK = 0, // complete request
    IORING_BPF_WAIT,   // wait on CQ for completions
};

struct io_uring_bpf_ctx { // BPF io_uring context
    __u64   user_data;    // sqe->user_data specified at submission
    __u32   wait_nr;      // number of requests to wait for
    __u32   wait_idx;     // CQ index to wait on
};

// Returns the number of submitted requests or a negative error if failed.
long (*bpf_io_uring_submit)(void *ctx, void *sqe, __u32 size);
// Returns 0 on success, -ENOMEM if the CQE has been dropped.
long (*bpf_io_uring_emit_cqe)(void *ctx, __u32 cq_idx, __u64 user_data, __s32 res, __u32 cflags);
// Returns 0 on success, -ENOENT if there are no CQEs in the CQ.
long (*bpf_io_uring_reap_cqe)(void *ctx, __u32 cq_idx, struct io_uring_cqe *cqe, __u32 size);
```

API: libbpf example

```
SEC("iouring") // io_uring BPF program
int bpf_program_name(struct io_uring_bpf_ctx *ctx) {
    struct io_uring_cqe cqe;
    ret = bpf_io_uring_reap_cqe(ctx, cq_idx, &cqe, sizeof(cqe));

    struct io_uring_sqe sqe;
    io_uring_prep_nop(&sqe); // helper copy-pasted from liburing
    sqe.user_data = 42;
    ret = bpf_io_uring_submit(ctx, &sqe, sizeof(sqe));

    u64 data, *uptr = (u64 *)ctx->user_data;
    bpf_copy_from_user(&data, sizeof(data), uptr);

    if (exit) return IORING_BPF_OK;
    ctx->wait_idx = cq_idx_to_wait;
    ctx->wait_nr = nr_cqes_to_wait;
    return IORING_BPF_WAIT; // wait for @nr_cqes_to_wait CQEs in @cq_idx_to_wait CQ
}
```

API: ideas?

- make BPF registration optional
- extra data to pass in SQE, e.g. maps or shared memory
- more convenient bpf_copy_[from,to]_user(), e.g. plain pointers
- other synchronisation, e.g. futex
- batched version of bpf_io_uring_submit()
- anything missing?

Testing

Not yet conclusive. Test case:

- Copy a file by 4KB at a time into /dev/zero, buffered and fully cached

Mitigations	Test case	Time (ms)
ON	read(2)/write(2)	1350
ON	io_uring, simple QD=1	1630
any	io_uring + BPF	810
OFF	read(2)/write(2)	550

However, let’s take another CPU:

Mitigations	Test case	Time (ms)
ON	read(2)/write(2)	1320
any	io_uring + BPF	1250

Applicability

Applicability: shouldn’t be of interest if batching is naturally “high enough”.

High queue depth is not always possible and/or desirable.

- batching hurts latency
- may care about ordering, e.g. TCP sockets.
- slow devices and memory/responsiveness restrictions

Use cases to try:

- databases / engines, caching systems
- Intelligent file-file splicing, e.g. based on data
- broadcast / collect
- mentioned that may be of use to QUIC
- explore applicability to FUSE
- ideas are welcome

Next steps

Need to explore more test cases …

… and more “interesting” tests.

Each new case requires some tuning and optimisation.

Upside: also usually benefits non-BPF io_uring.

Have to solve some slight performance regressions from multi-CQ

- good chance extra CQs will only be visible to BPF

TODO: selftests, bpf_link, API changes

Resources

Kernel
https://github.com/isilence/linux.git bpf_v3

Liburing, see <liburing>/examples/bpf/*
https://github.com/isilence/liburing.git bpf_v3

io_uring mailing list
io-uring@vger.kernel.org

io_uring guide
https://kernel.dk/io_uring.pdf

benchmark, <fio>/t/io_uring.c
git://git.kernel.dk/fio