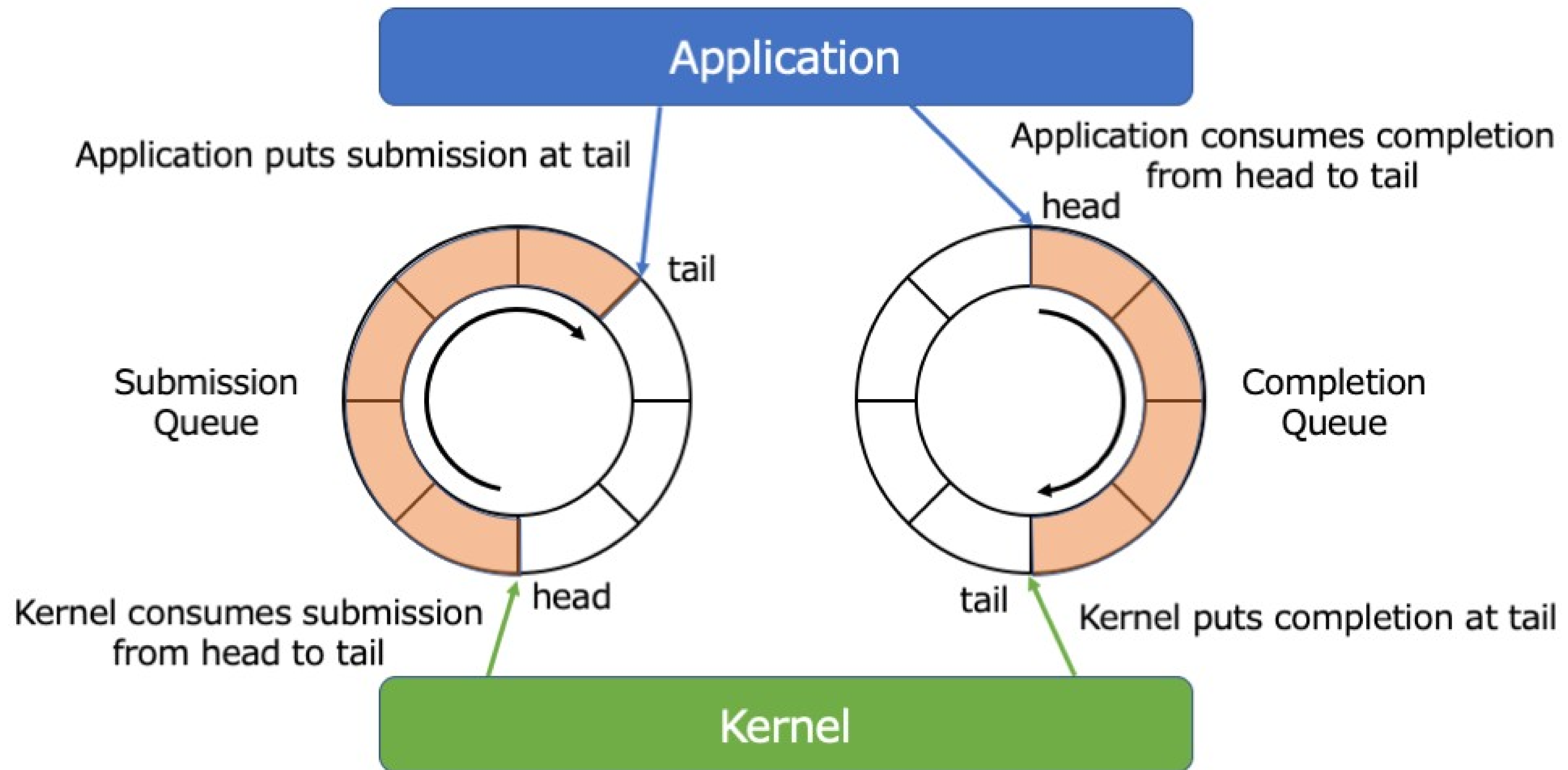
What’s new with io_uring

FASTER IO WITH A CONSISTENT MODEL? YES PLEASE

Jens Axboe
Software Engineer

Meta

Quick primer



Source: https://medium.com/nttlabs/rust-async-with-io-uring-db3fa2642dd4

System calls

`io_uring_setup``(2)`

Sets up an io_uring instance, application then `mmap(2)`'s the SQ and CQ ring memory. Returns a file descriptor, application closes fd when done (or on process exit).

`io_uring_enter``(2)`

Informs the kernel about work to be done, waits for work to be completed, or both.

`io_uring_register``(2)`

Auxiliary functions, like registering file, buffers, setting async worker CPU affinities, etc.

Why?

AIO API is pretty horrible and inefficient. Opinions are subjective, but I think we have pretty universal agreement on that one. libaio is just a useless wrapper.

AIO not widely used because it only supports one niche use case, even 20 years later.

Why not do a proper API that could work in an efficient manner for a wide range of use cases?

Key features

Actually, you know, async!

Zero-copy submissions, no indirections.

Lock-less communication.

Extendable.

Easy to use.

Feature rich.

liburing

Easy to use, minimize boilerplate code in app.

Kernel independent. Use any version with any kernel.

Helps hide some of the quirkiness that inevitably ends up in APIs that can never get broken.

More future proof for kernel additions and changes.

Example

```c
#include <liburing.h>

struct io_uring_sqe *sqe;
struct io_uring_cqe *cqe;
struct io_uring ring;

io_uring_queue_init(8, &ring, 0);

/* get request slot, prepare request */
sqe = io_uring_get_sqe(&ring);
io_uring_prep_read(sqe, fd, buf, sizeof(buf), offset);

/* submit request(s) to the kernel */
io_uring_submit(&ring);

/* wait for a completion */
io_uring_wait_cqe(&ring, &cqe);
if (cqe->res < 0)
    printf("Read error: %s\n", strerror(-cqe->res);
else
    printf("Read %d from file\n", cqe->res);

/* mark cqe as seen, increments CQ ring head */
io_uring_cqe_seen(&ring, cqe);
```

Lifetimes

SQE lifetime is from get → submit. Hence SQ ring size only limits batch size, not in-flight IO count.

Requests passing in data structs need to ensure validity only until submit is done, not until completion.

Store `sqe→user_data` and retrieve it as `cqe→user_data`, tying a completion to a specific submission.

CQE wait functions tell you nothing about the result of a request, only if waiting was successful or not.

What’s new

Native workers

Originally io-wq used kernel threads that assumed the identity of the original task when needed. This was risky.

Available in 5.12, io-wq is based on io-threads. These are normal task threads, except they never leave the kernel and they don’t take signals.

Source: https://blog.cloudflare.com/missing-manuals-io_uring-worker-pool/

Native workers

Native io-threads eliminate security concerns with io-wq offload, for the requests that need that.

It also makes offload a bit more efficient, as no identify switching is needed (files_struct, mm, creds, etc).

It also fixes cases that didn't previously work, like **`/proc/self`**, reading from signalfd, etc.

Enables **`IORING_SETUP_SQPOLL`** to work with any file type, or any request in general, and without privilege requirements.

Available in 5.12, identified by **`IORING_FEAT_NATIVE_WORKERS`.**

io-wq poll

io-wq used to just block when offloaded.

With hybrid mode, even io-wq can take advantage of the internal poll support.

Not user visible, just faster and more efficient.

Available since 5.16.

`TIF_NOTIFY_SIGNAL`

io_uring relies on a signal-like mechanism for interrupting in-kernel waits just like normal signals.

Signals and threads are not happy partners.

Support for all architectures was added for **`TIF_NOTIFY_SIGNAL`**, which decouples the signal interruption from the shared **`struct`** `sighand_struct`.

Was miserable work, but yielded very nice performance improvements. Available since 5.10.

Direct descriptors

Normal file descriptors can be slow, particularly for threaded applications. **`fget`** / **`fput`** per system call is an atomic inc and dec in shared data. Not unusual to see 3-5% overhead.

Direct descriptors exist only within the ring itself, but can be used for any request within that ring.

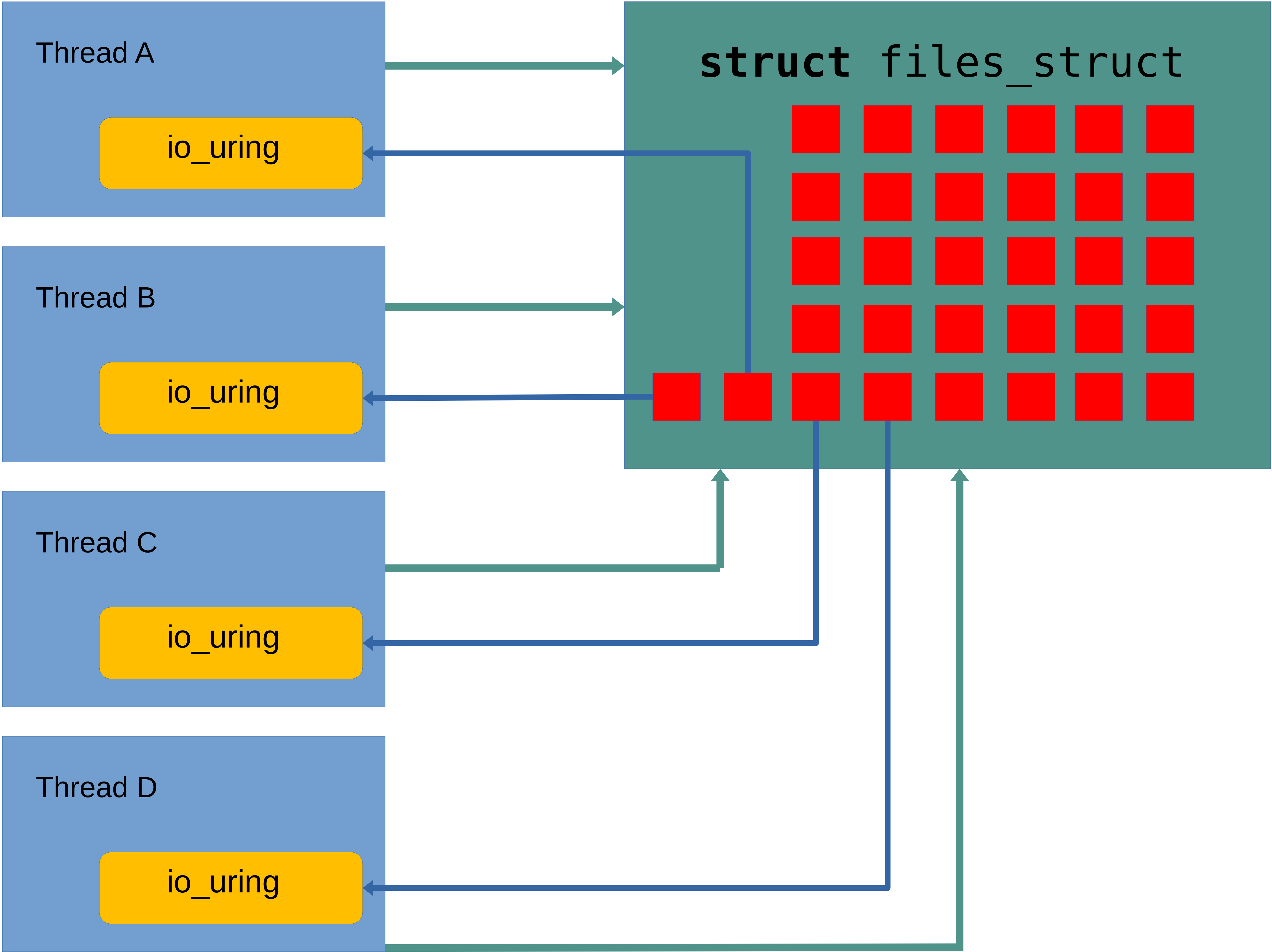
Enables use of links for

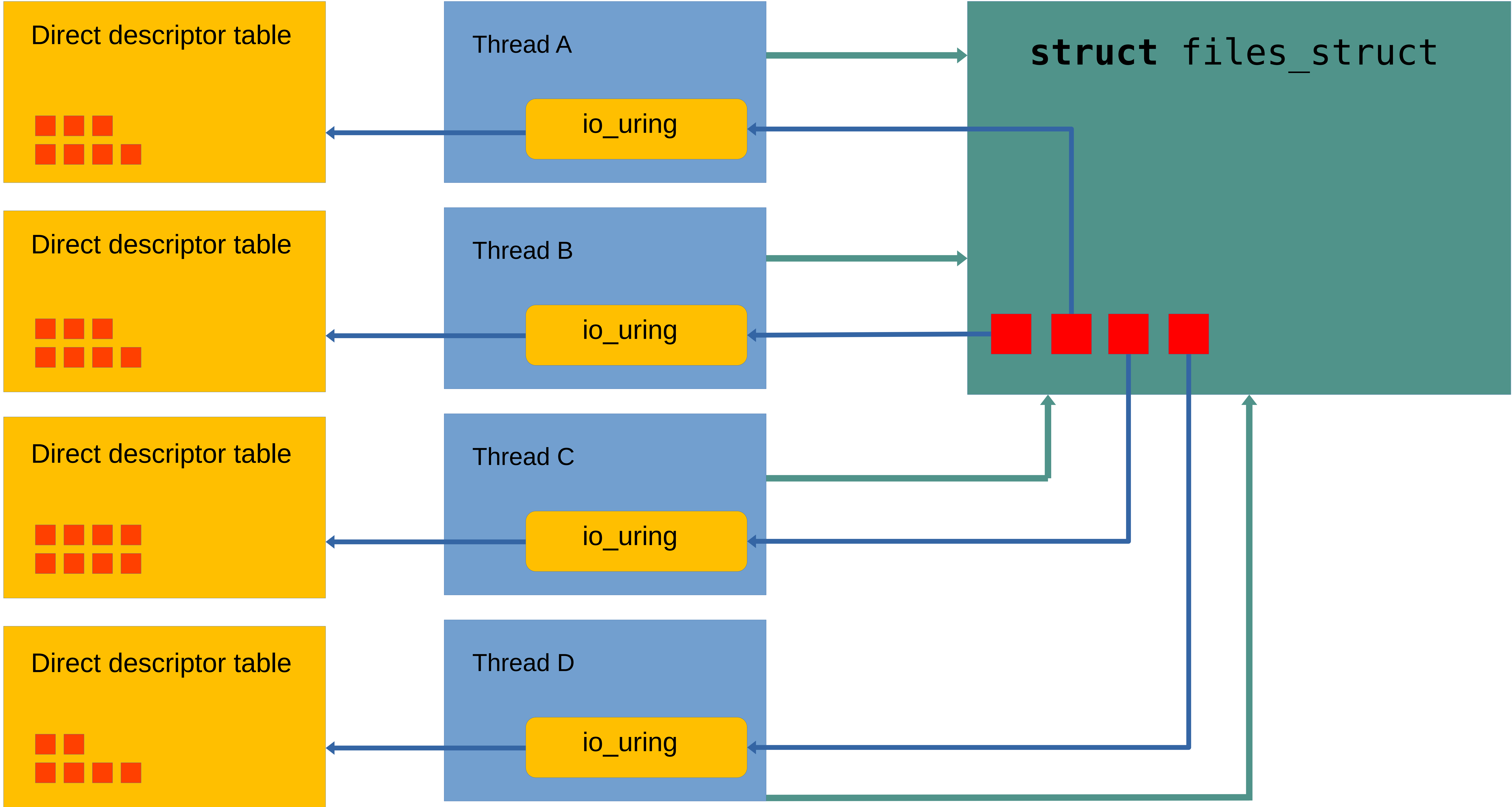
[*open file X***]→[***read file X***]→[***close file X***]**

operations since the descriptor can be known in advance.

Also referred to as fixed or registered files.

[1] *https://lwn.net/Articles/863071/*


Thread A
io_uring
struct files_struct
Thread B
io_uring
Thread C
io_uring
Thread D
io_uring


Direct descriptor table
Thread A
io_uring
struct files_struct
Direct descriptor table
Thread B
io_uring
Direct descriptor table
Thread C
io_uring
Direct descriptor table
Thread D
io_uring

How to use direct descriptors

```
io_uring_register_files(ring, files, nfiles);
```

Files is array of valid descriptors, or **-1**

```
io_uring_register_files_sparse(ring, nfiles);
```

Register existing normal file descriptor, use registered index:

```
sqe→flags |= IOSQE_FIXED_FILE;
sqe→fd = fixed_file_index;
```

Or instantiate directly with the io_uring socket, accept, openat/openat2. Open into existing slot to close + replace.

```
io_uring_prep_openat_direct();
io_uring_prep_socket_direct();
io_uring_prep_accept_direct();
```

Managed direct descriptors

Prior to 5.19, applications had to manage their own direct descriptor space.

5.19 enables io_uring to manage it, like the normal file descriptor table.

Use **`IORING_FILE_INDEX_ALLOC`** as the index, allocated descriptor value returned in `cqe→res`.

io_uring has prep helpers for the direct cases too, making this easy.

Registered ring fd

Why not get rid of the **fget** / **fput** for the **io_uring_enter**() system call as well?

Not usable on a shared ring. But don’t do those in general!

```
int io_uring_register_ring_fd(struct io_uring *ring);
int io_uring_unregister_ring_fd(struct io_uring *ring);
```

Provided buffers

Rather than pass in an IO buffer for a `recv()` or `recvmsg()` type operation, provide a buffer pool upfront. When the file or socket is ready to transfer data, pick a buffer and tell the application about it in the **CQE**.

Enables efficient use of memory with a completion based IO model.

How to use provided buffers

```
io_uring_register_buffers(ring, vecs,
                          nvecs);
io_uring_prep_provide_buffers();
```

Good use case for **IOSQE_CQE_SKIP_SUCCESS**

```
sqe→flags |= IOSQE_BUFFER_SELECT;
sqe→buf_group = buffer_group;
```

CQE has **IORING_CQE_F_BUFFER** set, `cqe→flags` contains buffer ID of selected buffer.

Ring provided buffers

Even with **`IORING_CQE_SKIP_SUCCESS`**, the existing provided buffers added some overhead to the request. Batching helps, but can be hard and/or expensive to do.

App allocates memory for the buffer ring, kernel maps it.

Cannot share a buffer group ID with *classic* provided buffers.

Available in 5.19.

```
io_uring_register_buf_ring(ring, reg, flags);
io_uring_buf_ring_add(br, buf, size, id, off);
io_uring_buf_ring_advance(br, count);
io_uring_buf_ring_cq_advance(ring, br, count);
```

Results

To gauge overhead of the existing scheme and evaluate the mapped ring approach, a simple NOP benchmark was written. It uses a ring of 128 entries, and submits/completes 32 at the time. 'Replenish' is how many buffers are *provided* back at the time after they have been consumed:

Test	Replenish	NOPs/sec
No *provided* buffers	NA	~30M
Provided buffers	32	~16M
Provided buffers	1	~10M
Ring buffers	32	~27M
Ring buffers	1	~27M

The ring mapped buffers perform almost as well as not using *provided* buffers at all, and they don't care if you *provided* 1 or more back at the same time. This means application can just replenish as they go, rather than need to batch and compact, further reducing overhead in the application. The NOP benchmark above doesn't need to do any compaction, so that overhead isn't even reflected in the above test.

:

Support for app driven issue and poll

Normal flow of request is attempt to issue, arm poll if data / space not available.

`IORING_RECVSEND_POLL_FIRST`

Don’t attempt issue first, go straight to poll.

`IORING_CQE_F_SOCK_NONEMPTY`

Previous eg **`recv`**`()` returns if there was more data available.

Available in 5.19.

`->uring_cmd()`

Communicate through the entire stack, file type specific requests (aka async ioctls).

`IORING_SETUP_SQE128, IORING_SETUP_CQE32`

NVMe passthrough support, both IO and admin queues.

Many potential use cases. Return bytes left in socket after receive?

Trivial **`setsockopt`**() / **`getsockopt`**() for direct descriptors.

Available in 5.19.

[1] *https://lwn.net/Articles/844875/*

Cooperative completion scheduling

io_uring uses `task_work` for retries or posting completions. This uses IPI based signaling with **`TIF_NOTIFY_SIGNAL`**, which forces a task preemption if running in user space. This can cause unnecessary re-schedules. Setup flags:

`IORING_SETUP_COOP_TASKRUN`
`IORING_SETUP_TASKRUN_FLAG`

If set, task_work runs happen when the task transitions anyway. liburing supports it for peek, too.

`IORING_SQ_TASKRUN`

Flag set in the SQ ring `flags` if events are available.

Available in 5.19.

Cancelations

Support for cancelations beyond just matching `user_data` in 5.19.

```
IORING_ASYNC_CANCEL_ALL
IORING_ASYNC_CANCEL_FD
IORING_ASYNC_CANCEL_ANY
```

FD matches using the file descriptor of the original request, rather than `user_data`. **ALL** keeps canceling matching requests. **ANY** matches any request.

```
io_uring_prep_cancel();
io_uring_prep_cancel_fd();
```

Multishot accept

Usually a io_uring request will post a single completion. Some can post more, and will inform the app of more coming by setting **`IORING_CQE_F_MORE`** in cqe→flags. Multi-shot poll is one example.

5.19 supports this for accept as well. Application can post a single accept request and get a completion event every time a connection request comes in. One accept request to rule them all.

```
io_uring_prep_multishot_accept();
io_uring_prep_multishot_accept_direct();
```

OP_MSG_RING

Supports sending a “message” from one ring to another – one 64-bit value and one 32-bit value.

```
io_uring_prep_msg_ring(…, fd, len, data, …);
```

Useful for passing eg a work item pointer between threads that each have their own ring.

Direct descriptor passing a future potential use case.

Available in 5.18

EXT_ARG

Timeouts used to be done by posting a timeout command. liburing hid this.

Unhandy for split submit+complete threads.

`IORING_ENTER_EXT_ARG`

Passes in a struct with signal and timeout information.

Handled internally in liburing, but worth knowing about because of the previous implied submission.

Generic optimizations

Request (and other structs) memory recycling

Request reference counting

Request completion batching, inline completions

`task_work` optimizations

Locking optimizations (split and IRQ less)

`LOOKUP_CACHED` support for opening files

Pass batching information all the way down the stack

Many many more optimizations. Cycle counting and cacheline layout work is not a forgotten art here.

liburing 2.2

Release coming shortly, synced with 5.19.

Bug fixes, optimizations, and helpers for all the new features.

2.1 had 8 man pages, 2.2 has **80**. Almost all of liburing is documented at this point. ~5200 new lines of man pages was added.

~7000 added lines of regression tests.

Should you upgrade? Yes!

`git://git.kernel.dk/liburing`

Cross platform

Microsoft introduced “I/O Rings” [1] with Windows 11, which DirectStorage is built on top of.

Eerily similar to io_uring, and even later additions mostly mimic io_uring functionality.

Still fairly simplistic and limited in functionality.

Will make cross platform applications feasible [2].

Check out Yarden Shafir’s blog posts and P99 talk for more details.

FreeBSD version in the works?

[1] https://docs.microsoft.com/en-us/windows/win32/api/ioringapi/
[2] https://github.com/CarterLi/libwinring

Upcoming features

Support for true async buffered writes. Targeting 5.20 with XFS support, btrfs in the works.

Further networking features to improve efficiency, and improvements in this area in general. NAPI, zc, etc.

Incrementally consumed provided buffers.

Level triggered poll support [1].

Not io_uring specific, but support for **`ITER_UBUF`**.

Faster io-wq offload.

Code split. Moving `fs/io_uring.c` into `io_uring/` and splitting it into related opcodes and topical files [2].

[1] *OK so I ended up doing this while writing slides...*

[2] *`https://git.kernel.dk/cgit/linux-block/log/?h=for-5.20/io_uring`*

Final words

Completion based is a new IO model on networking for Linux and related operating systems.

Retrofitting can be harder to do right because of that.

Applications or library adaptions of io_uring that simply switch an **`epoll`**`(7)` (or <insert event library here>) based readiness model to io_uring are trivial, but also woefully uninteresting.

The model unifies IO across all types of files and sockets. Finally!

We’re in it for the long run. Who doesn’t need another decade long project?

Meta