Efficient IO with io_uring

This article is intended to serve as an introduction to the newest Linux IO interface, io_uring, and compare it to the existing offerings. We'll go over the reasons for its existence, inner workings of it, and the user visible interface. The article will not go into details about specific commands and the likes, as that would just be duplicating the information available in the associated man pages. Rather, it will attempt to provide an introduction to io_uring and how it works, with the goal hopefully being that the reader will have gained a deeper understanding of how it all ties together. That said, there will be some overlap between this article and the man pages. It's impossible to provide a description of io_uring without including some of those details.

1.0 Introduction

There are many ways to do file based IO in Linux. The oldest and most basic are the `read(2)` and `write(2)` system calls. These were later augmented with `pread(2)` and `pwrite(2)` versions which allow passing in of an offset, and later still we got `preadv(2)` and `pwritev(2)` which are vector-based versions of the former. Because that still wasn't quite enough, Linux also has `preadv2(2)` and `pwritev2(2)` system calls, which further extend the API to allow modifier flags. The various differences of these system calls aside, they share the common trait that they are synchronous interfaces. This means that the system calls return when the data is ready (or written). For some use cases that is sub-optimal, and an asynchronous interface is desired. POSIX has `aio_read(3)` and `aio_write(3)` to satisfy that need, however the implementation of those is most often lackluster and performance is poor.

Linux does have a native async IO interface, simply dubbed aio. Unfortunately, it suffers from a number of limitations:

- The biggest limitation is undoubtedly that it only supports async IO for `O_DIRECT` (or un-buffered) accesses. Due to the restrictions of `O_DIRECT` (cache bypassing and size/alignment restraints), this makes the native aio interface a no-go for most use cases. For normal (buffered) IO, the interface behaves in a synchronous manner.
- Even if you satisfy all the constraints for IO to be async, it's sometimes not. There are a number of ways that the IO submission can end up blocking - if meta data is required to perform IO, the submission will block waiting for that. For storage devices, there are a fixed number of request slots available. If those slots are currently all in use, submission will block waiting for one to become available. These uncertainties mean that applications that rely on submission always being async are still forced to offload that part.
- The API isn't great. Each IO submission ends up needing to copy 64 + 8 bytes and each completion copies 32 bytes. That's 104 bytes of memory copy, for IO that's supposedly zero copy. Depending on your IO size, this can definitely be noticeable. The exposed completion event ring buffer mostly gets in the way by making completions slower, and is hard (impossible?) to use correctly from an application. IO always requires at least two system calls (submit + wait-for-completion), which in these post spectre/meltdown days is a serious slowdown.

Over the years there has been various efforts at lifting the first limitation mentioned (I also made a stab at it back in 2010), but nothing succeeded. In terms of efficiency, arrival of devices that are capable of both sub-10usec latencies and very high IOPS, the interface is truly starting to show its age. Slow and non-deterministic submission latencies are very much an issue for these types of devices, as is the lack of performance that you can extract out of a single core. On top of that, because of the aforementioned limitations, it's safe to say that native Linux aio doesn't have a lot of use cases. It's been relegated to a niche corner of applications, with all the issues that come with that (long term undiscovered bugs, etc).

Furthermore, the fact that "normal" applications have no use for aio means that Linux is still lacking an interface that provides the features that they desire. There is absolutely no reason that applications or libraries continue to need to create private IO offload thread pools to get decent async IO, especially when that can be done more efficiently in the kernel.

2.0 Improving the status quo

Initial efforts were focused on improving the aio interface, and work progressed fairly far down that path before being abandoned. There are multiple reasons why this initial direction was chosen:

- If you can extend and improve an existing interface, that's preferable to providing a new one. Adoption of new interfaces take time, and getting new interfaces reviewed and approved is a potentially long and arduous task.
- It's a lot less work in general. As a developer, you're always looking to accomplish the most with the least amount of work. Extending an existing interface gives you many advantages in terms of existing test infrastructure.

The existing aio interface is comprised of three main system calls: a system call to setup an aio context (`io_setup(2)`), one to submit IO (`io_submit(2)`), and one to reap or wait for completions of IO (`io_getevents(2)`). Since a change in behavior was required for multiple of these system calls, we needed to add new system calls to pass in this information. This created both multiple entry points to the same code, as well as shortcuts in other places. The end result wasn't very pretty in terms of code complexity and maintainability, and it only ended up fixing one of the highlighted deficiencies from the previous section. On top of that, it actually made one of them worse, since now the API was even more complicated to understand and use.

While it's always hard to abandon a line of work to start from scratch, it was clear that we needed something new entirely. Something that would allow us to deliver on all points. We needed it to be performant and scalable, while still making it easy to use and having the features that existing interfaces were lacking.

3.0 New interface design goals

While starting from scratch was not an easy decision to make, it did allow us full artistic freedom in coming up with something new. In rough ascending order of importance, the main design goals were:

- Easy to use, hard to misuse. Any user/application visible interface should have this as a main goal. The interface should be easy to understand and intuitive to use.
- Extendable. While my background is mostly storage related, I wanted the interface to be usable for more than just block oriented IO. That meant networking and non-block storage interfaces that may be coming down the line. If you're creating a brand new interface, it should be (or at least attempt to be) future proof in some shape or form.
- Feature rich. Linux aio caters to a subset (of a subset) of applications. I did not want to create yet another interface that only covered some of what applications need, or that required applications to reinvent the same functionality over and over again (like IO thread pools).
- Efficiency. While storage IO is mostly still block based and hence at least 512b or 4kb in size, efficiency at those sizes is still critical for certain applications. Additionally, some requests may not even be carrying a data payload. It was important that the new interface was efficient in terms of per-request overhead.

- Scalability. While efficiency and low latencies are important, it's also critical to provide the best performance possible at the peak end. For storage in particular, we've worked very hard to deliver a scalable infrastructure. A new interface should allow us to expose that scalability all the way back to applications.

Some of the above goals may seem mutually exclusive. Interfaces that are efficient and scalable are often hard to use, and more importantly, hard to use correctly. Both feature rich and efficient can also be hard to achieve. Nevertheless, these were the goals we set out with.

4.0 Enter io_uring

Despite the ranked list of design goals, the initial design was centered around efficiency. Efficiency isn't something that can be an afterthought, it has to be designed in from the start - you can't wring it out of something later on once the interface is fixed. I knew I didn't want any memory copies for either submissions or completion events, and no memory in-directions either. At the end of the previous aio based design, both efficiency and scalability were visibly harmed by the multiple separate copies that aio had to do to handle both sides of the IO.

As copies aren't desirable, it's clear that the kernel and the application have to graciously share the structures defining the IO itself, and the completion event. If you're taking the idea of sharing that far, it was a natural extension to have the coordination of shared data also reside in memory shared between the application and the kernel. Once you've made that leap, it also becomes clear that synchronization between the two has to be managed somehow. An application can't share locking with the kernel without invoking system calls, and a system call would surely reduce the rate at which we communicate with the kernel. This was at odds with the efficiency goal. One data structure that would satisfy our needs would be a single producer and single consumer ring buffer. With a shared ring buffer, we could eliminate the need to have shared locking between the application and the kernel, getting away with some clever use of memory ordering and barriers instead.

There are two fundamental operations associated with an async interface: the act of submitting a request, and the event that is associated with the completion of said request. For submitting IO, the application is the producer and the kernel is the consumer. The opposite is true for completions - here the kernel produces completion events and the application consumes them. Hence, we need a pair of rings to provide an effective communication channel between an application and the kernel. That pair of rings is at the core of the new interface, io_uring. They are suitably named submission queue (SQ), and completion queue (CQ), and form the foundation of the new interface.

4.1 DATA STRUCTURES

With the communication foundation in place, it was time to look at defining the data structures that would be used to describe the request and completion event. The completion side is straight forward. It needs to carry information pertaining to the result of the operation, as well as some way to link that completion back to the request it originated from. For io_uring, the layout chosen is as follows:

```
struct io_uring_cqe {
    __u64    user_data;
    __s32    res;
    __u32    flags;
};
```

The io_uring name should be recognizable by now, and the _cqe postfix refers to a Completion Queue Event. For the rest of this article, commonly referred to as just a cqe. The cqe contains a ***user_data*** field. This field is carried from the

initial request submission, and can contain any information that the the application needs to identify said request. One common use case is to have it be the pointer of the original request. The kernel will not touch this field, it's simply carried straight from submission to completion event. ***res*** holds the result of the request. Think of it like the return value from a system call. For a normal read/write operation, this will be like the return value from *read(2)* or *write(2)*. For a successful operation, it will contain the number of bytes transferred. If a failure occurred, it will contain the negative error value. For example, if an I/O error occurred, ***res*** will contain **-EIO**. Lastly, the ***flags*** member can carry meta data related to this operation. As of now, this field is unused.

Definition of a request type is more complicated. Not only does it need to describe a lot more information than a completion event, it was also a design goal for io_uring to be extendable for future request types. What we came up with is as follows:

```
struct io_uring_sqe {
    __u8    opcode;
    __u8    flags;
    __u16    ioprio;
    __s32    fd;
    __u64    off;
    __u64    addr;
    __u32    len;
    union {
        __kernel_rwf_t    rw_flags;
        __u32        fsync_flags;
        __u16        poll_events;
    };
    __u64    user_data;
    union {
        __u16    buf_index;
        __u64    __pad2[3];
    };
};
```

Akin to the completion event, the submission side structure is dubbed the Submission Queue Entry, or sqe for short. It contains an ***opcode*** field that describes the operation code (or op-code for short) of this particular request. One such op-code is **IORING_OP_READV**, which is a vectored read. ***flags*** contains modifier flags that are common across command types. We'll get into this a bit later in the advanced use case section. ***ioprio*** is the priority of this request. For normal read/writes, this follows the definition as outlined for the *ioprio_set(2)* system call. ***fd*** is the file descriptor associated with the request, and ***off*** holds the offset at which the operation should take place. ***addr*** contains the address at which the operation should perform IO, if the op-code describes an operation that transfers data. If the operation is a vectored read/write of some sort, this will be a pointer to an struct iovec array, as used by *preadv(2)*, for example. For a non-vectored IO transfer, ***addr*** must contain the address directly. This carries into ***len***, which is either a byte count for a non-vectored IO transfer, or a number of vectors described by ***addr*** for a vectored IO transfer.

Next follows a union of flags that are specific to the op-code. For example, for the mentioned vectored read (**IORING_OP_READV**), the flags follow those described for the *preadv2(2)* system call. ***user_data*** is common across op-codes, and is untouched by the kernel. It's simply copied to the completion event, cqe, when a completion event is posted for this request. ***buf_index*** will be described in the advanced use cases section. Lastly, there's some padding at the end of the structure. This serves the purpose of ensuring that the sqe is aligned nicely in memory at 64 bytes in size, but also for future use cases that may need to contain more data to describe a request. A few use cases for that comes to mind - one would be a key/value store set of commands, another would be for end-to-end data protection where the application passes in a pre-computed checksum for the data it wants to write.

4.2 COMMUNICATION CHANNEL

With the data structures described, let's go into some detail on how the rings work. Even though there is symmetry in the sense that we have a submission and completion side, the indexing is different between the two. Like in the previous section, let's start with less complicated one, the completion ring.

The cqes are organized into an array, with the memory backing the array being visible and modifiable by both the kernel and the application. However, since the cqe's are produced by the kernel, only the kernel is actually modifying the cqe entries. The communication is managed by a ring buffer. Whenever a new event is posted by the kernel to the CQ ring, it updates the tail associated with it. When the application consumes an entry, it updates the head. Hence, if the tail is different than the head, the application knows that it has one or more events available for consumption. The ring counters themselves are free flowing 32-bit integers, and rely on natural wrapping when the number of completed events exceed the capacity of the ring. One advantage of this approach is that we can utilize the full size of the ring without having to manage a "ring is full" flag on the side, which would have complicated the management of the ring. With that, it also follows that the ring must be a power of 2 in size.

To find the index of an event, the application must mask the current tail index with the size mask of the ring. This commonly looks something like the below:

```
    unsigned head;

    head = cqring->head;
    read_barrier();
    if (head != cqring->tail) {
        struct io_uring_cqe *cqe;
         unsigned index;

         index = head & (cqring->mask);
         cqe = &cqring->cqes[index];
         /* process completed cqe here */
         ...

         /* we've now consumed this entry */
       head++;
   }

    cqring->head = head;
    write_barrier();
```

ring→cqes[] is the shared array of io_uring_cqe structures. In the next sections, we'll get into the inner details of how this shared memory (and the io_uring instance itself) is setup and managed, and what the magic read and write barrier calls are doing here.

For the submission side, the roles are reversed. The application is the one updating the tail, and the kernel consumes entries (and updates) the head. One important difference is that while the CQ ring is directly indexing the shared array of cqes, the submission side has an indirection array between them. Hence the submission side ring buffer is an index into this array, which in turn contains the index into the sqes. This might initially seem odd and confusing, but there's some reasoning behind it. Some applications may embed request units inside internal data structures, and this allows them the flexibility to do so while retaining the ability to submit multiple sqes in one operation. That in turns allows for easier conversion of said applications to the io_uring interface.

Adding an sqe for consumption by the kernel is basically the opposite operation of reaping an cqe from the kernel. A typical example would look something like this:

```
struct io_uring_sqe *sqe;
unsigned tail, index;

tail = sqring->tail;
index = tail & (*sqring->ring_mask);
sqe = &sqring->sqes[index];

/* this call fills in the sqe entries for this IO */
init_io(sqe);

/* fill the sqe index into the SQ ring array */
sqring->array[index] = index;
tail++;

write_barrier();
sqring->tail = tail;
write_barrier();
```

As with the CQ ring side, the read and write barriers will be explained later. The above is a simplified example, it assumes that the SQ ring is currently empty, or at least that it has room for one more entry.

As soon as an sqe is consumed by the kernel, the application is free to reuse that sqe entry. This is true even for cases where the kernel isn't completely done with a given sqe yet. If the kernel does need to access it after the entry has been consumed, it will have made a stable copy of it. Why this can happen isn't necessarily important, but it has an important side effect for the application. Normally an application would ask for a ring of a given size, and the assumption may be that this size corresponds directly to how many requests the application can have pending in the kernel. However, since the sqe lifetime is only that of the actual submission of it, it's possible for the application to drive a higher pending request count than the SQ ring size would indicate. The application must take care not to do so, or it could risk overflowing the CQ ring. By default, the CQ ring is twice the size of the SQ ring. This allows the application some amount of flexibility in managing this aspect, but it doesn't completely remove the need to do so. If the application does violate this restriction, it will be tracked as an overflow condition in the CQ ring. More details on that later.

Completion events may arrive in any order, there is no ordering between the request submission and the association completion. The SQ and CQ ring run independently of each other. However, a completion event will always correspond to a given submission request. Hence, a completion event will always be associated with a specific submission request.

5.0 io_uring interface

Just like aio, io_uring has a number of system calls associated with it that define its operation. The first one is a system call to setup an io_uring instance:

```
int io_uring_setup(unsigned entries, struct io_uring_params *params);
```

The application must provide a desired number of entries for this io_uring instance, and a set of parameters associated with it. ***entries*** denotes the number of sqes that will be associated with this io_uring instance. It must be a power of 2, in the range of 1..4096 (both inclusive). The ***params*** structure is both read and written by the kernel, it is defined as follows:

```
struct io_uring_params {
    __u32 sq_entries;
    __u32 cq_entries;
    __u32 flags;
    __u32 sq_thread_cpu;
    __u32 sq_thread_idle;
    __u32 resv[5];
    struct io_sqring_offsets sq_off;
    struct io_cqring_offsets cq_off;
};
```

The ***sq_entries*** will be filled out by the kernel, letting the application know how many sqe entries this ring supports. Likewise for the cqe entries, the ***cq_entries*** member tells the application how big the CQ ring is. Discussion of the rest of this structure is deferred to the advanced use cases section, with the exception of the ***sq_off*** and ***cq_off*** fields as they are necessary to setup the basic communication through the io_uring.

On a successful call to *io_uring_setup(2)*, the kernel will return a file descriptor that is used to refer to this io_uring instance. This is where the ***sq_off*** and ***cq_off*** structures come in handy. Given that the sqe and cqe structures are shared by the kernel and the application, the application needs a way to gain access to this memory. This is done through *mmap(2)*'ing it into the application memory space. The application uses the ***sq_off*** member to figure out the offsets of the various ring members. The io_sqring_offsets structure looks as follows:

```
struct io_sqring_offsets {
    __u32 head;         /* offset of ring head */
    __u32 tail;         /* offset of ring tail */
    __u32 ring_mask;    /* ring mask value */
    __u32 ring_entries; /* entries in ring */
    __u32 flags;        /* ring flags */
    __u32 dropped;      /* number of sqes not submitted */
    __u32 array;        /* sqe index array /
    __u32 resv1;
    __u64 resv2;
};
```

To access this memory, the application must call *mmap(2)* using the io_uring file descriptor and the memory offset associated with the SQ ring. The io_uring API defines the following mmap offsets for use by the application:

```
#define IORING_OFF_SQ_RING          0ULL
#define IORING_OFF_CQ_RING          0x8000000ULL
#define IORING_OFF_SQES             0x10000000ULL
```

where **IORING_OFF_SQ_RING** is used to map the SQ ring into the application memory space, **IORING_OFF_CQ_RING** for the CQ ring ditto, and finally **IORING_OFF_SQES** to map the sqe array. For the CQ ring, the array of cqes is a part of the CQ ring itself. Since the SQ ring is an index of values into the sqe array, the sqe array must be mapped separately by the application.

The application will define its own structure holding these offsets. One example might look like the following:

```
struct app_sq_ring {
    unsigned *head;
    unsigned *tail;
    unsigned *ring_mask;
    unsigned *ring_entries;
    unsigned *flags;
    unsigned *dropped;
    unsigned *array;
};
```

and a typical setup case will thus look like:

```
struct app_sq_ring app_setup_sq_ring(int ring_fd, struct io_uring_params *p)
{
    struct app_sq_ring sqring;
    void *ptr;

    ptr = mmap(NULL, p->sq_off.array + p->sq_entries * sizeof(__u32),
            PROT_READ | PROT_WRITE, MAP_SHARED | MAP_POPULATE,
            ring_fd, IORING_OFF_SQ_RING);

    sring->head = ptr + p->sq_off.head;
    sring->tail = ptr + p->sq_off.tail;
    sring->ring_mask = ptr + p->sq_off.ring_mask;
    sring->ring_entries = ptr + p->sq_off.ring_entries;
    sring->flags = ptr + p->sq_off.flags;
    sring->dropped = ptr + p->sq_off.dropped;
    sring->array = ptr + p->sq_off.array;

    return sring;
}
```

The CQ ring is mapped similarly to this, using `IORING_OFF_CQ_RING` and the offset defined by the io_cqring_offsets `cq_off` member. Finally, the sqe array is mapped using the `IORING_OFF_SQES` offset. Since this is mostly boiler plate code that can be reused between applications, the liburing library interface provides a set of helpers to accomplish the setup and memory mapping in a simple manner. See the io_uring library section for details on that. Once all of this is done, the application is ready to communicate through the io_uring instance.

The application also needs a way to tell the kernel that it has now produced requests for it to consume. This is done through another system call:

```
int io_uring_enter(unsigned int fd, unsigned int to_submit,
          unsigned int min_complete, unsigned int flags,
          sigset_t sig);
```

`fd` refers to the ring file descriptor, as returned by *io_uring_setup(2)*. `to_submit` tells the kernel that there are up to that amount of sqes ready to be consumed and submitted, while `min_complete` asks the kernel to wait for completion of that amount of requests. Having the single call available to both submit and wait for completions means that the

application can both submit and wait for request completions with a single system call. ***flags*** contains flags that modify the behavior of the call. The most important one being:

```
#define IORING_ENTER_GETEVENTS  (1U << 0)
```

If **IORING_ENTER_GETEVENTS** is set in ***flags***, then the kernel will actively wait for ***min_complete*** events to be available. The astute reader might be wondering what we need this flag for, if we have ***min_complete*** as well. There are cases where the distinction is important, which will be covered later. For now, if you wish to wait for completions, **IORING_ENTER_GETEVENTS** must be set.

That essentially covers the basic API of io_uring. *io_uring_setup(2)* will create an io_uring instance of the given size. With that setup, the application can start filling in sqes and submitting them with *io_uring_enter(2)*. Completions can be waited for with the same call, or they can be done separately at a later time. Unless the application wants to wait for completions to come in, it can also just check the cq ring tail for availability of any events. The kernel will modify CQ ring tail directly, hence completions can be consumed by the application without necessarily having to call *io_uring_enter(2)* with **IORING_ENTER_GETEVENTS** set.

For the types of commands available and how to use them, please see the *io_uring_enter(2)* man page.

6.0 Memory ordering

One important aspect of both safe and efficient communication through an io_uring instance is the proper use of memory ordering primitives. Covering memory ordering of various architectures in detail is beyond the scope of this article. If you're happy using the simplified io_uring API exposed through the liburing library, then you can safely ignore this section and skip to the liburing library section instead. If you have an interest in using the raw interface, understanding this section is important.

To keep things simple, we'll reduce it to two simple memory ordering operations. The explanations are somewhat simplified to keep it short.

`read_barrier()`: *Ensure previous writes are visible before doing subsequent memory reads.*

`write_barrier()`: *Order this write after previous writes.*

Depending on the architecture in question, either one or both of these may be no-ops. While using io_uring, that doesn't matter. What matters is that we'll need them on some architectures, and hence the application writer should understand how to do so. A write_barrier() is needed to ensure ordering of writes. Let's say an application wants to fill in an sqe and inform the kernel that one is available for consumption. This is a two stage process - first the various sqe members are filled in and the sqe index is placed in the SQ ring array, and then the SQ ring tail is updated to show the kernel that a new entry is available. Without any ordering implied, it's perfectly legal for the processor to reorder these writes in any order it deems the most optimal. Let's take a look at the following example, with each number indicating a memory operation:

```
1: sqe->opcode = IORING_OP_READV;
2: sqe->fd = fd;
3: sqe->off = 0;
4: sqe->addr = &iovec;
5: sqe->len = 1;
6: sqe->user_data = some_value;
7: sqring->tail = sqring->tail + 1;
```

There's no guarantee that the write 7, which makes the sqe visible to the kernel, will take place as the last write in the sequence. It's critical that all writes prior to write 7 are visible before write 7 is, otherwise the kernel could be seeing a half written sqe. From the application point of view, before notifying the kernel of the new sqe, you will need a write barrier to ensure proper ordering of the writes. Since it doesn't matter in which order the actual sqe stores happen, as long as they are visible before the tail write, we can get by with an ordering primitive after write 6, and before write 7. Hence the sequence then looks like the following:

```
    1: sqe->opcode = IORING_OP_READV;
    2: sqe->fd = fd;
    3: sqe->off = 0;
    4: sqe->addr = &iovec;
    5: sqe->len = 1;
    6: sqe->user_data = some_value;
    write_barrier(); /* ensure previous writes are seen before tail write */
    7: sqring->tail = sqring->tail + 1;
    write_barrier(); /* ensure tail write is seen */
```

The kernel will include a read_barrier() before reading the SQ ring tail, to ensure that the tail write from the application is visible. From the CQ ring side, since the consumer/producer roles are reversed, the application merely needs to issue a read_barrier() before reading the CQ ring tail to ensure it sees any writes made by the kernel.

While the memory ordering types have been condensed to two specific types, the architecture implementation will of course be different depending on what machine the code is being run on. Even if the application is using the io_uring interface directly (and not the liburing helpers), it still needs architecture specific barrier types. The liburing library provides these defines, and it's recommended to use those from the application.

With this basic explanation of memory ordering, and with the helpers that liburing provides to manage them, go back and read the previous examples that referenced read_barrier() and write_barrier(). If they didn't fully make sense before, hopefully they do now.

7.0 liburing library

With the inner details of the io_uring out of the way, you'll now be relieved to learn that there's a simpler way to do much of the above. The liburing library serves two purposes:

- Remove the need for boiler plate code for setup of an io_uring instance.
- Provide a simplified API for basic use cases.

The latter ensures that the application doesn't have to worry about memory barriers at all, or do any ring buffer management on its own. This makes the API much simpler to use and understand, and in fact removes the need to understand all the gritty details of how it works. This article could have been much shorter if we had just focused on providing liburing based examples, but it's often beneficial to at least have some understanding of the inner workings to extract the most performance out of an application. Additionally, liburing is currently focused on reducing boiler plate code and providing basic helpers for standard use case. Some of the more advanced features are not yet available through liburing. However, that doesn't mean you can't mix and match the two. Underneath the covers they both operate on the same structures. Applications are generally encouraged to use the setup helpers from liburing, even if they are using the raw interface.

7.1 LIBURING IO_URING SETUP

Let's start with an example. Instead of calling *io_uring_setup(2)* manually and subsequently doing an *mmap(2)* of the three necessary regions, liburing provides the following basic helper to accomplish the very same task:

```
struct io_uring ring;
io_uring_queue_init(ENTRIES, &ring, 0);
```

The io_uring structure holds the information for both the SQ and CQ ring, and the *io_uring_queue_init(3)* call handles all the setup logic for you. For this particular example, we're passing in 0 for the ***flags*** argument. Once an application is done using an io_uring instance, it simply calls:

```
io_uring_queue_exit(&ring);
```

to tear it down. Similarly to other resources allocated by an application, once the application exits, they are automatically reaped by the kernel. This is also true for any io_uring instances the application may have created.

7.2 LIBURING SUBMISSION AND COMPLETION

One very basic use case is submitting a request and, later on, waiting for it to complete. With the liburing helpers, this looks something like this:

```
struct io_uring_sqe sqe;
struct io_uring_cqe cqe;

/* get an sqe and fill in a READV operation */
sqe = io_uring_get_sqe(&ring);
io_uring_prep_readv(sqe, fd, &iovec, 1, offset);

/* tell the kernel we have an sqe ready for consumption */
io_uring_submit(&ring);

/* wait for the sqe to complete */
io_uring_wait_cqe(&ring, &cqe);

/* read and process cqe event */
app_handle_cqe(cqe);
io_uring_cqe_seen(&ring, cqe);
```

This should be mostly self explanatory. The last call to *io_uring_wait_cqe(3)* will return the completion event for the sqe that we just submitted, provided that you have no other sqes in flight. If you do, the completion event could be for another sqe.

If the application merely wishes to peek at the completion and not wait for an event to become available, *io_uring_peek_cqe(3)* does that. For both use cases, the application must call *io_uring_cqe_seen(3)* once it is done

with this completion event. Repeated calls to *io_uring_peek_cqe(3)* or *io_uring_wait_cqe(3)* will otherwise keep returning the same event. This split is necessary to avoid the kernel potentially overwriting the existing completion even before the application is done with it. *io_uring_cqe_seen(3)* increments the CQ ring head, which enables the kernel to fill in a new event at that same slot.

There are various helpers for filling in an sqe, *io_uring_prep_readv(3)* is just one example. I would encourage applications to always take advantage of the liburing provided helpers to the extent possible.

The liburing library is still in its infancy, and is continually being developed to expand both the supported features and the helpers available.

8.0 Advanced use cases and features

The above examples and uses cases work for various types of IO, be it **O_DIRECT** file based IO, buffered IO, socket IO, and so on. No special care needs to be taken to ensure the proper operation, or async nature, of them. However, io_uring does offer a number of features that the application needs to opt in to. The following sub-sections will describe most of those.

8.1 FIXED FILES AND BUFFERS

Every time a file descriptor is filled into an sqe and submitted to the kernel, the kernel must retrieve a reference to said file. Once IO has completed, the file reference is dropped again. Due to the atomic nature of this file reference, this can be a noticeable slowdown for high IOPS workloads. To alleviate this issue, io_uring offers a way to pre-register a file-set for an io_uring instance. This is done through a third system call:

```
int io_uring_register(unsigned int fd, unsigned int opcode, void *arg,
                      unsigned int nr_args);
```

fd is the io_uring instance ring file descriptor, and ***opcode*** refers to the type of registration that is being done. For registering a file-set, **IORING_REGISTER_FILES** must be used. ***arg*** must then point to an array of file descriptors that the application already has open, and ***nr_args*** must contain the size of the array. Once *io_uring_register(2)* completes successfully for a file-set registration, the application can use these files by assigning the index of the file descriptor in the array (instead of the actual file descriptor) to the sqe→fd field, and marking it as an file-set fd by setting **IOSQE_FIXED_FILE** in the sqe→flags field. The application is free to continue to use non-registered files even when a file-set is registered by setting sqe→fd to the non-registered fd and not setting **IOSQE_FIXED_FILE** in the flags. The registered file-set is automatically freed when the io_uring instance is torn down, or it can be done manually by using **IORING_UNREGISTER_FILES** in the ***opcode*** for *io_uring_register(2)*.

It's also possible to register a set of fixed IO buffers. When **O_DIRECT** is used, the kernel must map the application pages into the kernel before it can do IO to them, and subsequently unmap those same pages when IO is done. This can be a costly operation. If an application reuses IO buffers, then it's possible to do the mapping and unmapping once, instead of per IO operation. To register a fixed set of buffers for IO, *io_uring_register(2)* must be called with an opcode of **IORING_REGISTER_BUFFERS**. ***args*** must then contain an array of struct iovec, which have been filled in with the address and length for each iovec. ***nr_args*** must contain the size of the iovec array. Upon successful registration of the buffers, the application can use the **IORING_OP_READ_FIXED** and **IORING_OP_WRITE_FIXED** to perform IO to and from these buffers. When using these fixed op-codes, sqe→addr must contain an address that is within one of these buffers, and sqe→len must contain the length (in bytes) of the request. The application may register buffers larger than any given IO operation, it's perfectly legal for a fixed read/write to just be a subset of a single fixed buffer.

8.2 POLLED IO

For applications chasing the very lowest of latencies, io_uring offers support for polled IO for files. In this context, polling refers to performing IO without relying on hardware interrupts to signal a completion event. When IO is polled, the application will repeatedly ask the hardware driver for status on a submitted IO request. This is different than non-polled IO, where an application would typically go to sleep waiting for the hardware interrupt as its wakeup source. For very low latency devices, polling can significantly increase the performance. The same is true for very high IOPS applications as well, where high interrupt rates makes a non-polled load have a much higher overhead. The boundary numbers for when polling makes sense, either in terms of latency or overall IOPS rates, vary depending on the application, IO device(s), and capability of the machine.

To utilize IO polling, **`IORING_SETUP_IOPOLL`** must be set in the flags passed in to the *`io_uring_setup(2)`* system call, or to the *`io_uring_queue_init(3)`* liburing library helper. When polling is utilized, the application can no longer check the CQ ring tail for availability of completions, as there will not be an async hardware side completion event that triggers automatically. Instead the application must actively find and reap these events by calling *`io_uring_enter(2)`* with **`IORING_ENTER_GETEVENTS`** set and ***`min_complete`*** set to the desired number of events. It is legal to have **`IORING_ENTER_GETEVENTS`** set and ***`min_complete`*** set to 0. For polled IO, this asks the kernel to simply check for completion events on the driver side and not continually loop doing so.

Only op-codes that makes sense for a polled completion may be used on an io_uring instance that was registered with **`IORING_SETUP_IOPOLL`**. These include any of the read/write commands: **`IORING_OP_READV`**, **`IORING_OP_WRITEV`**, **`IORING_OP_READ_FIXED`**, **`IORING_OP_WRITE_FIXED`**. It's illegal to issue a non-pollable op-code on an io_uring instance that is registered for polling. Doing so will result in an **`-EINVAL`** return from *`io_uring_enter(2)`*. The reason behind this is that the kernel cannot know if a call to *`io_uring_enter(2)`* with **`IORING_ENTER_GETEVENTS`** set can safely sleep waiting for events, or if it should be actively polling for them.

8.3 KERNEL SIDE POLLING

Even though io_uring is generally more efficient in allowing more requests to be both issued and completed through fewer system calls, there are still cases where we can improve the efficiency by further reducing the number of system calls required to perform IO. One such feature is kernel side polling. With that enabled, the application no longer has to call *`io_uring_enter(2)`* to submit IO. When the application updates the SQ ring and fills in a new sqe, the kernel side will automatically notice the new entry (or entries) and submit them. This is done through a kernel thread, specific to that io_uring.

To use this feature, the io_uring instance must be registered with **`IORING_SETUP_SQPOLL`** specific for the io_uring_params ***`flags`*** member, or passed in to *`io_uring_queue_init(3)`*. Additionally, should the application wish to limit this thread to a specific CPU, this can be done by flagging **`IORING_SETUP_SQ_AFF`** as well, and also setting the io_uring_params ***`sq_thread_cpu`*** to the desired CPU.

To avoid wasting too much CPU while the io_uring instance is inactive, the kernel side thread will automatically go to sleep when it has been idle for a while. When that happens, the thread will set **`IORING_SQ_NEED_WAKEUP`** in the SQ ring ***`flags`*** member. When that is set, the application cannot rely on the kernel automatically finding new entries, and it must then call *`io_uring_enter(2)`* with **`IORING_ENTER_SQ_WAKEUP`** set. The application side logic typically looks something like this:

```
    /* fills in new sqe entries */
    add_more_io();
```

```
    /*
     * need to call io_uring_enter() to make the kernel notice the new IO
     * if polled and the thread is now sleeping.
     */
    if ((*sqring->flags) & IORING_SQ_NEED_WAKEUP)
        io_uring_enter(ring_fd, to_submit, to_wait, IORING_ENTER_SQ_WAKEUP);
```

As long as the application keeps driving IO, **`IORING_SQ_NEED_WAKEUP`** will never be set, and we can effectively perform IO without performing a single system call. However, it's important to always keep logic similar to the above in the application, in case the thread does go to sleep. The specific grace period before going idle can be configured by setting the io_uring_params ***`sq_thread_idle`*** member. The value is in milliseconds. If this member isn't set, the kernel defaults to one second of idle time before putting the thread to sleep.

For "normal" IRQ driven IO, completion events can be found by looking at the CQ ring directly in the application. If the io_uring instance is setup with **`IORING_SETUP_IOPOLL`**, then the kernel thread will take care of reaping completions as well. Hence for both cases, unless the application wants to wait for IO to happen, it can simply peek at the CQ ring to find completion events.

9.0 Performance

In the end, io_uring met the design goals that was set out for it. We have a very efficient delivery mechanism between the kernel and the application, in the shape of two distinct rings. While the raw interface takes some care to use correctly in an application, the main complication is really the need for explicit memory ordering primitives. Those are relegated to a few specifics on both the submission and completion side of issuing and handling events, and will generally follow the same pattern across applications. As the liburing interface continues to mature, I expect that most applications will be quite satisfied using the API provided there.

While it's not the intent of this note to go into full details about the achieved performance and scalability of io_uring, this section will briefly touch upon some of the wins observed in this area. For more details, see [1]. Do note that due to further improvements on the block side of the equation, these results are a bit outdated. For example, peak per-core performance with io_uring is now approximately 1700K 4k IOPS, not 1620K, on my test box. Note that these values don't carry a lot of absolute meaning, they are mostly useful in terms of gauging relative improvements. We'll continue finding lower latencies and higher peak performance through using io_uring, now that the communication mechanism between the application and the kernel is no longer the bottleneck.

9.1 RAW PERFORMANCE

There are many ways to look at the raw performance of the interface. Most testing will involve other parts of the kernel as well. One such example are the numbers in the section above, where we measure performance by randomly reading from the block device or file. For peak performance, io_uring helps us get to 1.7M 4k IOPS with polling. aio reaches a performance cliff much lower than that, at 608K. The comparison here isn't quite fair, since aio doesn't support polled IO. If we disable polling, io_uring is able to drive about 1.2M IOPS for the (otherwise) same test case. The limitations of aio is quite clear at that point, with io_uring driving twice the amount of IOPS for the same workload.

io_uring supports a no-op command as well, which is mainly useful for checking the raw throughput of the interface. Depending on the system used, anywhere from 12M messages per second (my laptop) to 20M messages per second (test box used for the other quoted results) have been observed. The actual results vary a lot based on the specific test

case, and are mostly bound by the number of system calls that have to be performed. The raw interface is otherwise memory bound, and with both submission and completion messages being small and linear in memory, the achieved messages per second rate can be very high.

9.2 BUFFERED ASYNC PERFORMANCE

I previously mentioned that an in-kernel buffered aio implementation could be more efficient than one done in user-space. A major reason for that has to do with cached vs un-cached data. When doing buffered IO, the application generally relies heavily on the kernels page cache to get good performance. A userspace application has no way of knowing if the data it is going to ask for next is cached or not. It can query this information, but that requires more system calls and the answer is always going to be racy by nature - what is cached this very instant might not be so a few milliseconds from now. Hence an application with an IO thread pool always has to bounce requests to an async context, resulting in at least two context switches. If the data requested was already in page cache, this causes a dramatic slowdown in performance.

io_uring handles this condition like it would for other resources that potentially could block the application. More importantly, for operations that will not block, the data is served inline. That makes io_uring just as efficient for IO that is already in the page cache as the regular synchronous interfaces. Once the IO submission call returns, the application will already have a completion event in the CQ ring waiting for it and the data will have already been copied.

10.0 Further reading

Given that this is an entirely new interface, we don't have a lot of adoption yet. As of this writing, a kernel with the interface is in the -rc stages. Even with a fairly complete description of the interface, studying programs utilizing io_uring can be advantageous in fully understanding how best to use it.

One example is the *io_uring* engine that ships with fio [2]. It is capable of using all of the described advanced features as well, with the exception of registering a file-set.

Another example is the t/io_uring.c sample benchmark application that also ships with fio. It simply does random reads to a file or device, with configurable settings that explore the entire feature set of the advanced use cases.

The liburing library [3] has a full set of man pages for the system call interface, which are worth a read. It also comes with a few test programs, both unit tests for issues that were found during development, as well as tech demos.

LWN also wrote an excellent article [4] about the earlier stages of io_uring. Note that some changes were made to io_uring after this article was written, hence I'd recommend deferring to this article for cases where there are discrepancies between the two.

11.0 References

[1] https://lore.kernel.org/linux-block/20190116175003.17880-1-axboe@kernel.dk/
[2] git://git.kernel.dk/fio
[3] git://git.kernel.dk/liburing
[4] https://lwn.net/Articles/776703/

Version: 0.2d, 2019-04-18