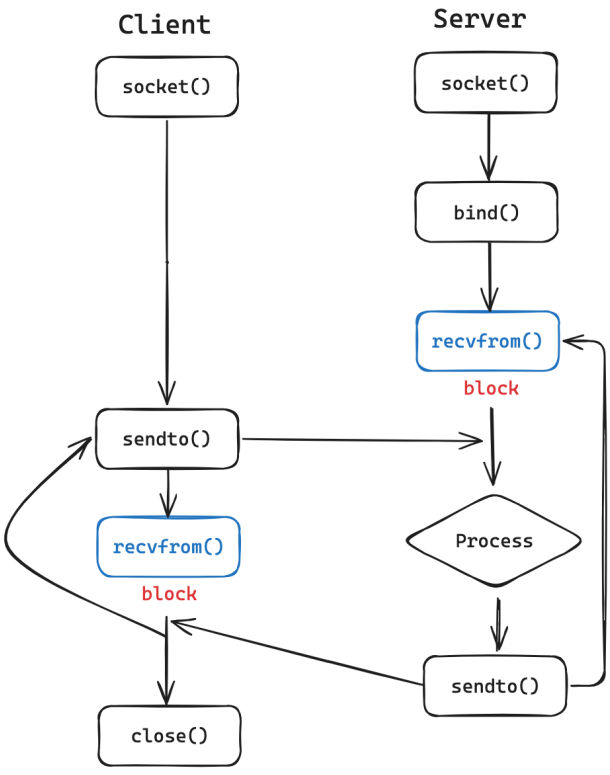
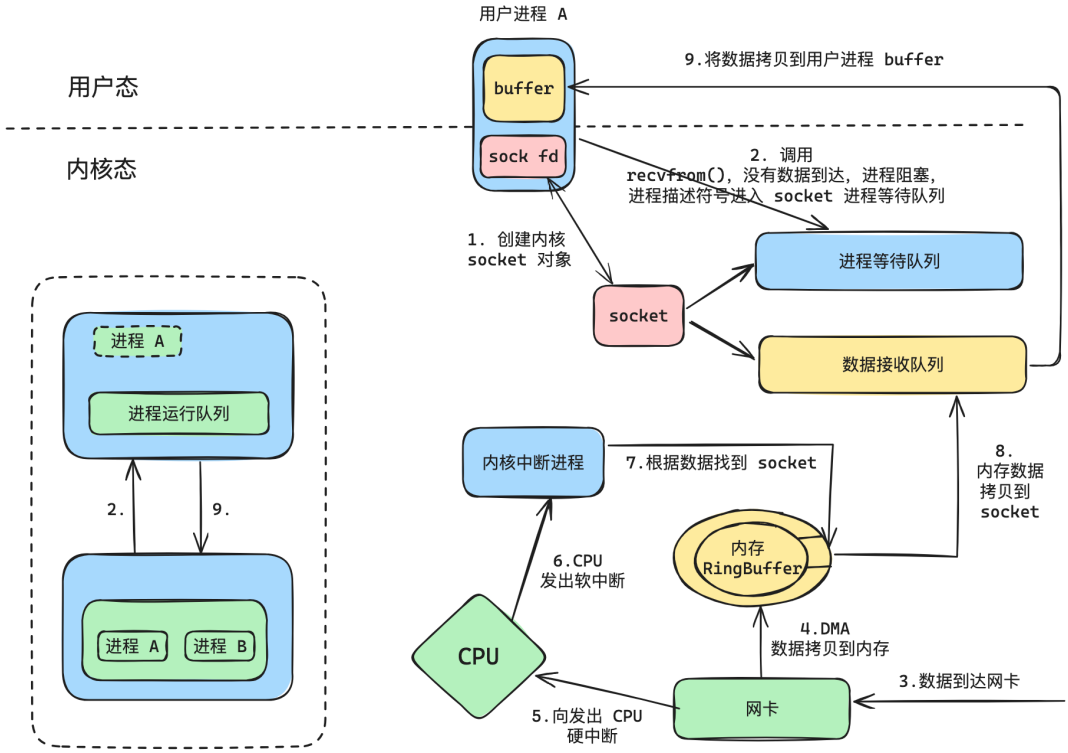
Linux IO_Uring

一、网络 IO

下图是一个 UDP 连接中 socket 的 IO 流程：



可以看到，在每次调用 recvfrom 时，由于网络数据传输的时延缘故，Client/Server 都需要一定程度的等待，如果不做特殊的处理，这里就只能阻塞进程，被迫放弃 CPU，内核调度其他的进程运行。在内核中，IO 更为具体的流程如下：



1. 用户进程通过 socket 系统调用，创建 socket
2. 在进程 socket 配置之后，发起 recvfrom 系统调用陷入内核。此时，由于 socket 的数据接收队列还没有数据到来，所以内核将用户进程阻塞，且将进程描述符添加到进程等待队列，之后调度其他进程执行
3. 在其他进程执行的过程中，数据到达网卡
4. 网卡通过 DMA 将数据拷贝到内存环形缓冲区 RingBuffer
5. 网卡随后就向 CPU 发出硬件中断
6. CPU 向内核中断进程发出软中断
7. 中断进程首先通过 RingBuffer 中的数据（IP + 端口），找到对应的 socket
8. 将 RingBuffer 中的数据拷贝到 socket 中的接收队列，并唤醒用户进程
9. 用户进程进入运行队列，继续从之前被阻塞的 recvfrom 处执行，此时接收队列有数据，将数据拷贝进用户空间的 buffer。

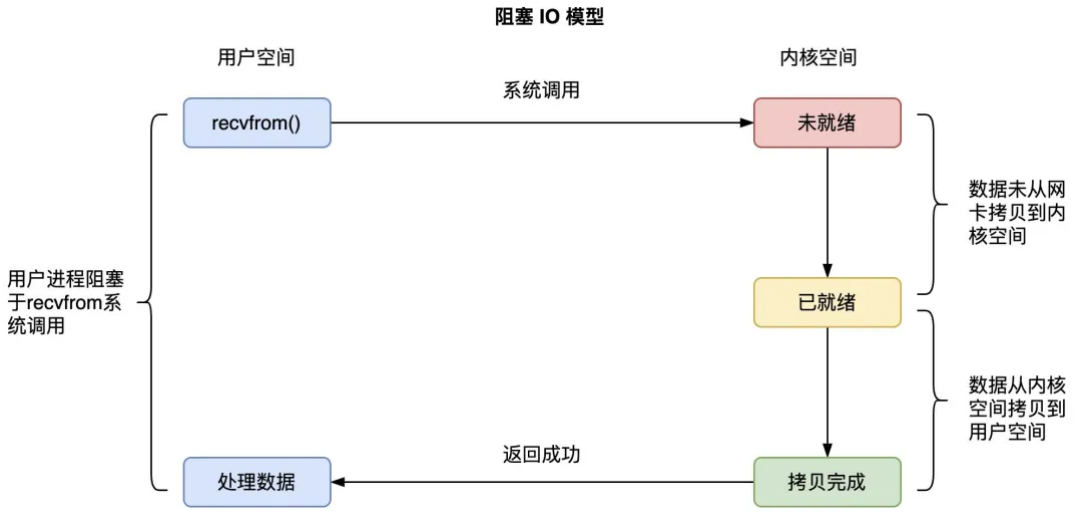
根据上面的流程可知，在 Client/Server 执行一次网络 IO 的过程中，需要等待两次：

- 在调用 recvfrom 陷入内核时，由于 socket 的接收队列没有数据，所以需要等待数据从网卡输入最终进入 socket 的接收队列
- 在 socket 的接收队列准备好数据，原用户进程被唤醒以后，还需要等待将数据从内核中的 socket 接收队列拷贝进用户空间

其中，根据第一次等待需不需要发生，将 IO 分为阻塞 IO 和 非阻塞 IO。

1. 阻塞 IO（BIO）

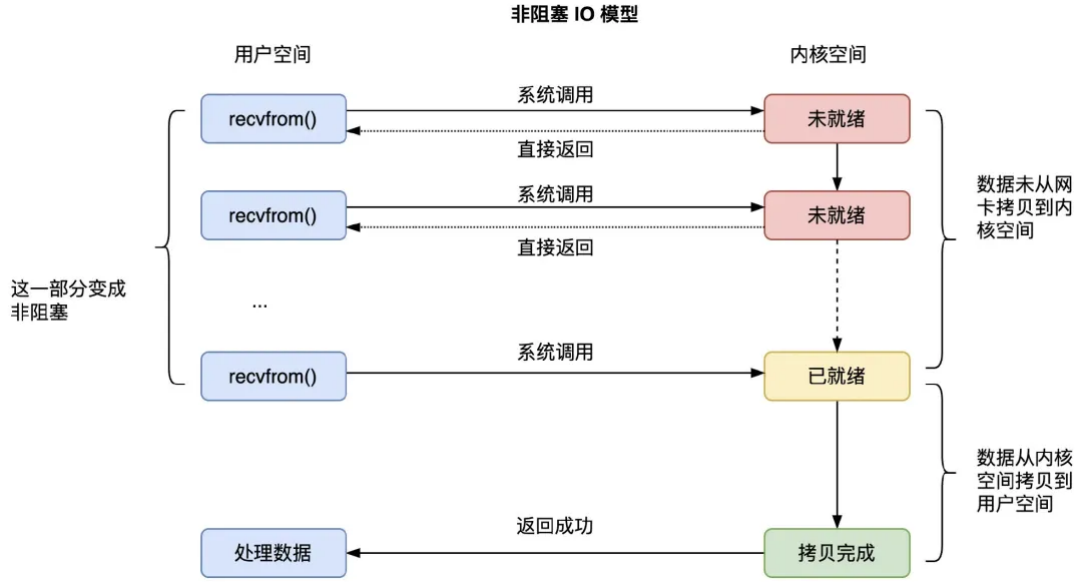
上文中，清晰地展示了阻塞 IO 模型。当进程调用 recvfrom 时，由于此时 socket 中数据为空，进程阻塞。只有当内核将数据准备好，并且拷贝到用户空间后，进程才能继续执行，recvfrom 系统调用返回。



在阻塞 IO 中，在调用 recvfrom 时，会发生阻塞，导致进程切换，数据需要经历从网卡到内核，从内核到用户空间的两次转换。一次进程切换或者或者是数据拷贝均会导致许多不必要的 CPU 开销。

2. 非阻塞 IO（NIO）

非阻塞 IO 即不发生阻塞的网络 IO。Linux 中为 socket 实现了 non_blocking 选项，这样在调用 recvfrom 时，socket 接收队列没有准备好数据时，进程不会阻塞，没有上下文切换，而是直接返回。因此，在使用是，需要轮询数据是否就绪，如下图：



虽然不会发生阻塞，但当数据已经到达内核空间的 socket 的接收队列后，用户进程依然要等待 recvfrom() 函数将数据从内核空间拷贝到用户空间，才会从 recvfrom() 系统调用函数中返回。

小结：

- NIO 解决了 BIO 的进程阻塞，上下文切换的问题，但是需要频繁的系统调用，消耗系统资源
- 在 NIO 和 BIO 中，都需要两次数据的拷贝，一次从网卡到内核空间，一次从内核空间到用户空间。即便对于 NIO 来说，当 socket 中的数据已经就绪，仍然需要等待数据从内核到用户的时间开销，这个过程对数据来说是同步的，也因此，他们都是同步 IO。

3. IO 多路复用

为了解决非阻塞 IO 频繁系统调用的问题，引出 IO 多路复用机制，一次性管理多个连接的 IO。我们不用频繁的系统调用，而可以非阻塞的处理多个连接。不过，在没有连接就绪时，我们仍然需要轮询或者阻塞进程进行等待，并且数据还是需要进行两轮拷贝。由于 IO 多路复用不是本文的重点，这里忽略详细介绍，具体请看 这里。

4. 异步 IO（AIO）

在之前的 IO 模型中，当数据在内核态就绪时，在内核态拷贝用用户态的过程中，仍然会进行短暂时间的阻塞等待。而 AIO 的使用则是为了避免这一点：内核态拷贝数据到用户态交给内核来实现，不由用户线程完成。这样，当数据在内核态就绪时，进程可以直接在用户态读取数据并处理，没有额外的等待时间。

在之前的 Linux 中已经存在异步 IO 框架，AIO，尽管 AIO 已经在数据库领域得到较好应用。但是 AIO 仍然存在一些问题：

- **仅支持direct IO。**在采用AIO的时候，只能使用O_DIRECT，不能借助文件系统缓存来缓存当前的IO请求，还存在size对齐（直接操作磁盘，所有写入内存块数量必须是文件系统块大小的倍数，而且要与内存页大小对齐。）等限制，这直接影响了aio在很多场景的使用。对常规的非数据库应用几乎是无用的；
- **仍然可能被阻塞。语义不完备。**
- **拷贝开销大。**每个IO提交需要拷贝64+8字节，每个IO完成需要拷贝32字节，总共104字节的拷贝。这个拷贝开销是否可以承受，和单次IO大小有关：如果需要发送的IO本身就很大，相较之下，这点消耗可以忽略，而在大量小IO的场景下，这样的拷贝影响比较大。
- **API 不友好。**每一个IO至少需要两次系统调用才能完成（submit和wait-for-completion），需要非常小心地使用完成事件以避免丢事件。
- **系统调用开销大。**也正是因为上一条，io_submit/io_getevents造成了较大的系统调用开销，在存在spectre/meltdown（CPU熔断幽灵漏洞，CVE-2017-5754）的机器上，若要避免漏洞问题，则系统调用性能会大幅下降。所以在存储场景下，高频系统调用的性能影响较大。

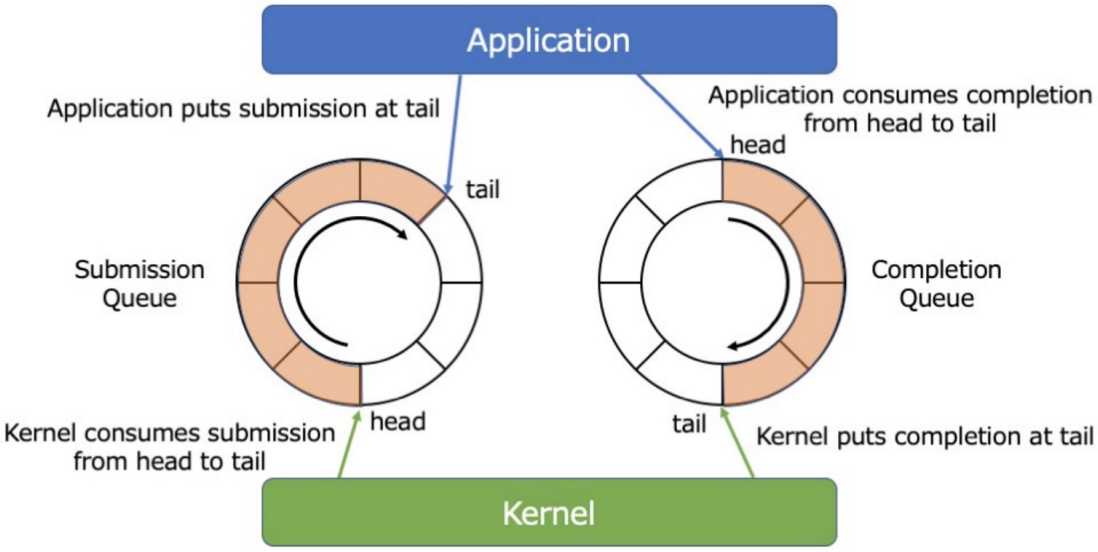
也正是因此，设计出了新的异步 IO 框架，IO_Uring。

二、IO_Uring 框架结构

在上文的介绍中，可以知道同步 IO 影响性能的主要问题在于：

- 数据的多次拷贝（从网卡到内核，从内核到用户）
- 频繁多次的系统调用（利用特殊的机制：IO 多路复用解决）

如果要解决从内核到用户的数据拷贝，最自然的思路就是在内核和用户空间共享一块内存，同时采用一种同步的机制去使用它。然而，不经过系统调用，内核和用户空间是无法共同使用锁机制的，而系统调用又会影响性能。因此 IO_Uring 采用 **单生产者单消费者** 的环形缓冲区，通过**内存排序**和**内存屏障**而不是共享锁来实现 IO 操作。因为对于一个异步接口有两个最基本的操作：提交请求的行为、请求完成的事件。对于 提交 IO 的操作：应用是生产者、内核是消费者，对于 IO 完成的事件：应用是消费者、内核是生产者。因此需要一对 channel 以供应用和内核进行通信：submission queue（SQ）、（Completion Queue）（CQ）。其大致的架构如下：



1. SQ && CQ

这里详细介绍 SQ 和 CQ 的数据结构设计。

SQE

内核作为消费者，在 SQ 中消费 SQE，应用则生成带有新数据的 SQE，提交到 SQ 中：

```
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;
        __u32 sync_range_flags;
        __u32 msg_flags;
    };
    __u64 user_data;
    union {
        __u16 buf_index;
        __u64 __pad2[3];
    };
};
```

- opcode：操作码，例如 IORING_OP_READV，代表向量读
- flags：标志位集合
- ioprio：请求的优先级，对于普通的读写，具体定义可以参照ioprio_set(2)
- fd：与这个请求相关的文件描述符
- off：操作的偏移量
- addr：表示这次IO操作执行的地址，如果操作码 opcode 描述了一个传输数据的操作，这个操作是基于向量的，addr就指向 struct iovec 的数组首地址；如果不是基于向量的，那么addr必须直接包含一个地址，len 这里（非向量场景）就表示这段 buffer 的长度，而向量场景就表示 iovec 的数量

- 接下来的是一个 union，表示一系列针对特定操作码 opcode 的一些 flag。例如，对于上文所提的 IORING_OP_READV ，随后的flags就遵循 preadv2 系统调用
- user_data：对于各个操作码都是通用的，内核不会尝试 touch
- 结构的最后用于内存对齐，对齐到64字节，为了更丰富的特性，未来这个请求结构应该会包含更多的内容

CQE

应用作为消费者，在 CQ 中消费 CQE，内核则在内核空间中生产，其数据结构为：

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

- user_data：来自于这个 CQE 对应的 SQE，内核不会修改它的数据，一个常见的应用就是作为一个指针指向原始请求
- res：请求完成后的结果。对于成功的请求：是传送的字节数，如果失败：是一个负的错误值
- flags：是标志位集合。在这一版文档中，还没有使用，但是在 man page 中已经存在一些用例了

communication channel

已知 SQE 和 CQE 的数据结构，这里介绍 SQ 和 CQ 的使用方式。虽然 SQ 和 CQ 在原理上来说，相对于用户进程是对称的，一个是作为生产者，一个是作为消费者，但是在索引上还是有区别的。这里从稍微简单一点的 CQ 开始：

```
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();
```

对于 CQ 来说，应用是消费者，会从 head 开始取出对应的 CQE。当 `head != cqring->tail` 时，可知 CQ ring 不为空。CQE 在 CQ 内部通过数组的形式存储，通过 CQ 的 head 索引具体的 CQE。

> 为什么这里需要 `index = head & (cqring->mask)` 创建索引?
>
> 这是 CQ 之所以被设计为 ring buffer 的关键。这里的 head 和 tail 都是 32 位的无符号整数，IO_Uring 框架不会维护 head 和 tail 的累加溢出，在超过内存所能表示的范围以后 head 和 tail 会自动从 0 开始继续累加。通过 `index = head & (cqring->mask)`，可以将 index 的范围限制在 mask 以内。例如当 mask 的原码为 1111 时，无论 head 和 tail 的大小为多少，得到的 index 都是落在 0~1111 范围之内的，且按照 head、tail 的环形迭代，index 也是环形的。当然这里的 mask 被限制为 2^n - 1。在 IO_Uring 框架中，mask 设置为 **ring_entries - 1**，这也要求 ring 的大小是 2 的幂。

SQ 的使用稍微有一点不同：

```
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();
```

和 CQ 直接通过共享内存索引 CQE 不同，SQ 在这中间还添加了一层 array，这个 array 的作用是什么呢?

- 应用程序可能需要一次性提交多个 SQE
- 内核可以通过索引数组知道一次提交所需要添加的 SQE

接下来介绍 IO_Uring 的三个主要系统调用。

2. io_uring_setup

应用通过 `io_uring_setup(unsigned entries, struct io_uring_params *params);` 设置一个 io_uring 的实例。entries 表示期望 SQ ring 中 SQE 的数量，要求必须是 2 的 幂（[1..4096]）。params 结构体如下：

```
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;
};
```

params 由内核读写，其中：

- sq_entries 表示 SQ ring 支持的 SQE 数量
- cq_entries 表示 CQ ring 支持的大小
- sq_off 和 cq_off 分别描述了 SQ 和 CQ 的指针在 mmap 中的 offset
- 其他的结构成员涉及到高级用法：
    - flags 用来设置当前整个 io_uring 的标志的，它将决定是否启动 sq_thread，是否采用 iopoll 模式等等
    - sq_thread_cpu、sq_thread_idle 也由用户设置，用来指定 io_sq_thread 内核线程 CPU、idle 时间

对于 io_sqring_offsets，其结构如下：

```
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;
};
```

为了能够访问到对应地址的共享内存，应用还必须使用 mmap 将对应内存 map 到用户空间使用。由于内核 需要和应用映射到同一个地址空间，所以这里的 mmap 偏移量是固定的：

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

一个简单的用户设置 IO_uring 初始化的函数如下（来自 man page）：

```
int app_setup_uring(void) {
    struct io_uring_params p;
    void *sq_ptr, *cq_ptr;

    /* See io_uring_setup(2) for io_uring_params.flags you can set */
    memset(&p, 0, sizeof(p));
    ring_fd = io_uring_setup(QUEUE_DEPTH, &p);
    if (ring_fd < 0) {
        perror("io_uring_setup");
        return 1;
    }

    /*
     * io_uring communication happens via 2 shared kernel-user space ring
     * buffers, which can be jointly mapped with a single mmap() call in
     * kernels >= 5.4.
     */

    int sring_sz = p.sq_off.array + p.sq_entries * sizeof(unsigned);
    int cring_sz = p.cq_off.cqes + p.cq_entries * sizeof(struct io_uring_cqe);

    /* Rather than check for kernel version, the recommended way is to
     * check the features field of the io_uring_params structure, which is a
     * bitmask. If IORING_FEAT_SINGLE_MMAP is set, we can do away with the
     * second mmap() call to map in the completion ring separately.
     */
    if (p.features & IORING_FEAT_SINGLE_MMAP) {
        if (cring_sz > sring_sz)
            sring_sz = cring_sz;
        cring_sz = sring_sz;
    }

    /* Map in the submission and completion queue ring buffers.
     *  Kernels < 5.4 only map in the submission queue, though.
     */
```

```
    sq_ptr = mmap(0, sring_sz, PROT_READ | PROT_WRITE,
                  MAP_SHARED | MAP_POPULATE,
                  ring_fd, IORING_OFF_SQ_RING);
    if (sq_ptr == MAP_FAILED) {
        perror("mmap");
        return 1;
    }
    if (p.features & IORING_FEAT_SINGLE_MMAP) {
        cq_ptr = sq_ptr;
    } else {
        /* Map in the completion queue ring buffer in older kernels separately
*/
        cq_ptr = mmap(0, cring_sz, PROT_READ | PROT_WRITE,
                      MAP_SHARED | MAP_POPULATE,
                      ring_fd, IORING_OFF_CQ_RING);
        if (cq_ptr == MAP_FAILED) {
            perror("mmap");
            return 1;
        }
    }
    /* Save useful fields for later easy reference */
    sring_tail = sq_ptr + p.sq_off.tail;
    sring_mask = sq_ptr + p.sq_off.ring_mask;
    sring_array = sq_ptr + p.sq_off.array;

    /* Map in the submission queue entries array */
    sqes = mmap(0, p.sq_entries * sizeof(struct io_uring_sqe),
                PROT_READ | PROT_WRITE, MAP_SHARED | MAP_POPULATE,
                ring_fd, IORING_OFF_SQES);
    if (sqes == MAP_FAILED) {
        perror("mmap");
        return 1;
    }

    /* Save useful fields for later easy reference */
    cring_head = cq_ptr + p.cq_off.head;
    cring_tail = cq_ptr + p.cq_off.tail;
    cring_mask = cq_ptr + p.cq_off.ring_mask;
    cqes = cq_ptr + p.cq_off.cqes;

    return 0;
}
```

在设置好 IO_Uring 的实例之后，可以对 SQ 和 CQ 进行 IO 的提交和处理，包括为 SQ 提交新的 SQE 请求，处理 来自 CQ 的 CQE 事件。同时，IO_Uring 还提供了一些其他接口使用。

3. io_uring_enter

函数原型为：

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

- fd 表示 IO_Uring 描述符（由 setup 返回）
- to_submit 指定了 SQ 中提交的 I/O 数量
- 依据不同模式：

    - 默认模式，如果指定了 `min_complete`，会等待这个数量的 I/O 事件完成再返回；
    - 如果 io_uring 是 polling 模式，即设置 flags 有 `IORING_ENTER_GETEVENTS`，这个参数表示：如果有事件完成，内核仍然立即返回；如果没有完成事件，内核会 poll，等待指定的次数完成，或者这个进程的时间片用完

4. Advanced features

FIXED FILE AND BUFFER

每次将一个 fd 填入 SQE 并提交给内核时，内核必须获取该文件的引用。一旦 IO 操作完成，内核会释放该文件引用。由于这个文件引用的操作是原子性的，因此在高 IOPS（每秒输入输出操作）工作负载中，这可能会导致显著的性能下降。为了缓解这个问题，io_uring 提供了一种为 io_uring 实例预先注册一组文件的方法。通过这种方式，可以减少每次 IO 操作中对文件引用的获取和释放开销。该系统调用为：

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

- fd：IO_Uring 实例 fd
- opcode：操作码，表示注册的类型，例如`IORING_REGISTER_FILES`，表示注册一个 fd_set
- arg：指向应用已经打开的，想要注册的 fd 数组
- nr_args：fd_set 中 fd 数量

一旦注册成功，应用可以在 SQE 的 fd 中使用其在这个 arg 中的 index，并且需要把 SQE 中的 flags 设置有 `IOSQE_FIXED_FILE`，以表明这个 fd 是一个 fd_set 的索引。这些注册的 fd_set 会在 IO_Uring 销毁的时候释放，或者也可以通过设置 opcode 为 `IORING_UNREGISTER_FILES`调用此接口。同理也可以使用 iovec 注册缓冲区。通过注册文件或用户缓冲区，使内核能长时间持有对该文件在内核内部的数据结构引用， 或创建应用内存的长期映射，以此进一步提高 IO 性能。

POLLED IO

如果应用追求非常低的延时性能，IO_Uring 提供了对文件的 poll IO 机制。在这里，poll 指不再依靠硬件中断信号传递事件完成，而是采用应用不断地向驱动请求提交的 IO 状态来判断事件是否完成。实现方式是在 `io_uring_setup`中设置`IORING_SETUP_IOPOLL`flags。当使用了 poll IO，应用就不能通过检查 CQ 来判断 IO 是否完成，而是需要使用io_uring_enter 系统调用，并且传入 flags 设置 `IORING_ENTER_GETEVENTS`。

KERNEL SIDE POLLING

IO_Uring 已经通过更少的系统调用发起和完成更多请求，但仍有一些情况下可以通过进一步减少执行 IO 所需的系统调用数量来提高效率。也就是**内核侧轮询**。IO_Uring 启用该功能后，应用不再需要调用 `io_uring_enter` 来提交 IO。当应用更新 SQ 并填充新的 SQE 时，内核会自动检测到新的条目并提交它们。而这是通过一个特定于该 io_uring 的内核线程来完成的。

具体使用为：

- 在 setup 的 params 中设置 flags 有 `IORING_SETUP_SQPOLL`
- 额外的，可以通过添加 `IORING_SETUP_SQ_AFF`设置 CPU 亲和性，设置 params 中的 sq_thread_cpu 为亲和 CPU
- 但是，这是一个特权功能，需要应用获得足够的权限，否则失败

由于这里内核线程会一直占用 CPU，所以在长时间没有时间完成时，为防止资源浪费，该进程会进入休眠，并在 SQ 的 flags 中设置 `IORING_SQ_NEED_WAKEUP`，这时候是不能再依靠内核检查 CQ 了，而是需要一段这样的代码：

```
/* 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);
```

同时内核进程休眠的时间默认是一秒，但是可以再 params 中设置，参数为 sq_thread_idle。对于内核侧轮询模式，要得到完成 IO 的事件，直接检查 CQ 的 head。

三、liburing 学习

liburing 是官方支持 io_uring 库，这里简单学习一下它为 io_uring 做的封装。

1. 数据结构

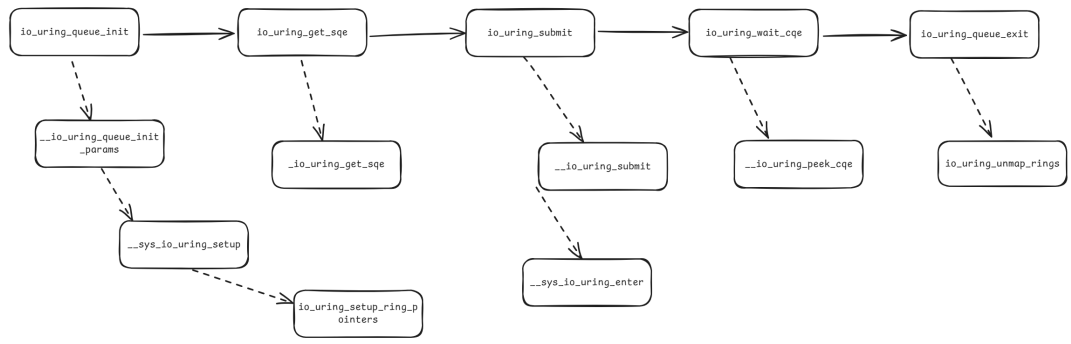
liburing中，核心的结构有 io_uring、io_uring_sq、io_uring_cq

```
/*
 * Library interface to io_uring
 */
struct io_uring_sq {
    unsigned *khead;
    unsigned *ktail;
    // Deprecated: use `ring_mask` instead of `*kring_mask`
    unsigned *kring_mask;
    // Deprecated: use `ring_entries` instead of `*kring_entries`
    unsigned *kring_entries;
    unsigned *kflags;
    unsigned *kdropped;
    unsigned *array;
    struct io_uring_sqe *sqes;

    unsigned sqe_head;
    unsigned sqe_tail;

    size_t ring_sz;
    void *ring_ptr;

    unsigned ring_mask;
    unsigned ring_entries;

    unsigned pad[2];
};

struct io_uring_cq {
    unsigned *khead;
    unsigned *ktail;
    // Deprecated: use `ring_mask` instead of `*kring_mask`
    unsigned *kring_mask;
    // Deprecated: use `ring_entries` instead of `*kring_entries`
    unsigned *kring_entries;
    unsigned *kflags;
    unsigned *koverflow;
```

```
    struct io_uring_cqe *cqes;

    size_t ring_sz;
    void *ring_ptr;

    unsigned ring_mask;
    unsigned ring_entries;

    unsigned pad[2];
};

struct io_uring {
    struct io_uring_sq sq;
    struct io_uring_cq cq;
    unsigned flags;
    int ring_fd;

    unsigned features;
    int enter_ring_fd;
    __u8 int_flags;
    __u8 pad[3];
    unsigned pad2;
};
```

2. 主要接口

简单梳理了一下 `/liburing/examples/io_uring-test.c` 简单使用一个 IO_Uring 的函数调用链：



四、参考文档

- Efficient IO with io_uring
- How io_uring and eBPF Will Revolutionize Programming in Linux
- An Introduction to the io_uring Asynchronous I/O Framework
- 深入学习IO多路复用 select/poll/epoll 实现原理
- 网络 IO 演变发展过程和模型介绍
- 操作系统与存储：解析Linux内核全新异步IO引擎io_uring设计与实现
- liburing