Linux 文件 I/O 进化史（四）：io_uring —— 全新的异步 I/O

基本原理

io_uring 是 2019 年 5 月发布的 Linux 5.1 加入的一个重大特性 —— Linux 下的全新的异步 I/O 支持，希望能彻底解决长期以来 Linux AIO 的各种不足。

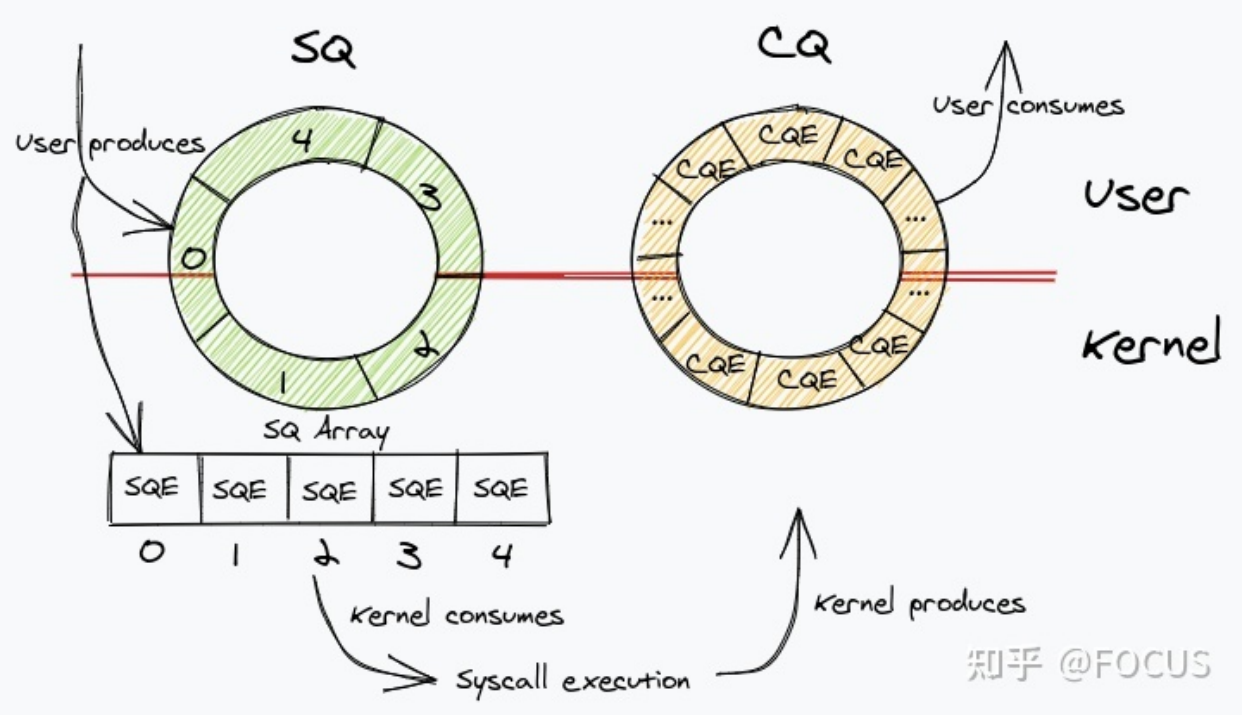
io_uring 实现异步 I/O 的方式其实是一个生产者-消费者模型：

1. 用户进程生产 I/O 请求，放入提交队列（Submission Queue，后续简称 SQ）。
2. 内核消费 SQ 中的 I/O 请求，完成后将结果放入完成队列（Completion Queue，后续简称 CQ）。
3. 用户进程从 CQ 中收割 I/O 结果。

SQ 和 CQ 是内核初始化 io_uring 实例的时候创建的。为了减少系统调用和减少用户进程与内核之间的数据拷贝，io_uring 使用 mmap 的方式让用户进程和内核共享 SQ 和 CQ 的内存空间。

另外，由于先提交的 I/O 请求不一定先完成，SQ 保存的其实是一个数组索引（数据类型 uint32），真正的 SQE（Submission Queue Entry）保存在一个独立的数组（SQ Array）。所以要提交一个 I/O 请求，得先在 SQ Array 中找到一个空闲的 SQE，设置好之后，将其数组索引放到 SQ 中。

用户进程、内核、SQ、CQ 和 SQ Array 之间的基本关系如下：



（图片来自：Getting Hands on with io_uring using Go）

初始化

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

内核提供了 `io_uring_setup` 系统调用来初始化一个 io_uring 实例。

`io_uring_setup` 的返回值是一个文件描述符，暂且称为 ring_fd，用于后续的 mmap 内存映射和其它相关系统调用的参数。

io_uring 会创建 SQ、CQ 和 SQ Array， `entries` 参数表示的是 SQ 和 SQ Array 的大小，CQ 的大小默认是 `2 * entries`。

`params` 参数既是输入参数，也是输出参数，其定义如下：

```
struct io_uring_params {
    __u32 sq_entries;
    __u32 cq_entries;
    __u32 flags;
    __u32 sq_thread_cpu;
    __u32 sq_thread_idle;
    __u32 features;
    __u32 resv[4];
    struct io_sqring_offsets sq_off;
    struct io_cqring_offsets cq_off;
};

struct io_sqring_offsets {
    __u32 head;
    __u32 tail;
    __u32 ring_mask;
    __u32 ring_entries;
    __u32 flags;
    __u32 dropped;
    __u32 array;
    __u32 resv[3];
};

struct io_cqring_offsets {
    __u32 head;
    __u32 tail;
    __u32 ring_mask;
    __u32 ring_entries;
    __u32 overflow;
    __u32 cqes;
    __u32 flags;
```

```
    __u32 resv[3];
};
```

- `flags`、`sq_thread_cpu`、`sq_thread_idle` 是输入参数，用于设置 io_uring 的一些特性。
- `resv[4]` 是保留字段，我们暂且忽略。
- 其它参数都是由内核设置的输出参数，用户进程可能需要使用这些参数进行一些初始化和判断：
  - `sq_entries` 是提交队列的大小。
  - `cq_entries` 是完成队列的大小。
  - `features` 描述当前内核版本支持的 io_uring 特性。其中，`IORING_FEAT_SINGLE_MMAP` 是 io_uring 一个比较重要的特性：内核支持通过一次 mmap 完成 SQ 和 CQ 的内存映射，具体可以参考 liburing 的 io_uring_mmap。
  - `sq_off` 描述了 SQ 的一些属性的 offset。
  - `cq_off` 描述了 CQ 的一些属性的 offset。

用户进程需要根据相关参数对 SQ、CQ 和 SQ Array 进行 mmap 之后，才能和内核共享使用。下面是示例代码：

```
int io_uring_mmap(int ring_fd, struct io_uring_params *p) {
  unsigned sq_ring_sz = p->sq_off.array + p->sq_entries * sizeof(unsigned);
  unsigned cq_ring_sz =
      p->cq_off.cqes + p->cq_entries * sizeof(struct io_uring_cqe);
  unsigned sq_array_size = p->sq_entries * sizeof(struct io_uring_sqe);

  // 建立 SQ 和 CQ 的内存映射
  if (p->features & IORING_FEAT_SINGLE_MMAP) {
    if (cq_ring_sz > sq_ring_sz) {
      sq_ring_sz = cq_ring_sz;
    }
    cq_ring_sz = sq_ring_sz;
  }

  void *sq_ring_ptr =
      mmap(nullptr, sq_ring_sz, PROT_READ | PROT_WRITE,
           MAP_SHARED | MAP_POPULATE, ring_fd, IORING_OFF_SQ_RING);
  if (sq_ring_ptr == MAP_FAILED) {
    return -errno;
  }

  void *cq_ring_ptr = nullptr;
  if (p->features & IORING_FEAT_SINGLE_MMAP) {
    cq_ring_ptr = sq_ring_ptr;
  } else {
    cq_ring_ptr = mmap(nullptr, cq_ring_sz, PROT_READ | PROT_WRITE,
                       MAP_SHARED | MAP_POPULATE, ring_fd,
```

```
IORING_OFF_CQ_RING);
    if (cq_ring_ptr == MAP_FAILED) {
      return -errno; // FIXME: unmap sq_ring_ptr
    }
  }
  // 建立 SQ Array 的内存映射
  void *sq_array_ptr =
      mmap(nullptr, sq_array_size, PROT_READ | PROT_WRITE,
           MAP_SHARED | MAP_POPULATE, ring_fd, IORING_OFF_SQES);
  if (sq_array_ptr == MAP_FAILED) {
    return -errno; // FIXME: unmap sq_ring_ptr and cp_range_ptr
  }

  // ......
}
```

如何提交 I/O 请求

初始化完成之后，我们需要向 io_uring 提交 I/O 请求。默认情况下，使用 io_uring 提交 I/O 请求需要：

1. 从 SQ Arrary 中找到一个空闲的 SQE。
2. 根据具体的 I/O 请求设置这个 SQE。
3. 将 SQE 的数组索引放到 SQ 中。
4. 调用系统调用 `io_uring_enter` 提交 SQ 中的 I/O 请求。

为了进一步提升性能，io_uring 提供了内核轮询的方式来提交 I/O 请求（设置 `params.flags` 的 `IORING_SETUP_SQPOLL` 位）：创建一个内核线程（简称 SQ 线程）对 SQ 进行轮询（polling），发现有未提交的 I/O 请求就自动进行提交：

- 如果 I/O 请求源源不断，SQ 线程会一直轮询并提交 I/O 请求给内核，这个过程不需要经过系统调用。
- 如果 SQ 线程的空闲时间超过 `sq_thread_idle` 毫秒，会自动睡眠，并设置 io_sq_ring 的 flags（`sq_ring_ptr + p.sq_off.flags`）的 `IORING_SQ_NEED_WAKEUP` 位。
- 用户进程需要根据 flags 是否设置了 `IORING_SQ_NEED_WAKEUP` 来决定是否需要调用 `io_uring_enter` 来唤醒 SQ 线程：

```
/* 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, min_complete,
IORING_ENTER_SQ_WAKEUP, NULL);
```

如何收割 I/O 结果

默认情况下，调用 `io_uring_enter` 来收割 I/O：

```
io_uring_enter(ring_fd, to_submit, min_complete, IORING_ENTER_GETEVENTS,
NULL);
```

如果已完成的 I/O 数量小于 `min_complete`，请求会阻塞。

当调用 `io_uring_setup` 时， `params.flags` 的 `IORING_SETUP_IOPOLL` 位被设置时，调用 `io_uring_enter` 收割 I/O，如果已完成的 I/O 的数量不为 0，则不阻塞，立刻返回。

用户进程可以通过遍历 CQ 的 [head, tail) 区间获取已完成的 CQE 并进行处理，然后移动 head 指针到 tail，完成 I/O 收割。

如此，io_uring 的 I/O 提交和收割都可以做到不经过系统调用。

io_uring_register

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

`io_uring_register` 可以将一个文件描述符数组、iovec 数组注册到某个 io_uring 实例，提升 I/O 操作的性能。

- `IORING_REGISTER_FILES`

每次提交 I/O 请求的时候，内核需要将文件 fd 对应的文件增加引用计数。每次 I/O 完成之后，内核需要将文件 fd 对应的文件减少引用计数。而引用计数的修改是原子的，在高 IOPS 的场景下会严重影响性能。

用户进程可以先将文件 fd 数组注册到 io_uring，注册的时候内核会将这些文件的引用计数加 1，只有当取消注册或 io_uring 销毁时，才会将引用计数减 1。

之后提交 I/O 请求的时候将 SQE 的 flags 的 `IOSQE_FIXED_FILE` 设置上，将 fd 参数设置为注册到 io_uring 的 fd 数组的索引。这样，便可以避免每次 I/O 都会导致引用计数原子地加 1 减 1 的开销。

- `IORING_REGISTER_BUFFERS`

当使用 O_DIRECT 的时候：提交 I/O 操作时，内核需要将用户进程的内存映射到内核；完成 I/O 操作后，内核需要将用户进程的内存在内核的映射取消。

在高 IOPS 的时候，频繁的建立、取消内存映射会造成比较大的开销。用户进程可以提前将一个 iovec 数组注册到某个 io_uring 实例，建立相关的内存映射，只有当主动取消注册或 io_uring 实例销毁时，才会取消内存映射。

之后提交 I/O 请求的时候可以使用 `IORING_OP_READ_FIXED` 、 `IORING_OP_WRITE_FIXED` 来配合这些已注册的 buffer （将 io_uring_sqe.addr 指向 iovec 数组相关的 buffer）完成 I/O 操作。

小结

io_uring 通过用户进程和内核共享两个 ring queue 的方式，来减少系统调用和内存拷贝，解决了 Linux AIO 的一些槽点。

io_uring 原理上很简单。但是，这是一套细节很多的接口，很多参数还没搞明白。想要详细了解 io_uring 的话，建议仔细看看下面的参考资料。