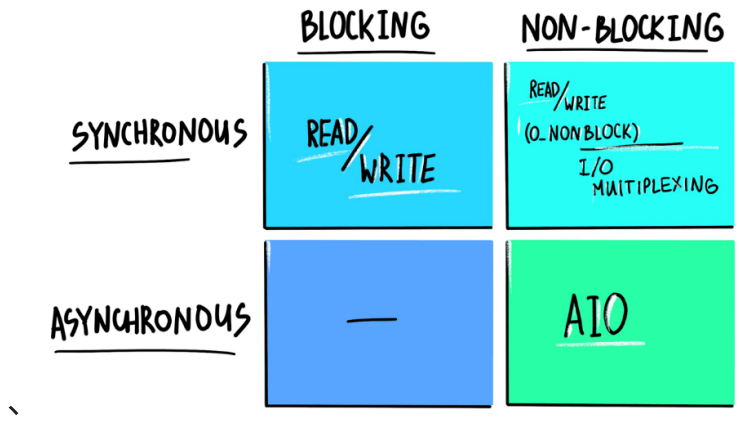
深入理解Linux异步I/O框架 io_uring

赵亚楠 极客重生 2021-12-07 05:02

收录于合集

#深入理解Linux系统 49 #精通后端开发 61



来源：云原生实验室

hi，大家好，今天分享一篇Linux异步IO编程框架文章，对比IO复用的epoll框架，到底性能提高多少？让我们看一看。

译者序

本文组合翻译了以下两篇文章的干货部分，作为 `io_uring` 相关的入门参考：

- **How io_uring and eBPF Will Revolutionize Programming in Linux**[1], ScyllaDB, 2020
- **An Introduction to the io_uring Asynchronous I/O Framework**[2], Oracle, 2020

`io_uring` 是 2019 年 **Linux 5.1** 内核首次引入的高性能**异步 I/O 框架**，能显着加速 I/O 密集型应用的性能。但如果你的应用 **已经在使用** 传统 Linux AIO 了， **并且使用方式恰当**， 那 `io_uring` **并不会带来太大的性能提升** —— 根据原文测试（以及我们 自己的复现），即便打开高级特性，也只有 5%。除非你真的需要这 5% 的额外性能，否则切换成 `io_uring` **代价可能也挺大**，因为要重写应用来适配 `io_uring` （或者让依赖的平台或框架去适配，总之需要改代码）。

既然性能跟传统 AIO 差不多，那为什么还称 `io_uring` 为革命性技术呢？

1. 它首先和最大的贡献在于：**统一了 Linux 异步 I/O 框架**，
   - Linux AIO **只支持 direct I/O** 模式的 **存储文件**（storage file），而且主要用在 **数据库这一细分领域**；
   - `io_uring` 支持存储文件和网络文件（network sockets），也支持更多的异步系统调用（`accept/openat/stat/...`），而非仅限于 `read/write` 系统调用。
2. 在 **设计上是真正的异步 I/O**，作为对比，Linux AIO 虽然也 是异步的，但仍然可能会阻塞，某些情况下的行为也无法预测；

   似乎之前 Windows 在这块反而是领先的，更多参考：
   - **浅析开源项目之 io_uring**[3]，“分布式存储”专栏，知乎
   - **Is there really no asynchronous block I/O on Linux?**[4]，stackoverflow
3. **灵活性和可扩展性**非常好，甚至能基于 `io_uring` 重写所有系统调用，而 Linux AIO 设计时就没考虑扩展性。

eBPF 也算是异步框架（事件驱动），但与 `io_uring` 没有本质联系，二者属于不同子系统，并且在模型上有一个本质区别：

1. **eBPF 对用户是透明的**，只需升级内核（到合适的版本），**应用程序无需任何改造**；
2. `io_uring` 提供了 **新的系统调用和用户空间 API**，因此 **需要应用程序做改造**。

eBPF 作为动态跟踪工具，能够更方便地排查和观测 `io_uring` 等模块在执行层面的具体问题。

本文介绍 Linux 异步 I/O 的发展历史，`io_uring` 的原理和功能，并给出了一些程序示例和性能压测结果（我们在 5.10 内核做了类似测试，结论与原文差不多）。

由于译者水平有限，本文不免存在遗漏或错误之处。如有疑问，请查阅原文。

以下是译文。

很多人可能还没意识到，Linux 内核在过去几年已经发生了一场革命。这场革命源于两个激动人心的新接口的引入：**eBPF 和 io_uring**。我们认为，二者将会完全 **改变应用与内核交互的方式**，以及**应用开发者思考和看待内核的方式**。

本文介绍 io_uring（我们在 ScyllaDB 中有 io_uring 的深入使用经验），并略微提及一下 eBPF。

1 Linux I/O 系统调用演进

1.1 基于 fd 的阻塞式 I/O：read()/write()

作为大家最熟悉的读写方式，Linux 内核提供了 **基于文件描述符的系统调用**， 这些描述符指向的可能是 **存储文件**（storage file），也可能是 **network sockets**：

```
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

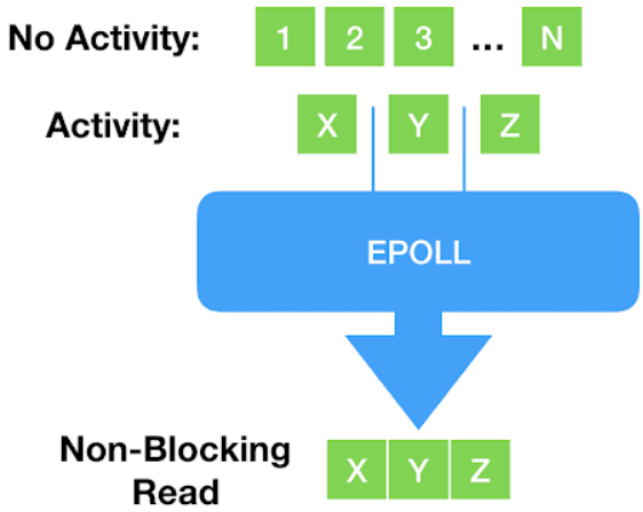
二者称为 **阻塞式系统调用**（blocking system calls），因为程序调用 这些函数时会进入 sleep 状态，然后被调度出去（让出处理器），直到 I/O 操作完成：

- 如果数据在文件中，并且文件内容 **已经缓存在 page cache 中**，调用会 **立即返回**；
- 如果数据在另一台机器上，就需要通过网络（例如 TCP）获取，会阻塞一段时间；
- 如果数据在硬盘上，也会阻塞一段时间。

但很容易想到，随着存储 **设备越来越快，程序越来越复杂**， 阻塞式（blocking）已经这种最简单的方式已经不适用了。

1.2 非阻塞式 I/O：select()/poll()/epoll()

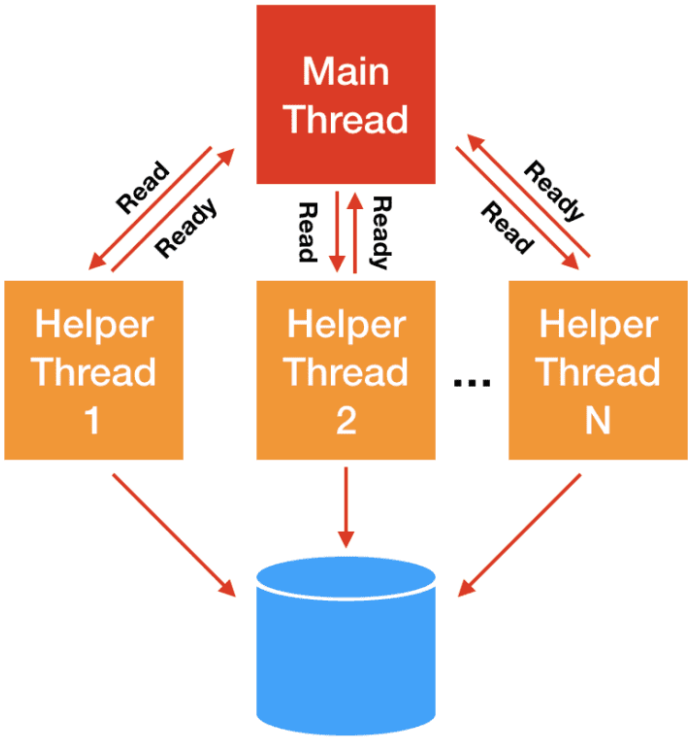
阻塞式之后，出现了一些新的、非阻塞的系统调用，例如 `select()` 、 `poll()` 以及更新的 `epoll()` 。应用程序在调用这些函数读写时不会阻塞，而是 **立即返回**，返回的是一个**已经 ready 的文件描述符列表**。



但这种方式存在一个致命缺点：**只支持 network sockets 和 pipes** —— `epoll()` 甚至连 storage files 都不支持。

1.3 线程池方式

对于 storage I/O，经典的解决思路是 **thread pool**[5]：主线程将 I/O 分发给 worker 线程，后者代替主线程进行阻塞式读写，主线程不会阻塞。



这种方式的问题是 **线程上下文切换开销可能非常大**，后面性能压测会看到。

1.4 Direct I/O（数据库软件）：绕过 page cache

随后出现了更加灵活和强大的方式：**数据库软件**（database software） 有时 **并不想使用操作系统的 page cache**[6]， 而是希望打开一个文件后， **直接从设备读写这个文件**（direct access to the device）。这种方式称为 **直接访问**（direct access）或 **直接 I/O**（direct I/O），

- 需要指定 `O_DIRECT` flag；
- 需要 **应用自己管理自己的缓存** —— 这正是数据库软件所希望的；
- 是 **zero-copy I/O**，因为应用的缓冲数据直接发送到设备，或者直接从设备读取。

1.5 异步 IO（AIO）

前面提到，随着存储设备越来越快，主线程和 worker 线性之间的上下文切换开销占比越来越高。现在市场上的一些设备，例如 **Intel Optane**[7]，**延迟已经低到和上下文切换一个量级**（微秒 `us`）。换个方式描述，更能让我们感受到这种开销：**上下文每切换一次，我们就少一次 dispatch I/O 的机会**。

因此，Linux **2.6** 内核引入了异步 I/O（asynchronous I/O）接口，方便起见，本文简写为 `linux-aio`。AIO **原理**是很简单的：

- 用户通过 `io_submit()` 提交 I/O 请求，
- 过一会再调用 `io_getevents()` 来检查哪些 events 已经 ready 了。
- 使程序员 **能编写完全异步的代码**。

近期，**Linux AIO 甚至支持了**[8] `epoll()`：也就是说 不仅能提交 storage I/O 请求，还能提交网络 I/O 请求。照这样发展下去，linux-aio**似乎能成为一个王者**。但由于它糟糕的演进之路，这个愿望几乎不可能实现了。我们从 **Linus 标志性的激烈言辞中就能略窥一斑**：

> Reply to: **to support opening files asynchronously**[9]
>
> *So I think this is ridiculously ugly.*
>
> *AIO is a horrible ad-hoc design, with the main excuse being "other, less gifted people, made that design, and we are implementing it for compatibility because database people — who seldom have any shred of taste — actually use it".*
>
> — Linus Torvalds (on lwn.net)

首先，作为数据库从业人员，我们想借此机会为我们的没品（lack of taste）向 Linus 道歉。但更重要的是，我们要进一步解释一下 **为什么 Linus 是对的**：Linux AIO 确实问题缠身，

1. **只支持 `O_DIRECT` 文件**，因此 **对常规的非数据库应用**（normal, non-database applications）**几乎是无用的**；
2. 接口在 **设计时并未考虑扩展性**。虽然可以扩展 —— 我们也确实这么做了 —— 但每加一个东西都相当复杂；
3. 虽然从 **技术上说接口是非阻塞的**，但实际上有**很多可能的原因都会导致它阻塞**[10]，而且引发的方式难以预料。

1.6 小结

以上可以清晰地看出 Linux I/O 的演进：

- 最开始是同步（阻塞式）系统调用；
- 然后随着 **实际需求和具体场景**，不断加入新的异步接口，还要保持与老接口的兼容和协同工作。

另外也看到，在非阻塞式读写的问题上 **并没有形成统一方案**：

1. Network socket 领域：添加一个异步接口，然后去轮询（poll）请求是否完成（readiness）；
2. Storage I/O 领域：**只针对某一细分领域**（数据库）在某一特定时期的需求，添加了一个定制版的异步接口。

这就是 Linux I/O 的演进历史 —— 只着眼当前，出现一个问题就引入一种设计，而并没有多少前瞻性 —— 直到 `io_uring` 的出现。

2 io_uring

io_uring 来自资深内核开发者 Jens Axboe 的想法，他在 Linux I/O stack 领域颇有研究。从最早的 patch **aio: support for IO polling**[11]可以看出，这项工作始于一个很简单的观察：随着设备越来越快，**中断驱动（interrupt-driven）模式效率已经低于轮询模式**（polling for completions） —— 这也是高性能领域最常见的主题之一。

- `io_uring` 的 **基本逻辑与 linux-aio 是类似的**：提供两个接口，一个将 I/O 请求提交到内核，一个从内核接收完成事件。
- 但随着开发深入，它逐渐变成了一个完全不同的接口：设计者开始从源头思考**如何支持完全异步的操作**。

2.1 与 Linux AIO 的不同

`io_uring` 与 `linux-aio` 有着本质的不同：

1. **在设计上是真正异步的**（truly asynchronous）。只要 设置了合适的 flag，它在 **系统调用上下文中就只是将请求放入队列**， 不会做其他任何额外的事情， **保证了应用永远不会阻塞**。

2. **支持任何类型的 I/O**：cached files、direct-access files 甚至 blocking sockets。

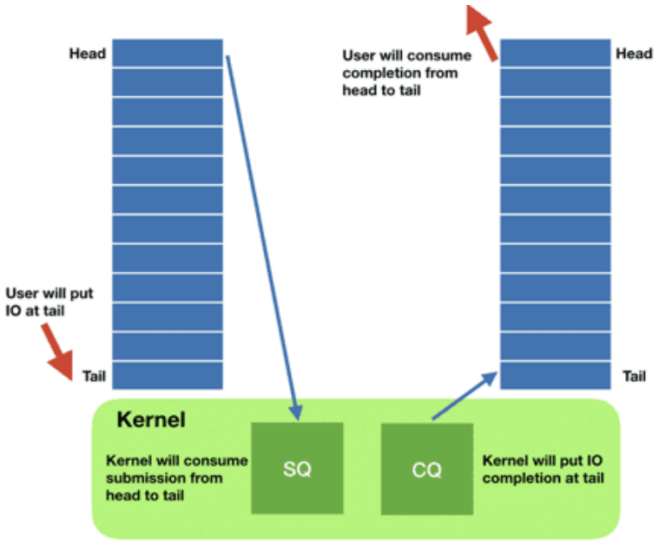
   由于设计上就是异步的（async-by-design nature），因此 **无需 poll+read/write 来处理 sockets**。只需提交一个阻塞式读（blocking read），请求完成之后，就会出现在 completion ring。

3. **灵活、可扩展**：基于 `io_uring` 甚至能重写（re-implement）Linux 的每个系统调用。

2.2 原理及核心数据结构：SQ/CQ/SQE/CQE

每个 io_uring 实例都有 **两个环形队列**（ring），在内核和应用程序之间共享：

- **提交队列**：submission queue (SQ)
- **完成队列**：completion queue (CQ)



这两个队列：

- 都是 **单生产者、单消费者**，size 是 2 的幂次；
- 提供 **无锁接口**（lock-less access interface），内部使用 **内存屏障**做同步（coordinated with memory barriers）。

使用方式：

- 请求
  - 应用创建 SQ entries (SQE)，更新 SQ tail；
  - 内核消费 SQE，更新 SQ head。
- 完成
  - 内核为完成的一个或多个请求创建 CQ entries (CQE)，更新 CQ tail；
  - 应用消费 CQE，更新 CQ head。
  - 完成事件（completion events）可能以任意顺序到达，到总是与特定的 SQE 相关联的。
  - 消费 CQE 过程无需切换到内核态。

2.3 带来的好处

`io_uring` 这种请求方式还有一个好处是：原来需要多次系统调用（读或写），现在变成批处理一次提交。

还记得 Meltdown 漏洞吗？当时我还写了**一篇文章**[12]解释为什么我们的 Scylla NoSQL 数据库受影响很小：`aio` 已经将我们的 I/O 系统调用批处理化了。

`io_uring` **将这种批处理能力带给了** storage I/O 系统调用之外的**其他一些系统调用**，包括：

- `read`
- `write`
- `send`
- `recv`
- `accept`
- `openat`
- `stat`
- 专用的一些系统调用，例如 `fallocate`

此外，`io_uring` 使异步 I/O 的使用场景也不再仅限于数据库应用，**普通的非数据库应用也能用**。这一点值得重复一遍：

> 虽然 `io_uring` 与 `aio` 有一些相似之处，但它的**扩展性和架构是革命性的**：它**将异步操作的强大能力带给了所有应用**（及其开发者），而**不再仅限于是数据库应用这一细分领域**。

我们的 CTO Avi Kivity 在 the Core C++ 2019 event 上**有一次关于 async 的分享**[13]。核心点包括：**从延迟上来说**，

1. 现代多核、多 CPU 设备，其内部本身就是一个基础网络；
2. **CPU 之间**是另一个网络；
3. **CPU 和磁盘 I/O 之间**又是一个网络。

因此网络编程采用异步是明智的，而现在开发自己的应用也应该考虑异步。这**从根本上改变了 Linux 应用的设计方式**：

- 之前都是一段顺序代码流，需要系统调用时才执行系统调用，
- 现在需要思考一个文件是否 ready，因而自然地引入 event-loop，不断通过共享 buffer 提交请求和接收结果。

2.4 三种工作模式

io_uring 实例可工作在三种模式：

1. **中断驱动模式**（interrupt driven）

   默认模式。可通过 io_uring_enter() 提交 I/O 请求，然后直接检查 CQ 状态判断是否完成。

2. **轮询模式**（polled）

   Busy-waiting for an I/O completion，而不是通过异步 IRQ（Interrupt Request）接收通知。

   这种模式需要文件系统（如果有）和块设备（block device）支持轮询功能。相比中断驱动方式，这种方式延迟更低（**连系统调用都省了**[14]），但可能会消耗更多 CPU 资源。

   目前，只有指定了 O_DIRECT flag 打开的文件描述符，才能使用这种模式。当一个读或写请求提交给轮询上下文（polled context）之后，应用（application）必须调用 `io_uring_enter()` 来轮询 CQ 队列，判断请求是否已经完成。

   对一个 io_uring 实例来说，**不支持混合使用轮询和非轮询模式**。

3. **内核轮询模式**（kernel polled）

这种模式中，会 **创建一个内核线程**（kernel thread）来执行 SQ 的轮询工作。

使用这种模式的 io_uring 实例，**应用无需切到到内核态** 就能触发（issue）I/O 操作。通过 SQ 来提交 SQE，以及监控 CQ 的完成状态，应用无需任何系统调用，就能提交和收割 I/O（submit and reap I/Os）。

如果内核线程的空闲时间超过了用户的配置值，它会通知应用，然后进入 idle 状态。这种情况下，应用必须调用 `io_uring_enter()` 来唤醒内核线程。如果 I/O 一直很繁忙，内核线性是不会 sleep 的。

2.5 io_uring 系统调用 API

有三个：

- `io_uring_setup(2)`
- `io_uring_register(2)`
- `io_uring_enter(2)`

下面展开介绍。完整文档见 **manpage**[15]。

2.5.1 io_uring_setup()

执行异步 I/O 需要先 **设置上下文**：

```
int io_uring_setup(u32 entries, struct io_uring_params *p);
```

这个系统调用

- **创建一个 SQ 和一个 CQ**，
- queue size 至少 `entries` 个元素，
- 返回一个文件描述符，随后用于在这个 io_uring 实例上执行操作。

SQ 和 CQ 在应用和内核之间共享，避免了在初始化和完成 I/O 时（initiating and completing I/O）拷贝数据。

参数 p：

- 应用用来配置 io_uring，
- 内核返回的 SQ/CQ 配置信息也通过它带回来。

io_uring_setup() 成功时返回一个文件描述符（fd）。应用随后可以将这个 fd 传给 mmap(2) 系统调用，来 map the submission and completion queues 或者传给 to the io_uring_register() or io_uring_enter() system calls.

2.5.2 io_uring_register()

注册用于异步 I/O 的 **文件或用户缓冲区**（files or user buffers）：

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

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**（internal kernel data structures associated with the files），或创建 **应用内存的长期映射**（long term mappings of application memory associated with the buffers），这个操作只会在注册时执行一次，而不是每个 I/O 请求都会处理，因此减少了 per-I/O overhead。

注册的缓冲区（buffer）性质

- Registered buffers 将会 **被锁定在内存中**（be locked in memory），并 **计入用户的 RLIMIT_MEMLOCK** 资源限制。
- 此外，每个 buffer 有 **1GB 的大小限制**。
- 当前，buffers 必须是 **匿名、非文件后端的内存**（anonymous, non-file-backed memory），例如 malloc(3) or mmap(2) with the MAP_ANONYMOUS flag set 返回的内存。
- Huge pages 也是支持的。整个 huge page 都会被 pin 到内核，即使只用到了其中一部分。
- 已经注册的 buffer 无法调整大小，想调整只能先 unregister，再重新 register 一个新的。

通过 `eventfd()` 订阅 completion 事件

可以用 `eventfd(2)` 订阅 io_uring 实例的 completion events。将 eventfd 描述符通过这个系统调用注册就行了。

> The credentials of the running application can be registered with io_uring which returns an id associated with those credentials. Applications wishing to share a ring between separate users/processes can pass in this credential id in the SQE personality field. If set, that particular SQE will be issued with these credentials.

2.5.3 `io_uring_enter()`

```
int io_uring_enter(unsigned int fd, unsigned int to_submit, unsigned int min_complete, unsigned
```

这个系统调用用于初始化和完成（initiate and complete）I/O，使用共享的 SQ 和 CQ。单次调用同时执行：

1. 提交新的 I/O 请求
2. 等待 I/O 完成

参数：

1. `fd` 是 `io_uring_setup()` 返回的文件描述符；
2. `to_submit` 指定了 SQ 中提交的 I/O 数量；
3. 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，这个参数表示：

a. 0：要求内核返回当前以及完成的所有 events，无阻塞；
b. 非零：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。

注意：对于 interrupt driven I/O， **应用无需进入内核就能检查 CQ 的 event completions**。

`io_uring_enter()` 支持很多操作，包括：

- Open, close, and stat files
- Read and write into multiple buffers or pre-mapped buffers
- Socket I/O operations
- Synchronize file state
- Asynchronously monitor a set of file descriptors
- Create a timeout linked to a specific operation in the ring
- Attempt to cancel an operation that is currently in flight
- Create I/O chains
- Ordered execution within a chain
- Parallel execution of multiple chains

当这个系统调用返回时，表示一定数量的 SEQ 已经被消费和提交了，此时可以安全的重用队列中的 SEQ。此时 IO 提交有可能还停留在异步上下文中，即实际上 SQE 可能还没有被提交 —— 不过 用户不用关心这些细节 —— 当随后内核需要使用某个特定的 SQE 时，它已经复制了一份。

2.6 高级特性

`io_uring` 提供了一些用于特殊场景的高级特性：

1. **File registration**（文件注册）：每次发起一个指定文件描述的操 作，内核都需要 **花费一些时钟周期**（cycles） **将文件描述符映射到内部表示**。对于那些 **针对同一文件进行重复操作**的场景， `io_uring` 支持 **提前注册这些文件**，后面直接查找就行了。
2. **Buffer registration**（缓冲区注册）：与 file registration 类 似，direct I/O 场景中，内核需要 map/unmap memory areas。 `io_uring` 支持提前 注册这些缓冲区（buffers）。
3. **Poll ring**（轮询环形缓冲区）：对于非常快是设备，处理中断的开 销是比较大的。 `io_uring` 允许用户关闭中断，使用轮询模式。前面“三种工作模式”小节 也介绍到了这一点。
4. **Linked operations**（链接操作）：允许用户发送串联的请求。这两 个请求同时提交，但后面的会等前面的处理完才开始执行。

2.7 用户空间库 liburing

`liburing`[16] 提供了一个简单的高层 API， 可用于一些基本场景，应用程序避免了直接使用更底层的系统调用。此外，这个 API 还避免了一些重复操作的代码，如设置 io_uring 实例。

举个例子，在 `io_uring_setup()` 的 manpage 描述中，调用这个系统调用获得一个 ring 文件描述符之后，应用必须调用 `mmap()` 来这样的逻辑需要一段略长的代码，而用 `liburing` 的话，下面的函数已经将上述流程封装好了：

```
int io_uring_queue_init(unsigned entries, struct io_uring *ring, unsigned flags);
```

下一节来看两个例子基于 liburing 的例子。

3 基于 liburing 的示例应用

编译：

```
$ git clone https://github.com/axboe/liburing.git
$ git co -b liburing-2.0 tags/liburing-2.0

$ cd liburing
$ ls examples/
io_uring-cp  io_uring-cp.c  io_uring-test  io_uring-test.c  link-cp  link-cp.c  Makefile  ucont

$ make -j4

$ ./examples/io_uring-test <file>
Submitted=4, completed=4, bytes=16384

$ ./examples/link-cp <in-file> <out-file>
```

3.1 io_uring-test

这个程序使用 4 个 SQE，从输入文件中 **读取最多 16KB 数据**。

源码及注释

为方便看清主要逻辑，忽略了一些错误处理代码，完整代码见**io_uring-test.c**[17]。

```
/* SPDX-License-Identifier: MIT */
/*
 * Simple app that demonstrates how to setup an io_uring interface,
 * submit and complete IO against it, and then tear it down.
 *
 * gcc -Wall -O2 -D_GNU_SOURCE -o io_uring-test io_uring-test.c -luring
 */
#include "liburing.h"

#define QD    4 // io_uring 队列长度

int main(int argc, char *argv[]) {
    int i, fd, pending, done;
    void *buf;

    // 1. 初始化一个 io_uring 实例
    struct io_uring ring;
    ret = io_uring_queue_init(QD,    // 队列长度
                              &ring, // io_uring 实例
                              0);    // flags，0 表示默认配置，例如使用中断驱动模式

    // 2. 打开输入文件，注意这里指定了 O_DIRECT flag，内核轮询模式需要这个 flag，见前面介绍
    fd = open(argv[1], O_RDONLY | O_DIRECT);
    struct stat sb;
    fstat(fd, &sb); // 获取文件信息，例如文件长度，后面会用到

    // 3. 初始化 4 个读缓冲区
    ssize_t fsize = 0;              // 程序的最大读取长度
    struct iovec *iovecs = calloc(QD, sizeof(struct iovec));
    for (i = 0; i < QD; i++) {
        if (posix_memalign(&buf, 4096, 4096))
            return 1;
        iovecs[i].iov_base = buf;  // 起始地址
        iovecs[i].iov_len = 4096;  // 缓冲区大小
```

```
    fsize += 4096;
}

// 4. 依次准备 4 个 SQE 读请求，指定将随后读入的数据写入 iovecs
struct io_uring_sqe *sqe;
offset = 0;
i = 0;
do {
    sqe = io_uring_get_sqe(&ring);  // 获取可用 SQE
    io_uring_prep_readv(sqe,        // 用这个 SQE 准备一个待提交的 read 操作
                        fd,         // 从 fd 打开的文件中读取数据
                        &iovecs[i], // iovec 地址，读到的数据写入 iovec 缓冲区
                        1,          // iovec 数量
                        offset);    // 读取操作的起始地址偏移量
    offset += iovecs[i].iov_len;    // 更新偏移量，下次使用
    i++;

    if (offset > sb.st_size)        // 如果超出了文件大小，停止准备后面的 SQE
        break;
} while (1);

// 5. 提交 SQE 读请求
ret = io_uring_submit(&ring);       // 4 个 SQE 一次提交，返回提交成功的 SQE 数量
if (ret < 0) {
    fprintf(stderr, "io_uring_submit: %s\n", strerror(-ret));
    return 1;
} else if (ret != i) {
    fprintf(stderr, "io_uring_submit submitted less %d\n", ret);
    return 1;
}

// 6. 等待读请求完成（CQE）
struct io_uring_cqe *cqe;
done = 0;
pending = ret;
fsize = 0;
for (i = 0; i < pending; i++) {
    io_uring_wait_cqe(&ring, &cqe);  // 等待系统返回一个读完成事件
    done++;
```

```
        if (cqe->res != 4096 && cqe->res + fsize != sb.st_size) {
            fprintf(stderr, "ret=%d, wanted 4096\n", cqe->res);
        }

        fsize += cqe->res;
        io_uring_cqe_seen(&ring, cqe);    // 更新 io_uring 实例的完成队列
    }

    // 7. 打印统计信息
    printf("Submitted=%d, completed=%d, bytes=%lu\n", pending, done, (unsigned long) fsize);

    // 8. 清理工作
    close(fd);
    io_uring_queue_exit(&ring);
    return 0;
}
```

其他说明

代码中已经添加了注释，这里再解释几点：

- 每个 SQE 都执行一个 allocated buffer，后者是用 `iovec` 结构描述的；
- 第 3 & 4 步：初始化所有 SQE，用于接下来的 `IORING_OP_READV` 操作，后者**提供了 `readv(2)` 系统调用的异步接口**。
- 操作完成之后，这个 SQE iovec buffer 中存放的是相关 `readv` 操作的结果；
- 接下来调用 `io_uring_wait_cqe()` 来 reap CQE，并通过 `cqe->res` 字段验证读取的字节数；
- `io_uring_cqe_seen()` 通知内核这个 CQE 已经被消费了。

3.2 link-cp

link-cp 使用 io_uring 高级特性 SQE chaining 特性来复制文件。

I/O chain

io_uring 支持创建 I/O chain。一个 chain 内的 I/O 是顺序执行的，多个 I/O chain 可以并行执行。

`io_uring_enter()` manpage 中对 `IOSQE_IO_LINK` 有 **详细解释**[18]：

> When this flag is specified, it forms a link with the next SQE in the submission ring. That next SQE will not be started before this one completes. This, in effect, forms a chain of SQEs, which can be arbitrarily long. The tail of the chain is denoted by the first SQE that does not have this flag set. This flag has no effect on previous SQE submissions, nor does it impact SQEs that are outside of the chain tail. This means that multiple chains can be executing in parallel, or chains and individual SQEs. Only members inside the chain are serialized. A chain of SQEs will be broken, if any request in that chain ends in error. io_uring considers any unexpected result an error. This means that, eg, a short read will also terminate the remainder of the chain. If a chain of SQE links is broken, the remaining unstarted part of the chain will be terminated and completed with -ECANCELED as the error code. Available since 5.3.

为实现复制文件功能，link-cp 创建一个长度为 2 的 SQE chain。

- 第一个 SQE 是一个读请求，将数据从输入文件读到 buffer；
- 第二个请求，与第一个请求是 linked，是一个写请求，将数据从 buffer 写入输出文件。

源码及注释

```c
/* SPDX-License-Identifier: MIT */
/*
 * Very basic proof-of-concept for doing a copy with linked SQEs. Needs a
 * bit of error handling and short read love.
 */
#include "liburing.h"

#define QD    64         // io_uring 队列长度
#define BS    (32*1024)

struct io_data {
    size_t offset;
    int index;
    struct iovec iov;
};

static int infd, outfd;
```

```c
static unsigned inflight;

// 创建一个 read->write SQE chain
static void queue_rw_pair(struct io_uring *ring, off_t size, off_t offset) {
    struct io_uring_sqe *sqe;
    struct io_data *data;
    void *ptr;

    ptr = malloc(size + sizeof(*data));
    data = ptr + size;
    data->index = 0;
    data->offset = offset;
    data->iov.iov_base = ptr;
    data->iov.iov_len = size;

    sqe = io_uring_get_sqe(ring);                               // 获取可用 SQE
    io_uring_prep_readv(sqe, infd, &data->iov, 1, offset);      // 准备 read 请求
    sqe->flags |= IOSQE_IO_LINK;                                // 设置为 LINK 模式
    io_uring_sqe_set_data(sqe, data);                           // 设置 data

    sqe = io_uring_get_sqe(ring);                               // 获取另一个可用 SQE
    io_uring_prep_writev(sqe, outfd, &data->iov, 1, offset);    // 准备 write 请求
    io_uring_sqe_set_data(sqe, data);                           // 设置 data
}

// 处理完成（completion）事件：释放 SQE 的内存缓冲区，通知内核已经消费了 CQE。
static int handle_cqe(struct io_uring *ring, struct io_uring_cqe *cqe) {
    struct io_data *data = io_uring_cqe_get_data(cqe);          // 获取 CQE
    data->index++;

    if (cqe->res < 0) {
        if (cqe->res == -ECANCELED) {
            queue_rw_pair(ring, BS, data->offset);
            inflight += 2;
        } else {
            printf("cqe error: %s\n", strerror(cqe->res));
            ret = 1;
        }
    }

    if (data->index == 2) {         // read->write chain 完成，释放缓冲区内存
```

```
        void *ptr = (void *) data - data->iov.iov_len;
        free(ptr);
    }

    io_uring_cqe_seen(ring, cqe);  // 通知内核已经消费了 CQE 事件
    return ret;
}

static int copy_file(struct io_uring *ring, off_t insize) {
    struct io_uring_cqe *cqe;
    size_t this_size;
    off_t offset;

    offset = 0;
    while (insize) {                        // 数据还没处理完
        int has_inflight = inflight;        // 当前正在进行中的 SQE 数量
        int depth;  // SQE 阈值，当前进行中的 SQE 数量（inflight）超过这个值之后，需要阻塞等待 CQE 完成

        while (insize && inflight < QD) { // 数据还没处理完，io_uring 队列也还没用完
            this_size = BS;
            if (this_size > insize)         // 最后一段数据不足 BS 大小
                this_size = insize;

            queue_rw_pair(ring, this_size, offset); // 创建一个 read->write chain，占用两个 SQE
            offset += this_size;
            insize -= this_size;
            inflight += 2;                  // 正在进行中的 SQE 数量 +2
        }

        if (has_inflight != inflight)       // 如果有新创建的 SQE，
            io_uring_submit(ring);          // 就提交给内核

        if (insize)                         // 如果还有 data 等待处理，
            depth = QD;                     // 阈值设置 SQ 的队列长度，即 SQ 队列用完才开始阻塞等待 CQE；
        else                                // data 处理已经全部提交，
            depth = 1;                      // 阈值设置为 1，即只要还有 SQE 未完成，就阻塞等待 CQE

        // 下面这个 while 只有 SQ 队列用完或 data 全部提交之后才会执行到
        while (inflight >= depth) {         // 如果所有 SQE 都已经用完，或者所有 data read->write 请求
            io_uring_wait_cqe(ring, &cqe);// 等待内核 completion 事件
            handle_cqe(ring, cqe);          // 处理 completion 事件：释放 SQE 内存缓冲区，通知内核 CQE
```

```
            inflight--;                     // 正在进行中的 SQE 数量 -1
        }
    }

    return 0;
}

static int setup_context(unsigned entries, struct io_uring *ring) {
    io_uring_queue_init(entries, ring, 0);
    return 0;
}

static int get_file_size(int fd, off_t *size) {
    struct stat st;

    if (fstat(fd, &st) < 0)
        return -1;
    if (S_ISREG(st.st_mode)) {
        *size = st.st_size;
        return 0;
    } else if (S_ISBLK(st.st_mode)) {
        unsigned long long bytes;

        if (ioctl(fd, BLKGETSIZE64, &bytes) != 0)
            return -1;

        *size = bytes;
        return 0;
    }

    return -1;
}

int main(int argc, char *argv[]) {
    struct io_uring ring;
    off_t insize;
    int ret;

    infd = open(argv[1], O_RDONLY);
    outfd = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0644);

    if (setup_context(QD, &ring))
```

```
        return 1;
    if (get_file_size(infd, &insize))
        return 1;

    ret = copy_file(&ring, insize);

    close(infd);
    close(outfd);
    io_uring_queue_exit(&ring);
    return ret;
}
```

其他说明

代码中实现了三个函数：

1. `copy_file()`：高层复制循环逻辑；它会调用 `queue_rw_pair(ring, this_size, offset)` 来构造 SQE pair；并通过一次 `io_uring_submit()` 调用将所有构建的 SQE pair 提交。

   这个函数维护了一个最大 DQ 数量的 inflight SQE，只要数据 copy 还在进行中；否则，即数据已经全部读取完成，就开始等待和收割所有的 CQE。

2. `queue_rw_pair()` 构造一个 read-write SQE pair.

   read SQE 的 `IOSQE_IO_LINK` flag 表示开始一个 chain，write SQE 不用设置这个 flag，标志着这个 chain 的结束。用户 data 字段设置为同一个 data 描述符，并且在随后的 completion 处理中会用到。

3. `handle_cqe()` 从 CQE 中提取之前由 `queue_rw_pair()` 保存的 data 描述符，并在描述符中记录处理进展（index）。

   如果之前请求被取消，它还会重新提交 read-write pair。

   一个 CQE pair 的两个 member 都处理完成之后（`index==2`），释放共享的 data descriptor。最后通知内核这个 CQE 已经被消费。

4 io_uring 性能压测（基于 fio）

对于已经在使用 linux-aio 的应用，例如 ScyllaDB，**不要期望换成 io_uring 之后能获得大幅的性能提升**，这是因为： `io_uring` 性能相关的底层机制与 `linux-aio` 并无本质不同（都是异步提交，轮询结果）。

在此，本文也希望使读者明白： `io_uring` 首先和最重要的贡献在于：**将 linux-aio 的所有优良特性带给了普罗大众**（而非局限于数据库这样的细分领域）。

4.1 测试环境

本节使用 `fio` 测试 4 种模式：

1. synchronous reads
2. `posix-aio` (implemented as a thread pool)
3. `linux-aio`
4. `io_uring`

硬件：

- NVMe 存储设备，物理极限能打到 **3.5M IOPS**。
- 8 核处理器

4.2 场景一：direct I/O 1KB 随机读（绕过 page cache）

第一组测试中，我们希望所有的读请求都能 **命中存储设备**（all reads to hit the storage）， **完全绕开操作系统的页缓存**（page cache）。

测试配置：

- 8 个 CPU 执行 72 `fio` job，
- 每个 job 随机读取 4 个文件，
- `iodepth=8` （number of I/O units to keep in flight against the file.）。

这种配置 **保证了 CPU 处于饱和状态**，便于观察 I/O 性能。如果 CPU 数量足够多，那每组测试都可能会打满设备带宽，结果对 I/O 压测就没意义了。

表 1. Direct I/O（绕过系统页缓存）：1KB 随机读，CPU 100% 下的 I/O 性能

backend	IOPS	context switches	IOPS ±% vs io_uring
sync	814,000	27,625,004	**-42.6%**
posix-aio (thread pool)	433,000	64,112,335	**-69.4%**

linux-aio	1,322,000	10,114,149	**-6.7%**
io_uring (basic)	1,417,000	11,309,574	**—**
io_uring (enhanced)	1,486,000	11,483,468	**4.9%**

几点分析：

1. `io_uring` 相比 `linux-aio` 确实有一定提升，但并非革命性的。
2. 开启高级特性，例如 buffer & file registration 之后性能有进一步提升 —— 但也还 没有到为了这些性能而重写整个应用的地步，除非你是搞数据库研发，想榨取硬件的最后一分性能。
3. `io_uring` and `linux-aio` **都比同步 read 接口快 2 倍，而后者又比 posix-aio 快 2 倍** —— 初看有点差异。但看看 **上下文切换次数**，就不难理解为什么 posix-aio 这么慢了。
   - 同步 read 性能差是因为：在这种没有 page cache 的情况下，**每次 read 系统调用都会阻塞，因此就会涉及一次上下文切换**。
   - `posix-aio` 性能更差是因为：不仅内核和应用程序之间要频繁上下文切换，线程池的 **多个线程之间也在频繁切换**。

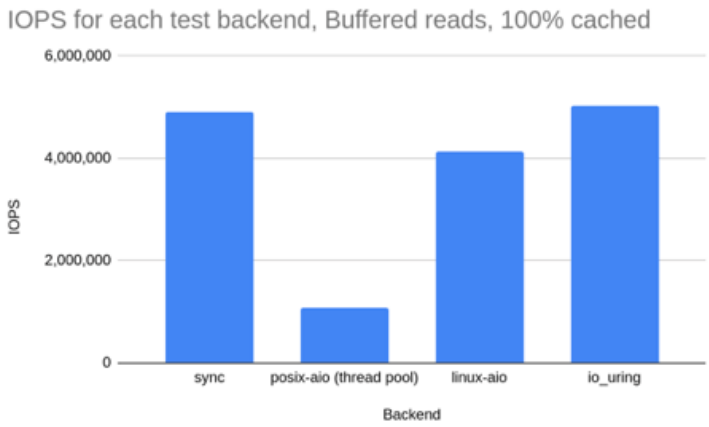
4.2 场景二：buffered I/O 1KB 随机读（数据提前加载到内存，100% hot cache）

第二组测试 buffered I/O：

1. 将文件数据提前加载到内存，然后再测随机读。
   - 由于 **数据全部在 page cache**，因此 **同步 read 永远不会阻塞**。
   - 这种场景下，我们预期 **同步读和 io_uring 的性能差距不大（都是最好的）**。
2. 其他测试条件不变。

表 2. Buffered I/O（数据全部来自 page cache，100% hot cache）：1KB 随机读，100% CPU 下的 I/O 性能

Backend	IOPS	context switches	IOPS ±% vs io_uring
sync	4,906,000	105,797	-2.3%
posix-aio (thread pool)	1,070,000	114,791,187	-78.7%
linux-aio	4,127,000	105,052	-17.9%
io_uring	5,024,000	106,683	—



结果分析：

1. 同步读和 `io_uring` 性能差距确实很小，二者都是最好的。

   但注意， **实际的应用**不可能一直 100% 时间执行 IO 操作，因此 基于同步读的真实应用性能 **还是要比基于 io_uring 要差的**，因为 io_uring 会将多个系统调用批处理化。

2. `posix-aio` 性能最差，直接原因是 **上下文切换次数太多**，这也和场景相关：在这种 **CPU 饱和的情况下**，它的线程池反而是累赘，会完全拖慢性能。

3. `linux-aio` 并 **不是针对 buffered I/O 设计的**，在这种 page cache 直接返回的场景， 它的 **异步接口反而会造成性能损失** —— 将操作分 为 dispatch 和 consume 两步不但没有性能收益，反而有额外开销。

4.3 性能测试小结

最后再次提醒，本节是极端应用/场景（ **100% CPU + 100% cache miss/hit**）测试， 真实应用的行为通常处于同步读和异步读之间：时而一些阻塞操作，时而一些非阻塞操作。但不管怎么说，

用了 io_uring 之后，用户就无需担心同步和异步各占多少比例了，因为它 **在任何场景下都表现良好**。

1. 如果操作是非阻塞的， io_uring 不会有额外开销；
2. 如果操作是阻塞式的，也没关系， io_uring 是完全异步的，并且不依赖线程池或昂贵的上下文切换来实现这种异步能力；

本文测试的都是随机读，但对 **其他类型的操作**， io_uring 表现也是非常良好的。例如：

1. 打开/关闭文件
2. 设置定时器
3. 通过 network sockets 传输数据

而且 **使用的是同一套 io_uring 接口**。

4.4 ScyllaDB 与 io_uring

Scylla 重度依赖 direct I/O，从一开始就使用 linux-aio 。在我们转向 io_uring 的过程中，最开始测试显示对某些 workloads，能取得 50% 以上的性能提升。但 **深入研究之后发现**，这是因为我们 **之前的 linux-aio 用的不够好**。这也揭示了一个 **经常被忽视的事实**：获得高性能没有那么难（前提是你得弄对了）。在对比 io_uring 和 linux-aio 应用之后，我们 **很快改进了一版，二者的性能差距就消失了**。但坦率地说，解决这个问题 **需要一些工作量**，因为要改动一个已经使用 了很多年的基于 linux-aio 的接口。而对 io_uring 应用来说，做类似的改动是轻而 易举的。

以上只是一个场景， io_uring 相比 linux-aio 的 **优势**是能应用于 file I/O 之外的场景。此外，它还自带了特殊的 **高性能**[19] 接口，例如 buffer registration、file registration、轮询模式等等。

启用 io_uring 高级特性之后，我们看到性能确实有提升：Intel Optane 设备，单个 CPU 读取 512 字节，观察到 5% 的性能提升。与 表 1 & 2 对得上。虽然 5% 的提升 看上去不是太大，但对于希望压榨出硬件所有性能的数据库来说，还是非常宝贵的。

linux-aio:	io_uring, with buffer and file registration and poll:
Throughput : 330 MB/s Lat average : 1549 usec Lat quantile= 0.5 : 1547 usec Lat quantile= 0.95 : 1694 usec Lat quantile= 0.99 : 1703 usec	Throughput : 346 MB/s Lat average : 1470 usec Lat quantile= 0.5 : 1468 usec Lat quantile= 0.95 : 1558 usec

Lat quantile=0.999 : 1950 usec Lat max : 2177 usec	Lat quantile= 0.99 : 1613 usec Lat quantile=0.999 : 1674 usec Lat max : 1829 usec
使用 1 个 CPU 从 Intel Optane 设备读取 512 字节。1000 并发请求。linux-aio 和 io_uring basic interface 性能差异很小。 但启用 io_uring 高级特性后，有 5% 的性能差距。	

5 eBPF

eBPF 也是一个 **事件驱动框架**（因此也是异步的），允许用户空间程序动态向内核注入字节码，主要有两个使用场景：

1. Networking：**本站**[20] 已经有相当多的文章
2. Tracing & Observability：例如 **bcc**[21] 等工具

eBPF 在内核 4.9 首次引入，4.19 以后功能已经很强大。更多关于 eBPF 的演进信息，可参考：（译）大规模微服务利器：eBPF + Kubernetes（KubeCon, 2020）。

谈到与 io_uring 的结合，就是用 bcc 之类的工具跟踪一些 I/O 相关的内核函数，例如：

1. Trace how much time an application spends sleeping, and what led to those sleeps. (`wakeuptime`)
2. Find all programs in the system that reached a particular place in the code (`trace`)
3. Analyze network TCP throughput aggregated by subnet (`tcpsubnet`)
4. Measure how much time the kernel spent processing softirqs (`softirqs`)
5. Capture information about all short-lived files, where they come from, and for how long they were opened (`filelife`)

6 结束语

`io_uring` 和 eBPF 这两大特性 **将给 Linux 编程带来革命性的变化**。有了这两个特性的加持，开发者就能更充分地利用 **Amazon i3en meganode systems**[22]之类的多核/多处理器系统，以及 **Intel Optane 持久存储**[23]之类的 `us` 级延迟存储设备。

参考资料

- **Efficient IO with io_uring**[24], pdf
- **Ringing in a new asynchronous I/O API**[25], lwn.net
- **The rapid growth of io_uring**[26], lwn.net

- **System call API**[27], manpage

引用链接

[1] How io_uring and eBPF Will Revolutionize Programming in Linux: *https://thenewstack.io/how-io_uring-and-ebpf-will-revolutionize-programming-in-linux*

[2] An Introduction to the io_uring Asynchronous I/O Framework: *https://medium.com/oracledevs/an-introduction-to-the-io-uring-asynchronous-i-o-framework-fad002d7dfc1*

[3] 浅析开源项目之 io_uring: *https://zhuanlan.zhihu.com/p/361955546*

[4] Is there really no asynchronous block I/O on Linux?: *https://stackoverflow.com/questions/13407542/is-there-really-no-asynchronous-block-i-o-on-linux*

[5] thread pool: *https://en.wikipedia.org/wiki/Thread_pool*

[6] 并不想使用操作系统的 page cache: *https://www.scylladb.com/2018/07/26/how-scylla-data-cache-works*

[7] Intel Optane: *https://pcper.com/2018/12/intels-optane-dc-persistent-memory-dimms-push-latency-closer-to-dram*

[8] Linux AIO 甚至支持了: *https://lwn.net/Articles/742978/*

[9] to support opening files asynchronously: *https://lwn.net/Articles/671657/*

[10] 很多可能的原因都会导致它阻塞: *https://lwn.net/Articles/724198*

[11] aio: support for IO polling: *https://lwn.net/ml/linux-fsdevel/20181221192236.12866-9-axboe@kernel.dk*

[12] 一篇文章: *https://www.scylladb.com/2018/01/07/cost-of-avoiding-a-meltdown/*

[13] 有一次关于 async 的分享: *https://www.scylladb.com/2020/03/26/avi-kivity-at-core-c-2019*

[14] 连系统调用都省了: *https://www.phoronix.com/scan.php?page=news_item&px=Linux-io_uring-Fast-Efficient*

[15] manpage: *https://github.com/axboe/liburing/tree/master/man*

[16] `liburing` : *https://github.com/axboe/liburing/*

[17] io_uring-test.c: *https://github.com/axboe/liburing/blob/liburing-2.0/examples/io_uring-test.c*

[18] 详细解释: *https://www.mankier.com/2/io_uring_enter#Description-IOSQE_IO_LINK*

[19] 高性能: *https://www.p99conf.io/*

[20] 本站: *https://arthurchiao.art*

[21] bcc: *https://github.com/iovisor/bcc*

[22] Amazon i3en meganode systems: *https://www.scylladb.com/2019/05/28/aws-new-i3en-meganode*

[23] Intel Optane 持久存储: *https://www.scylladb.com/2017/09/27/intel-optane-scylla-providing-speed-memory-database-persistency*

[24] Efficient IO with io_uring: *https://kernel.dk/io_uring.pdf*

[25] Ringing in a new asynchronous I/O API: *https://lwn.net/Articles/776703/*

[26] The rapid growth of io_uring: *https://lwn.net/Articles/810414/*