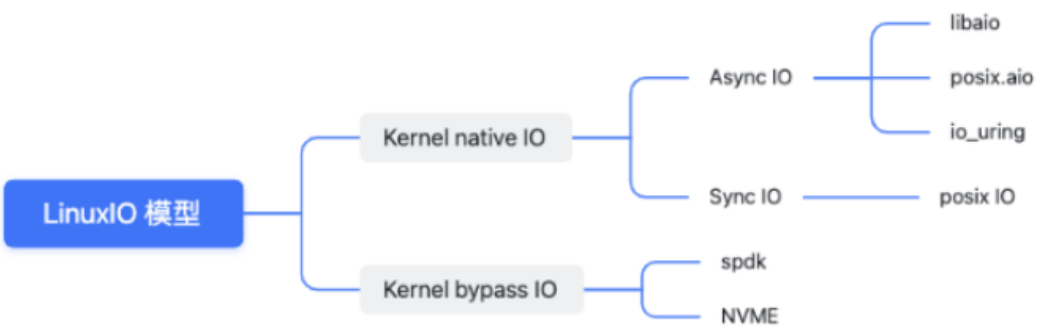
新一代异步IO框架 io

原文链接

1.Linux IO 模型分类



相比于kernel bypass 模式需要结合具体的硬件支撑来讲，native IO是日常工作中接触到比较多的一种，其中同步IO在较长一段时间内被广泛使用，通常我们接触到的IO操作主要分为网络IO和存储IO。在大流量高并发的今天，提到网络IO，很容易想到大名鼎鼎的epoll 以及reactor架构。但是epoll并不属于异步IO的范畴。本质上是一个同步非阻塞的架构。关于同步异步，阻塞与非阻塞的概念区别这里做简要概述：

- **什么是同步**

指进程调用接口时需要等待接口处理完数据并相应进程才能继续执行。这里重点是数据处理活逻辑执行完成并返回，如果是异步则不必等待数据完成，亦可以继续执行。同步强调的是逻辑上的次序性；

- **什么是阻塞**

当进程调用一个阻塞的系统函数时，该进程被 置于睡眠(Sleep)状态，这时内核调度其它进程运行，直到该进程等待的事件发生了(比 如网络上接收到数据包，或者调用sleep指定的睡眠时间到了)它才有可能继续运行。与睡眠状态相对的是运行(Running)状态，在Linux内核中，处于运行状态的进程分为两种情况，一种是进程正在被CPU调度，另一种是处于就绪状态随时可能被调度的进程；阻塞强调的是函数调用下进程的状态。

2.Linux常见文件操作方式

2.1 open/close/read/write

基本操作API 如下：

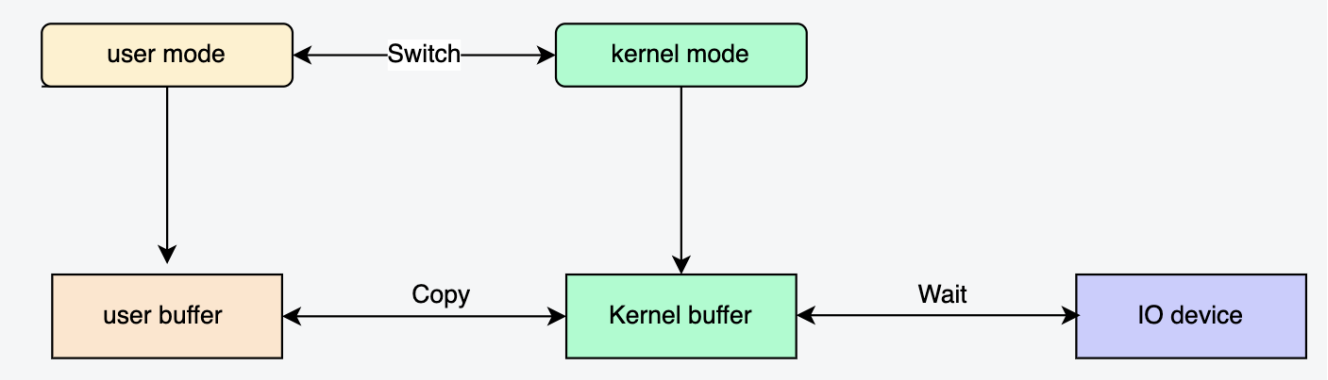
```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```
//  返回值:成功返回新分配的文件描述符，出错返回-1并设置errno
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);

// 返回值:成功返回0，出错返回-1并设置errno
int close(int fd);

// 返回值:成功返回读取的字节数，出错返回-1并设置errno，如果在调read之前已到达文件末尾，则这次read返回0
ssize_t read(int fd, void *buf, size_t count);

// 返回值:成功返回写入的字节数，出错返回-1并设置errno
ssize_t write(int fd, const void *buf, size_t count);
```

在打开文件时可以指定为，只读，只写，读写等权限，以及阻塞或者非阻塞操作等；具体通过open函数的flags 参数指定 。这里以打开一个读写文件为例，同时定义了写文件的方式为追加写，以及使用直接IO模式操作文件，具体什么是直接IO下文会细述。
open("/path/to/file", O_RDWR|O_APPEND|O_DIRECT);flags 可选参数如下：

Flag 参数	含义
O_CREATE	创建文件时，如果文件存在则出错返回
O_EXCL	如果同时指定了O_CREAT，并且文件已存在，则出错返回。
O_TRUC	把文件截断成0
O_RDONLY	只读
O_WRONLY	只写
O_RDWR	读写
O_APPEND	追加
O_NONBLOCK	非阻塞标记
O_SYNC	每次读写都等待物理IO操作完成
O_DIRECT	提供最直接IO支持

通常读写操作的数据首先从用户缓冲区进入内核缓冲区，然后由内核缓冲区完成与IO设备的同步：

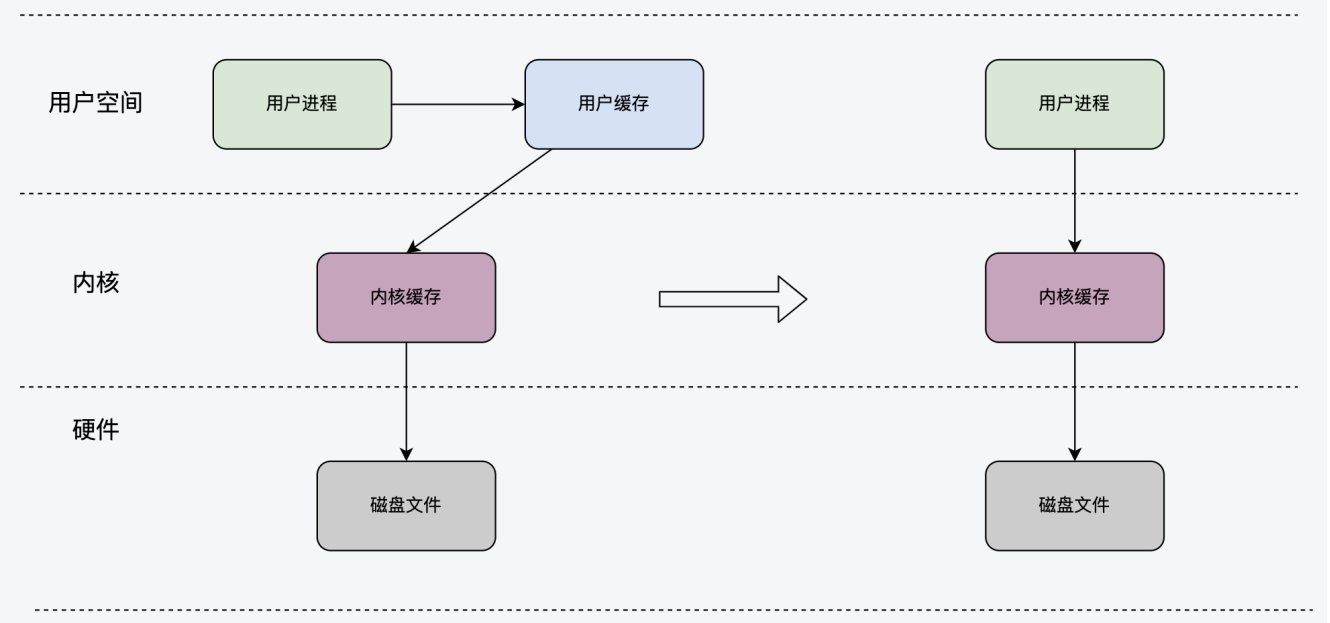


2.2 Mmap

```
// 成功执行时，mmap()返回被映射区的指针。失败时，mmap()返回MAP_FAILED[其值为(void *)-1],
// error被设为以下的某个值:
```

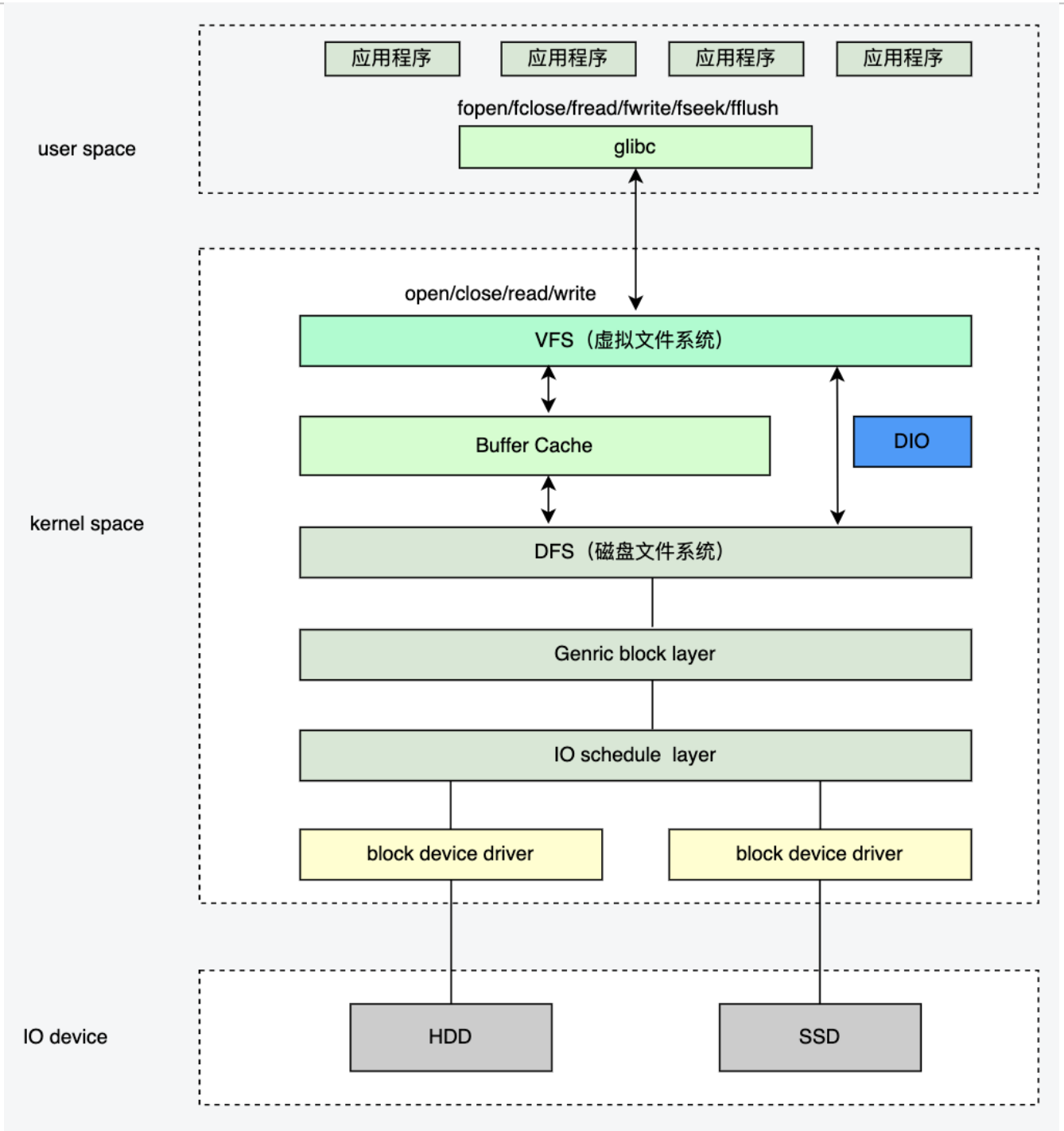
```
// 1 EACCES：访问出错
// 2 EAGAIN：文件已被锁定，或者太多的内存已被锁定
// 3 EBADF：fd不是有效的文件描述词
// 4 EINVAL：一个或者多个参数无效
// 5 ENFILE：已达到系统对打开文件的限制
// 6 ENODEV：指定文件所在的文件系统不支持内存映射
// 7 ENOMEM：内存不足，或者进程已超出最大内存映射数量
// 8 EPERM：权能不足，操作不允许
// 9 ETXTBSY：已写的方式打开文件，同时指定MAP_DENYWRITE标志
//10 SIGSEGV：试着向只读区写入
//11 SIGBUS：试着访问不属于进程的内存区
void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t of
fset);

// 成功执行时，munmap()返回0。失败时，munmap返回-1，error返回标志和mmap一致；
// 该调用在进程地址空间中解除一个映射关系，addr是调用mmap()时返回的地址，len是映射
区的大小；
int munmap( void * addr, size_t len )

// 进程在映射空间的对共享内容的改变并不直接写回到磁盘文件中，往往在调用munmap（）后
才执行该操作。
// 如果期望内存的数据变化能够立刻反应到磁盘上，可以通过调用msync()实现。
int msync( void *addr, size_t len, int flags )
```

Mmap 是一种内存映射方法，通过将文件映射到内存的某个地址空间上，在对该地址空间的读写操作时，会触发相应的缺页异常以及脏页回写操作，从而实现文件数据的读写操作；



2.3 直接IO

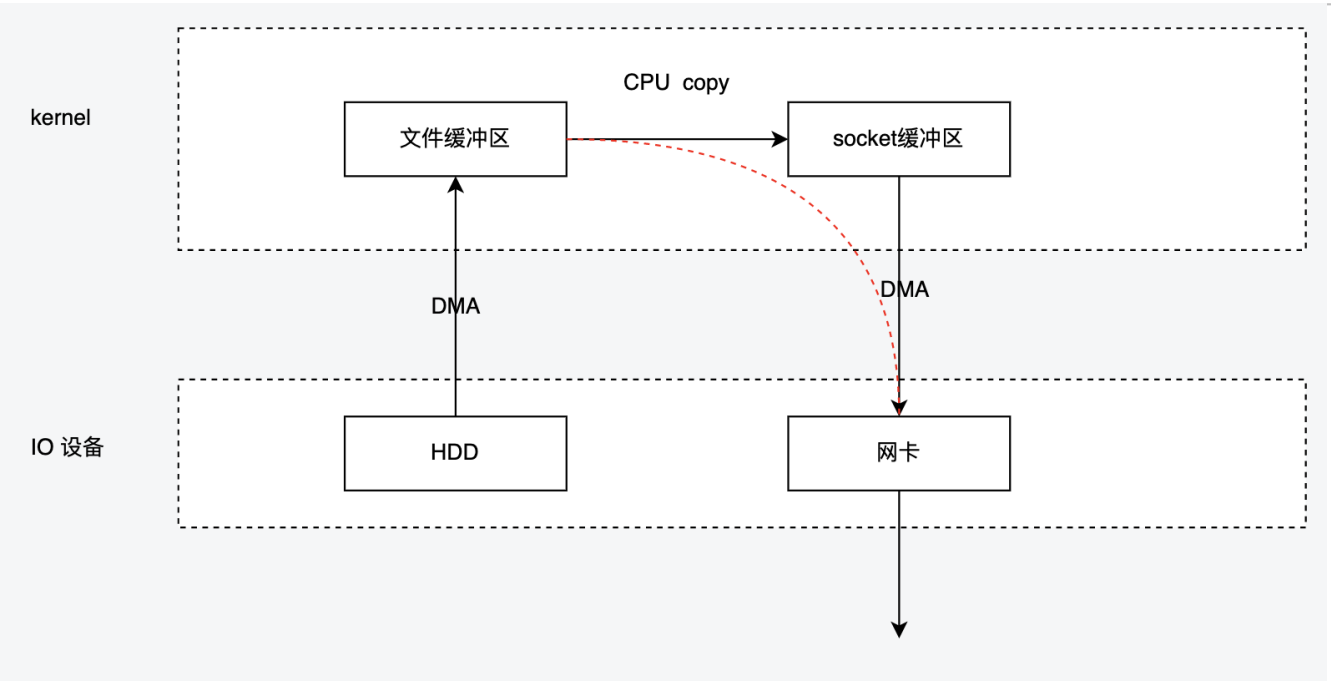
直接IO的方式比较简单，直接上文提及的open函数入参中指定 O_DIRECT 即可，相比普通IO操作，略过了内核的缓冲区直接操作下一层的文件文件。该操作比较底层，相比普通的文件读写少了一次数据复制，一般需要结合用户态缓存来使用；下图所示为 DIO 透过 buffer层直接操作磁盘文件系统：



2.4 sendFile

严格来讲，sendfile 并不提供完整的读写能力，仅用于加速读取数据到网络的能力，由于数据不经过用户空间，因此无法对数据进行二次处理，也就是说从磁盘中读出来原封不动的发给网卡，下图展示了sendFile 的工作流程，

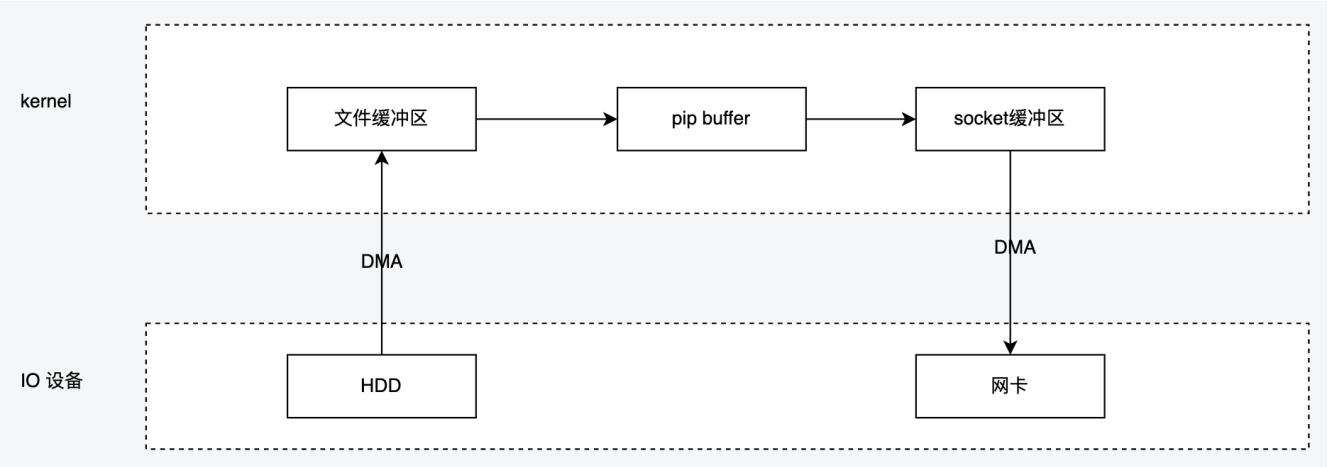
- 数据首先以DMA的方式从磁盘上读取到内核的文件缓冲区，
- 然后再从文件缓冲区读取到了socket的缓冲区，该过程由CPU负责完成。
- 接着网卡再以DMA的方式从socket缓冲区 拷贝到自己网卡缓冲区，然后进行发送



Linux 内核2.4 版本以后对 sendFile 进行了进一步优化，提供了带有 scatter/gather 的 sendfile 操作，将仅有一次的CPU参与copy 环节去掉，该操作需要网卡硬件的支持。其原理就是在内核空间 Read Buffer 和 Socket Buffer 不做数据复制，而是将 Read Buffer 的内存地址、偏移量记录到相应的 Socket Buffer 中。其本质和虚拟内存的解决方法思路一致，就是内存地址的记录。

2.5 splice

splice 调用和 sendfile 很相似，应用程序必须拥有两个已经打开的文件描述符，一个表示输入设备，一个表示输出设备。splice允许任意两个文件互相连接，而并不只是文件与 socket 进行数据传输。对于从一个文件描述符发送数据到 socket 这种特例来说，简化为使用 sendfile 系统调用，splice 适用范围更广且不需要硬件支持， sendfile 是 splice 的一个子集。



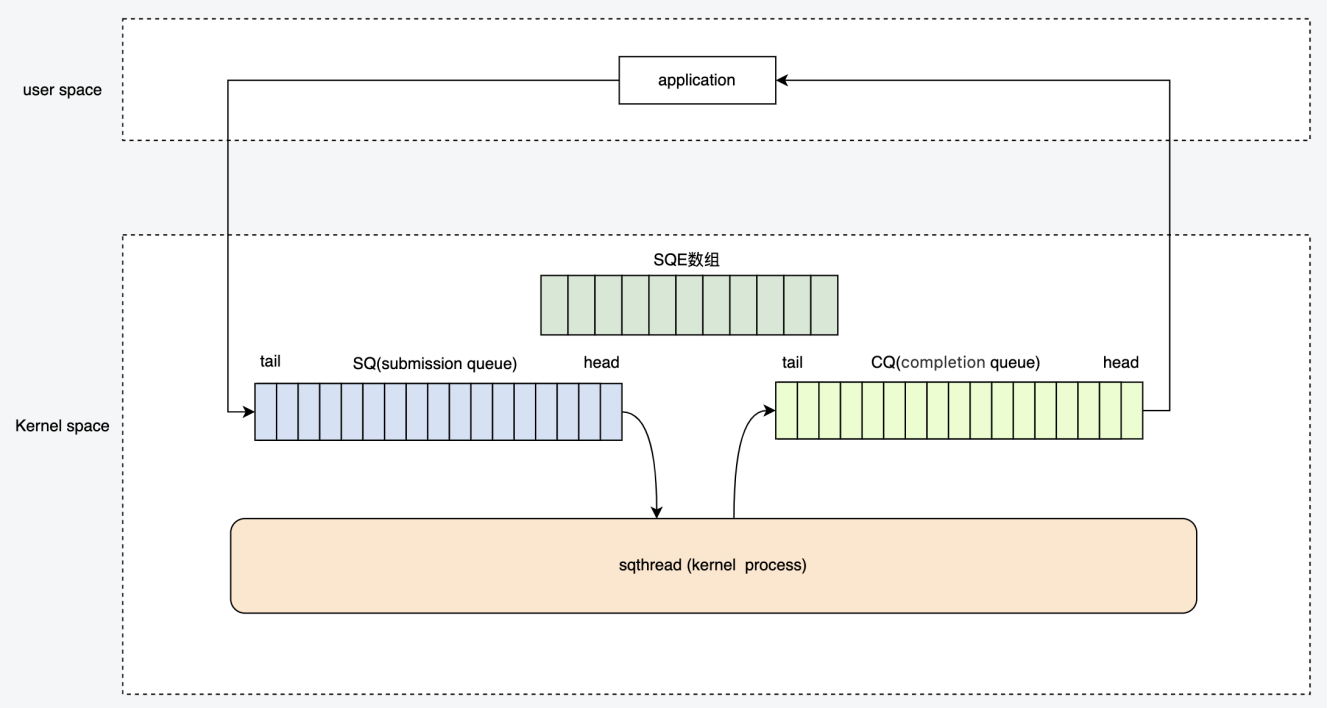
- 用户进程调用 pipe()陷入内核态；创建匿名单向管道 pipe() 返回，从内核态切换回用户态；
- 用户进程调用 splice()从用户态陷入内核态，DMA 控制器将数据从硬盘拷贝到内核缓冲区，从管道的写入端"拷贝"进管道，splice() 返回，从内核态切换为用户态；
- 用户进程再次调用 splice()，从用户态陷入内核态，内核把数据从管道的读取端拷贝到socket缓冲区，DMA 控制器将数据从 socket 缓冲区拷贝到网卡，splice() 返

回，上下文从内核态切换回用户态。

3.IO_URING是什么

`io_uring`是 Linux 提供的一个异步非阻塞 I/O 接口，他既能支持磁盘IO也能支持网络IO，只是存储IO支持的比较早较为成熟。IO_URING的使用需要较高的linux 内核版本，一般建议5.12 版本以后。下面会分别从存储和网络两个角度来介绍IO_URING 。

3.1 IO_URING 架构



- 应用程序提交的IO 请求会直接进入submission queue 队列的尾部，内核进程会不断的从SQ 队列的头部消费请求
- 内核处理完的SQ后会更新CQ tail 部分 ，应用程序读取到CQ 的head时，会更新CQ 的head
- SQ 中的任务称之为 SQE（entry），CQ中的任务称之为CQE

3.2 系统调用API

```
// 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。
// 参数p 有两个作用：
// 1.作为入参：应用用来配置 io_uring 的一些行为
// 2.作为出参：内核返回的 SQ/CQ 地址信息等也通过它带回来。
int io_uring_setup(u32 entries, struct io_uring_params *p);


// 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, unsig
ned int nr_args);


// 用于初始化和完成I/O，使用共享的 SQ 和 CQ。 单次调用同时提交新的 I/O 请求和等待
```

```
 I/O 完成操作
// fd 是 io_uring_setup() 返回的文件描述符;
// to_submit 指定了 SQ 中提交的 I/O 数量;
// 默认模式下如果指定了 min_complete，会等待这个数量的 I/O 事件完成再返回;
// 轮询模式（2种）：
//   0：要求内核返回当前以及完成的所有 events，无阻塞;
//   非0：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。
int io_uring_enter(unsigned int fd, unsigned int to_submit, unsigned int min_complete, unsigned int flags, sigset_t *sig);
```

3.3 三种工作模式

3.3.1中断驱动模式：

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。也可以通过 min_complete 来睡在 enter 方法上，等待完成事件到达 ；

3.3.2 轮询模式

相比中断驱动方式，这种方式延迟更低，但是会消耗更多的CPU，应用线程需要不断的调用 enter 函数，然后陷入内核态后持续地 polling，等到一个 min_complete 到达。但是注意的是此时 polling 关注的是**完成事件。3.3.3 内核轮询模式**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(是否有新的SQE提交)。使用这种模式应用无需切到到内核态 就能触发（issue）I/O 操作。应用线程通过mmap 机制更新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，内核线程是不会 sleep 的。在日常的使用中一般建议选择后两种轮训模式，用户线程轮存在用户态到内核态的切换，相比内核轮询存在一定的性能损耗；io_uring 之所以能达到超高性能的原因主要在以下几个方面：

1. *Mmap 机制减少了 内存复制*
2. *内核轮询模式下，没有用户态和内核态的切换降低了损耗*
3. *基于SQ和CQ 机制下的数据竞争消除，即没有并发竞争损耗*

3.4 liburing

io_uring的核心系统调用只有三个，但使用起来较为复杂，开发者在io_uring 之上封装了新的liburing 库，简化使用。

```
// io_uring 结构体中包含需要使用到的 SQ和CQ ，以及需要关联的文件FD， 和相关的配置参数falgs;
struct io_uring {
        struct io_uring_sq sq;
        struct io_uring_cq cq;
        unsigned flags;
        int ring_fd;
};

struct io_uring_sq {
        unsigned *khead;
        unsigned *ktail;
```

```
        unsigned *kring_mask;
        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;
};

struct io_uring_cq {
        unsigned *khead;
        unsigned *ktail;
        unsigned *kring_mask;
        unsigned *kring_entries;
        unsigned *koverflow;
        struct io_uring_cqe *cqes;

        size_t ring_sz;
        void *ring_ptr;
};

// 用户初始化 io_uring。该方法中包含了内存空间的初始化以及mmap 调用,entries：队列深度
int io_uring_queue_init(unsigned entries, struct io_uring *ring, unsigned flags);

// 为了提交IO请求，需要获取里面queue的一个空闲项
struct io_uring_sqe *io_uring_get_sqe(struct io_uring *ring);

// 非系统调用，准备阶段，和libaio封装的io_prep_writev一样
void io_uring_prep_writev(struct io_uring_sqe *sqe, int fd,const struct iovec *iovecs, unsigned nr_vecs, off_t offset)

// 非系统调用，准备阶段，和libaio封装的io_prep_readv一样
void io_uring_prep_readv(struct io_uring_sqe *sqe, int fd, const struct iovec *iovecs, unsigned nr_vecs, off_t offset)

// 提交sq的entry，不会阻塞等到其完成，内核在其完成后会自动将sqe的偏移信息加入到cq，在提交时需要加锁
int io_uring_submit(struct io_uring *ring);

// 提交sq的entry，阻塞等到其完成，在提交时需要加锁。
int io_uring_submit_and_wait(struct io_uring *ring, unsigned wait_nr);

// 非系统调用 遍历时，可以获取cqe的data
void *io_uring_cqe_get_data(const struct io_uring_cqe *cqe)

// 清理io_uring
void io_uring_queue_exit(struct io_uring *ring);
```

liburing github地址 ： https://github.com/axboe/liburing

3.5 使用方式

3.5.1 读取文件

1. 调用 io_uring_queue_init 初始化
2. 获取一个空 SQE用于提交任务
3. io_uring_prep_readv 方法填充SQE 任务内容
4. io_uring_submit 提交SQE
5. io_uring_wait_cqe 获取已完成的CQE
6. io_uring_cqe_seen 更新CQ 队列的head ,避免CQE被重复处理
7. io_uring_queue_exit 退出 io_uring

下面是liburing github 上的example 代码适当精简后的代码

```
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include "liburing.h"

#define QD	4
int main(int argc, char *argv[]){
	struct io_uring ring;
	int i, fd, ret, pending, done;
	struct io_uring_sqe *sqe;
	struct io_uring_cqe *cqe;
	struct iovec *iovecs;
	struct stat sb;
	ssize_t fsize;
	off_t offset;
	void *buf;

	ret = io_uring_queue_init(QD, &ring, 0);
	if (ret < 0) {
		fprintf(stderr, "queue_init: %s\n", strerror(-ret)); return
1;
	}
	fd = open(argv[1], O_RDONLY | O_DIRECT);
	fsize = 0;
	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;
		iovecs[i].iov_len = 4096;
		fsize += 4096;
	}

	offset = 0;
	i = 0;
	do {
		sqe = io_uring_get_sqe(&ring);
		if (!sqe)  break;
```

```
                io_uring_prep_readv(sqe, fd, &iovecs[i], 1, offset);
                offset += iovecs[i].iov_len;
                i++;
                if (offset > sb.st_size) break;
        } while (1);

        ret = io_uring_submit(&ring);
        if (ret < 0) {
                fprintf(stderr, "io_uring_submit: %s\n", strerror(-ret)); re
turn 1;
        } else if (ret != i) {
                fprintf(stderr, "io_uring_submit submitted less %d\n", ret);
return 1;
        }

        done = 0;
        pending = ret;
        fsize = 0;
        for (i = 0; i < pending; i++) {
                ret = io_uring_wait_cqe(&ring, &cqe);
                if (ret < 0) {
                        fprintf(stderr, "io_uring_wait_cqe: %s\n", strerror
(-ret));return 1;
                }
                done++;
                ret = 0;
                if (cqe->res != 4096 && cqe->res + fsize != sb.st_size) {
                        fprintf(stderr, "ret=%d, wanted 4096\n", cqe->res);
                        ret = 1;
                }
                fsize += cqe->res;
                io_uring_cqe_seen(&ring, cqe);
                if (ret) break;
        }
        printf("Submitted=%d, completed=%d, bytes=%lu\n", pending, done, (un
signed long) fsize);
        close(fd);
        io_uring_queue_exit(&ring);
        return 0;
}
```

3.5.2 网络服务

网络服务这里直接参考 Github 地址：GitHub - frevib/io_uring-echo-server: io_uring echo server

```
#include <errno.h>
#include <fcntl.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include <sys/poll.h>
#include <sys/socket.h>
#include <unistd.h>
```

```c
#include "liburing.h"

#define MAX_CONNECTIONS     4096
#define BACKLOG             512
#define MAX_MESSAGE_LEN     2048
#define BUFFERS_COUNT       MAX_CONNECTIONS

void add_accept(struct io_uring *ring, int fd, struct sockaddr *client_addr,
socklen_t *client_len, unsigned flags);
void add_socket_read(struct io_uring *ring, int fd, unsigned gid, size_t siz
e, unsigned flags);
void add_socket_write(struct io_uring *ring, int fd, __u16 bid, size_t size,
unsigned flags);
void add_provide_buf(struct io_uring *ring, __u16 bid, unsigned gid);

enum {
    ACCEPT,
    READ,
    WRITE,
    PROV_BUF,
};

typedef struct conn_info {
    __u32 fd;
    __u16 type;
    __u16 bid;
} conn_info;

char bufs[BUFFERS_COUNT][MAX_MESSAGE_LEN] = {0};
int group_id = 1337;

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Please give a port number: ./io_uring_echo_server [port]\n"
);
        exit(0);
    }

    // some variables we need
    int portno = strtol(argv[1], NULL, 10);
    struct sockaddr_in serv_addr, client_addr;
    socklen_t client_len = sizeof(client_addr);

    // setup socket
    int sock_listen_fd = socket(AF_INET, SOCK_STREAM, 0);
    const int val = 1;
    setsockopt(sock_listen_fd, SOL_SOCKET, SO_REUSEADDR, &val, sizeof(val));

    memset(&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(portno);
    serv_addr.sin_addr.s_addr = INADDR_ANY;

    // bind and listen
    if (bind(sock_listen_fd, (struct sockaddr *)&serv_addr, sizeof(serv_add
r)) < 0) {
        perror("Error binding socket...\n");
```

```
        exit(1);
    }
    if (listen(sock_listen_fd, BACKLOG) < 0) {
        perror("Error listening on socket...\n");
        exit(1);
    }
    printf("io_uring echo server listening for connections on port: %d\n", p
ortno);

    // initialize io_uring
    struct io_uring_params params;
    struct io_uring ring;
    memset(&params, 0, sizeof(params));

    if (io_uring_queue_init_params(2048, &ring, &params) < 0) {
        perror("io_uring_init_failed...\n");
        exit(1);
    }

    // check if IORING_FEAT_FAST_POLL is supported
    if (!(params.features & IORING_FEAT_FAST_POLL)) {
        printf("IORING_FEAT_FAST_POLL not available in the kernel, quitin
g...\n");
        exit(0);
    }

    // check if buffer selection is supported
    struct io_uring_probe *probe;
    probe = io_uring_get_probe_ring(&ring);
    if (!probe || !io_uring_opcode_supported(probe, IORING_OP_PROVIDE_BUFFER
S)) {
        printf("Buffer select not supported, skipping...\n");
        exit(0);
    }
    free(probe);

    // register buffers for buffer selection
    struct io_uring_sqe *sqe;
    struct io_uring_cqe *cqe;

    sqe = io_uring_get_sqe(&ring);
    io_uring_prep_provide_buffers(sqe, bufs, MAX_MESSAGE_LEN, BUFFERS_COUNT,
group_id, 0);

    io_uring_submit(&ring);
    io_uring_wait_cqe(&ring, &cqe);
    if (cqe->res < 0) {
        printf("cqe->res = %d\n", cqe->res);
        exit(1);
    }
    io_uring_cqe_seen(&ring, cqe);

    // add first accept SQE to monitor for new incoming connections
    add_accept(&ring, sock_listen_fd, (struct sockaddr *)&client_addr, &clie
nt_len, 0);

    // start event loop
    while (1) {
```

```
        io_uring_submit_and_wait(&ring, 1);
        struct io_uring_cqe *cqe;
        unsigned head;
        unsigned count = 0;

        // go through all CQEs
        io_uring_for_each_cqe(&ring, head, cqe) {
            ++count;
            struct conn_info conn_i;
            memcpy(&conn_i, &cqe->user_data, sizeof(conn_i));

            int type = conn_i.type;
            if (cqe->res == -ENOBUFS) {
                fprintf(stdout, "bufs in automatic buffer selection empty, t
his should not happen...\n");
                fflush(stdout);
                exit(1);
            } else if (type == PROV_BUF) {
                if (cqe->res < 0) {
                    printf("cqe->res = %d\n", cqe->res);
                    exit(1);
                }
            } else if (type == ACCEPT) {
                int sock_conn_fd = cqe->res;
                // only read when there is no error, >= 0
                if (sock_conn_fd >= 0) {
                    add_socket_read(&ring, sock_conn_fd, group_id, MAX_MESSA
GE_LEN, IOSQE_BUFFER_SELECT);
                }

                // new connected client; read data from socket and re-add ac
cept to monitor for new connections
                add_accept(&ring, sock_listen_fd, (struct sockaddr *)&client
_addr, &client_len, 0);
            } else if (type == READ) {
                int bytes_read = cqe->res;
                int bid = cqe->flags >> 16;
                if (cqe->res <= 0) {
                    // read failed, re-add the buffer
                    add_provide_buf(&ring, bid, group_id);
                    // connection closed or error
                    close(conn_i.fd);
                } else {
                    // bytes have been read into bufs, now add write to sock
et sqe
                    add_socket_write(&ring, conn_i.fd, bid, bytes_read, 0);
                }
            } else if (type == WRITE) {
                // write has been completed, first re-add the buffer
                add_provide_buf(&ring, conn_i.bid, group_id);
                // add a new read for the existing connection
                add_socket_read(&ring, conn_i.fd, group_id, MAX_MESSAGE_LEN,
IOSQE_BUFFER_SELECT);
            }
        }

        io_uring_cq_advance(&ring, count);
    }
```

```
}

void add_accept(struct io_uring *ring, int fd, struct sockaddr *client_addr,
socklen_t *client_len, unsigned flags) {
    struct io_uring_sqe *sqe = io_uring_get_sqe(ring);
    io_uring_prep_accept(sqe, fd, client_addr, client_len, 0);
    io_uring_sqe_set_flags(sqe, flags);

    conn_info conn_i = {
        .fd = fd,
        .type = ACCEPT,
    };
    memcpy(&sqe->user_data, &conn_i, sizeof(conn_i));
}

void add_socket_read(struct io_uring *ring, int fd, unsigned gid, size_t mes
sage_size, unsigned flags) {
    struct io_uring_sqe *sqe = io_uring_get_sqe(ring);
    io_uring_prep_recv(sqe, fd, NULL, message_size, 0);
    io_uring_sqe_set_flags(sqe, flags);
    sqe->buf_group = gid;

    conn_info conn_i = {
        .fd = fd,
        .type = READ,
    };
    memcpy(&sqe->user_data, &conn_i, sizeof(conn_i));
}

void add_socket_write(struct io_uring *ring, int fd, __u16 bid, size_t messa
ge_size, unsigned flags) {
    struct io_uring_sqe *sqe = io_uring_get_sqe(ring);
    io_uring_prep_send(sqe, fd, &bufs[bid], message_size, 0);
    io_uring_sqe_set_flags(sqe, flags);

    conn_info conn_i = {
        .fd = fd,
        .type = WRITE,
        .bid = bid,
    };
    memcpy(&sqe->user_data, &conn_i, sizeof(conn_i));
}

void add_provide_buf(struct io_uring *ring, __u16 bid, unsigned gid) {
    struct io_uring_sqe *sqe = io_uring_get_sqe(ring);
    io_uring_prep_provide_buffers(sqe, bufs[bid], MAX_MESSAGE_LEN, 1, gid, b
id);

    conn_info conn_i = {
        .fd = 0,
        .type = PROV_BUF,
    };
    memcpy(&sqe->user_data, &conn_i, sizeof(conn_i));
}
```

4.性能对比

4.1 存储IO

- **Synchronous I/O、 Libaio和IO_uring 特性对比**

SW Overhead	Synchronous I/O	Libaio	IO_uring
System Calls	At least 1 per I/O	At least 2 per I/O	At least 1 per batch, zero when using SQ submission thread. Batching reduces per I/O overhead
Memory Copy	Yes	Yes - SQE/CQE	Zero-copy. Shared SQ& CQ
Context Switches	Yes	Yes	Minimal context switching
Interrupts	Interrupt driven	Interrupt driven	Supports both Interrupts and polling I/O
Blocking I/O	Synchronous	Asynchronous	Asynchronous
Buffered I/O	Yes	No	Yes

- **io_uring和spdk的特性对比**

SPDK 全名 Storage Performance Development Kit，是一种存储性能开发套件 。针对于支持nvme协议的SSD设备。是一种高性能的解决方案。

项目	io_uring	spdk
驱动程序	内核态驱动程序有锁	用户态驱动程序、无锁、轮询、线程绑定
run_to_completion	非 rtc 模型，可能会有上下文切换	rtc 模型，单线程撸到底
内存管理	mmu、4k	2MB 大页
提交任务有无锁	无锁	无锁
系统调用	可有可无	无系统调用
用户内核态切换	轻量级的	无内核切换
poll 模型	可选	polling

!

- **io_uring和spdk的性能对比**

- 3d xpoint, 4k random read

Interface	QD	Polled	Latency	IOPS
io_uring	1	0	9.5usec	77K
io_uring	2	0	8.2usec	183K
io_uring	4	0	8.4usec	383K
io_uring	8	0	13.3usec	449K
libaio	1	0	9.7usec	74K
libaio	2	0	8.5usec	181K
libaio	4	0	8.5usec	373K
libaio	8	0	15.4usec	402K
io_uring	1	1	6.1usec	139K
io_uring	2	1	6.1usec	272K
io_uring	4	1	6.3usec	519K
io_uring	8	1	11.5usec	592K
spdk	1	1	6.1usec	151K
spdk	2	1	6.2usec	293K
spdk	4	1	6.7usec	536K
spdk	8	1	12.6usec	586K

非polling模式，io_uring相比libaio提升不是很明显；在polling模式下，io_uring能与spdk接近，甚至在queue depth较高时性能更好，性能超越libaio。

- Peak IOPS, 512b random read

Interface	QD	Polled	Latency	IOPS
io_uring	4	1	6.8usec	513K
io_uring	8	1	8.7usec	829K
io_uring	16	1	13.1usec	1019K
io_uring	32	1	20.6usec	1161K
io_uring	64	1	32.4usec	1244K
spdk	4	1	6.8usec	549K
spdk	8	1	8.6usec	865K
spdk	16	1	14.0usec	1105K
spdk	32	1	25.0usec	1227K
spdk	64	1	47.3usec	1251K

在queue depth较低时有约7%的差距，但在queue depth较高时基本接近。

- Peak per-core, multiple devices, 4k random read

Interface	QD	Polled	IOPS
io_uring	128	1	1620K
libaio	128	0	608K
spdk	128	1	1739K

对比结论：

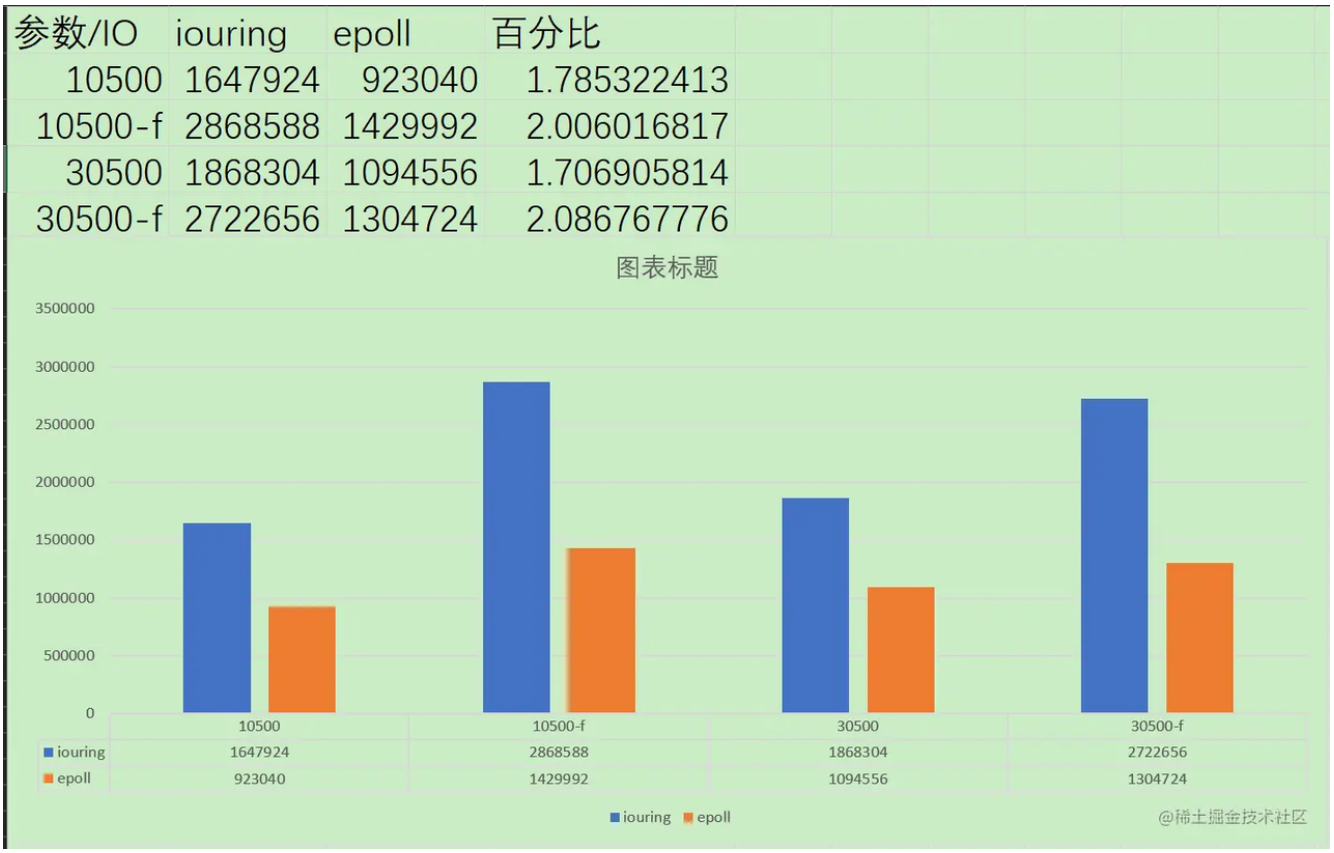
****io_uring在非polling模式下，相比libaio，性能提升不是非常显著。*io_uring在polling模式下，性能提升显著，与spdk接近，在队列深度较高时性能更好。***

4.2 网络IO

- **Epoll 性能对比**

与epoll的性能对比差异还是很大的，参考这篇文章的数据 https://juejin.cn/post/7074212680071905311测试环境：wsl2，内核版本5.10.60.1，发行版为Debian硬件：I5-9400，16gDDR4使用webbench进行简易测试，模拟10500、30500台客户端，持续时间为5s，分别在正常访问和不等待返回两种模式下进行测试，两个客户端均关闭日志记录，epoll开启双ET模式，比较每分钟发送页面数，结果如下：

参数/IO	iouring	epoll	百分比
10500	1647924	923040	1.785322413
10500-f	2868588	1429992	2.006016817
30500	1868304	1094556	1.706905814
30500-f	2722656	1304724	2.086767776



对比结论：

毋庸置疑，碾压性的结果。

5.总结

得益于精妙的设计，io_uring的性能基本超越linux 内核以往任何软件层面的IO解决方案，达到了与硬件级解决方案媲美的性能。io_uring 需要较高版本的内核支持，目前还没有大面积普及，但可以预料他是 linux 内核 IO未来的核心发展方向。