腾讯云

uringlet

a new way to do async IO in io_uring

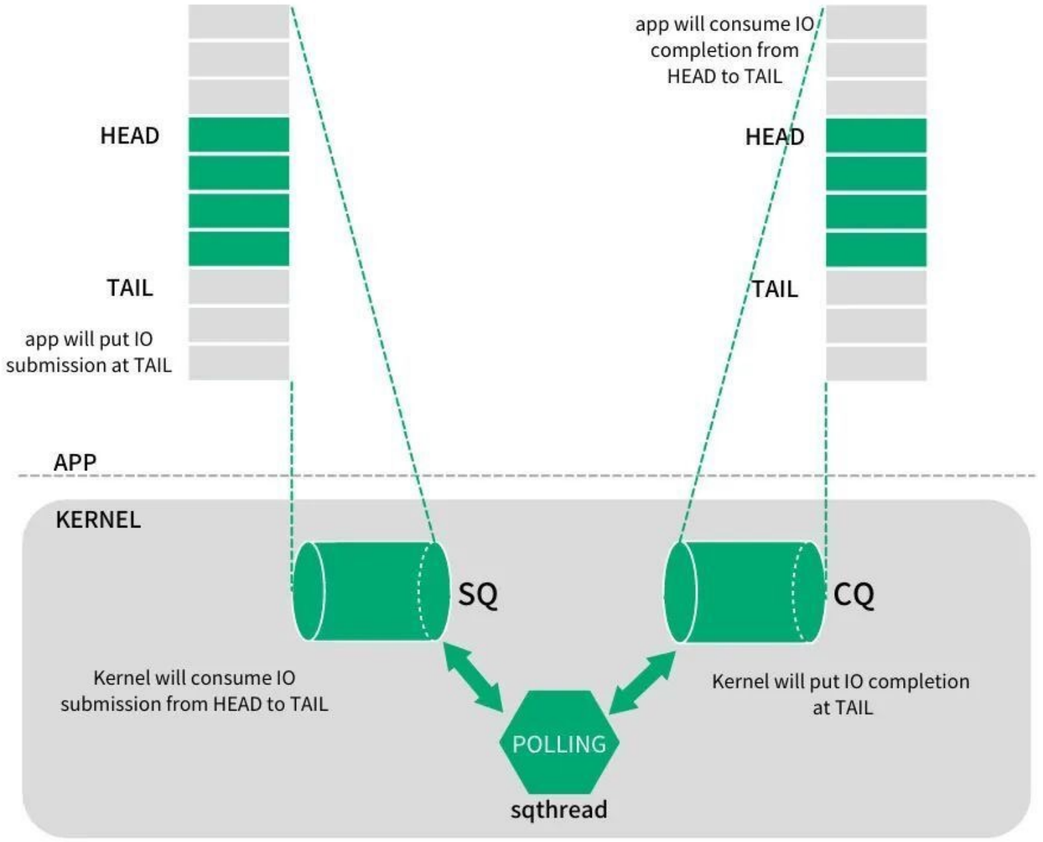
徐浩
hao.xu@linux.dev

腾讯云

01.io_uring介绍

快速发展的Linux内核最新异步IO框架

io_uring架构


app will consume IO
completion from
HEAD to TAIL
HEAD
TAIL
HEAD
TAIL
app will put IO
submission at TAIL
APP
KERNEL
SQ
CQ
Kernel will consume IO
submission from HEAD to TAIL
Kernel will put IO completion
at TAIL
POLLING
sqthread

io_uring API

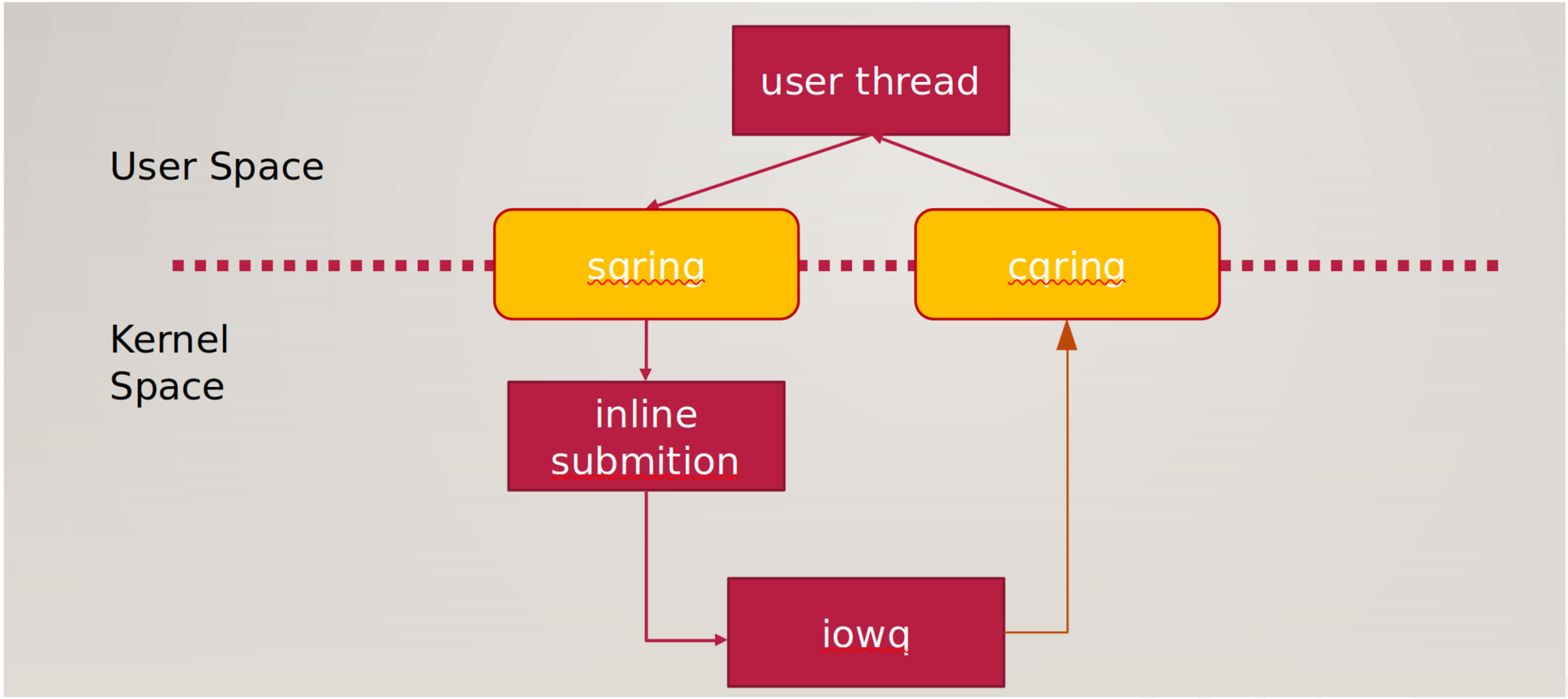
```
int io_uring_setup(u32 entries, struct io_uring_params *p);
```

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

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

```c
int main(int argc, char *argv[])
{
        struct io_uring ring;
        struct io_uring_cqe *cqe;
        struct io_uring_sqe *sqe;
        int ret;
        const int sqring_len = 200;
        const int batch = 100;

        if (argc > 1)
                return 0;

        ret = io_uring_queue_init(sqring_len, &ring, 0);

        for (i = 0; i < batch; i++) {
                sqe = io_uring_get_sqe(ring);
                io_uring_prep_nop(sqe);
        }

        ret = io_uring_submit(ring);

        for (i = 0; i < batch; i++) {
                ret = io_uring_wait_cqe(ring, &cqe);
                if (ret < 0) {
                        fprintf(stderr, "wait completion %d\n", ret);
                        goto err;
                }

                io_uring_cqe_seen(ring, cqe);
        }

        return 0;
}
```

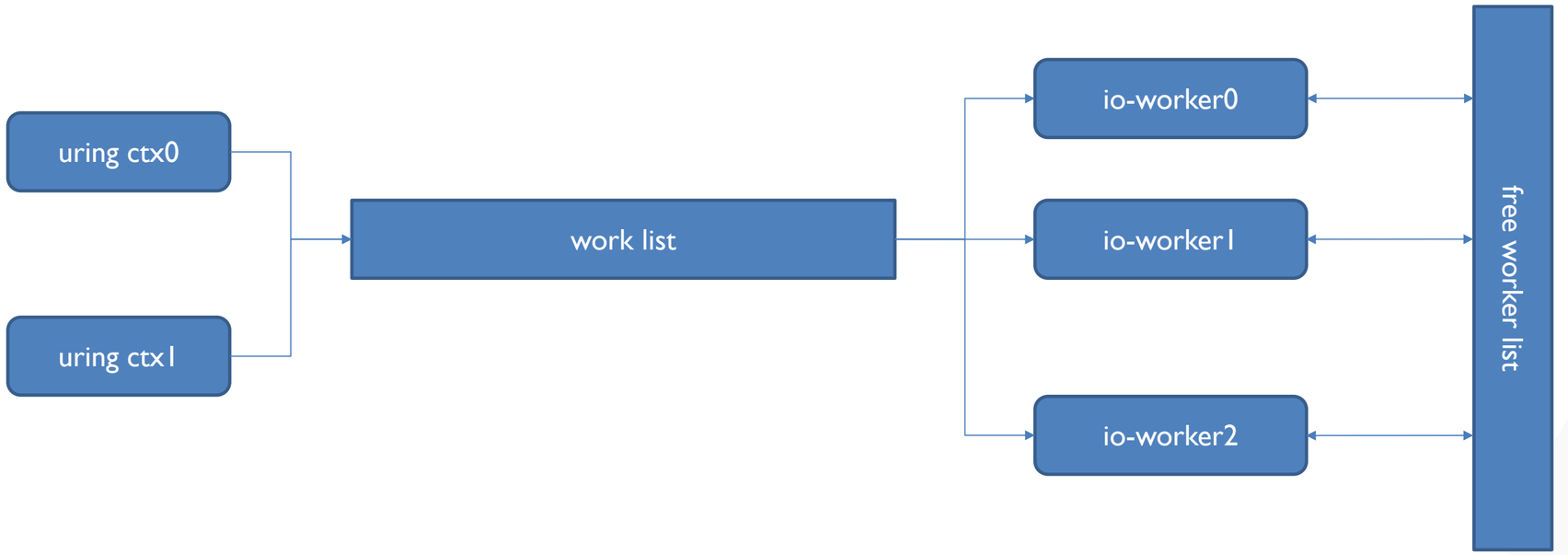
IO请求的生命周期


user thread
User Space
sqring
cqring
Kernel
Space
inline
submition
iowq

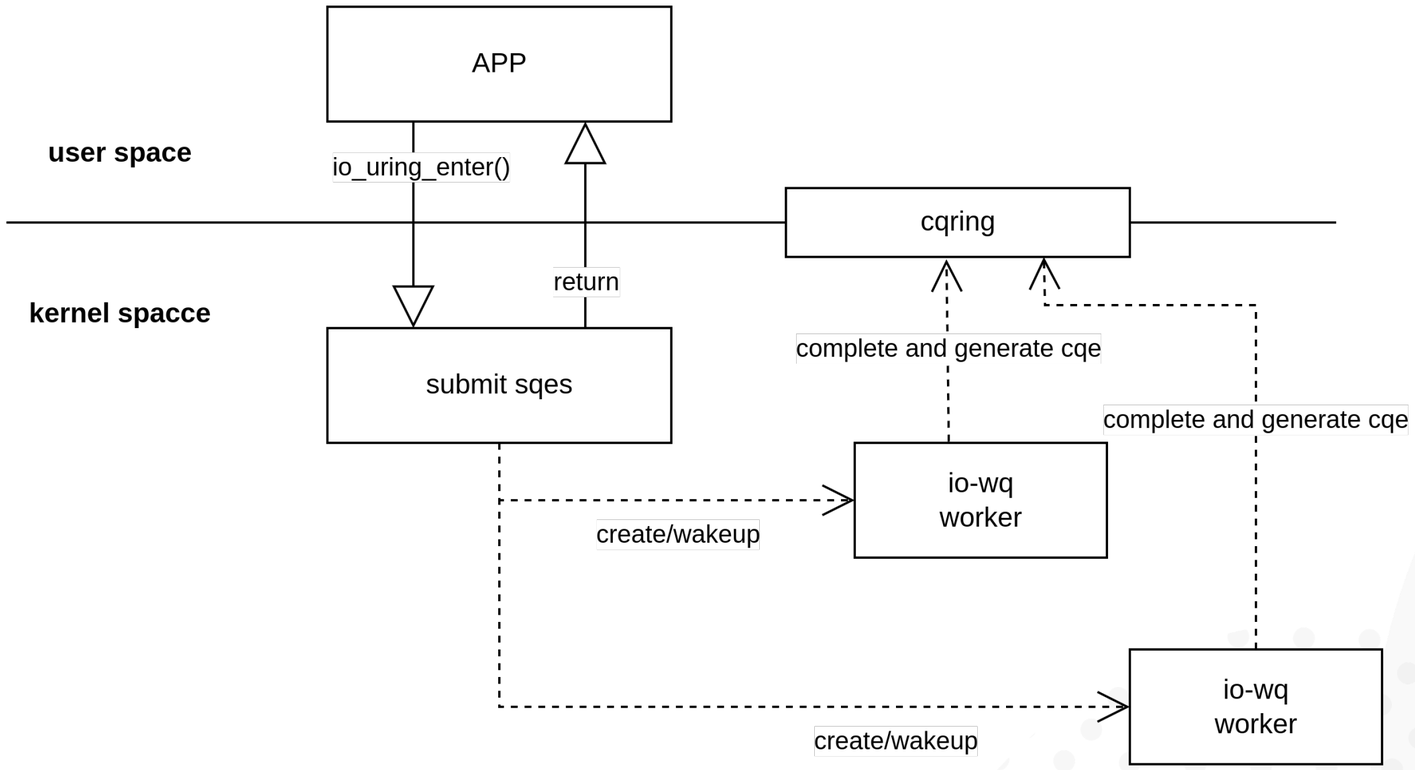
02.io-wq机制

io_uring实现异步IO的关键组件

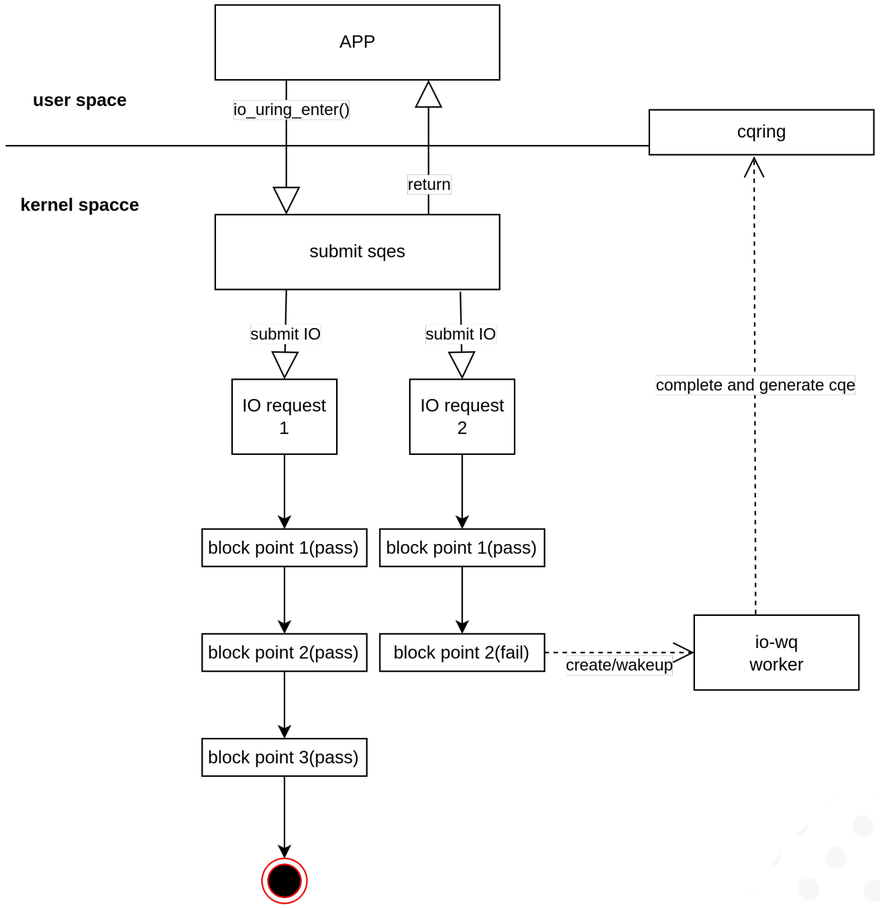
io-wq架构


uring ctx0
uring ctx1
work list
io-worker0
io-worker1
io-worker2
free worker list

利用io-wq实现异步化:初级



利用io-wq实现异步化:改进



03.现有的问题

腾讯云

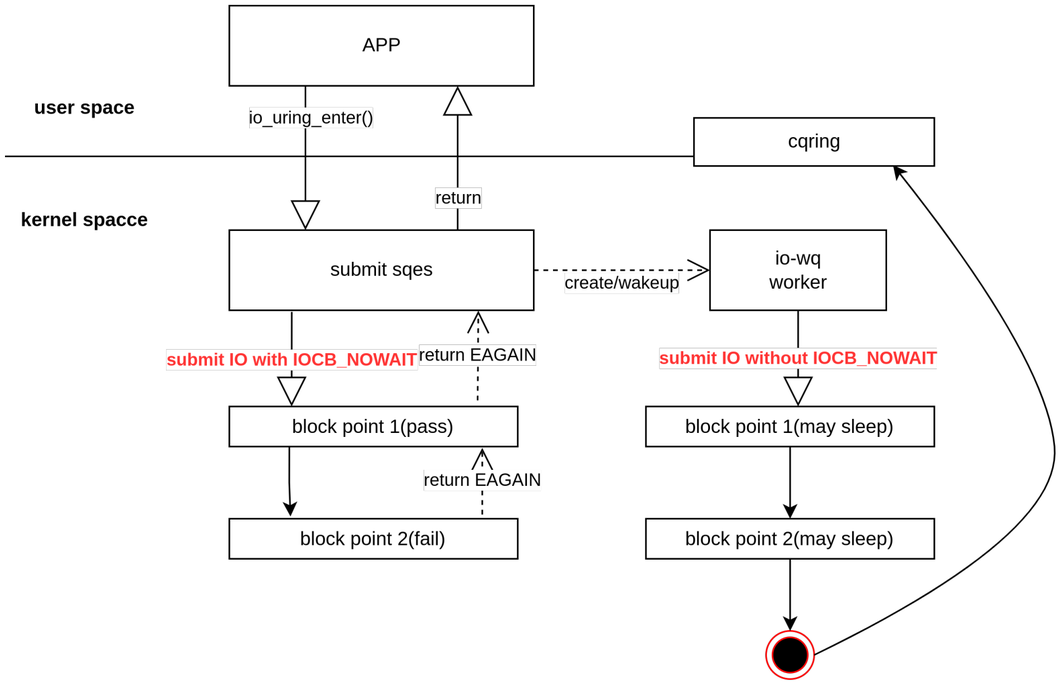
从一段代码开始

```
static ssize_t ext4_dio_read_iter(struct kiocb *iocb, struct iov_iter *to)
{
        ssize_t ret;
        struct inode *inode = file_inode(iocb->ki_filp);

        if (iocb->ki_flags & IOCB_NOWAIT) {
                if (!inode_trylock_shared(inode))
                        return -EAGAIN;
        } else {
                inode_lock_shared(inode);
        }
```

- IOCB_NOWAIT语义
- trylock操作
- -EAGAIN

腾讯云

剖析执行流



- 阻塞前的操作
- 阻塞后的操作
- Io-wq worker的操作

04.可能的解决方案

问题一：从schedule()入手

问题二：？？？

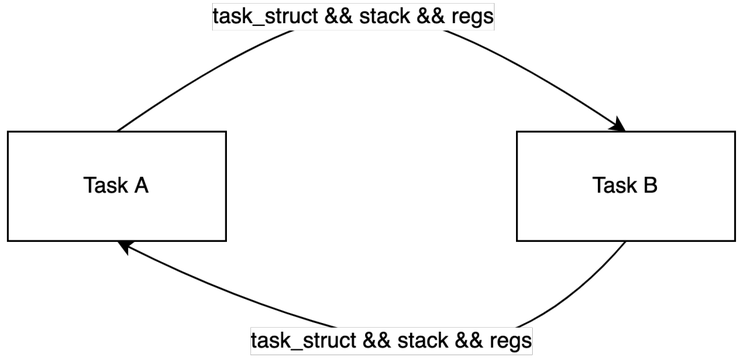
问题二的解决方案

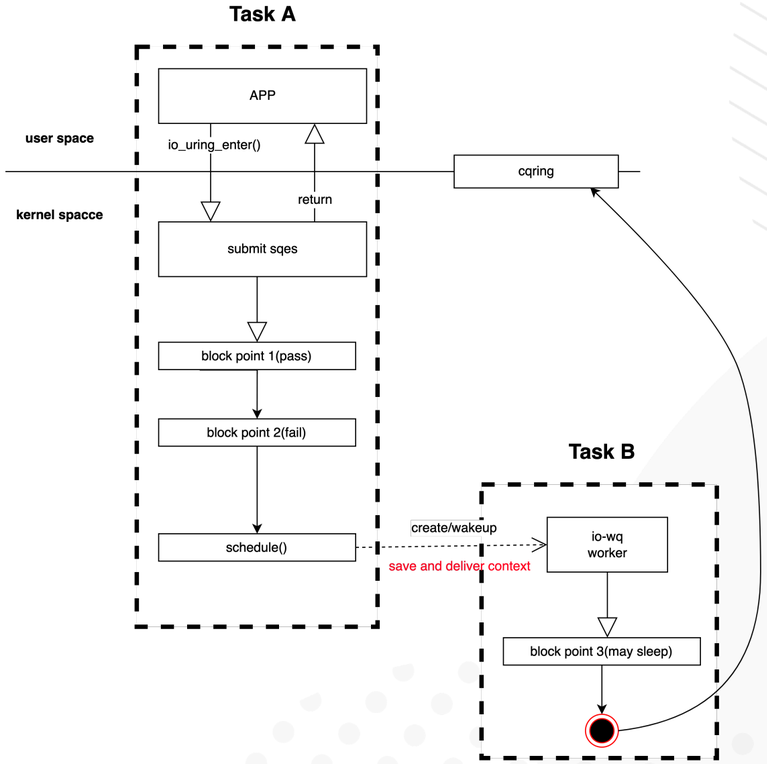
- *如何利用阻塞前的工作？*

  ------→　　保存上下文

- *如何让io-wq worker从阻塞点开始继续执行？*

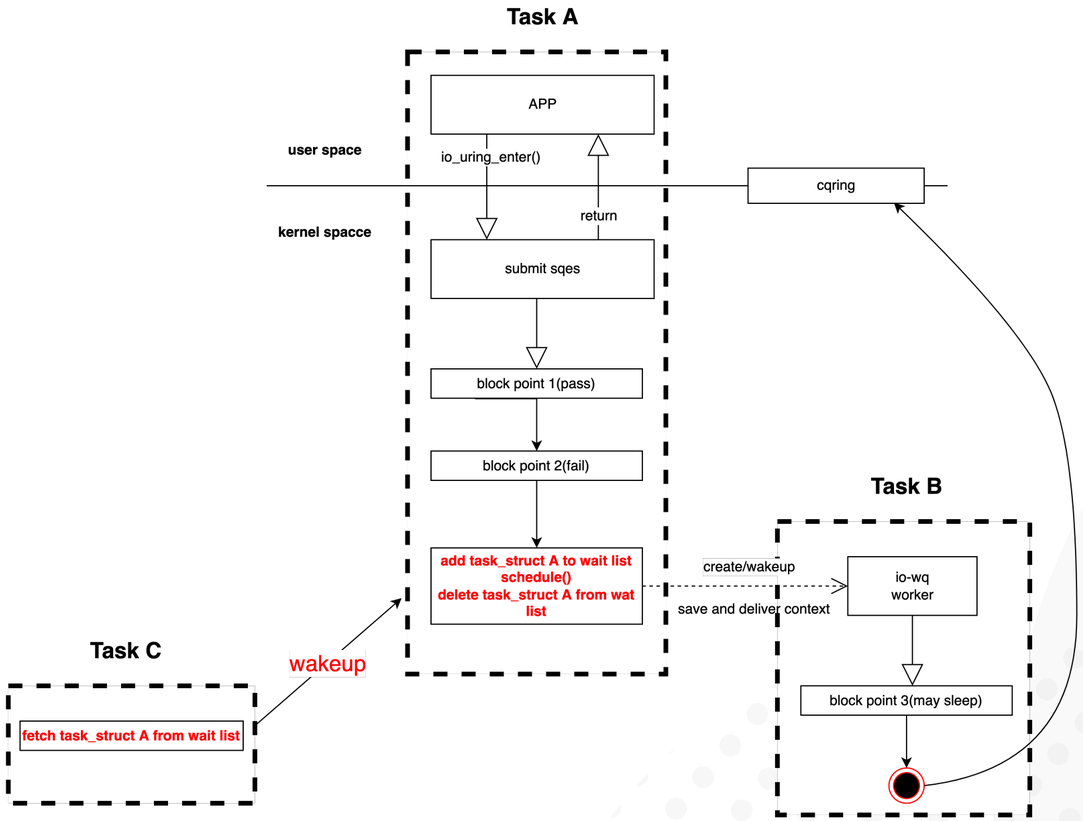
  ------→　　上下文传递





方案的问题

如何唤醒Task B



方案的问题

如何唤醒Task B

- 仍然让Task A阻塞，但不返回用户态
  - 在task A阻塞时在合适的地方加上标记，然后在其完成IO并返回io_uring层时进行判断
- Task B提交其他请求并返回用户态
  - 设置task B的上下文另其提交其他IO请求，修改pt_regs等信息使其返回

```
syscall1(add task to some place X)
do IO through io_uring // (1)
syscall2(fetch task from X, and do something) // (2)
```

05.uringlet

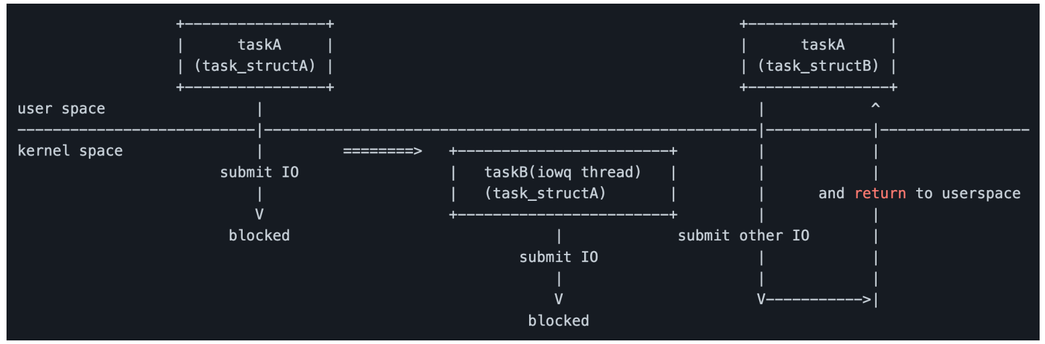
- 关键点：“让taskA继续阻塞，让taskB提交其他IO”
- Hack操作的源头：taskB需要返回到用户态以实现异步语义

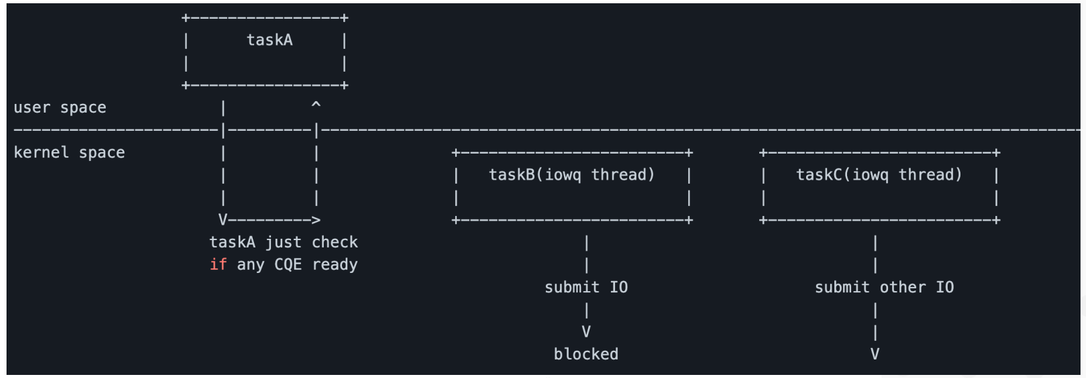
- 原上下文直接返回
- 用一个iowq线程提交IO请求，阻塞时创建/唤醒新的iowq线程用于提交其它IO，原iowq线程让其自然阻塞

前述方案

VS

uringlet





06.uringlet改进

低io depth情况下主线程与io-wq worker线程组高频率睡眠与唤醒

io-wq worker使用合适的polling策略

iodepth	1	2	4	8
uringlet	499K	681K	783K	718K
current	510K	529K	537K	541K

Reference

- [RFC] (https://lore.kernel.org/io-uring/20220819152738.1111255-1-hao.xu@linux.dev/)
- [Article] (https://howhsu.github.io/Linux-Kernel-Notes/kernel/io_uring/uringlet/uringlet.html)

THANK YOU!

感谢聆听！