项目申请书

项目名称：在Occlum 库操作系统中实现Linux AIO系统调用
项目主导师：吴新月
申请人：蔡加明
日期：2025.5.30
邮箱：jiam.cai@outlook.com

一. 项目背景

为了使得应用程序可以无需修改地从 Linux 迁徙到 Occlum 上，Occlum 期望实现一个尽可能兼容所有 Linux 的系统调用的 LibOS。目前 LibOS 已经实现了一个较为完整的 file system，支持 channel、pipe、eventfd、socket 以及 inode fd 等多种文件类型；兼容了 Linux open、write、ioctl 等多种系统调用，但目前还不支持 Linux 原生 AIO 系统调用。

1. 基本需求

本项目的基本需求即完成 Occlum LibOS 中对 AIO 相关系统调用的兼容实现，包括：

1. io_setup、io_destory、io_getevents、io_submit、io_cancel 共五种系统调用
2. 通过 gvisor 或 ltp 相关的完整性测试
3. 产出开发设计文档，使用 fio 等进行性能测试

2. AIO 系统调用

Linux 在 2.6 内核中引入了 AIO 支持，提供了以下 5 个系统调用：

系统调用	功能说明
`io_setup()`	为 AIO 创建一个上下文环境
`io_destroy()`	销毁之前创建的 AIO 上下文
`io_submit()`	用户向内核空间提交异步 I/O 请求
`io_getevents()`	获取内核已完成的 I/O 请求
`io_cancel()`	用户尝试取消尚未完成的 I/O 请求

首先介绍 AIO 中的几个结构：

1. io_context_t：AIO 的上下文结构的指针，`typedef struct io_context *io_context_t;`，在 io_setup 时交给内核初始化，之后的 IO 请求相关的操作都需要关联一个 AIO 的上下文
2. iocb：一个 IO 请求的控制块，代表一个 IO 请求，里面封装着对应 IO 的参数

```
struct iocb {
  __u64   aio_data; // 用户自定义指针，完成后返回，类似 io_uring 的 userdata
  __u32   PADDED(aio_key, aio_rw_flags);
  __u16   aio_lio_opcode; // IO 的操作码，即 r/w 等
  __s16   aio_reqprio;  // 请求优先级
  __u32   aio_fildes; // IO 请求对应的 fd
  __u64   aio_buf;  // 缓冲区
  __u64   aio_nbytes; // 缓冲区长度
  __s64   aio_offset; // 偏移量
  __u64   aio_reserved2;
  __u32   aio_flags;  // 一组和 iocb 结构关联的 flag
  __u32   aio_resfd;  // 指示 IO 请求事件是否完成的 fd，通常用法是与 eventfd 关联
};
```

3. io_event：io 请求完成后，返回的事件

```
struct io_event {
    __u64   data;     // 和 iocb.aio_data 相对应
    __u64   obj;      // 指向 iocb 本身
    __s64   res;      // 结果（读写了多少字节）
    __s64   res2;     // 错误码（如果 res < 0）
};
```

下面介绍用户空间 AIO 框架的使用流程：

- 创建 io_context_t 上下文句柄、创建 iocb 数组、创建 io_event 数组
- 打开文件，获取 fd（仅支持普通文件和设备文件），且打开方式为（O_DIRECT）
- **调用 io_setup，创建一个 AIO 上下文环境**
- 填充 iocb 数组，封装 io 请求的参数
- **调用 io_submit，提交 iocb 数组给内核，内核执行 IO 操作**
- 定期检查完成的 IO 请求，**调用 io_getevent，完成的请求填充到 io_event 数组中**
- **调用 io_cancel 取消对应的还未完成的 iocb 任务**
- 在完成所有的 IO 请求后，或者出现严重错误时，**用户显式调用 io_destory 销毁内核中的 AIO 上下文**

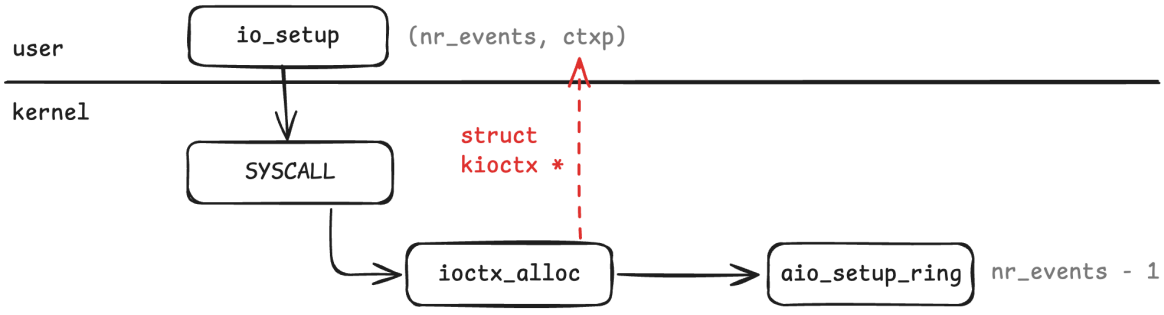
二. 技术背景及实现方案

1. AIO 基本原理

从内核角度，简单分析 AIO 系统调用的流程。将 AIO 框架主要分成三个阶段：setup 上下文设置、submit 提交异步请求、getevents 收割完成事件。（以下分析代码皆来自 github：Linux）

1.1 setup 上下文设置

将内核中 io_setup 的初始化逻辑简化为如下图所示：



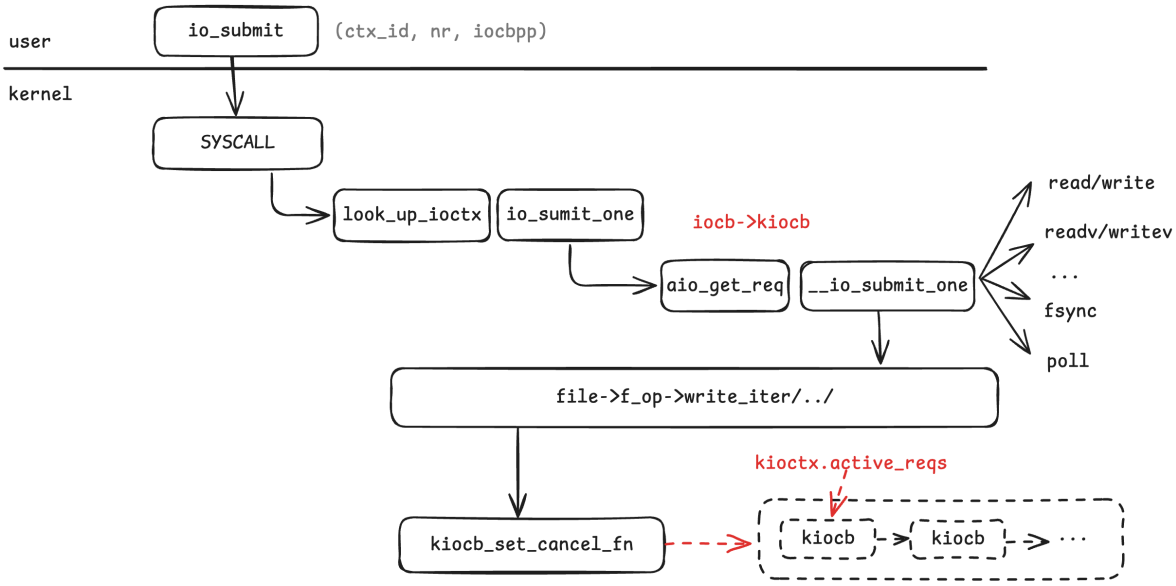
在用户空间调用 io_setup 系统调用，传入两个参数：nr_events, ctxp。其中 nr_events 表示请求事件的数量限制，ctxp 是一个用户空间的指针，指向一个 aio 上下文（也即 struct kioctx），由内核进行初始化并返回给 userspace。而一个 kioctx 结构为：

```
// 这里面的字段较多，重点观察几个重要的字段
struct kioctx {
  ...
  unsigned    req_batch;  // cpu 一次从 ring 中拿到的请求数量
```

```
  unsigned    max_reqs; // 表示请求的最大的数量，等于 setup 的 nr_events - 1
  unsigned    nr_events;  // 存储已完成请求的 ring 的大小（这里和 max_reqs）是不一致的，
主要是为了多 CPU 并行设计
  ...
  struct {
    atomic_t  reqs_available;
  } ____cacheline_aligned_in_smp; // 当前 ring 中可用的大小
  struct {
    spinlock_t  ctx_lock;
    struct list_head active_reqs; /* used for cancellation */
  } ____cacheline_aligned_in_smp; // 当前还未结束的 IO 请求

  struct {
    struct mutex  ring_lock;
    wait_queue_head_t wait;
  } ____cacheline_aligned_in_smp; // 当前阻塞等待的 进程队列

  struct {
    unsigned  tail;
    unsigned  completed_events;
    spinlock_t  completion_lock;
  } ____cacheline_aligned_in_smp; // 已经完成的事件

  struct folio    *internal_folios[AIO_RING_PAGES];
  struct file   *aio_ring_file; // 缓存所有请求的 ring_buffer
  unsigned    id;
};
```

1.2 submit 提交异步请求

在内核中的简要执行流程如下图所示：

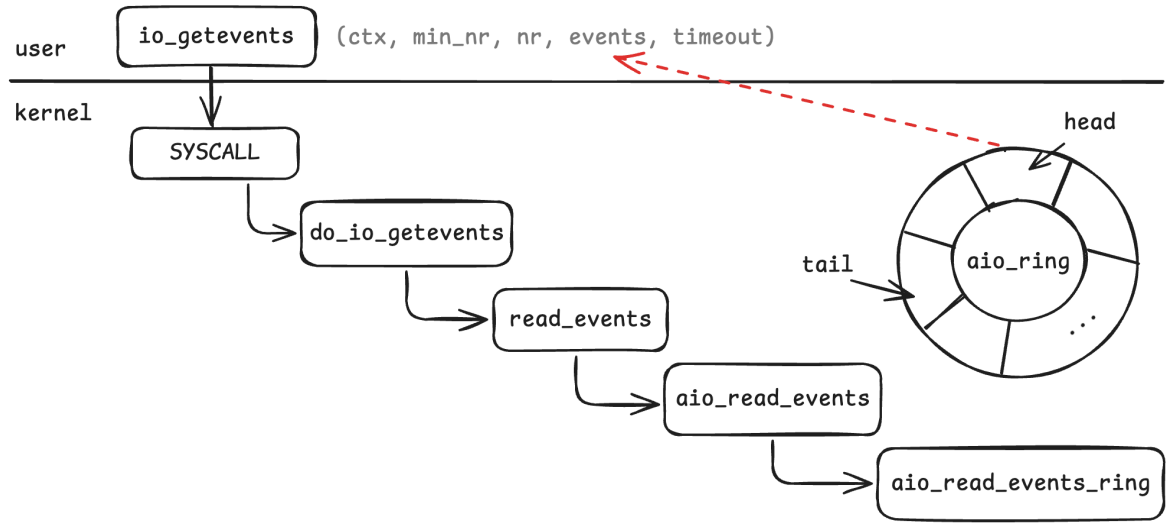


可以看到，在用户空间调用 io_submit 后，传入一个关于 iocb 的数组，内核会依次进行操作，首先创建一个 kiocb，获得用户 iocb 的信息，并根据 opcode 执行对应的 IO 操作，并且在最后的回调中将正在进行的 req 添加到 kioctx 的链表中。一个 kioctx 的结构为：

```
struct aio_kiocb {
  union {
    struct file   *ki_filp;
    struct kiocb    rw;
    struct fsync_iocb fsync;
    struct poll_iocb  poll;
  };  // 指示文件类型
  struct kioctx   *ki_ctx;  // 上下文
  kiocb_cancel_fn   *ki_cancel; // 取消 req 的回调方法
  struct io_event   ki_res; // 对应的完成事件
  struct list_head  ki_list;  /* the aio core uses this
            * for cancellation */
  refcount_t    ki_refcnt;
  struct eventfd_ctx  *ki_eventfd;  // 绑定的 evenfd
};
```

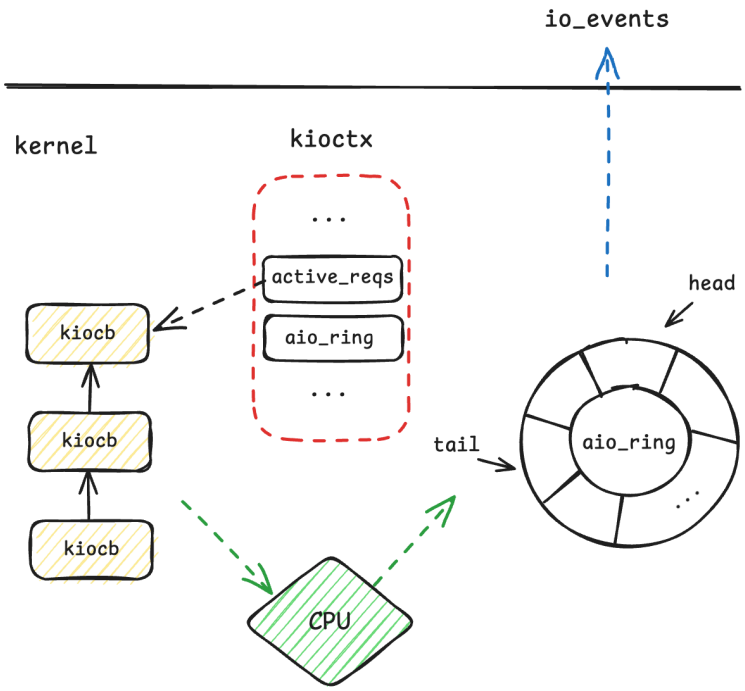
1.3 getevents 收割完成事件

内核中 io_getevents 的函数调用链如下所示：



用户在调用 io_submit 以后，提交的 IO 请求在完成以后，会填充在一个内核的 ringbuffer 当中。当用户主动发起一个 getevents 系统调用，会去收割这个 ring 中已经完成的 IO 事件，并且在这里注意到，这个系统调用是阻塞的，当完成的事件小于 min_nr 并且时间小于 timeout，当前的系统调用阻塞，一直轮询 aio_ring，等待 IO 事件完成，或者超时返回。而这个 aio_ring 是一个多生产者多消费者模型，在内核中通过内存屏障实现原子更新 head 和 tail。

至此，可以简单勾勒 Linux 原生 AIO 的抽象逻辑：



- 在 io_setup 时，创建 kioctx 结构，初始化 active_reqs 和 aio_ring 结构
- 用户通过 io_submit 提交 iocb 到 内核，正在进行的 kiocb 以链表的形式保存在 active_reqs 中
- 当 IO 事件完成后，生成 io_event，填充 aio_ring 的 tail 上
- 当用户调用 io_getevents，内核检查 aio_ring，并将完成的事件返回给用户。

2. Occlum 实现方案

整个 Occlum 的虚拟文件系统建立在 sgxfs 之上，和 HostOS 的 fs 是隔离的，所以无法使用 HostOS 上的 Linux AIO 原生系统调用，考虑在 Occlum 内部设计实现一个 AIO 框架以支持其系统调用。由于当前对 AIO 和 Occlum 的了解还不够深入，具体实现可能会做一些修改。

2.1 抽象出 AIO 的接口

将 AIO 的各项系统调用聚合为一个 Rust trait，然后在 Occlum 中支持 AIO 的 fd 类型中实现这个 trait：

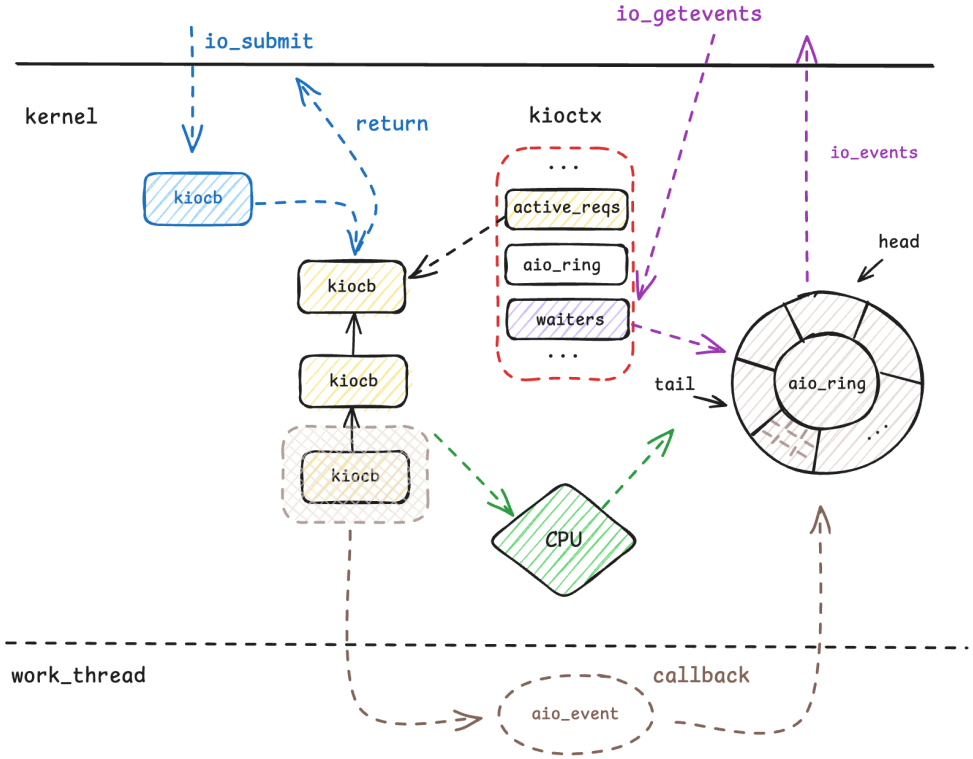
```
pub trait Aio {
  fn do_aio_setup {...}
  fn do_aio_submit {...}
  fn do_aio_cancel {...}
  fn do_aio_cancel {...}
  fn do_aio_destory {...}
}

pub struct Kioctx {
  ...
  waiters: WaiterQueue, // Occlum 中实现的线程 waiter 队列
  active_reqs: Vec<Kiocb>,  // 基于 Vec 的 IO 请求链表
  aio_ring: VecDeque<aio_event>,  //  基于 VecDeque 的 ring 结构
  ...
```

```
}
```

所有从 syscall mod 下发而来的 aio 系统调用在做完异常检查以后，根据传入的 fd 和其实现的 Aio trait 在这里进行真正的内核操作。同时在这里设计 kioctx 的内核结构，具体内容参考 Linux 的内核实现，重点关注`wait_queue_head_t wait`、`struct list_head active_reqs`、`struct file *aio_ring_file`这三个结构，分别表示阻塞进程队列、当前正在异步执行的IO请求、完成事件的ringbuffer。具体的设计如上。

2.2 设计异步模型

仿照 Linux 内核的实现，对于异步的任务可以起一个 work 线程去执行，并在 io_submit 返回用户之前，标识 submit 成功并将对应的 req 插入 active_reqs，并设置请求执行完成后的回调，在 callback 中将对应的 aio_event 添加到 aio_ring 中，并移除 active_reqs 中的 req。大致的流程如下图所示：



三. 项目实现规划

> 由于之前已经参与过 Occlum 社区，为 LibOS 实现了 netlink、IP 层和数据链路层 socket 的支持，且已经合并到 0.31.0-dev 分支，详细可见 #1653，所以实际开发速度可能会比规划日期超前。

1. 项目准备（6.16 - 7.01）

- 熟悉 Occlum 文件系统代码
- 继续深入对 Linux 原生 AIO 的了解

2. 项目编码（7.01 - 8.15）

- 完成 Occlum 对 AIO 系统调用的实现

- 编写开发设计文档

3. 完整性测试（8.15 - 9.10）

- 为完成的 AIO 系统调用添加 Occlum 测试用例
- 通过 gvisor 中 AIO 系统调用的完整性测试

4. 性能测试（9.10 - 9.30）

- 利用 fio 完成 Occlum AIO 性能测试
- 编写性能测试报告