组件间单向依赖的观点

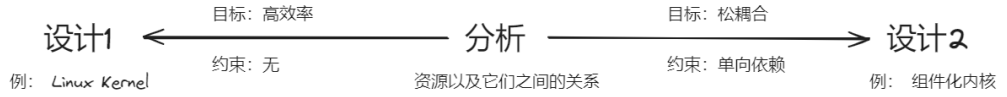
组件间的**单向依赖**是一个**设计问题**，不存在能否实现的问题，只有代价权衡的问题。

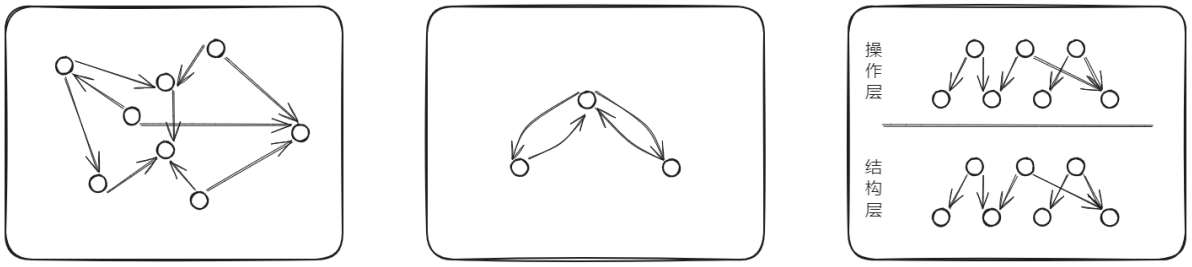
类似于应用开发，内核开发也包括分析和设计两个阶段。

在分析阶段，如果资源A管理资源B 或 A使用B，与此同时，也可以说，B从属于A 或 B被A使用，因此，资源对象之间只要有关系，则关系就必然是双向的。

但是到了设计层面，是把双向关系都表示出来，还是仅表示单向关系，这个取决于设计的**目标**与**约束**。

在设计时，如果把双向关系都表示出来，就是双向依赖；如果仅选择一个方向，就是单向依赖。无论哪一种表示方法，在下步的功能实现阶段都是能够实现的，只是在实现的难易程度、性能和耦合性方面有所差异。





依赖的定义

如果资源(组件) A 的构建、启动和运行必须基于另一个资源(组件) B才能完成，那么A依赖B。否则，不依赖。

具体分析依赖时，也可以从构建、启动和运行这三个方面分别分析，再综合考虑依赖的强弱。

设计目标考虑的问题

是更重视性能效率与开发的方便性？还是更重视模块的松耦合性？

Linux kernel选择了前者，追求极致性能与开发便利，内部模块之间大量采用了双向指针；组件化内核的设计目标则可能优先选择后者。

设计约束考虑的问题

如果选择前者目标，那么基本无约束；如果是后者目标，必须限制双向依赖。最理想的情况是，**仅允许单向依赖**。

约束考虑的问题：

1. 资源间关系的主次方向

   > 假定：管理者/使用者作为上层，从属/被使用者作为下层。

   以上层组件为起点，自上而下去逐层搜索定位目标资源，然后操作目标资源，这个应该是各种设计的共识。自上而下是主方向。

   例如，Linux设计与实现，主要遵循syscall -> (current)task -> parent_resource -> child_resource；我们组件化内核也类似。

   因此：自上而下关系称为**正方向**，默认建立**依赖**，也就是说无论Linux还是我们的组件化内核，上层模块/组件 依赖 下层的模块/组件完成构建、启动和运行。

分歧在于**反向关系**：Linux允许反向依赖，实现简单、性能好；组件化内核尽力避免反向依赖，当下层需要访问上层时，采取间接方式解决。

2. 依赖和反向关系采取的实现形式

   三种实现形式：

   (1) 直接指针或引用

   (2) 注册回调

   (3) 中间层解耦

   在不同阶段各个方式的耦合性对比：

实现形式	构建	启动	运行
(1) 直接指针或引用	强	强	强
(2) 注册回调	无	无	强
(3) 中间层解耦	无	无	无

   自上而下的正向依赖就是基于第1种 - 直接指针或引用。Linux和组件化内核无差别。

   自下向上的方向关系：Linux设计实现中依然大量采取第1种；组件化内核**禁止第1种**，优先选择第3种，第2种作为备选。

3. 结构与操作是否需要分离?

   Linux在设计上没有考虑结构/操作分离，资源的结构、状态与操作耦合在一起。

   组件化内核则考虑一种分层设计：

   把资源的结构定义和操作分散到两层，下层组件包含结构定义维护状态，上层组件包含各种操作。目的：操作可能跨不同资源的状态，把操作放到上层以解决部分循环依赖问题。

   两层分别解决不同的问题。

   下层 - 结构层内部：各个组件内的资源**数据结构**根据**数据包含或引用关系**建立依赖。

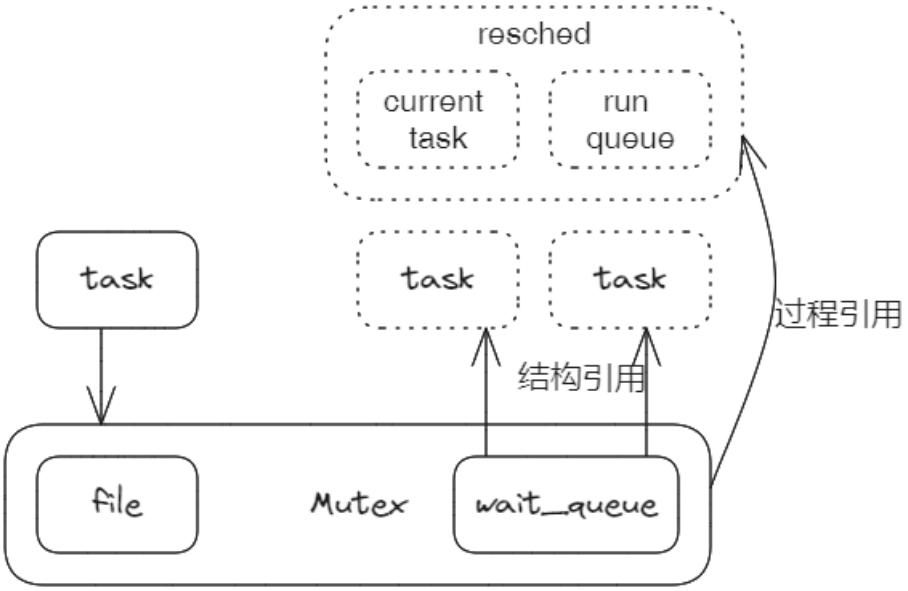
   上层 - 操作层内部：各个组件内的操作过程方法根据**过程调度**关系建立依赖。

   需要解决的特例问题：在两个层的内部，如果出现循环依赖，在环的非关键路径上找到一个点，插入中间层打破循环关系。

互斥锁层次问题分析

互斥锁Mutex与保护的资源、等待者之间天然存在一种循环关系。

以Task访问File为例，Task是资源的等待访问者，File代表某种I/O资源，可能存在长时间访问延迟，因此需要被Mutex保护。此时是从上向下的依赖。与此同时，Mutex机制需要一个等待者队列暂存那些处于等待状态的Task，这需要一个结构上的反向引用；还需要唤醒操作，这也需要一个自下向上的过程引用。



无论是Linux还是ArceOS之前的设计，两个方向的关系都是通过上述第1种方式 - 直接指针或引用来实现的，形成比较紧密的耦合关系。

在单向依赖约束下的解决思路，从反向关系的起点入口，切断关系环，提出两个解决方案：

1. 基于第2种 - 注册回调方式：wait_queue不是直接维护Task指针，而是抽象的Waiter Trait；运行中，等待者Task把自己的指针(引用)注册到wait_queue。这样在构建和启动阶段，都是自上而下的单向依赖；运行中，存在自下而上的反向依赖。
2. 基于第3种 - 引入中间层：wait_queue仅维护ID的队列，它不知道ID对应什么，仅负责按照要求写入和读出。上层组件维护ID到Task的映射。如此，消除了自下而上的反向依赖，保持自上而下的单向依赖。

原始SpinLock实现 - 作为参照

以fileops组件中read的实现为例，modules/fileops/src/lib.rs

```rust
pub fn read(fd: usize, ubuf: &mut [u8]) -> usize {
    let count = ubuf.len();
    let current = task::current();
    let file = current.filetable.lock().get_file(fd).unwrap();
    let mut pos = 0;
    assert!(count < 1024);
    let mut kbuf: [u8; 1024] = [0; 1024];
    while pos < count {
        let ret = file.lock().read(&mut kbuf[pos..]).unwrap();
        if ret == 0 {
            break;
        }
        pos += ret;
    }

    axhal::arch::enable_sum();
    ubuf.copy_from_slice(&kbuf[..count]);
    axhal::arch::disable_sum();
    pos
}
```

第9行，对file加自旋锁read。资源对象file由filetable维护，它的类型：

```
pub struct FileTableEntry {
    file: Arc<SpinNoIrq<File>>,
}
```

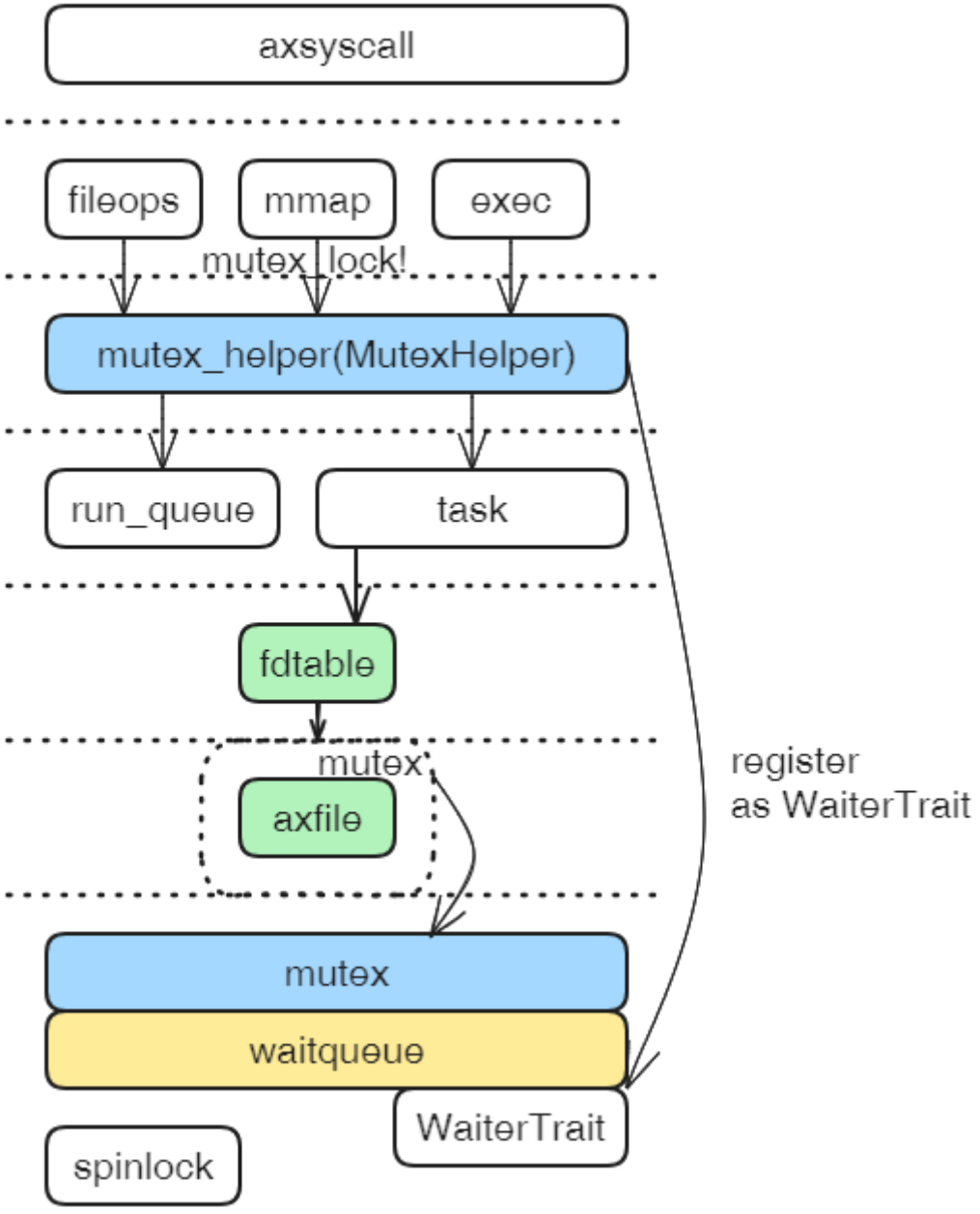
以下两个方案，在使用形式上基本保持一致，只是把自旋锁替换为可以Sleep的Mutex互斥锁。

另外，在底层组件的实现方式上，各有相应的改造。

以下两个方案的验证代码在lk_model仓库的两个分支上：mutex_model和mutex_model_2，可以checkout出来查看。

方案1 - 基于注册回调方式

验证分支：mutex_model



调用方式上替换为mutex锁的宏，modules/fileops/src/lib.rs改造前后的对比：

```
- let ret = file.lock().read(&mut kbuf[pos..]).unwrap();
+ let ret = mutex_lock!(file).read(&mut kbuf[pos..]).unwrap();
```

上述file的类型是

```
use mutex::AxMutex;
pub struct FileTableEntry {
    file: Arc<AxMutex<File>>,
}
```

AxMutex来自底层的那个mutex组件，因此能够调到。

关于mutex_lock!宏，modules/mutex_helper/src/lib.rs：

```
#[macro_export]
macro_rules! mutex_lock {
    ($arg:tt) => {
        $arg.lock(mutex_helper::MutexHelper::new())
    }
}

pub struct MutexHelper {
    task: TaskRef,
}

impl MutexHelper {
    pub fn new() -> Arc<Self> {
        Arc::new(Self {
            task: current().as_task_ref().clone(),
        })
    }
}

impl Waiter for MutexHelper {
    fn wid(&self) -> u64 {
        self.task.get_task_pid() as u64
    }

    fn block(&self) {
        let rq = run_queue::task_rq(&self.task);
        rq.lock().resched(false);
    }

    fn unblock(&self, resched: bool) {
        let rq = run_queue::task_rq(&self.task);
        rq.lock().add_task(self.task.clone());
        if resched {
            self.task.set_preempt_pending(true);
        }
    }

    fn on_waked(&self) {
        unimplemented!("");
    }
```

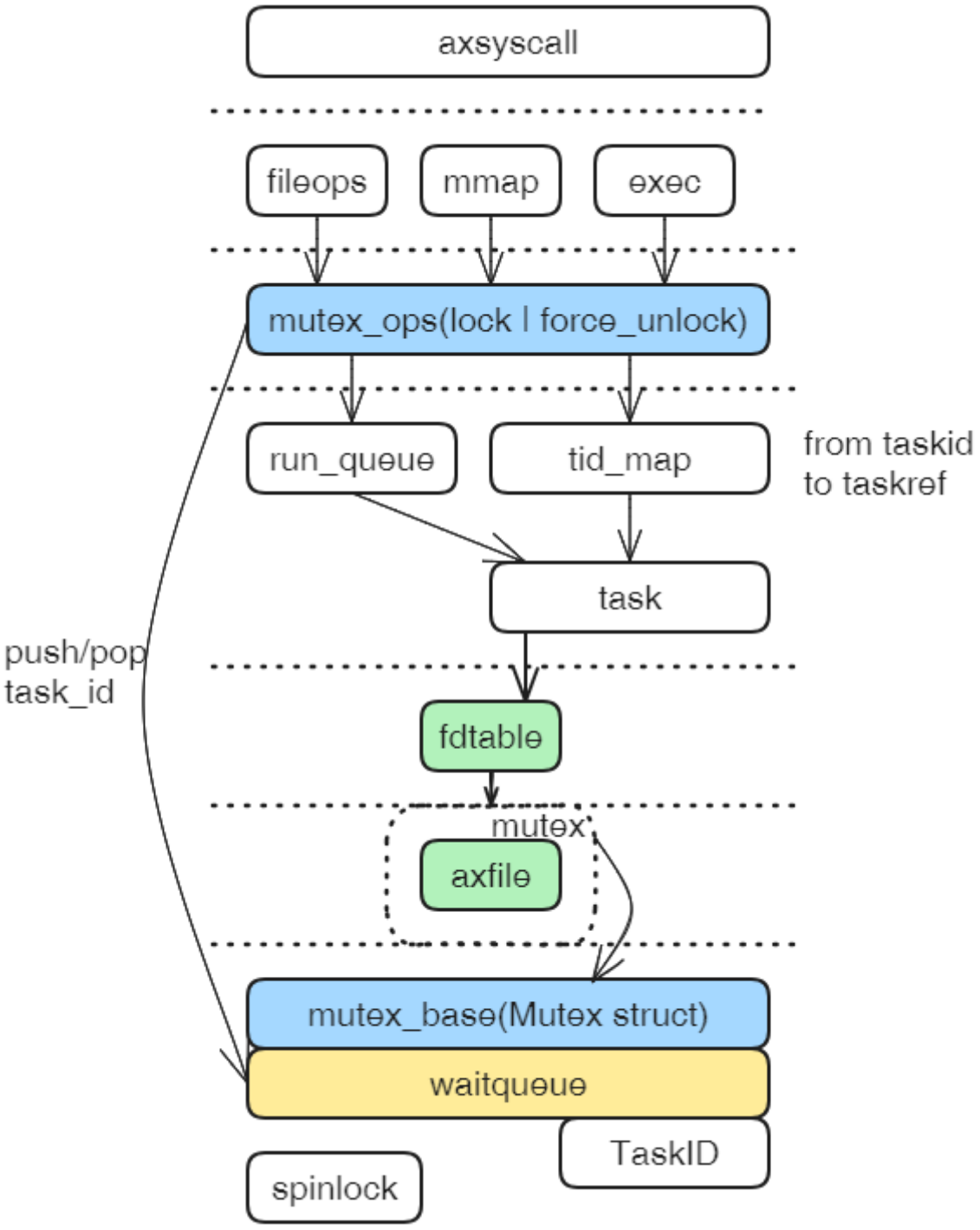
```
41 }
```

主要是在上层的mutex_helper组件中，实现MutexHelper，通过下层组件mutex中Mutex.lock的参数传下去，注册给WaitQueue，实现自底向上的回调。

总结：注册回调方式形成了运行时的反向依赖，但这条依赖局限在下层mutex和上层mutex_helper这一对组件内部。需要判断这种反向依赖在我们的组件化内核设计中是否允许。

方案2 - 引入中间层

验证分支：mutex_model_2



涉及三个方面的改造：

1. WaitQueue的元素类型仅是TaskID，它不解释不依赖该ID背后的东西，因此，从此处解除了对上层组件的耦合。
2. 原始的Mutex实现(axsync/Mutex)按照结构与操作，被切分为上下两个组件。上层组件mutex_ops包含lock/try_lock/force_unlock这些操作，事实上，仅是这些操作涉及对高层组件的调用，所以把它们提出来放到上层。
3. 增加一个高层组件tid_map，它负责从id映射到task的引用。用于配合上层组件mutex_ops的工作。

首先看对mutex锁的调用，除了替换自旋锁之外，与参照实现几乎没有差别：

```
+ use mutex_ops::MutexTrait;
let ret = file.lock().read(&mut kbuf[pos..]).unwrap();
```

上面加了对mutex_ops::MutexTrait的引用，基于这个Trait把lock/force_unlock方法提升到上层组件mutex_ops中实现：

```
pub trait MutexTrait<T: ?Sized> {
    fn lock(&self) -> MutexGuard<T>;
    fn try_lock(&self) -> Option<MutexGuard<T>>;
    fn force_unlock(&self);
}

impl<T: ?Sized> MutexTrait<T> for Mutex<T> {
    fn lock(&self) -> MutexGuard<T> {
        let current_id = current().pid() as u64;
        loop {
            // Can fail to lock even if the spinlock is not locked. May be more efficient than `try_lock`
            // when called in a loop.
            match self.owner_id.compare_exchange_weak(
                0,
                current_id,
                Ordering::Acquire,
                Ordering::Relaxed,
            ) {
                Ok(_) => break,
                Err(owner_id) => {
                    assert_ne!(
                        owner_id,
                        current_id,
                        "{} tried to acquire mutex it already owns.",
                        current().pid()
                    );
                    // Wait until the lock looks unlocked before retrying
                    loop {
                        let curr = task::current();
                        let mut rq = run_queue::task_rq(curr.as_task_ref()).lock();
                        if !self.is_locked() {
                            break;
                        }
                        self.wq.push_back(curr.pid());
                        rq.resched(false);
```

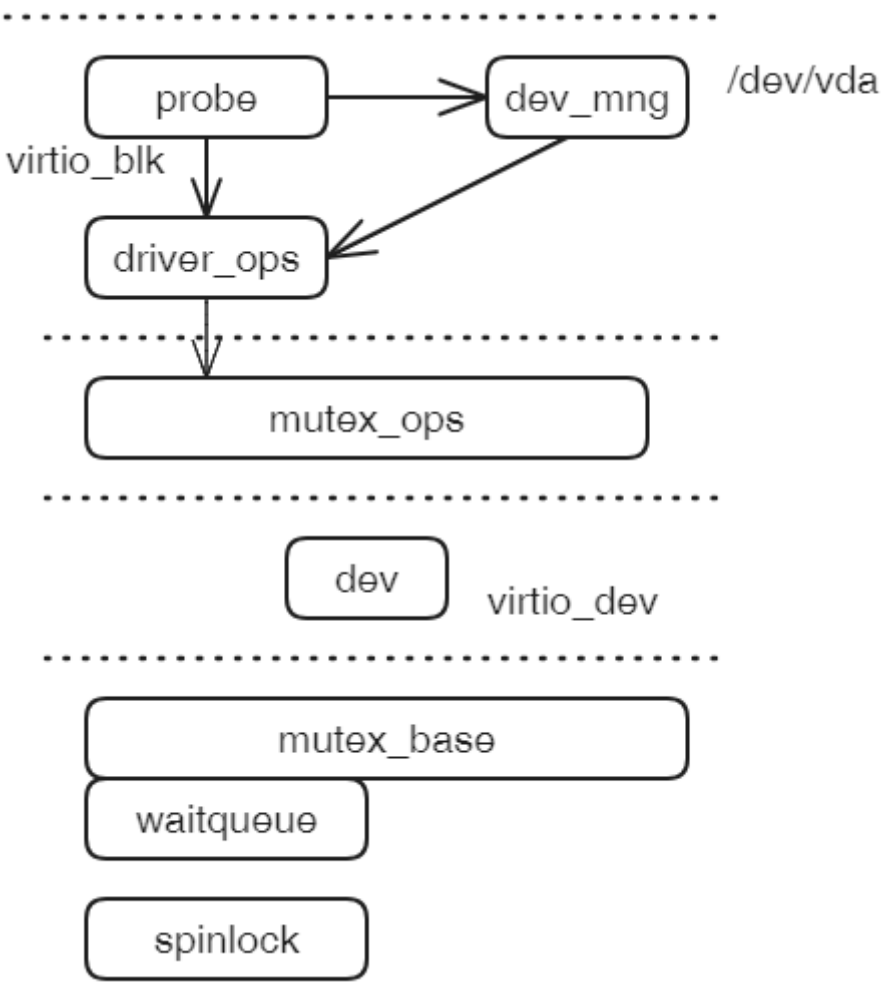
```
36                      }
37                      //self.cancel_events(crate::current());
38                  }
39              }
40          }
41      }
42      ... ...
43      fn force_unlock(&self) {
44          let owner_id = self.owner_id.swap(0, Ordering::Release);
45          assert_eq!(
46              owner_id,
47              current().pid() as u64,
48              "{} tried to release mutex it doesn't own",
49              current().pid()
50          );
51          if let Some(tid) = self.wq.pop_front() {
52              let task = tid_map::get_task(tid).unwrap();
53              //task.set_in_wait_queue(false);
54              let task2 = task.clone();
55              let mut rq = run_queue::task_rq(&task2).lock();
56              rq.add_task(task);
57          }
58      }
59  }
```

因为mutex_ops的位置高，它自然能够调用到task/run_queue等高层组件。

另外一个问题，底层的WaitQueue只是保持TaskID，去除了反向调用上层Task的功能。所以这部分功能也是在上层由mutex_ops完成的，见上面第34和第51行，那个WaitQueue仅负责push/pop任务的TaskID，如何使用和映射TaskID是上层组件的职责。

总结：方案2通过引入中间层tid_map解耦WaitQueue对Task的引用，通过按结构和操作切分Mutex完成操作过程的反向依赖。它的问题主要是，引入中间层tid_map后，多了一级映射，不确定对性能的影响有多大。考虑这个过程是唤醒睡眠等待的Task时才会被触发的，而等待I/O完成的时间，可能是通过自旋锁访问tid_map完成映射时间的数百倍，也许这个额外损耗并不显著？！（待验证！）

驱动与设备 - 涉及互斥锁的问题



类似于上一个问题，driver_ops相当于fileops层次，原始设备dev相当于file层次。互斥锁Mutex按照结构与操作分为上下两层。

相对的复杂性在于，驱动可能是多层，驱动组件之间也存在个上下关系；各级总线可以看作是设备的容器，也有相对上下关系。

例如，virtio块设备的过程涉及：

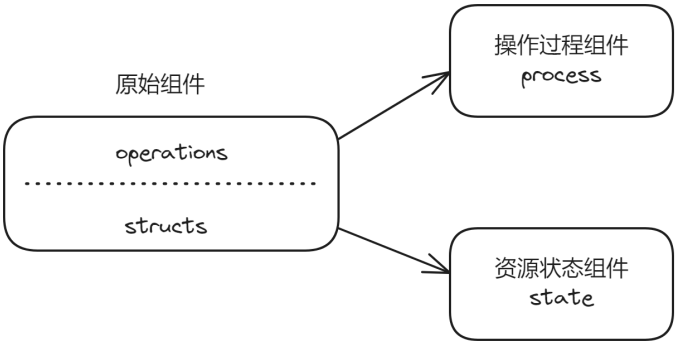
1. 通过virtio_mmio发现裸设备，挂到virtio总线上，识别为virtio_dev
2. 通过virtio_blk继续probe它，发现它是块设备，把它编号并基于/dev/vda管理
3. 上层组件通过/dev/vda访问这个设备

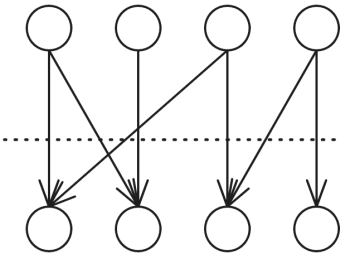
确实需要真正实践一下，把驱动、设备、总线理出个清晰的层次关系。

其他问题

1. 组件是否可能过于琐碎

   有可能，不好说。要看我们实践的情况。目前看，这种拆分主要涉及：对原本耦合到一起的一个组件，可能拆分成状态组件和过程组件这样上下两个。





2. unsafe/safe问题与单向依赖问题

   避免crate_interface不是因为不安全，而是它出现的场景就是反向依赖。

   而且crate_interface的依赖相当于第1种方式 - 直接指针或引用的耦合性。

3. OS中的相互依赖广泛存在，单向依赖是否过于理想化?

   确实是理想化目标。

   首先需要考虑的是：是否值得这样去尝试，这样做如果成功的收益是否值得我们做出这样的努力和冒这样的风险。

   然后，我们工作策略是：可以先朝着这个理想化目标去努力，争取最理想的效果，根据实践情况评估实际上能做到多少，那些不适用的特例是什么。

   这方面的一个看法：我们有可能受到过去经验的影响，存在思维定式。因为过去无论Linux kernel和我们自己的实践，都很少把组件化，尤其是单向依赖作为一个重要目标去考虑。都是怎么方便，怎么高效就怎么来，所以那些设计是在没有单向依赖这种约束下产生的。但那些设计是否是唯一合理设计？在单向依赖约束下能否找到其他的合理设计，可能很少有人尝试过。

4. 兄弟组件间出现一个中间组件，导致不再平级，例如log组件?

   有可能出现这种情况，开发过程中需要建立新组件，它导致之前的兄弟组件之间不再平级，那么也应当满足单向依赖。

   至于log组件其实是个特例，因为它就是个辅助组件，不参与真正的业务流程。我们可以允许它通过crate_interface的方式与其他组件产生隐形的反向依赖，这个不算是破坏组件间依赖。