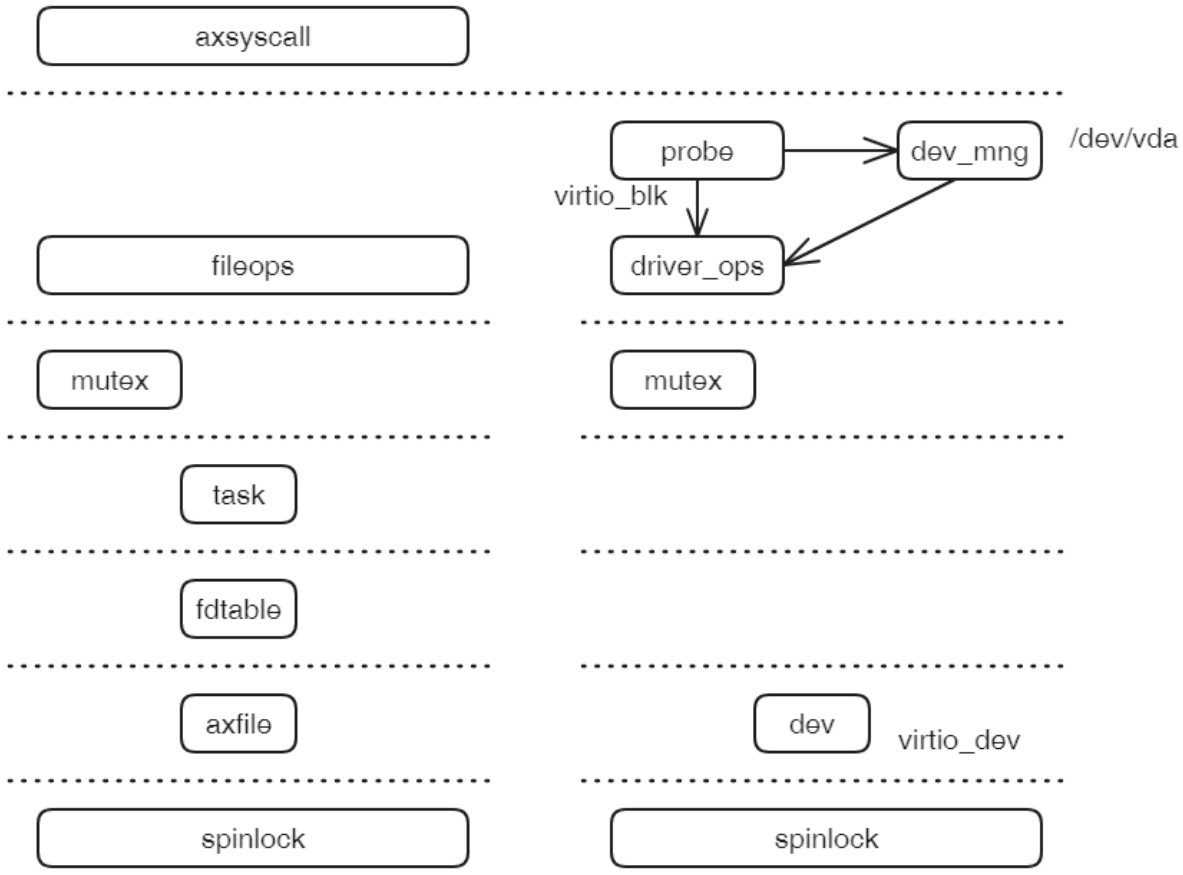


文件操作方面，互斥&自旋锁的层次的问题

目前fileops组件中read的实现，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
}
```

为讨论方便，简化为

```
1 pub fn read(fd: usize, ubuf: &mut [u8]) -> usize {
2     let current = task::current();
3     let file = current.filetable.lock().get_file(fd).unwrap();
4     file.lock().read(buf)
5 }
```

第3行返回的那个file类型是，Arc<SpinLock>；所以第4行file.lock()返回的是Guard。

上面代码中第4行等价为下面第4~6三行：

```
1 pub fn read(fd: usize, ubuf: &mut [u8]) -> usize {
2     let current = task::current();
3     let file = current.filetable.lock().get_file(fd).unwrap();
4     file.lock() => spinlock_guard(file_inner)
5     file_inner.read(buf)
6     spinlock_guard.drop()
7 }
```

按照目前的层次，guard只能是SpinLock自旋锁的guard，无sleep&resched，有效率问题。

假定我们让filetable维护和返回的是Arc，把加锁的操作延迟到fileops组件的read方法中，是否就可以选择带睡眠Mutex。如下：

```
1 pub fn read(fd: usize, ubuf: &mut [u8]) -> usize {
2     let current = task::current();
3     let file = current.filetable.lock().get_file(fd).unwrap();
4     Mutex(file).lock => mutex_guard(file)
5     file_inner.read(buf)
6     mutex_guard.drop()
7 }
```

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。中间是互斥锁Mutex。

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：

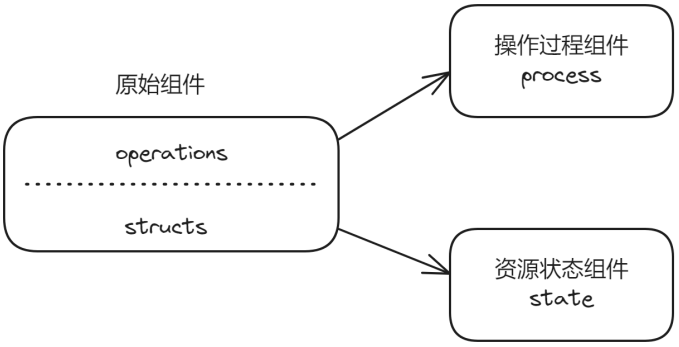
1. 通过virtio_mmio发现裸设备，挂到virtio总线上，识别为virtio_dev
2. 通过virtio_blk继续probe它，发现它是块设备，把它编号并基于/dev/vda管理
3. 上层组件通过/dev/vda访问这个设备

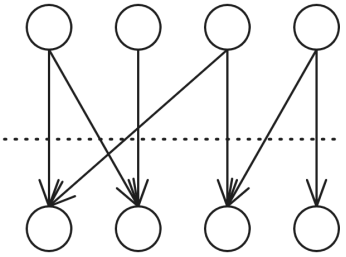
确实需要真正实践一下，把驱动、设备、总线理出个清晰的层次关系。

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。

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。