Lab13 IO and Storage

0. 前言

实验概述

了解设备驱动所需的 MMIO、PMIO 以及 DMA 机制。以 VirtIO 驱动为例，学习 virtio-blk 块设备驱动，并从主机硬盘读取数据。

实验代码

拉取实验代码：

```
$ git clone https://github.com/sdww0/sustech-os-lab.git && cd sustech-os-lab/lab13-io
```

或更新实验代码：

```
# In sustech-os-lab
$ git pull && cd lab13-io
```

本次实验新增代码包括：

1. **VirtIO 设备驱动框架**，包含：
    1. VirtIO MMIO 传输层
    2. Virtqueue 数据交互队列
    3. `DmaSliceAlloc` 与 `DmaSlice`，用于将 DMA 内存块划分为若干切片
2. **QEMU 块设备挂载**
3. **VirtIO-blk 块设备驱动**

1. 设备驱动基础知识

设备驱动是操作系统内核中的关键组件，充当硬件设备与操作系统之间的桥梁。它负责驱动和管理硬件设备，为操作系统提供必需的核心功能，例如硬盘存储、网络通信、图形显示等。一个完整的设备驱动通常需要实现以下基本功能：

1. **识别与初始化设备**：通过总线或其他机制检测设备存在，并完成其初始化配置。
2. **数据上行处理**：从硬件读取数据，经过必要的处理后提交给内核。
3. **数据下行处理**：接收来自内核的数据，经过格式化或转换后发送至硬件。
4. **错误处理**：监控设备状态，检测并处理设备运行中出现的异常或错误。

1.1 设备发现与总线

操作系统通常通过总线（如 PCI/PCIe、USB、I²C、SPI 等）来发现和识别硬件设备。每种总线规范定义了设备枚举、寻址和通信的协议。驱动程序依赖总线控制器提供的机制，获取设备的厂商ID、设备ID、资源需求（如内存地址、中断号）等信息。随后，系统据此加载对应的驱动程序并完成设备初始化。

RISC-V采取设备树（Device Tree）这一静态数据结构来描述系统的硬件布局。设备树是一个以树状节点形式描述硬件（如CPU、内存、外设）以及相互关系的文件。它包含设备驱动所必需的信息，如：

- **寄存器地址范围**：设备寄存器在物理内存中的映射区域（MMIO）
- **中断号**：设备所使用的中断线
- **标识符**：用于匹配和绑定设备驱动

在启动时，引导加载程序（如OpenSBI）将设备树二进制文件（DTB）的地址传递给内核。内核随后解析设备树，根据标识符等属性匹配并加载对应的驱动程序，并根据节点信息完成设备的资源分配与初始化。

1.2 驱动与设备的交互

驱动与硬件设备之间的交互主要包括两个方面：（1）**数据交互**，即通过 I/O（输入/输出）访问或基于物理内存的 DMA（直接内存访问）进行数据传输；（2）**事件通知**，例如设备通过外部中断向 CPU 发出信号，或内核主动“踢”（kick）设备以触发其操作。

1.2.1 数据交互

I/O 访问

CPU 与设备间数据传输可通过专门的 I/O 指令或内存映射方式实现。主要存在两种模式：

- **MMIO（内存映射 I/O）**：将设备的寄存器或缓冲区映射到系统的物理地址空间。CPU 通过普通的访存指令（如 load/store）读写这些地址，实际上是在与设备通信。MMIO 简化了编程模型，是 RISC-V 等现代体系结构的主要 I/O 方式，也广泛用于 x86 系统。
- **PMIO（端口映射 I/O）**：为 I/O 设备设立独立的地址空间（I/O 端口空间），访问时需使用专用的 CPU 指令。例如在 x86-64 中，`IN` 和 `OUT` 指令用于从端口读取或向端口写入数据。PMIO 主要用于兼容早期 x86 架构的设备。

针对MMIO访问，OSTD提供了`IoMem`封装：

```
#[derive(Debug, Clone)]
pub struct IoMem {
    kvirt_area: Arc<KVirtArea<Untracked>>,
    offset: usize,
    limit: usize,
    pa: Paddr,
}
```

使用者在获取`IoMem`时，OSTD会从内核空间分配一个虚拟地址，并建立该虚拟地址到实际物理地址的映射，此外，相比物理内存的大块数据读写，`IoMem`额外提供了`read_once`/`write_once`接口，适用于触发单个 MMIO 请求的小数据写入场景。

DMA 机制

DMA 是一种允许外设直接与系统内存交换数据的技术，无需 CPU 参与每次传输。它通过 DMA 控制器在设备和内存之间建立高速数据通道，大幅提升批量数据传输效率，并释放 CPU 资源以处理其他任务。然而，DMA最终依赖于内存进行数据交换，这就意味着需要CPU与设备双方需要进行及时的数据同步，即CPU需要及时将缓存数据刷新到内存中。为此，OSTD 提供了针对 DMA 内存的两种封装：`DmaCoherent` 与 `DmaStream`：

```rust
/// A coherent (or consistent) DMA mapping,
/// which guarantees that the device and the CPU can
/// access the data in parallel.
///
/// The mapping will be destroyed automatically when
/// the object is dropped.
#[derive(Debug)]
pub struct DmaCoherent {
    segment: USegment,
    start_daddr: Daddr,
    is_cache_coherent: bool,
}

/// A streaming DMA mapping.
///
/// Users must synchronize data before reading or after writing to ensure
/// consistency.
#[derive(Debug)]
pub struct DmaStream {
    segment: USegment,
    start_daddr: Daddr,
    is_cache_coherent: bool,
    direction: DmaDirection,
}
```

其中，`DmaCoherent` 确保每次读写都直接操作物理内存，适用于对**缓存一致性**要求高的小数据场景，如标志位；`DmaStream` 则不保证此性质，需要用户手动刷新数据到物理内存，适用于大量数据传输的场景，如网络包、硬盘数据传输。

1.2.2 事件通知

设备在完成某项操作或状态改变时（如数据到达、发送完成、发生错误），通常通过触发**外部中断**来通知 CPU。驱动程序需在初始化时注册相应的中断处理函数，并在中断发生时快速响应。OSTD提供了中断号的封装`IrqLine`：

```rust
/// An Interrupt ReQuest (IRQ) line.
///
/// Users can use [`alloc`] or [`alloc_specific`] to allocate a (specific) IRQ line.
///
/// The IRQ number is guaranteed to be an external IRQ number and users can use
[`on_active`] to
/// safely register callback functions on this IRQ line. When the IRQ line is dropped, all
the
/// registered callbacks will be unregistered automatically.
///
/// [`alloc`]: Self::alloc
/// [`alloc_specific`]: Self::alloc_specific
/// [`on_active`]: Self::on_active
#[derive(Debug)]
#[must_use]
pub struct IrqLine {
    inner: Arc<InnerHandle>,
```

```
    callbacks: Vec<CallbackHandle>,
}
```

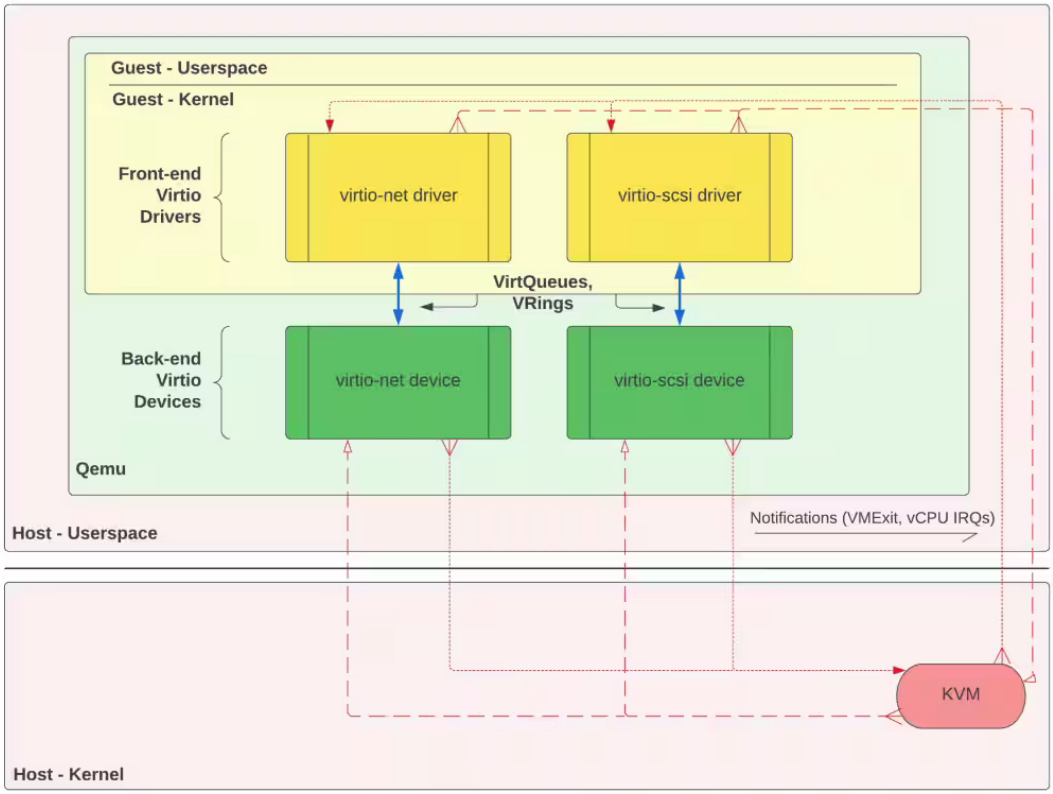
当外部中断发生时，OSTD会根据中断号找到对应的`IrqLine`，并回调其存储的一系列函数，以及时处理设备事件。

另一方面，内核也需要主动通知设备开始某项工作，这种机制也可被称为 **“kick”设备**。例如，当网络协议栈需要发送一个数据包时，它会将数据放入驱动管理的缓冲区，然后“kick”一下网卡，通知它“有数据待发送”。这种通知通常是驱动程序向设备的特定寄存器写入一个命令或值来实现的。

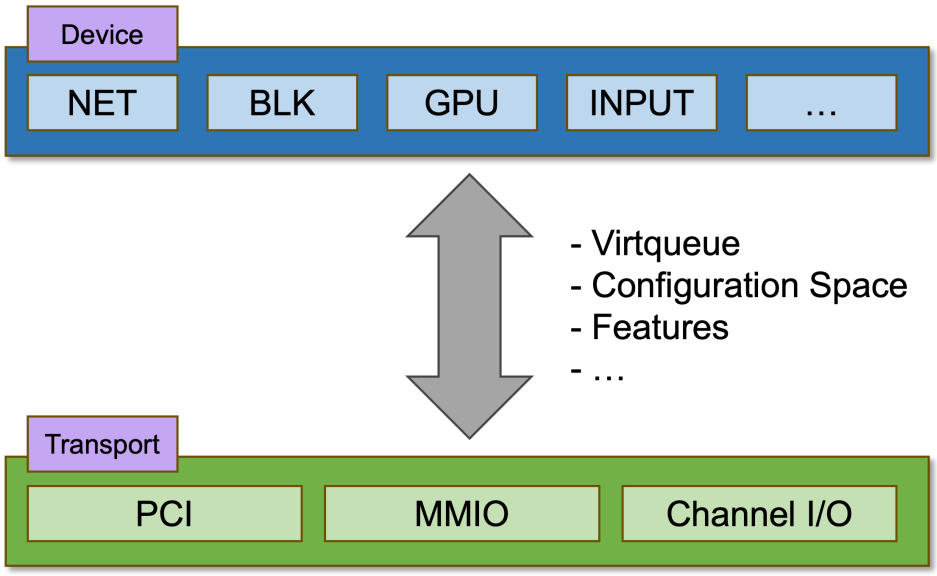
因此，驱动与设备之间的“事件通知”是双向的：**设备通过中断“通知”内核，内核则通过写寄存器“kick”设备**，从而形成一个完整的“命令-执行-完成通知”协作闭环。

2. VirtIO驱动

VirtIO 是一个位于客户机操作系统（Guest OS）和宿主机操作系统（Host OS）之间的、开放的半虚拟化 I/O 设备标准。其核心思想是：在Guest OS中实现一套标准化的、高效的“前端驱动”，在VMM或宿主机中实现对应的“后端设备”，双方通过一个共享的通信接口和数据结构（主要是“环形队列”，Virtqueue）来协作。来自Introduction to virtio对于VirtIO总体的分层结构（包括Host侧与Guest侧）：



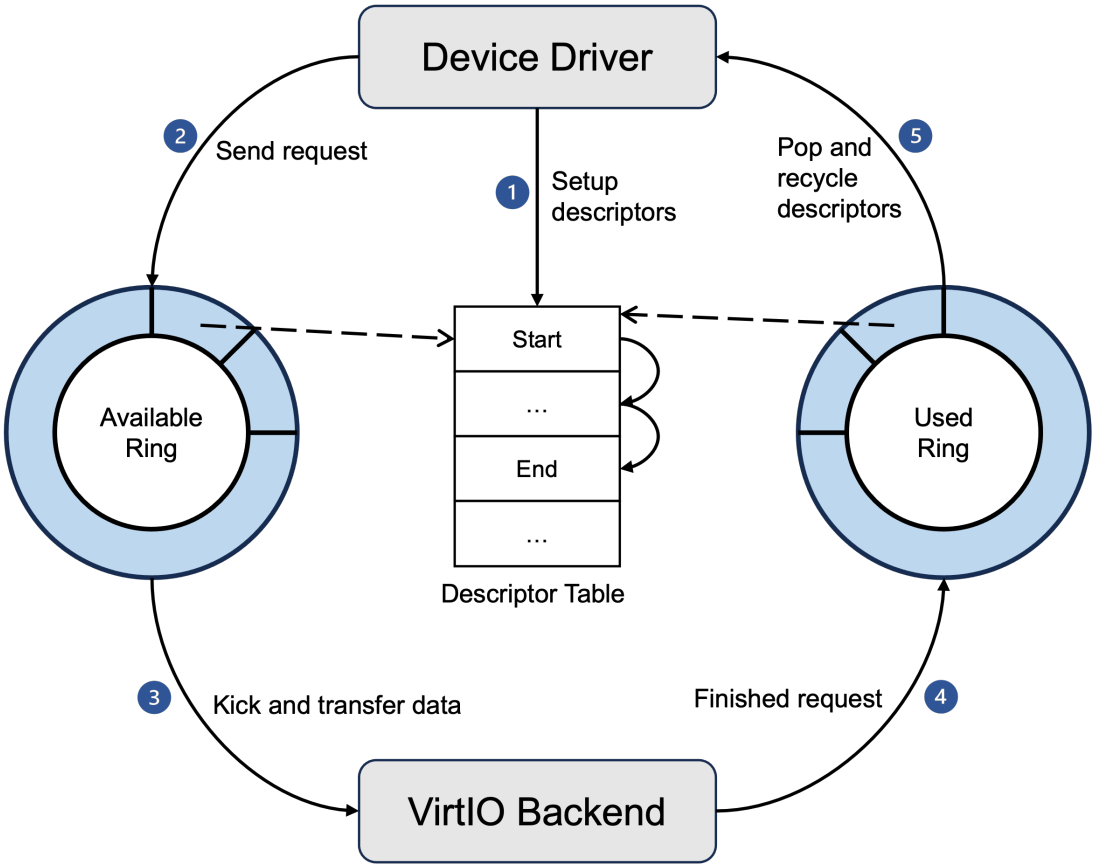
我们主要关注 Guest 侧的 VirtIO 驱动的分层结构，如下图所示：



Guest 侧的 VirtIO 驱动分为两部分：**传输层（Transport）**和**设备层（Device）**。传输层基于特定的传输协议，提供与设备交互的基础功能，例如 Virtqueue 基础设施的使能、设备配置空间访问、设备特性获取等；设备层则基于传输层提供的功能，实现特定的设备驱动，如网络、块设备、GPU 等。

需要注意的是，图中传输层的 MMIO 与“基于 MMIO 的设备访问”概念相似但不完全相同。图中的“MMIO”通常用于 RISC-V 平台架构，会通过设备树将 VirtIO 的 MMIO 地址传入内核，因此其依赖于 MMIO 进行初步的设备发现和后续交互。而“基于 MMIO 设备访问”是一个更广泛的概念，例如在基于 PCI 总线的交互中，会结合 PCI 总线与 MMIO 访问共同完成设备交互。

图中还展示了一个重要的数据结构——**Virtqueue**，它是设备驱动与设备进行数据交换的核心。Virtqueue 本质上是一个循环队列，其结构和操作流程如下图所示：



Virtqueue包含三个组件，即Descriptor Table、Available Ring、Used Ring，它们的描述如下：

- **Descriptor Table（描述符表）**：由一系列描述符（Descriptor）组成，数量由单个 Virtqueue 的队列大小决定。描述符之间以链表形式链接。每个描述符包含四个部分：（1）数据地址，指向存放数据的 DMA 内存；（2）数据长度；（3）标志位，指示该地址是否可被设备写入，以及该描述符是否指向下一个描述符；（4）下一描述符索引。
- **Available Ring（可用环）**：一个循环队列，存储 VirtIO 设备应处理的一系列请求，每个请求指向描述符链的开头。
- **Used Ring（已用环）**：一个循环队列，与Available Ring类似，但存放的是已处理完毕的请求的描述符链开头。

3. VirtIO驱动实现

本次实验课新增的驱动模块结构如下：

```
$ tree
.
├── blk.rs
├── mod.rs
├── utils
│   └── mod.rs
└── virtio
    ├── blk.rs
    ├── mmio.rs
    ├── mod.rs
    └── queue.rs
```

其中`virtio`目录下包含了三个主要文件：

- `mmio.rs`：实现 virtio-mmio 传输层功能，用于发现设备、使能 Virtqueue 与修改设备特性。
- `queue.rs`：驱动Virtqueue，为上层设备提供发送请求，等待请求完成，回收描述符等接口。
- `blk.rs`：VirtIO块设备的设备驱动，提供读写块的接口

本次实验代码新增较多，无法介绍全部的驱动内容。下面将会简要介绍virtio-blk中读取数据块的流程，代码如下：

```
fn read_block(&self, index: usize, data: &mut DmaSlice<[u8; SECTOR_SIZE], DmaStream>) {
    // 1. Prepare request and response structure
    let req_dma = self.request_alloc.lock().alloc().unwrap();
    let resp_dma = self.resp_alloc.lock().alloc().unwrap();

    let req = BlockReq {
        type_: ReqType::In as _,
        reserved: 0,
        sector: index as u64,
    };
    req_dma.write(&req);

    let resp = BlockResp::default();
    resp_dma.write(&resp);

    // 2. Put request, data, and response together
```

```
    let request1 = VirtqueueCoherentRequest::from_dma_slice(&req_dma, false);
    let request2 = VirtqueueStreamRequest::from_dma_slice(data, true);
    let request3 = VirtqueueCoherentRequest::from_dma_slice(&resp_dma, true);
    let requests: Vec<&dyn VirtqueueRequest> = vec![&request1, &request2, &request3];

    // 3. Send request
    let mut queue = self.request_queue.lock();
    queue.send_request(&requests).unwrap();
    if queue.should_notify() {
        queue.notify_device();
    }

    // 4. Wait for completion
    while !queue.can_pop() {
        core::hint::spin_loop();
    }
    queue.pop_finish_request();

    // 5. Check response
    let resp_read: BlockResp = resp_dma.read();
    if resp_read.status != RespStatus::Ok as u8 {
        error!("Block device read error: {:?}", resp_read.status);
    }
}
```

代码大致分为三个步骤：

1. **请求结构体配置**：从 DMA 内存中分配请求与响应结构体，准备发送给设备。
2. **数据发送与通知**：利用 Virtqueue 发送数据，并通知设备处理。
3. **等待与结果检查**：等待设备完成请求，随后检查请求状态。

运行实验代码，可以看到系统成功从文件中读取字符串：

```
$ make run
...
Enter riscv_boot
Virtio MMIO device found at 0x10008000 with size 0x1000, device id 2, version: 1
Testing block device read...
Read string: Hello World from Image!
...
```

4. 上手练习

块设备写入

目前我们只实现了从硬盘读取数据。请你模仿 `read_block` 函数，实现 `write_block` 函数（提示：复制后只需修改两处，请参考 VirtIO 白皮书 5.2.6 节，并注意描述符标志位中的设备可写位）。实现完成后，在宿主机运行 `strings blk.img`，应该输出 `Hello, Virtio Block Device!` 而非原来的 `Hello World from Image!`。

参考资料：[VirtIO白皮书]