跨操作系统异步块设备驱动模块设计与实现

Design and Implementation of a Cross-Operating System Asynchronous Block Device Driver Module

董若扬
1120202944@bit.edu.cn

School of Computer Science
Beijing Institute of Technology

May 26, 2024

大纲

大纲

研究背景

驱动程序（Device Driver）是计算机系统中的关键组件，负责在操作系统与硬件设备之间建立桥梁。作为一种系统软件，驱动程序专门控制硬件设备的操作。它是操作系统核心的一部分，直接与硬件交互，为操作系统提供硬件的抽象层，使应用程序无需了解硬件的具体实现即可使用硬件资源。块设备驱动程序是驱动程序中非常重要的一类。用于管理块设备（如硬盘、SSD、光驱等），负责大块数据的传输和存储操作，提供对块设备的高效访问接口。

块设备驱动程序直接影响数据存储和管理的效率与可靠性。高效的驱动程序能够提高数据传输速度，减少存储设备的读写延迟，提升整体系统性能。稳定的块设备驱动程序能够减少系统崩溃和数据丢失的风险。块设备驱动程序使操作系统能够支持多种不同的存储设备，提高系统的硬件兼容性。

研究背景

随着计算机技术的发展，操作系统对设备驱动的要求也随之提高，同时社会对于国产操作系统的需求也越来越高，在这种背景下涌现了许多开源操作系统，其中不乏一批被广泛用于教学，即帮助初学者入门学习操作系统设计的操作系统，例如基于 Rust 语言编写的 rCore 和 Alien。这些操作系统需要部署在开发板上时通常要求在操作系统中编写特定开发板的外设驱动模块。这导致操作系统在不同开发板上的可移植性受到极大挑战，给教学工作带来了不必要的复杂性。同时，随着操作系统内核功能的不断增加，代码量也在不断膨胀，这增加了学习者理解和分析内核的难度和成本。因此，为了降低这种复杂性，并减少学习者的负担，有必要使操作系统的各个部分之间尽可能解耦。

研究目的和意义

为了应对这些挑战，本研究提出了将驱动模块从操作系统中剥离出来，以 crate 的形式与操作系统进行对接的解决方案。这种做法不仅便于驱动模块的开发和迭代，也能够增强其在不同系统中的可重用性。从理论上讲，基于 Rust 的操作系统可以通过简单地在 Cargo.toml 文件中导入所需的驱动模块 crate 来实现这一目标，从而降低了系统的复杂性，并提高了其可移植性。这一研究将减轻操作系统开发者的开发负担，当越来越多研究投入该领域，将会形成完善的操作系统驱动生态，降低操作系统开发的门槛，相信将能极大促进开源操作系统的进展。另外本研究将块设备驱动从操作系统本身分离，降低了系统的耦合性，将为初学者学习分析理解操作系统内核代码带来极大帮助，减轻学习者的负担。本研究将对操作系统的开发者和学习者都能起到促进作用，这种帮助是正反馈的良性循环，相信这一研究对于促进基于 Rust 语言的教学操作系统的发展具有重要的理论和实际意义。

大纲

主要研究工作

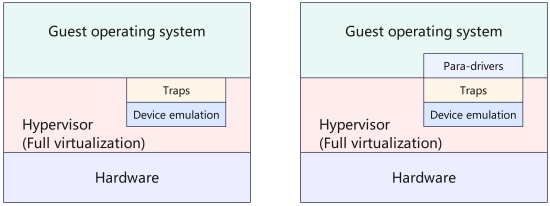
- 实现块设备的抽象接口
- 引入异步特性
- 支持更多 Feature
- 完成 Alien 操作系统接入工作
- 完成 rCore 操作系统接入工作

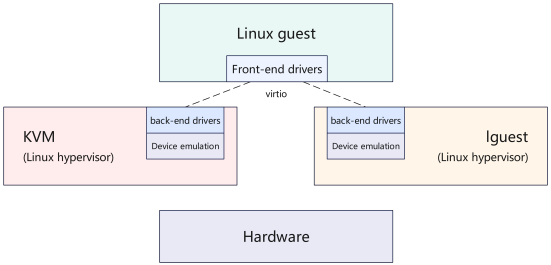
大纲

大纲

作为运行在硬件和操作系统层之间的一层，hypervisor 最早在 1960 年代被提出。hypervisor 使得计算环境能够在单个物理计算机上同时运行多个独立的操作系统，从而更有效地利用可用的计算能力、存储空间和网络带宽。VirtIO 规范的主要目的是简化和统一虚拟机（Hypervisor）的设备模拟，并提高虚拟机环境下的 I/O 性能。virtio 协议是对 hypervisor 中一组通用模拟设备的抽象，定义了虚拟设备的输入/输出接口。基于 virtio 协议的 I/O 设备被称为 virtio 设备。virtio 设备包括各种类型，如块设备（virtio-blk）、网络设备（virtio-net）、键盘鼠标类设备（virtio-input）、显示设备（virtio-gpu），它们具有共性特征和独有特征。共性特征通过统一抽象接口进行设计，而独有特征则尽量最小化各种类型设备的抽象接口，从而屏蔽了各种 hypervisor 的差异性，实现了 guest VM 和不同 hypervisor 之间的交互过程。

如图所示，左侧的虚拟机模拟外设的传统方案中，如果 guest VM 需要使用底层 host 主机的资源，那么 Hypervisor 必须截获所有的 I/O 请求指令，并模拟这些指令的行为。然而，这种方式会导致较大的性能开销。右侧的虚拟机模拟外设的 virtio 方案中，模拟的外设实现了功能最小化的原则。也就是说，虚拟外设的数据面接口主要与 guest VM 共享内存，而控制面接口主要基于内存映射的寄存器和中断机制。因此，当 guest VM 通过访问虚拟外设来使用底层 host 主机的资源时，Hypervisor 只需要处理少量的寄存器访问和中断机制，从而实现了高效的 I/O 虚拟化过程。





上图所描述的虚拟机模拟外设的 virtio 方案意味着，在 guest VM 上所见的虚拟设备具有简洁通用的优势。这对于运行在 guest VM 上的操作系统而言，意味着可以设计出轻量高效的设备驱动程序（即上图中的 Front-end drivers）。
在操作系统中，virtio 设备驱动程序（Front-end drivers）管理和控制着这些 virtio 虚拟设备。这些驱动程序仅需实现基本的发送和接收 I/O 数据，而位于 Hypervisor 中的 Back-end drivers 和设备模拟部分负责处理实际物理硬件设备上的设置、维护和处理，极大地减轻了 virtio 驱动程序的复杂性。

virtio 架构-1

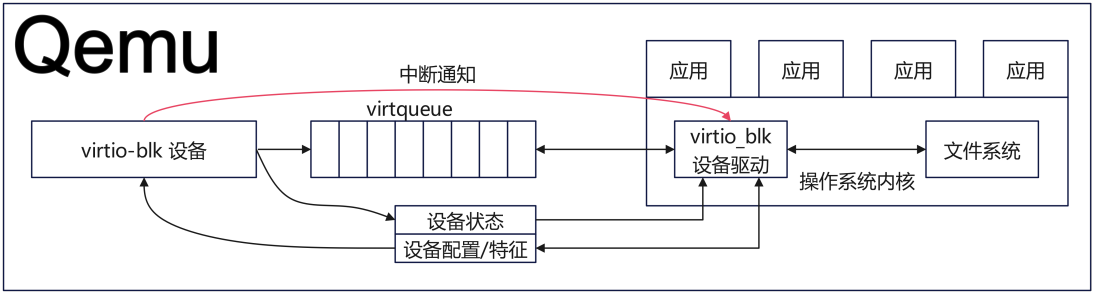
virtio 架构可以分为三层。

- 上层包括各种驱动程序，运行在 QEMU 模拟器中的前端操作系统
- 中间层则是传输层，负责驱动程序与虚拟设备之间的交互接口。传输层包含两部分:
  - 上半部是 virtio 接口定义，定义了 I/O 数据传输机制的 virtqueue
  - 下半部是 virtio 接口实现，具体实现了 I/O 数据传输机制的 virtio-ring，用于保存驱动程序和虚拟设备之间进行命令和数据交互的信息
- 下层是在 QEMU 中模拟的各种虚拟设备 Device

在操作系统中，virtio 驱动的主要功能包括接受来自用户进程或其他操作系统组件的 I/O 请求，将这些请求通过 virtqueue 发送到相应的 virtio 设备，并通过中断或轮询等方式查找并处理设备完成的 I/O 请求。

而在 QEMU 或 Hypervisor 中，virtio 设备的主要功能是通过 virtqueue 接受来自相应 virtio 驱动程序的 I/O 请求，然后通过设备仿真模拟或将 I/O 操作挂载到主机的物理硬件来处理这些请求，最终通过寄存器、内存映射或中断等方式通知 virtio 驱动程序处理已完成的 I/O 请求。

在运行在 Qemu 中的操作系统中，virtio 驱动程序与 virtio 设备驱动之间存在以下关系：

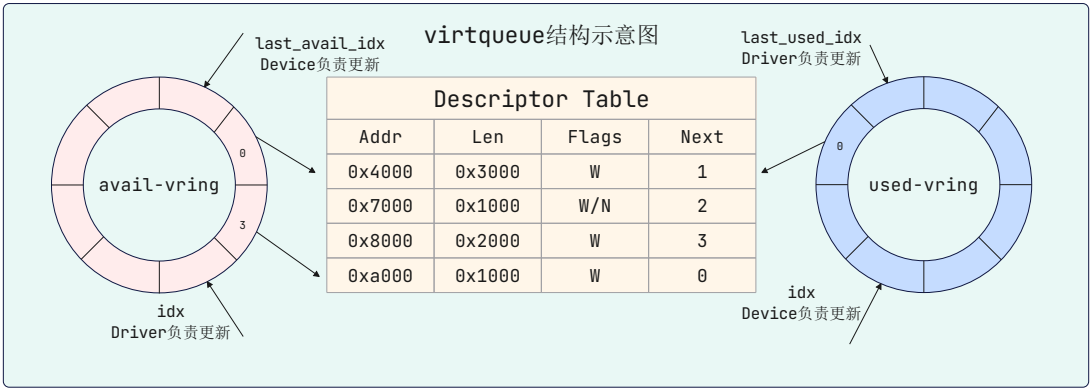


大纲

virtqueue 虚拟队列

为了实现批量数据传输，virtio 设备使用了 virtqueue 虚拟队列机制。每个 virtqueue 占用多个物理页。virtqueue 由三部分组成:

- 描述符表：描述符为组成元素的数组，每个描述符描述了一个内存 buffer 的 address/length。
- 可用环：记录了 virtio 设备驱动程序发出的 I/O 请求索引 (驱动更新的描述符索引的集合)，由设备进行读取。
- 已用环：记录了 virtio 设备发出的 I/O 完成索引 (设备更新的描述符索引的集合)，由驱动进行读取。



基于 virtqueue 进行 I/O 操作

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

初始化阶段（由驱动程序执行） 1

在设备初始化过程中，首先分配virtqueue的内存空间，包括描述符表、可用环以及已用环。然后，驱动程序将此部分的物理地址写入控制寄存器中，从而实现设备驱动程序和设备之间共享整个virtqueue的内存空间。

发起I/O请求阶段（由驱动程序执行） 2

当设备驱动程序发起I/O请求时，首先将请求的命令或数据存储到一个或多个buffer中。然后，在描述符表中分配描述符指向这些buffer。接着，将描述符的索引写入可用环中，并更新可用环的idx指针。最后，通知设备有新的请求。

完成I/O请求阶段（由设备执行） 3

当virtio设备接收到通知后，通过访问可用环的idx指针，解析出I/O请求。然后，设备执行I/O请求，并将结果存储到相应的buffer中。完成后，将描述符的索引写入已用环中，并更新已用环的idx指针。最后，通过中断机制通知设备驱动程序完成了I/O操作。

I/O后处理阶段（由驱动程序执行） 4

设备驱动程序读取已用环的idx信息，并从已用环中读取描述符索引，以获取I/O操作完成信息。

驱动程序与设备之间的交互

发出 I/O 请求的过程

驱动程序向设备提供新的 I/O 请求信息首先将包含一个 I/O 请求内容的缓冲区的地址和长度信息放入描述符表中的空闲描述符中，并根据需要将多个描述符进行链接，形成一个描述符链，表示一个 I/O 操作请求。接着，将描述符链头的索引放入可用环的下一个环条目中。如果可以进行批处理（batching），则重复执行前面的操作，通过可用环将多个 I/O 请求添加到描述符链中。随后，根据添加到可用环中的描述符链头的数量，更新可用环。最后，向设备发送通知，指示“有可用的缓冲区”。

接收设备 I/O 响应的操作

一旦设备完成了 I/O 请求，形成了 I/O 响应，就会更新描述符所指向的缓冲区，并向驱动程序发送已用缓冲区通知。通常，这种通知会采用中断这种高效的机制。设备驱动程序在收到中断后，会对 I/O 响应信息进行后续处理。

大纲

大纲

virtio 基本组成结构

组成要素	说明
设备状态域	表示 virtio 设备的当前状态，通常在设备初始化时使用
特征位	描述 virtio 设备的功能特性，由驱动程序和设备之间协商确定
通知	设备和驱动程序之间的异步事件通知，例如新数据到达或操作完成
设备配置空间	存储 virtio 设备的配置信息，包括设备的各种属性和参数
一个或多个虚拟队列	驱动程序和设备之间的数据传输和命令交互，实现高效通信的关键

virtio 设备的基本组成要素包括设备状态域（Device status field），特征位（Feature bits），通知（Notifications），设备配置空间（Device Configuration space），一个或多个虚拟队列（virtqueue）。其中，设备特征位和设备配置空间属于 virtio 设备的特征描述，用于描述设备的功能和配置；设备状态域用于初始化时表示设备的当前状态；而通知和虚拟队列则是 virtio 设备运行时的关键组成部分，用于实现设备和驱动程序之间的数据交换和通信。

特征位

特征位用于表示 VirtIO 设备具有的各种特性和功能。驱动程序与设备之间进行特性协商，以形成一致的共识，从而正确地管理设备。

```
bitflags! {
    #[derive(Copy, Clone, Debug, Default, Eq, PartialEq)]
    struct BlkFeature: u64 {
        const GEOMETRY   = 1 << 4; //指示磁盘的几何结构
        const RO         = 1 << 5; //设备是只读的
        const BLK_SIZE   = 1 << 6; //设定磁盘块大小，影响调度策略
        const FLUSH      = 1 << 9; //设备支持刷新操作
        const TOPOLOGY   = 1 << 10;//指示磁盘的逻辑结构
        const CONFIG_WCE = 1 << 11;//0:write-through, 1:write-back
        const MQ         = 1 << 12;//设备支持多队列
    }
}
```

特征位的协商

根据驱动支持的 Feature，初始化 SUPPORTED_FEATURE。

```
const SUPPORTED_FEATURES =
    BlkFeature::RO.union(FLUSH).union(BLK_SIZE)
    .union(CONFIG_WCE).union(GEOMETRY).union(TOPOLOGY)
    ...
```

驱动与设备进行协商，形成一致的 FEATURE 支持。

```
fn begin_init(&mut self,supported_features: F,) -> F {
    ...
    let device_features =
        from_bits_truncate(self.read_device_features());
    let negotiated_features = device_features & supported_features;
    self.write_driver_features(negotiated_features.bits());
    ...
    negotiated_features
}
```

特征位的协商

根据驱动支持的 Feature，初始化 SUPPORTED_FEATURE。

```
const SUPPORTED_FEATURES =
    BlkFeature::RO.union(FLUSH).union(BLK_SIZE)
    .union(CONFIG_WCE).union(GEOMETRY).union(TOPOLOGY)
    ...
```

驱动与设备进行协商，形成一致的 FEATURE 支持。

```
fn begin_init(&mut self,supported_features: F,) -> F {
    ...
    let device_features =
        from_bits_truncate(self.read_device_features());
    let negotiated_features = device_features & supported_features;
    self.write_driver_features(negotiated_features.bits());
    ...
    negotiated_features
}
```

特征位的使用-1

在初始化时，根据协商的特征位，设备支持自定义更多配置信息。

```
let writeback = if negotiated_features.contains(CONFIG_WCE) {
            unsafe { volread!(config, writeback) }
        } else {
            1
        };
```

在调用函数时，根据特征位的协商结果，判断该驱动方法是否为设备所支持。

```
pub fn flush(&mut self) -> Result {
    if self.negotiated_features.contains(BlkFeature::FLUSH) {
        ...
    } else {
        info!("device does not support flush");
        Ok(())
    }
}
```

特征位的使用-1

在初始化时，根据协商的特征位，设备支持自定义更多配置信息。

```
let writeback = if negotiated_features.contains(CONFIG_WCE) {
            unsafe { volread!(config, writeback) }
        } else {
            1
        };
```

在调用函数时，根据特征位的协商结果，判断该驱动方法是否为设备所支持。

```
pub fn flush(&mut self) -> Result {
    if self.negotiated_features.contains(BlkFeature::FLUSH) {
        ...
    } else {
        info!("device does not support flush");
        Ok(())
    }
}
```

特征位的使用-2

在执行过程中，根据协商的特征位，决定某些步骤是否应该执行。

```
pub fn write_blocks(
    &mut self,
    block_id: usize,
    buf: &[u8]
) -> Result {
    if self.negotiated_features.contains(BlkFeature::RO) {
        info!("device is read-only");
        Ok(())
    } else {
        ...
    }
}
```

设备状态域

在 virtio 设备初始化过程中，使用设备状态域来表示设备的状态。在设备驱动程序对 virtio 设备进行初始化过程中，会根据设备状态域的不同状态，经历一系列的初始化阶段。

1. EMPTY: 重置以确保设备处于初始状态
2. ACKNOWLEDGE: 操作系统设备已被识别
3. DRIVER: 操作系统已准备好驱动该设备
4. 写入操作系统和驱动程序能够处理的特性位子集至设备
5. FEATURES_OK: 指示驱动程序已完成特性协商
6. 此后，驱动程序不得接受新的特性位

状态	说明
ACKNOWLEDGE	驱动确认了一个有效的 virtio 设备
DRIVER	驱动知道如何驱动设备
FAILED	驱动无法正常驱动设备
FEATURES_OK	驱动已与设备就设备特性达成一致
DRIVER_OK	驱动加载完成，可以正常工作
DEVICE_NEEDS_RESET	设备触发了错误，需要重置

设备配置空间-1

配置中的信息是驱动相关的一些参数，当设备支持某些特征时，相关的参数就会被设置为配置中的数值。

基础配置信息有：capacity（容量），size_max（最大块数），seg_max（最大段数），blk_size（块大小）。

若协商特征包含GEOMETRY，则配置信息包含块设备的几何信息：cylinders（柱面数），heads（磁头数），sectors（单磁道扇区数）。

这些配置信息有助于操作系统和应用程序进行磁盘操作，如分区、格式化和文件系统管理。

若协商特征包含 TOPOLOGY，则配置信息包含块设备的拓扑结果信息：physical_block_size（物理块大小），alignment_offset（对齐偏移量），min_io_size（最小 I/O 规模），opt_io_size（最大 I/O 规模）。

这些配置信息有助于操作系统和应用程序优化存储访问和性能。

若协商特征包含CONFIG_WCE，则配置信息包含 writeback，用于表示缓存模式，0 表示write-through，1 表示write-back。若协商特征包含 MQ，则配置信息包含 num_queues，表示虚拟队列的数量。

设备配置空间-1

配置中的信息是驱动相关的一些参数，当设备支持某些特征时，相关的参数就会被设置为配置中的数值。

基础配置信息有：capacity（容量），size_max（最大块数），seg_max（最大段数），blk_size（块大小）。

若协商特征包含GEOMETRY，则配置信息包含块设备的几何信息：cylinders（柱面数），heads（磁头数），sectors（单磁道扇区数）。

这些配置信息有助于操作系统和应用程序进行磁盘操作，如分区、格式化和文件系统管理。

若协商特征包含 TOPOLOGY，则配置信息包含块设备的拓扑结果信息：physical_block_size（物理块大小），alignment_offset（对齐偏移量），min_io_size（最小 I/O 规模），opt_io_size（最大 I/O 规模）。

这些配置信息有助于操作系统和应用程序优化存储访问和性能。

若协商特征包含CONFIG_WCE，则配置信息包含 writeback，用于表示缓存模式，0 表示write-through，1 表示write-back。若协商特征包含 MQ，则配置信息包含 num_queues，表示虚拟队列的数量。

设备配置空间-2

在初始化阶段，根据协商的特征信息，逐一从配置空间中读取配置信息进行初始化。例如：

```
let mut blk_size: u32 = 512;
if negotiated_features.contains(BLK_SIZE) {
    blk_size = unsafe { volread!(config, blk_size) };
    info!("blk_size: {}", blk_size);
}
```

这些配置信息有的作为驱动的方法参数，有的作为硬件配置信息提供给操作系统，以供操作系统执行更高级的方法。

大纲

virtio-blk 设备结构

header 成员对应着 virtio 设备的共有属性，包括版本号、设备 ID、设备特征等信息。queue 是 virtio-blk 使用的虚拟队列。

Hal 是 virtio_drivers 库中定义的一个 trait，用于抽象出与具体操作系统相关的操作，主要涉及内存分配和虚实地址转换等功能。

```
pub struct VirtIOBlk<'a, H: Hal> {
    header: &'static mut VirtIOHeader,
    queue: VirtQueue<'a, H>,
    capacity: usize,
}

pub trait Hal {
    fn dma_alloc(pages: usize) -> PhysAddr;
    fn dma_dealloc(
        paddr: PhysAddr,
        pages: usize
    ) -> i32;
    fn phys_to_virt(paddr: PhysAddr)
        -> VirtAddr;
    fn virt_to_phys(vaddr: VirtAddr)
        -> PhysAddr;
 }
```

请求体 BlkReq

定义 BlkReq 结构体
表示块设备的请求。

```
#[repr(C)]
#[derive(AsBytes, Debug)]
pub struct BlkReq {
    type_: ReqType,//请求的类型
    reserved: u32, //保留字段
    sector: u64,   //请求涉及的扇区号
}
```

状态	取值	状态	取值	状态	取值
In	0	Out	1	Flush	4
GetId	8	GetLifetime	10	Discard	11
WriteZeroes	13	SecureErase	14	Append	15
Report	16	Open	18	Close	20
Finish	22	Reset	24	ResetAll	26

响应体 BlkResp

定义一个名为 BlkResp 的结构体，其包装了结构体 RespStatus 表示设备的响应状态。

```
#[repr(C)]
#[derive(...)]
pub struct BlkResp {
    status: RespStatus,
}
```

其中 RespStatus 的定义如下：

```
#[repr(transparent)]
#[derive(...)]
pub struct RespStatus(u8);
impl RespStatus {
    pub const OK = RespStatus(0);              //正常状态
    pub const IO_ERR = RespStatus(1);          //I/O错误
    pub const UNSUPPORTED = RespStatus(2);     //不支持的操作
    pub const NOT_READY = RespStatus(3);       //设备未准备好
}
```

大纲

Request

函数将块请求（BlkReq）转换为字节数组格式发送到 Virtio 设备，不带额外数据。接收并解析设备的响应，将响应状态作为函数的返回值。

该函数通过阻塞等待的方式，确保在收到设备响应后才返回，适用于需要同步 I/O 操作的场景。
request 方法主要用于刷新 flash 操作。

```
fn request(..., req: BlkReq) -> Result {
    let mut resp = BlkResp::default();
    self.queue.add_notify_wait_pop(
        &[req.as_bytes()],
        &mut [resp.as_bytes_mut()],
        &mut self.transport,
    )?;
    resp.status.into()
}
```

Request_read

函数将一个包含数据缓冲区的块请求发送到 Virtio 设备，包括给定的数据，并接收设备的响应。

函数的主体首先创建一个 BlkResp 类型的默认响应。然后将请求添加到队列中，并等待响应。最后，函数返回响应的状态，将其转换为 Result 类型的结果。request_read 方法主要用于封装读操作，包括同步读和异步读。

```
fn request_read(..., data: &mut [u8]
) -> Result {...
    self.queue.add_notify_wait_pop(
        &[req.as_bytes()],
        &mut [data, resp.as_bytes_mut()],
        &mut self.transport,
    )?;
    ...
}
```

Request_write

函数接收 BlkReq 类型的请求和字节切片作为数据，并返回 Result 类型的结果。函数将给定的请求和数据发送到设备，并等待设备的响应。

首先，函数创建了一个名为 resp 的变量，它是 BlkResp 类型的默认实例。然后，将请求和数据添加到队列中，然后通知设备有新的请求，接着等待设备的响应，最后从队列中弹出响应。request_write 方法主要用于封装写操作，包括同步写和异步写。

```
fn request_write(... , data: &[u8]
) -> Result {...
    self.queue.add_notify_wait_pop(
        &[request.as_bytes(), data],
        &mut [resp.as_bytes_mut()],
        &mut self.transport,
    )?;
    ...
}
```

flush

函数在设备支持的情况下，向设备发送一个刷新请求。

首先检查 negotiated_features 中是否包含 BlkFeature::FLUSH 特征，若包含则代表驱动和设备均支持刷新操作。如果设备支持刷新操作，那么函数会创建一个 BlkReq 类型的请求，请求类型为 ReqType::Flush，然后调用 request 方法发送这个请求。如果设备不支持刷新操作，函数返回一个 Ok(())，表示函数执行成功但没有产生有意义的结果。

```
pub fn flush(&mut self) -> Result {
    if self.negotiated_features
    .contains(FLUSH) {
        self.request(BlkReq {
            type_: ReqType::Flush,
            ..Default::default()
        })
    } else {...}
}
```

阻塞读-read_blocks

函数从设备中读取一个或多个块的数据，并将这些数据读入到给定的缓冲区中，阻塞直到读取完成或出现错误。

函数接受的参数中，buf 是一个可变引用到字节切片，表示用于存储读取的数据的缓冲区。

然后，函数调用封装的 request_read 方法来发送一个读取请求到设备。请求是一个 BlkReq 结构体的实例，其中 type_ 字段被设置为 ReqType::In，表示读取请求。以及按照协议完成一些其他工作（例如设置块 ID 等）。

```
pub fn read_blocks(..., buf: &mut [u8]
) -> Result {
    self.request_read(
        BlkReq {type_: ReqType::In, ...},
        buf,
    )
}
```

阻塞写-write_blocks

函数将给定的缓冲区内容写入到一个或多个块中。阻塞直到写入完成或出现错误。

这个方法接收的参数中，buf 是一个字节切片，包含了要写入的数据。

检查 negotiated_features 是否包含 RO。如果包含，说明设备是只读的，方法将打印信息并返回。否则，调用封装的 request_write 来写入数据。

在创建 BlkReq 实例时，设置 type_ 为 ReqType::Out，表示输出请求，设置 sector 为 block_id，表示要写入的块的 ID。其他的字段使用 Default::default() 方法的返回值进行初始化。

```
pub fn write_blocks(..., buf: &[u8]
) -> Result {
    if self.negotiated_features.contains(RO)
    else {
        self.request_write(
            BlkReq {type_: ReqType::Out, ...}
            buf,
        )
    }
}
```

异步读-read_blocks_nb

向 VirtIO 块设备提交请求，并返回一个标识第一个描述符在链中位置的令牌。如果没有足够的描述符可供分配，则返回 [Error::QueueFull]。然后调用者可以使用返回的令牌调用 peek_used 来检查设备是否已经完成处理请求。一旦设备完成处理，调用者必须在读取响应之前使用相同的缓冲区调用 complete_read_blocks。

```
pub unsafe fn read_blocks_nb(... ,
    req: &mut BlkReq,
    buf: &mut [u8],
    resp: &mut BlkResp,
) -> Result<u16> {
    *req = BlkReq {type_: ReqType::In, ...};
    let token = self.queue.add(
        &[req.as_bytes()],
        &mut [buf, resp.as_bytes_mut()]
    )?;
    if self.queue.should_notify() {
        self.transport.notify(QUEUE);
    }
    Ok(token)
}
```

异步读-complete_read_blocks

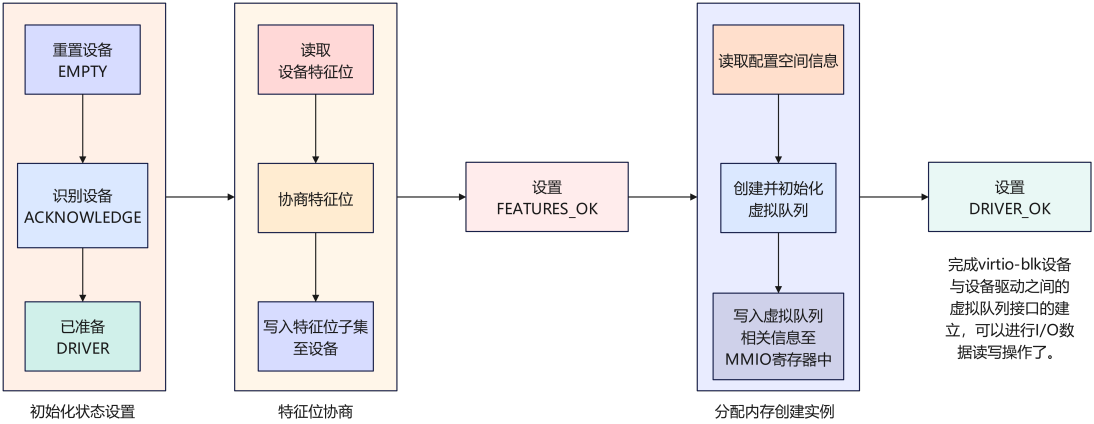
complete_read_blocks 函数的作用是完成由 read_blocks_nb 启动的读取操作。它接受一个令牌、一个 BlkReq 请求、一个用于存储数据的缓冲区以及一个 BlkResp 响应。它会从队列中弹出使用过的请求和缓冲区，并将响应的状态转换为 Result 类型。

```
pub unsafe fn complete_read_blocks(... ,
    token: u16,
    ...
) -> Result<()> {
    self.queue.pop_used(
        token,
        &[req.as_bytes()],
        &mut [buf, resp.as_bytes_mut()]
    )?;
    ...
}
```

需要注意的是，当 read_blocks_nb 返回令牌时，必须再次传递相同的缓冲区，作为参数传递给 complete_read_blocks。这是因为 read_blocks_nb 和 complete_read_blocks 之间存在数据依赖，如果传递的缓冲区不同，可能会导致数据错误。即使在此方法返回后，req、buf 和 resp 仍由底层的 VirtIO 块设备借用。因此，调用者有责任确保在请求完成之前不访问它们，以避免数据竞争。

大纲

virtio-blk 的初始化



设置 DRIVER_OK 状态位指示设备已准备好使用。驱动程序不得在设置 DRIVER_OK 位之前向设备发送任何缓冲区可用通知。

如果任何步骤不可恢复地失败，驱动程序应设置 FAILED 状态位。驱动程序不得在设置 FAILED 位后继续初始化。

virtio-blk 的 I/O 操作-1

在内存中，一个 I/O 请求的数据结构分为三个部分：Header（请求头部，包含操作类型和起始扇区）、Data（数据区，包含地址和长度）、Status（结果状态）。这些信息分别存放在三个 buffer 中，需要三个描述符。

virtio-blk 设备使用 `VirtQueue` 数据结构表示虚拟队列进行数据传输。该数据结构主要由三段连续内存组成：描述符表 `Descriptor[]`、环形队列结构的 `AvailRing` 和 `UsedRing`。驱动程序和 virtio-blk 设备都能访问到这个数据结构。

对于用户进程发出的 I/O 请求，经过系统调用、文件系统等一系列处理后，最终会形成对 virtio-blk 驱动程序的调用。

virtio-blk 的 I/O 操作-2

完整的 virtio-blk I/O 写请求过程:

1. 一个完整的 virtio-blk I/O 写请求包括:
   - 表示 I/O 写请求信息的结构体 `BlkReq`
   - 要传输的数据块 `buf`
   - 表示设备响应信息的结构体 `BlkResp`

   需要三个描述符表示。首先调用 `VirtQueue.add` 函数，从描述符表中申请三个空闲描述符，每个描述符指向一个内存块。驱动程序填写上述三部分的信息，并将这三个描述符连接成一个描述符链表。
2. 驱动程序调用 `VirtQueue.notify` 函数，通过写 MMIO 模式的 `queue_notify` 寄存器，向 virtio-blk 设备发出通知，指示有新的 I/O 请求待处理。virtio-blk 设备接收到通知后，通过比较 `last_avail` 和 `AvailRing` 中的 `idx` 来判断是否有新的请求。如果有新的请求，设备从描述符表中找到该 I/O 请求对应的描述符链，获取完整的请求信息，并执行存储块的 I/O 写操作。设备完成 I/O 写操作后，将已完成 I/O 的描述符放入 UsedRing 对应的 ring 中，并更新 `idx`，表明已经处理了一个请求响应。

   如果设置了中断机制，设备会产生中断通知操作系统响应中断。
3. 在驱动程序处理阶段，通过轮询或中断机制，驱动程序判断是否有新的响应。如果有新的响应，驱动程序取出响应，并回收完成响应的三个描述符。

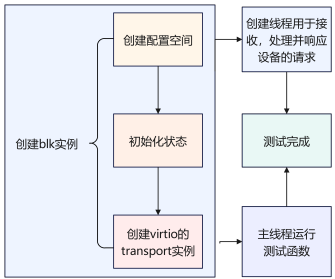
   最终，驱动程序将结果返回给用户进程。

大纲

大纲

单元测试

在单元测试中，对块设备驱动程序中的函数和方法进行测试，以验证其在模拟输入环境下的输出正确性。总体的测试流程如图所示。




```
#[test]
fn test() {
    ...
    let mut config_space = BlkConfig {...};
    let state = Arc::new(Mutex::new(State {...}))
    let transport = FakeTransport {...};
    let mut blk = VirtIOBlk::
        <FakeHal, FakeTransport<BlkConfig>>::
        new(transport).unwrap();
    let handle = thread::spawn(move || {...});
    # Test here
    fn(...);
    handle.join().unwrap();
}
```

刷新测试

flush() 函数模拟了一个 VirtIO 块设备的刷新操作。测试结果如图所示。
首先创建 VirtIOBlk 结构体实例 blk。接着，启动一个新线程，以模拟设备等待刷新请求的过程。在该线程中，首先打印出设备正在等待请求的信息，然后调用 State::wait_until_queue_notified 等待队列被通知。一旦队列被通知，打印出队列被通知的信息，然后锁定设备状态，并调用 read_write_queue 函数处理刷新请求。在处理刷新请求的过程中，首先断言请求的类型是刷新请求，然后创建一个包含响应状态的响应结构体，最后返回这个响应，以完成刷新请求的处理。

```
Device waiting for a request.
Transmit queue was notified.
Average execution time per request: 129.81 microseconds
Flush test passed.


successes:
    device::blk::tests::flush

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 23 filtered out; finished in 0.13s
```

阻塞读写测试

read() 函数模拟 VirtIO 块设备的读操作，其目的是验证 VirtIO 块设备能否正确地处理读请求并返回预期的数据。通过这种方式，可以验证 VirtIO 块设备在不同操作场景下的正确性和稳定性。

创建 VirtIOBlk 结构体实例 blk。然后，启动一个新线程来模拟设备等待读请求的过程。在这个线程中，首先打印出设备正在等待请求的信息，然后调用 State::wait_until_queue_notified 函数等待队列被通知。一旦队列被通知，打印出队列被通知的信息，然后锁定设备状态，并调用 read_write_queue 函数处理读请求。在处理读请求的过程中，首先断言请求的类型和扇区号，然后创建一个包含读取数据和响应状态的响应结构体，最后返回该响应。

在读操作部分，首先创建一个缓冲区 buffer，然后调用 blk.read_blocks 函数，从设备读取一个块的数据到该缓冲区中。接着，断言缓冲区中的数据是否与预期的数据一致。最后，等待设备模拟线程结束，以确保整个读操作完整执行。

写操作与读操作大同小异，不同之处在于数据的流向和处理方式。在读操作中，数据从设备读取到内存，而在写操作中，数据从内存写入到设备。此外，写操作结束后通常会紧接着一个刷新操作，以确保将暂存在内存中的缓冲区数据写入到持久性存储中。

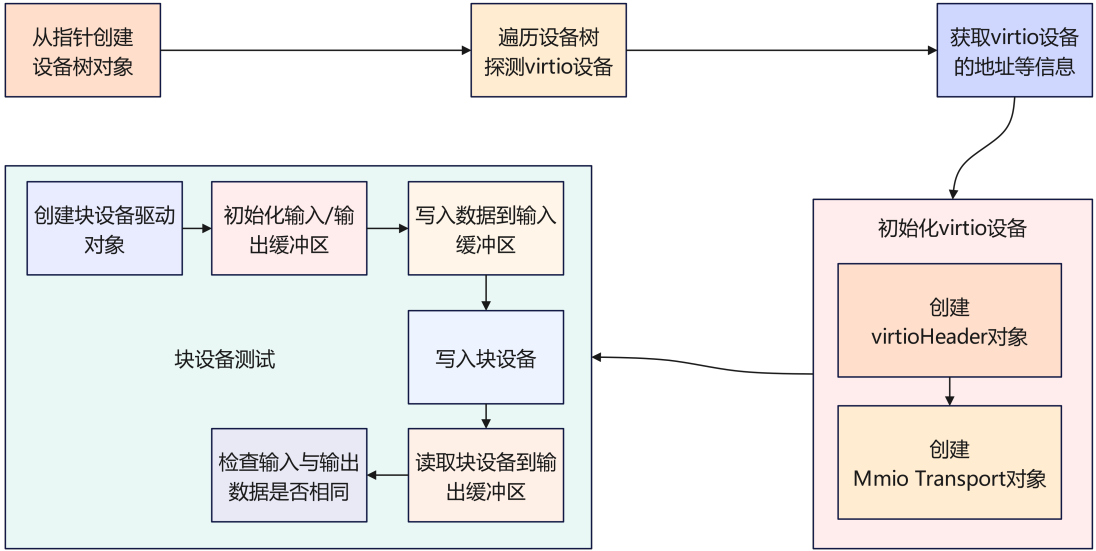
首先，向 VirtIO 块设备提交请求，并返回一个标识第一个描述符在链中位置的令牌。如果没有足够的描述符可供分配，则返回 Error::QueueFull 错误。然后，调用者可以使用返回的令牌调用 peek_used 函数来检查设备是否已经完成处理请求。一旦设备完成处理，调用者必须在读取响应之前使用相同的缓冲区调用 complete_read_blocks 函数。测试结果如图所示。
即使在该函数返回后，req、buf 和 resp 仍由底层的 VirtIO 块设备借用。因此，调用者有责任确保在请求完成之前不访问它们，以避免数据竞争。

```
running 1 test
test src/device/blk.rs - device::blk::VirtIOBlk<H,T>::read_blocks_nb (line 294) ... ok

successes:

successes:
    src/device/blk.rs - device::blk::VirtIOBlk<H,T>::read_blocks_nb (line 294)

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 7 filtered out; finished in 0.43s
```

写操作与读操作的测试环节几乎相同。

大纲

裸机环境集成测试-1


从指针创建
设备树对象
遍历设备树
探测virtio设备
获取virtio设备
的地址等信息
初始化virtio设备
创建
virtioHeader对象
创建
Mmio Transport对象
创建块设备驱动
对象
初始化输入/输
出缓冲区
写入数据到输入
缓冲区
块设备测试
写入块设备
读取块设备到输
出缓冲区
检查输入与输出
数据是否相同

裸机环境集成测试-2

1. 使用给定的设备树基地址（dtb）创建设备树对象
2. 遍历设备树节点。检查每个节点的兼容性属性，如果检测到兼容属性为"virtio,mmio"，则进行进一步处理
   - 2.1 获取设备的基地址和大小，并创建一个 VirtIOHeader 对象
   - 2.2 创建 VirtIO MMIO 传输实例
   - 2.3 进行设备类型识别，创建 VirtIOBlk 实例
3. 进行块设备的读写测试
   - 3.1 输入缓冲区填充为当前循环索引值
   - 3.2 将输入缓冲区的数据写入到第 i 个块
   - 3.3 从第 i 个块读取数据到输出缓冲区
   - 3.4 断言输入缓冲区和输出缓冲区的数据是否一致来判断块设备驱动的工作状况

测试结果如图所示，读取遍历设备树，探测到“virtio,mmio”类型，识别到类型是“Block”，读取其配置，是大小为 16KB 的 block device，其块大小为 512B。

```
[ INFO] device tree @ 0x87000000
[ INFO] walk dt addr=0x10008000, size=0x1000
[ INFO] Device tree node virtio_mmio@10008000: Some("virtio,mmio")
[ INFO] Detected virtio MMIO device with vendor id 0x554D4551, device type Block, version Legacy
[ INFO] config: 0x10008100
[ INFO] found a block device of size 16KB
[ INFO] blk_size: 512
[ INFO] virtio-blk test finished
```

大纲

对接 virtio-blk 的准备-1

在 Cargo.toml 中添加本项目的 virtio-blk 依赖。

```
virtio-drivers = { git = "https://github.com/semidry/virtio_crate.git" }
```

定义 Trait，用于满足对块设备驱动的 I/O 访问要求。

```
pub trait BlkDevice: Send + Sync + Any {
    fn read_block(&self, block_id: usize, buf: &mut [u8]);
    fn write_block(&self, block_id: usize, buf: &[u8]);
    fn handle_irq(&self);
}
```

建立用于表示 virtio_blk 设备的全局变量 BLOCK_DEVICE。

```
pub static BLK_DEVICE: Once<Arc<dyn BlkDevice>> = Once::new();
```

对接 virtio-blk 的准备-2

创建结构体 VirtIOBlkDeviceWrapper 用于表示块设备，其中包含一个实现了 BlkDevice Trait 的设备。然后为其实现方法以及实现 VfsFile 和 VfsInode 特征。
VfsFile 特征定义了与文件操作相关的方法，例如从指定偏移量读取数据到缓冲区，将缓冲区的数据写入指定偏移量。
VfsInode 特征定义了与索引节点相关的操作，例如返回或设置索引节点的类型（如字符设备），获取索引节点的属性。
这些特征方法应由该操作系统的开发者自行决定实现方式，这里仅进行最简单的实现。

```
pub struct VirtIOBlkDeviceWrapper {
    blk: Mutex<VirtIOBlk<HalImpl, MmioTransport>>,
}
impl VirtIOBlkDeviceWrapper {...}
impl BlkDevice for VirtIOBlkDeviceWrapper {...}
```

在 Makefile 中为 qemu 模拟器添加虚拟块设备

```
define boot_qemu
        ...
        -drive file=$(IMG),if=none,format=raw,id=x0 \
        -device virtio-blk-device,drive=x0 \
        ...
endef
```

在 Alien 中启动块设备

在内核态的主函数中，devices::init_device() 是启动设备的切入点。在此初始化函数中，先获取设备树块（DTB）的指针，然后将该指针转换为字节指针，以创建一个 Fdt 对象。接着调用方法探测设备，探测到 Virtio MMIO 设备，调用 init_virtio_mmio() 函数进行初始化。首先对设备树进行遍历，对每种设备执行相应的初始化操作。从 DeviceInfo 中提取设备的基地址 paddr。然后，尝试将该物理地址转换为指向 VirtIOHeader 结构体的非空指针，利用此指针创建一个 MmioTransport 实例。例如，对于块设备，其类型为 DeviceType::Block，字面值为 2，因此调用 init_block_device() 进行初始化。

完成上述所有操作后，qemu 模拟器中设备树上的块设备以及块设备驱动已成功加载，并能够被操作系统所识别，如图所示，加载出块设备的地址并初始化成功，就可以对块设备进行操作了。然后进入了 Alien 的控制台，如图所示表明 Alien 正常启动。

```
[0] Init blk device, base_addr:0x10008000,irq:8
[0] Init blk device success
```

```
[0] Initrd populate success
[0] Init filesystem success
[0] ++++ setup interrupt ++++
[0] ++++ setup interrupt done, enable:true ++++
[0] Init task success
[0] Begin run task...
[0] kthread_init start...
Init process is running
Alien:/#
```

大纲

对接 virtio-blk 设备初始化过程

在 Cargo.toml 中添加本项目的 virtio-blk 依赖
建立表示 virtio_blk 设备的全局变量 BLOCK_DEVICE，以便内核能够识别和使用该设备。

```
pub struct VirtIOBlock {
    virtio_blk: UPIntrFreeCell<VirtIOBlk<'static, VirtioHal>>,
    condvars: BTreeMap<u16, Condvar>,
}
```

condvars 条件变量，用于在进程等待 I/O 读或写操作完成之前挂起进程。VirtioHal 实现了

virtio_drivers 模块定义的 Hal trait，使得 VirtIOBlk 类型能够得到操作系统的服务。

```
impl Hal for VirtioHal
    fn dma_alloc(pages: usize) -> usize;
    fn dma_dealloc(pa: usize, pages: usize) -> i32;
    fn phys_to_virt(addr: usize) -> usize {addr};
    fn virt_to_phys(vaddr: usize) -> usize {vaddr};
```

对接 virtio-blk 设备 I/O 处理-1

操作系统中的文件系统模块与操作系统中的块设备驱动程序 VirtIOBlock 直接进行交互。VirtIOBlock 驱动程序封装了 virtio-drivers 模块中实现的 virtio_blk 设备驱动。
首先，在文件系统中定义对块设备驱动的方法。操作系统的 virtio_blk 设备驱动程序中的 VirtIOBlock 已实现了该方法，并且支持既支持轮询方式，也支持中断方式的块读写操作。

```
pub trait BlockDevice: Send + Sync + Any {
    fn read_block(&self, block_id: usize, buf: &mut [u8]);
    fn write_block(&self, block_id: usize, buf: &[u8]);
    fn handle_irq(&self);// 增加对块设备中断的处理
}
```

操作系统还需要对整体的中断处理过程进行调整，以支持基于中断方式的块读写操作。当系统发生中断时，BLOCK_DEVICE.handle_irq() 执行的实际上是 VirtIOBlock 实现的中断处理方法 handle_irq()，从而让等待在块读写的进程/线程得以继续执行
有了基于中断方式的块读写操作，当某个线程/进程由于块读写操作无法继续执行时，操作系统可以切换到其它处于就绪态的线程/进程执行，从而提升计算机系统的整体执行效率。

对接 virtio-blk 设备 I/O 处理-2

handle_irq 方法用于处理设备的中断。在一个循环中处理所有已完成的请求。对于每个已完成的请求，通过信号量通知等待的线程，表示该请求已处理完毕。read_block 方法用于读取指定

块设备上的数据块。若采用中断方式进行非阻塞访问，则发起非阻塞的读请求，并获取一个令牌。该令牌用于在 condvars 中查找相应的条件变量。完成后切换到其他线程或进程继续执行。若采用轮询方式进行阻塞访问，则阻塞式读取数据块，CPU 处于忙等待状态并期望操作成功完成。

```
impl BlockDevice for VirtIOBlock {
    fn handle_irq(&self) {...}
    fn read_block(...) {
        // 如果是中断方式
        if *DEV_NON_BLOCKING_ACCESS
          .exclusive_access() {
            ...
            let task_cx_ptr = ...;
            schedule(task_cx_ptr);
        } else {
        // 如果是轮询方式
            ...
        }
    }
    //write_block与read\_block处理类似
    fn write_block(...){...}
}
```

运行 rCore

进入 os 目录，使用命令 `make run` 启动 rCore，如图64所示可以看到 rCore 进入了控制台并且文件系统成功加载出了一系列文件，证明文件系统对接的块设备驱动程序工作正常。输入命令打开对应文件（例如 pipetest），可以看到 rCore 正常运行，并回到了控制台命令行。说明该 virtio-blk 块设备驱动程序可以在 rCore 上正常工作。

```
pipetest
adder_peterson_yield
huge_write_mt
gui_snake
inputdev_event
cat
eisenberg
**************/
Rust user shell
>> pipetest
Read OK, child process exited!
pipetest passed!
>>
```

大纲

可继续改进的部分

进一步提高模块化程度

由于本研究遵循 virtio 协议，因此应能与其他 virtio 协议驱动协调工作。然而，与其他 virtio 驱动共同工作时，势必会产生一些冗余代码。未来的研究希望能分离出 virtio 协议依赖部分，使操作系统开发者在使用多种 virtio 驱动时无需重复引入冗余代码。

实现更多 Feature

virtio 规定的许多功能尚未在本研究中实现，这使得对块设备的许多高级特性，例如分区块设备及其独特的空间布局和拓扑信息，无法获得更优支持，可能会阻碍操作系统在调度时的进一步优化。未来的研究将分析块设备的物理特性，逐步实现 virtio 支持的高级特性。

支持多队列

I/O 命令被挂起后携带唯一的标识符（token）在循环队列中排队。对于零散的 I/O 操作，这种方法对 CPU 利用率的提升较为显著，但当 I/O 操作较为密集时，CPU 仍会因为 I/O 操作未完成而不得不等待。因此，将来可以在队列上做进一步优化，例如多队列。该改进不属于异步本身，但可以提升异步效果。

总结

本研究设计并实现了一个遵循 virtio 协议的跨操作系统异步块设备驱动模块。该模块支持更多块设备特性，既能为块设备提供通用服务，又能依据块设备支持的高级特性提供更多支持。同时，该解决方案不依赖任何特定的异步运行时，提供了一种面向底层设备的异步解决方案。模块化设计极大提高了其可移植性。

本研究或能减轻操作系统开发者的开发负担，使其只需导入依赖即可在操作系统中直接使用块设备驱动，从而增强了操作系统的可移植性，降低了开发成本。同时，得益于模块化和低耦合性的设计，本研究也能为操作系统的学习者减轻分析和理解操作系统内核的难度。

本研究仍有大量改进工作可以继续完成，其前景十分广阔，具有巨大的实用性和经济价值。遵循 virtio 协议，是对跨操作系统外设驱动问题提出的一种具有实际意义的可持续改进的现实解决方案。

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

感谢各位专家老师

请您批评指正

答辩人：董若扬

导　师：陆慧梅

时　间：2024/5/27