摘要：

# 引言

## 研究背景及意义

## 研究现状

## 研究内容

# I/O设备

早期的UNIX把I/O设备分为两类：块设备（block device）和字符设备（character device。块设备每个块有独立的地址，例如磁盘。块设备的I/O传输以一个或多个完整的（连续的）块为单位。字符设备以单个字符为单位发送或接收一个字符流，例如串口，键盘等。字符设备不需要寻址，只需访问I/O设备提供的相关接口即可获得/发出字符信息流。

随着网络的普及，又出现了一类设备：网络设备。网络面向报文而不是面向字符流或数据块，因此将网络设备映射为常见的文件比较困难。随着个人电脑的普及，各种类型的I/O设备层出不穷。GPU、声卡、显卡等外设已经很难归类到上述的三种分类中，所以它们也就形成了各自独立的设备类型。各种设备出现时间有早晚，功能特点各异，这也使得现有的操作系统在设备驱动的设计和实现上面比较繁杂。

## CPU连接外设的方式

### 简单设备

在计算机发展的早期阶段，由于CPU连接的设备较少且性能较低，CPU可以直接通过I/O接口（例如嵌入式系统中的通用输入输出GPIO接口）控制I/O设备（例如简单的发光二极管等）。这种情况在简单的单片机和微处理器控制设备中非常常见。其特点是CPU发出I/O命令或数据，可以立即驱动I/O设备并产生相应的效果。

### 基于总线连接

随着计算机技术的发展，CPU与I/O设备的连接逐渐增多，因此引入了I/O控制器作为中间层，例如串口控制器。通过对I/O控制器进行编程，CPU可以控制各种设备，并通过访问相关寄存器获取设备的当前状态。然而，CPU需要轮询检查设备情况，导致对于低速设备（如串口）而言，CPU利用率较低。随着设备增多，I/O控制器也趋向通用化，能够连接不同设备并进行集中管理。

为了简化CPU与各种设备的连接，总线（bus）被引入。总线规定了连接设备需要共同遵循的连接方式和I/O时序等。不同总线（如SPI、I2C、USB、PCI等）具有不同的连接方式和I/O时序。

### 支持中断的设备

随着处理器技术的快速发展，CPU与外设之间的性能差距逐渐增大，导致CPU在等待外设完成操作时的利用率下降。为了解决这一问题，I/O控制器采用了中断机制的扩展，如Intel推出的8259可编程中断控制器。通过这一机制，CPU在发出I/O命令后无需忙等，而是可以执行其他任务。当外设完成I/O操作后，通过I/O控制器产生外部中断，从而促使CPU响应。这样，CPU和外设可以并行执行任务，提高整个系统的执行效率。

### 高吞吐量设备

随着外设技术的迅速发展，一些高性能外设（如SSD、网卡等）的性能不断提升。然而，当每次中断所产生的I/O数据传输量较少时，例如硬盘或SSD需要在短时间内传输大量数据时，频繁中断CPU会导致总体中断处理开销增大，从而降低系统效率。为解决这一问题，引入了DMA（Direct Memory Access，直接内存访问）控制器（如Intel推出的8237DMA控制器等）。通过DMA控制器，外设可以在CPU不访问内存的时间段内，以数据块的方式直接进行外设和内存之间的数据传输，而无需CPU干预。这样一来，I/O设备的传输效率大大提高。CPU只需在数据传输开始前发出DMA指令，并在外设完成DMA操作后响应其发出的中断信息即可。

## I/O传输方式

### Programmed I/O

PIO（Programmed I/O）是指CPU通过发出I/O指令的方式来进行数据传输的一种方式。PIO方式可以进一步细分为基于Memory-mapped的PIO（MMIO）和Port-mapped的PIO（PMIO）。在MMIO中，I/O设备的物理地址被映射到内存地址空间，这使得CPU可以通过普通的内存访问指令将数据传送到I/O设备在主存中的位置，从而完成数据传输。

相对而言，采用PMIO方式的I/O设备拥有独立的地址空间，与内存地址空间分离。若CPU要访问这些I/O设备，则需使用特殊的I/O指令，例如x86处理器中的`IN`和`OUT`指令。通过这些I/O指令，CPU可以直接访问设备，实现PMIO方式的数据传输。

### Interrupt based I/O

采用PIO方式让CPU获取外设执行结果时，I/O软件中存在一个循环，CPU不断读取外设相关寄存器，直到收到外设可继续执行I/O操作的信息后才能进行其他任务。当外设（如串口）的处理速度远低于CPU时，CPU将陷入忙等状态，效率低下。

中断机制的引入显著减轻了CPU的负担。CPU可以通过PIO方式通知外设，只要I/O设备有CPU需要的数据，便会发出中断请求信号。CPU发送通知后，即可继续执行与I/O设备无关的任务。中断控制器会检查I/O设备是否准备好传输数据，并向CPU发送中断请求信号。当CPU检测到中断信号时，会打断当前执行，并处理I/O传输。

### Direct Memory Access

外设每传输一个字节都触发一次中断会显著降低系统执行效率。为解决这一问题，DMA（Direct Memory Access，直接内存访问）技术被引入到计算机系统中，以实现快速数据传输。DMA允许外设直接将数据传输到内存，无需CPU直接参与。这样一来，CPU能够从I/O任务中解放出来，提升系统整体性能。DMA操作一般由DMA控制器管理。当CPU需要读取或写入设备数据时，它会向DMA控制器发出准备请求，然后DMA控制器在后续阶段直接完成数据传输到目标位置。

## I/O设备抽象

### I/O接口的交互协议

对于外设而言，其关键组成部分包括两个方面。首先是对外展示的设备I/O接口（hardware I/O interface），操作系统通过这一接口与外设进行通信和控制。每个设备都具有特定的接口和典型的交互协议。其次是内部结构，即对内的物理实现，包含设备内部的功能和结构。由于外设接口的多样性，操作系统在统一管理外设方面面临挑战。

在高度抽象的角度下，软件管理设备时，关注的焦点是简化设备接口而不是设备的内部结构。一个简化的抽象设备接口通常包括三个关键部分：状态、命令和数据。软件可以读取设备的当前状态，并基于此状态决定下一步的I/O访问请求；通过向设备发送一系列命令，软件可以请求设备执行特定的I/O访问操作；在I/O访问操作中，涉及将数据发送给设备或从设备接收数据。以下是CPU与设备之间的I/O接口交互协议：

```rust

while STATUS == BUSY {}; // 等待设备执行完毕

DATA = data; // 把数据传给设备

COMMAND = command; // 发命令给设备

while STATUS == BUSY {}; // 等待设备执行完毕

```

在引入中断机制后，简化的抽象设备接口需要涵盖四个关键部分：状态、命令、数据和中断。CPU与设备之间的I/O接口交互协议如下所示：

中断机制允许CPU的高速计算与外设的慢速I/O操作可以重叠，使得CPU无需等待外设完成操作，从而实现CPU与外设的并行执行，这是提高CPU利用率和系统效率的关键。从软件角度来看，引入DMA机制以提高大块数据传输效率并没有改变抽象设备接口的四个部分。仅仅是上面协议伪码中的 `data` 变成了 `data block`。这样，传输单个数据产生的中断频率会显著降低，进一步提高CPU利用率和系统效率。

### 基于文件的I/O设备抽象

在二十世纪七十到八十年代，计算机专家积极探索为I/O设备提供统一抽象的方法。最初，他们将专门用于存储类I/O设备的文件概念进行了扩展，认为所有的I/O设备都可以被视为文件。这一概念在传统UNIX系统中得到了体现，即设备文件。所有的I/O设备都以文件的形式呈现，可以通过诸如 `open/close/read/write` 的文件访问接口进行处理。在Linux系统下，可以通过执行 `$ ls /dev` 命令来查看各种设备文件。

然而，由于各种设备的功能多样化，仅仅依靠 `read/write` 这样的方式难以有效地与设备进行交互。因此，UNIX的后续设计者提出了一个独特的系统调用，即 `ioctl`（input/output control）系统调用。`ioctl` 是专门用于设备输入输出操作的系统调用，它接受与设备相关的请求码作为参数，系统调用的功能完全取决于设备驱动程序对请求码的解释和处理。例如，CD-ROM驱动程序可以弹出光驱，因此操作系统可以设定一个对应该操作的 `ioctl` 请求码。当应用程序发出带有CD-ROM设备文件描述符和 \*\*弹出光驱\*\* 请求码的 `ioctl` 系统调用请求时，操作系统中的CD-ROM驱动程序会识别出这个请求码，并执行弹出光驱的I/O操作。

尽管基于设备文件的设备管理表面上得到了大部分通用操作系统的支持，而且 `ioctl` 系统调用也非常灵活，但其问题在于它过于灵活。请求码的定义缺乏规律，文件接口过于面向用户应用，未能挖掘出操作系统在处理I/O设备时的共性特征。因此，文件这一抽象并未完全覆盖操作系统对设备管理的整个执行过程。

### 基于流的I/O设备抽象

在二十世纪八十到九十年代的UNIX操作系统发展过程中，随着网络等更加复杂的设备出现，人们提出了面向I/O设备管理的新抽象，即流（stream）。1984年，Dennis M. Ritchie撰写了一份技术报告，名为“A Stream Input-Output System”，详细介绍了基于流的I/O设备抽象设计。这一设计的目标在于将UNIX中管道（pipe）机制拓展到内核的设备驱动中。

流是用户进程和设备或伪设备之间的全双工连接，由多个线性连接的处理模块组成，类似于shell程序中的管道，用于数据双向流动。流中的模块通过消息传递进行通信，而模块不需要直接访问邻居模块的其他数据。每个模块只为每个邻居提供一个入口点，即一个接受消息的例程。

流的末端提供接口以与操作系统的其他部分交互。用户进程的写操作和输入/输出控制请求被转换成发送到流的消息，而读请求则从流中获取数据并传递给用户进程。流的另一端是设备驱动程序模块，它接收来自用户进程的数据并将其发送到设备；同时，它将设备检测到的数据和状态转换为消息，并发送到用户进程所在的流中。整个过程中会经过多个中间模块，这些模块以各种方式处理或过滤消息。

在实现上，当设备打开时，流中的两个末端的内核模块会自动连接；中间模块是根据用户程序的请求动态添加的。为了便于动态插入不同的流处理模块，这些中间模块的读写接口遵循相同的语义约束并相互兼容。

每个流处理模块都有一对队列，分别用于每个方向的数据传输。队列包括数据队列本身、两个例程和一些状态信息。其中，put例程用于将消息放入数据队列，而服务例程则在有工作要做时执行。状态信息包括指向下一个队列的指针、各种标志以及指向队列实例化所需的附加状态信息的指针。

### 基于virtio的I/O设备抽象

在二十一世纪，随着互联网和云计算的兴起，数据中心的物理服务器通过虚拟机技术运行多个虚拟机成为主流。然而，存在多种虚拟机技术，如Xen、VMware、KVM等，它们要求支持虚拟化不同处理器架构和各种外设，并且要求让以Linux为代表的guest操作系统能够高效地运行在其上。这为虚拟机和操作系统带来了复杂性和困难。

为解决这一问题，IBM工程师Rusty Russell提出了一组通用I/O设备抽象——virtio规范。虚拟机提供virtio设备的实现，而virtio设备具有统一的接口，因此guest操作系统只需实现这些通用接口即可管理和控制各种virtio设备。虚拟机与guest操作系统之间的通信通道采用基于共享内存的异步访问方式，效率非常高。虚拟机会将相关virtio设备的I/O操作转换成物理机上的物理外设的I/O操作，从而完成整个I/O处理过程。

由于virtio设备的设计，虚拟机无需模拟真实的外设，因此可以设计出一种统一且高效的I/O操作规范，让guest操作系统处理各种I/O操作。这种I/O操作规范形成了基于virtio的I/O设备抽象，并逐渐成为虚拟I/O设备的事实标准。

外部设备为CPU提供存储、网络等多种服务，是计算机系统中除了运算功能之外最为重要的功能载体。CPU与外设之间通过某种协议传递命令和执行结果；虽然virtio协议最初设计用于虚拟机外设，但随着应用范围的扩展，它正在朝着更适合物理机外设使用的方向演进。

## I/O执行模型

根据Richard Stevens的经典著作《UNIX Network Programming Volume 1: The Sockets Networking》第6.2节“I/O Models”所述，UNIX环境中的I/O系统调用具有多种不同类型的执行模型，可大致分为五种I/O执行模型（IO Model）。

在这些模型中，当用户进程发出一个 `read` 系统调用时，主要经历两个阶段：

1. 等待数据准备就绪（Waiting for the data to be ready）

2. 将数据从内核拷贝到用户进程中（Copying the data from the kernel to the process）

这五种IO模型在这两个阶段有不同的处理方式，并且阻塞与非阻塞关注的是进程的执行状态，而同步和异步关注的是消息通信机制：

- 阻塞：进程执行系统调用后会被阻塞，直到I/O操作完成。

- 非阻塞：进程执行系统调用后不会被阻塞，即使I/O操作未完成，进程也会继续执行。

- 同步：用户进程与操作系统（设备驱动）之间的操作是经过双方协调的，步调一致的。

- 异步：用户进程与操作系统（设备驱动）之间无需协调，可以各自独立进行操作。

这些模型的选择取决于应用程序的需求和设计考虑，不同的场景可能会选择不同的IO模型以达到最佳的性能和可维护性。

阻塞IO和非阻塞IO的区别在于第一阶段，即内核数据准备好之前是否阻塞用户进程；同步IO和异步IO的区别在于第二阶段，即数据从内核复制到用户空间时用户进程是否被阻塞或参与。

阻塞IO（Blocking IO）、非阻塞IO（Non-blocking IO）、多路复用IO（IO Multiplexing）、信号驱动IO（Signal-driven IO）都属于同步IO模型。这些模型在第二阶段（实际IO操作）中需要用户进程参与，因此称为同步IO模型。

虽然执行非阻塞IO系统调用的进程在第一阶段未被阻塞，但在第二阶段（实际IO操作）时，内核会阻塞用户进程，因为数据需要从内核复制到用户空间。

异步IO不同，用户进程发起IO操作后立即返回，直到内核通知IO完成。整个过程中，用户进程不会被阻塞。

### 阻塞IO（blocking IO）

阻塞IO（blocking IO）的特点：在I/O执行的两个阶段（等待数据和拷贝数据）中，用户进程都处于阻塞状态。

当用户进程发出 `read` 系统调用时，如果所需数据不在I/O缓冲区中，内核将用户进程置于阻塞状态，并向磁盘驱动程序发出I/O操作请求。直到数据从磁盘传输到I/O缓冲区，并且内核将数据从缓冲区拷贝到用户进程的buffer中，并唤醒用户进程，`read` 系统调用才完成，用户进程才从阻塞状态中恢复。

但是，如果所需数据恰好位于内存中，内核会立即将数据从I/O缓冲区拷贝到用户进程的buffer中，并且不会让用户进程处于阻塞状态。这是由于I/O缓冲的优化结果，减少了用户进程的等待时间。

### 非阻塞IO（non-blocking IO）

非阻塞IO（non-blocking IO）的特点：非阻塞系统调用在被调用之后会立即返回，而不会阻塞用户进程。

当用户进程发出 `read` 系统调用时，内核发现所需数据不在I/O缓冲区中，不会让用户进程处于阻塞状态，而是立即返回一个error。用户进程收到error后，知道数据还没有准备好，因此可以再次发送 `read` 操作，这个过程可以重复多次。

当磁盘驱动程序将数据从磁盘传输到I/O缓冲区并通知内核后，内核收到通知并再次收到用户进程的system call时，立即将数据从I/O缓冲区拷贝到用户进程的buffer中。用户进程不会被内核阻塞，而是需要不断主动询问内核所需数据是否准备好。

### 多路复用IO（IO multiplexing）

多路复用IO模型中，`select` 和 `epoll` 等系统调用对应的是事件驱动IO（event-driven IO）。它们的优势在于可以通过单个进程同时处理多个文件或网络连接的I/O操作。

基本工作机制是通过 `select` 或 `epoll` 系统调用不断轮询用户进程关注的所有文件句柄或socket。当某个文件句柄或socket有数据到达时，`select` 或 `epoll` 系统调用会返回到用户进程，然后用户进程再调用 `read` 系统调用，将数据从内核的I/O缓冲区拷贝到用户进程的buffer中。

在多路复用IO模型中，用户进程关注的每一个文件句柄或socket通常都设置为非阻塞，但用户进程被 `select` 或 `epoll` 系统调用阻塞。需要注意的是，`select/epoll` 的优势不会导致单个文件或socket的I/O访问性能更好，而是在存在大量文件或socket的I/O访问情况下，才能发挥总体效率的优势。

### 信号驱动IO（signal driven I/O）

进程发起 `read` 系统调用时，向内核注册信号处理函数后立即返回，进程不会被阻塞，而是继续执行。当内核中的IO数据准备就绪时，会向进程发送一个信号，进程在信号处理函数中调用IO读取数据。该模型的特点是采用了回调机制，增加了开发和调试应用的难度。

### 异步IO（Asynchronous I/O）

用户进程调用 `async\_read` 异步系统调用后，即可立即执行其他任务。从内核的角度看，收到 `async\_read` 异步系统调用后，内核会立即返回，不会造成用户进程阻塞。然后，内核会等待数据准备就绪，将数据复制到用户内存。当数据复制完成后，内核会通知用户进程，告知读取操作已完成。

# 外设平台

## 驱动程序概述

驱动程序是操作系统与外部设备之间的接口，允许它们之间进行通信。从操作系统的角度来看，驱动程序位于I/O设备和应用程序之间，处理各种I/O操作，需要站在协助所有进程的全局角度来处理I/O。

驱动程序的编写需要涉及以下操作：

1. 定义设备相关的数据结构，包括设备信息、状态和操作标识。

2. 设备初始化，配置设备，分配内存，设置中断处理例程。

3. 如果设备产生中断，需要编写中断处理例程。

4. 根据上层模块（如文件系统）的要求，向I/O设备发送命令，检测和处理错误。

5. 与上层模块或应用程序交互，满足其需求。

驱动程序的I/O操作模式主要有两种：同步和异步。在同步模式下，应用程序发出I/O请求后会等待操作完成，而在异步模式下，应用程序不会等待操作完成，而是继续执行其他任务。

编写驱动程序需要了解硬件规范，考虑同步互斥问题，并理解驱动程序可能处于的上下文环境，以确保程序正确执行。

## 硬件系统架构

### 设备树

首先，了解操作系统管理的计算机硬件系统——`QEMU riscv-64 virt machine`，其中包括各种外部设备。`virt`表示一台虚拟的RISC-V 64位计算机，可以通过参数`-cpu num`配置CPU数量，通过参数`-m numM/G`配置内存大小。虚拟计算机连接了许多外部设备，每个设备物理连接到父设备，最终通过总线等连接成设备树。QEMU可以将模拟的机器信息导出为dtb格式的二进制文件，并通过`dtc`（Device Tree Compiler）工具转换为可读的文本文件。

\*\*设备树与设备节点属性\*\*

设备树是一种数据结构，用于描述硬件系统的结构和功能，并提供给操作系统使用。它以文本文件形式存在，包含以下信息：

- 处理器类型和数量

- 板载设备（如存储器、网卡、显卡等）类型和数量

- 硬件接口（如 I2C、SPI、UART 等）类型和地址信息

设备树中的节点描述硬件设备的信息，每个节点包含一个或多个属性，每个属性都是键值对，用于描述设备的特定信息。操作系统通过这些节点信息识别和初始化设备。常见的设备节点属性包括：

- compatible：设备类型，如“virtio,mmio”表示设备通过virtio协议、MMIO方式驱动。

- reg：设备在系统中的地址空间位置。

- interrupts：设备支持的中断信号。

设备树在嵌入式系统中广泛应用，是一种将硬件信息传递给操作系统的常用方法。在桌面和服务器系统中，PCI总线类似于设备树，可用于访问PCI设备的地址空间信息。

### 传递设备树信息

在启动过程中，操作系统需要获取计算机系统中所有接入的设备信息。这一任务通常由引导加载程序（bootloader）完成，在RISC-V架构中，常见的引导加载程序包括OpenSBI或RustSBI固件。引导加载程序负责探测各个外设，包括物理内存，并将探测结果保存在物理内存中的某个位置，通常以设备树二进制对象（Device Tree Blob，DTB）的格式存储。随后，引导加载程序会启动操作系统，将DTB的物理地址加载到寄存器a1中，将硬件线程ID（HART ID）加载到寄存器a0中，然后跳转到操作系统的入口地址开始执行。

### 解析设备树信息

在virtio\_probe函数中，会查找virtio设备节点的reg属性，以确定virtio设备的具体类型（例如块设备类型）。这使得我们能够对特定的virtio设备进行初始化并执行I/O操作。通常，针对各种设备的控制主要基于对特定内存地址的读写，这称为内存映射I/O（MMIO）访问方式。

然而，将设备与操作系统结合后，设备的管理变得更加复杂。除了对设备硬件规范的深入了解外，还需要与操作系统内核中的其他模块（如文件系统等）进行交互。这种交互增加了系统的复杂性。

### 平台级中断控制器

\*\*中断控制器（Interrupt Controller）\*\*

中断控制器是一种硬件，用于处理来自多个不同I/O设备的中断请求（IRQ）。这些中断请求可能同时发生，中断控制器根据 IRQ 的优先级对它们进行排序，并将优先级最高的IRQ传递给处理器，以执行相应的中断处理例程（ISR）。

在之前的操作系统中，已经涉及到中断处理，但尚未处理外设（例如RISC-V处理器产生的时钟中断）产生的中断。若要让操作系统处理外设中断，就需要对中断控制器进行初始化设置。在RISC-V中，连接外设的I/O控制器的一个关键组成部分是平台级中断控制器（PLIC）。PLIC汇聚各种外设的中断信号，并将其转发给RISC-V CPU。CPU执行相应的中断处理程序来响应中断。通过RISC-V的mie寄存器中的meie位，可以控制是否接收外部中断信号。此外，通过RISC-V中M模式的中断委托机制，也可以在RISC-V的S模式下，通过sie寄存器中的seie位，对中断信号是否接收进行控制。

CPU可以通过MMIO方式来对PLIC进行管理，下面是一些与PLIC相关的寄存器：

寄存器 地址 功能描述

Priority 0x0c00\_0000 设置特定中断源的优先级

Pending 0x0c00\_1000 包含已触发（正在处理）的中断列表

Enable 0x0c00\_2000 启用/禁用某些中断源

Threshold 0x0c20\_0000 设置中断能够触发的阈值

Claim 0x0c20\_0004 按优先级顺序返回下一个中断

Complete 0x0c20\_0004 写操作表示完成对特定中断的处理

要让操作系统响应外设的中断，需要进行两方面的初始化工作。首先，完成中断初始化过程，包括将s模式下的sie寄存器中的seie位设置为1，以便CPU接收通过PLIC传来的外部设备中断信号。其次，通过MMIO方式对PLIC的寄存器进行初始设置，以确保外设产生的中断能够传递到CPU处。这些主要操作包括：

1. 设置外设中断的优先级。

2. 设置外设中断的阈值，优先级小于等于阈值的中断会被屏蔽。

3. 激活外设中断，即将Enable寄存器中外设中断编号对应的位设置为1。

但当外设产生中断后，CPU并不知道具体是哪个设备传来的中断。此时，可以通过读取PLIC的Claim寄存器来了解。Claim寄存器会返回PLIC接收到的优先级最高的中断；如果没有外设中断产生，读Claim寄存器会返回0。

操作系统在收到中断并完成中断处理后，还需要通知PLIC中断处理已完成。CPU需在PLIC的Complete寄存器中写入对应中断号为索引的位，以通知PLIC中断已处理完毕。

# virtio设备驱动程序

## virtio设备

### virtio概述

Rusty Russell在2008年左右设计了virtio协议，并开发了虚拟化解决方案lguest，形成了VirtIO规范（Virtual I/O Device Specification）。该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。

在传统的虚拟机模拟外设方案中，Hypervisor需要截获所有的I/O请求指令，并模拟这些指令的行为，导致较大的性能开销。而在virtio方案中，模拟的外设功能最小化，其数据面接口主要与guest VM共享内存，控制面接口主要基于内存映射的寄存器和中断机制。这样，当guest VM使用底层host主机资源时，Hypervisor只需处理少量寄存器访问和中断机制，实现了高效的I/O虚拟化。

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。

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。

总的来说，virtio是一个接口，允许运行在虚拟机上的操作系统和应用软件通过访问virtio设备使用其主机的设备。这些设备具备功能最小化的特征，使得操作系统能够设计出轻量高效的设备驱动程序。

### virtio架构

总体而言，virtio架构可以分为上、中、下三层。上层包括运行在QEMU模拟器上的前端操作系统中的各种驱动程序（Front-end drivers）；下层是在QEMU中模拟的各种虚拟设备（Device）；中间层是传输（transport）层，负责驱动程序与虚拟设备之间的交互接口，包含virtio接口定义和实现。

在操作系统中，virtio驱动程序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操作卸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。

### I/O设备基本组成结构

#### virtio设备基本组成要素

virtio设备的基本组成要素包括：

- 设备状态域（Device status field）：用于表示virtio设备的当前状态，通常在设备初始化时使用。

- 特征位（Feature bits）：用于描述virtio设备的功能特性，可以由驱动程序和设备之间协商确定。

- 通知（Notifications）：用于在设备和驱动程序之间进行异步事件通知，例如新数据到达或操作完成。

- 设备配置空间（Device Configuration space）：用于存储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（virtqueue）：用于驱动程序和设备之间的数据传输和命令交互，是实现高效通信的关键。

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。

#### virtio设备呈现模式

virtio设备支持三种不同的设备呈现模式：

1. Virtio Over MMIO：虚拟设备直接挂载到系统总线上，在内存映射的形式下呈现。在实验中的虚拟计算机通常采用这种呈现模式。

2. Virtio Over PCI BUS：遵循PCI规范，将设备挂载到PCI总线上，作为virtio-pci设备呈现。在QEMU虚拟的x86计算机上通常采用这种模式。

3. Virtio Over Channel I/O：主要用于虚拟IBM s390计算机，virtio-ccw使用这种基于channel I/O的机制。

在QEMU模拟的RISC-V计算机上，采用的是Virtio Over MMIO的呈现模式。因此，在实现设备驱动时，只需找到相应virtio设备的I/O寄存器等内存形式呈现的地址空间，即可对I/O设备进行初始化和管理。

#### virtio设备特征描述

virtio设备的特征描述包括设备特征位和设备配置空间。

\*\*特征位：\*\*

特征位用于表示VirtIO设备具有的各种特性和功能。其中，bit0 – 23是特定设备可以使用的feature bits，bit24 – 37用于预留给队列和特征协商机制，而bit38以上保留给未来其他用途。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。

\*\*设备配置空间：\*\*

设备配置空间通常用于配置不经常变动的设备参数（属性），或者在初始化阶段需要设置的设备参数。设备的特征位中包含一个表示配置空间是否存在的bit位，并可以通过在特征位的末尾添加新的bit位来扩展配置空间。

在初始化virtio设备时，设备驱动程序需要根据virtio设备的特征位和配置空间来了解设备的特性，并对设备进行适当的初始化。

#### virtio设备状态表示

\*\*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。这个状态域包括以下六种状态：

1. ACKNOWLEDGE (1)：驱动程序发现了设备，并确认这是一个有效的virtio设备。

2. DRIVER (2)：驱动程序知道如何驱动这个设备。

3. FAILED (128)：由于某种错误原因，驱动程序无法正常驱动这个设备。

4. FEATURES\_OK (8)：驱动程序已经与设备就设备特性达成一致。

5. DRIVER\_OK (4)：驱动程序已加载完成，设备可以正常工作。

6. DEVICE\_NEEDS\_RESET (64)：设备触发了错误，需要重置才能继续工作。

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。

\*\*I/O传输状态\*\*

在设备驱动程序控制virtio设备进行I/O传输过程中，会遇到以下几种状态：

1. I/O请求状态：驱动程序指示I/O请求队列的当前位置信息，设备可以根据这些信息执行相应的I/O传输操作。

2. I/O处理状态：设备正在处理收到的I/O请求。

3. I/O完成状态：设备已完成I/O请求的处理，并通知设备驱动程序。

4. I/O错误状态：发生了I/O请求处理过程中的错误。

5. I/O后续处理状态：设备驱动程序在接收到I/O完成通知后进行的后续处理。

例如，对于virtio\_blk设备驱动程序而言，当发出一个读设备块的I/O请求后，设备处于I/O请求状态。设备接收到请求后，进入I/O处理状态，处理读取操作。完成读取后，设备进入I/O完成状态，并通知设备驱动程序。驱动程序在接收到通知后，进行I/O后续处理，将数据传递给文件系统进行进一步处理，或者进行错误恢复处理。

#### virtio设备交互机制

\*\*Notification通知\*\*

在virtio设备的交互过程中，使用基于Notifications的事件通知机制。这意味着驱动程序和设备需要相互通知对方，以便处理数据。驱动程序可以通过门铃（doorbell）机制向设备发送通知，通常通过PIO或MMIO方式访问设备特定的寄存器。QEMU会拦截这些访问，并通知其模拟的设备。而设备通知驱动程序则通常使用中断机制。在QEMU中，中断被注入到CPU中，使其响应并执行相应的中断处理例程，从而完成对I/O执行结果的处理。

\*\*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virtio设备可以拥有零个或多个virtqueue。每个virtqueue占用多个物理页，并可以通过各种数据结构（如数组、环形队列等）来实现。virtqueue用于设备驱动程序向设备发送I/O请求命令和相关数据（例如磁盘块读写请求和读写缓冲区），同时也用于设备向设备驱动程序发送I/O数据（例如接收的网络包）。

#### virtqueue虚拟队列

在virtio协议中，virtqueue是一个关键部分，用于在virtio设备和驱动程序之间进行批量数据传输的机制和抽象表示。在设备驱动程序的实现和Qemu中的virtio设备模拟中，virtqueue被视为一种数据结构，用于执行各种数据传输操作。

当描述virtqueue时，有时会将其与vring（即virtio-rings或VRings）等同视，也有时会将二者单独描述为不同的对象。在这里，我们将它们单独描述，因为vring是virtqueues的主要组成部分，是实现virtio设备和驱动程序之间数据传输的数据结构。虽然vring本质上是virtio设备和驱动程序之间的共享内存，但virtqueue不仅仅是vring。

virtqueue由三个主要部分组成：

1. \*\*描述符表（Descriptor Table）\*\*：描述符表是一个由描述符组成的数组，每个描述符描述了一个内存缓冲区的地址和长度。这些内存缓冲区包含了I/O请求的命令/数据（由virtio设备驱动填写），也可能包含了I/O完成的返回结果（由virtio设备填写）等信息。

2. \*\*可用环（Available Ring）\*\*：这是一个vring，记录了virtio设备驱动程序发出的I/O请求索引，即被virtio设备驱动程序更新的描述符索引的集合。virtio设备需要读取这些索引并执行相关的I/O操作。

3. \*\*已用环（Used Ring）\*\*：这是另一个vring，记录了virtio设备发出的I/O完成索引，即被virtio设备更新的描述符索引的集合。virtio设备驱动程序需要读取这些索引，并进一步处理I/O操作的结果。

\*\*描述符表（Descriptor Table）\*\*

描述符表用于指向virtio设备中的I/O传输请求缓冲区信息，通常由一系列描述符组成，数量由队列大小（Queue Size）决定。每个描述符包含以下信息：

1. \*\*物理地址（addr字段）\*\*：指向I/O传输缓冲区的物理地址，即待读取或待写入数据的位置。

2. \*\*长度（len字段）\*\*：指定了缓冲区的长度，即待读取或待写入数据的大小。

3. \*\*下一个描述符指针（next指针）\*\*：指向下一个描述符的指针，用于将多个描述符链接成描述符链。这样的描述符链可以表示一个完整的I/O操作请求，包括I/O操作的命令、数据块和返回结果。

设备驱动程序在初始化过程中需要分配描述符表所需的内存空间，并在其中填写描述符的信息。在后续的I/O操作中，设备驱动程序将在描述符表中创建描述符链，并将其提交给virtio设备以执行相应的I/O操作。

\*\*可用环（Available Ring）\*\*

可用环是一个环形队列，用于驱动程序向设备发送I/O操作请求。其结构包含了一系列的条目（items），这些条目只能由驱动程序写入，而设备则会读取这些条目。每个条目包含了一个描述符链的头部描述符的索引值，用于指示设备需要执行的I/O操作请求的位置。

可用环的头指针（idx）和尾指针（last\_avail\_idx）用于表示可用条目的范围。当驱动程序写入新的描述符链时，头指针会递增，表示有新的可用条目。设备通过读取可用环中的条目来获取驱动程序发出的I/O操作请求对应的描述符链，然后执行相应的I/O操作。

描述符链通常由多个描述符组成，每个描述符指向一个缓冲区，用于存储数据或者命令。例如，在磁盘读取操作中，描述符链可能包含一个用于读取磁盘块的描述符、一个用于存储读取数据的缓冲区描述符，以及一个用于存储操作结果的描述符。设备会根据描述符链中的信息执行相应的I/O操作，例如读取磁盘块并将数据存储到缓冲区中。

总之，可用环是驱动程序向设备提交I/O操作请求的通道，设备通过读取可用环中的条目来获取需要执行的操作，并根据描述符链中的信息执行相应的I/O操作。

\*\*已用环（Used Ring）\*\*

已用环是一个环形队列，用于virtio设备向驱动程序发送已完成的I/O操作请求。其结构与可用环类似，包含一系列的条目，每个条目表示一个已完成的描述符链的头部描述符的索引值。

已用环的头指针（idx）和尾指针（last\_used\_idx）用于表示已用条目的范围。当设备完成一个I/O操作时，会将描述符链的头部描述符的索引值添加到已用环中，并更新头指针。驱动程序通过读取已用环中的条目来获取已完成的I/O操作请求的信息，包括操作的结果以及使用的描述符链。

例如，在磁盘读取操作完成后，virtio设备会将描述符链的头部描述符的索引值添加到已用环中，并在描述符链中存储操作的结果。驱动程序通过读取已用环中的条目来获取已完成的I/O操作请求的信息，然后进行进一步的处理，例如将数据传递给文件系统进行处理。

总之，已用环是virtio设备向驱动程序发送已完成的I/O操作请求的通道，驱动程序通过读取已用环中的条目来获取已完成的操作的信息。

基于virtqueue进行I/O操作的过程可以概括为以下步骤：

\*\*1. 初始化阶段（由驱动程序执行）：\*\*

在设备初始化过程中，virtio设备驱动程序首先分配virtqueue的内存空间，包括描述符表、可用环以及已用环。然后，驱动程序将这些部分的物理地址写入virtio设备的控制寄存器中，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。

\*\*2. 发起I/O请求阶段（由驱动程序执行）：\*\*

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，在描述符表中分配新的描述符（或描述符链），指向这些buffer。接着，将描述符（或描述符链的首描述符）的索引写入可用环中，并更新可用环的idx指针。最后，通过kick机制通知设备有新的请求。

\*\*3. 完成I/O请求阶段（由设备执行）：\*\*

当virtio设备接收到kick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（或描述符链的首描述符）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。

\*\*4. I/O后处理阶段（由驱动程序执行）：\*\*

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。

#### 基于MMIO方式的virtio设备

基于MMIO方式的virtio设备在操作系统中没有基于总线的设备探测机制，因此，操作系统采用Device Tree的方式来发现各种基于MMIO方式的virtio设备，以便获知设备的相关寄存器和中断信息。这些设备提供一组内存映射的控制寄存器，后跟一个设备特定的配置空间，形式上位于特定地址的内存区域。一旦操作系统找到这个内存区域，就可以获取与设备相关的各种寄存器信息。在virtio-drivers crate中，定义了基于MMIO方式的virtio设备的寄存器区域，以下是其中部分关键寄存器及其基本功能描述。在设备初始化和I/O操作中，将访问这些寄存器。

## virtio驱动程序

### 设备的初始化

操作系统在发现virtio设备后，驱动程序进行设备初始化的常规步骤如下：

1. 重启设备状态，将设备状态域设置为0。

2. 将设备状态域设置为“ACKNOWLEDGE”，表示已识别到设备。

3. 将设备状态域设置为“DRIVER”，表示驱动程序了解如何操作该设备。

4. 执行设备特定的安装和配置，包括协商特征位、建立virtqueue、访问设备配置空间等，将设备状态域设置为“FEATURES\_OK”。

5. 将设备状态域设置为“DRIVER\_OK”或者“FAILED”（如果出现错误）。

需要注意的是，上述步骤并非必须全部执行，但最终必须将设备状态域设置为“DRIVER\_OK”，以便驱动程序能够正常访问设备。

在virtio\_driver模块中，实现了通用的virtio驱动程序框架，各种virtio设备驱动程序共同的初始化过程包括：

1. 确定协商特征位，调用VirtIOHeader的begin\_init方法执行virtio设备初始化的第1-4步骤。

2. 读取配置空间，确定设备的配置情况。

3. 建立虚拟队列1至n个virtqueue。

4. 调用VirtIOHeader的finish\_init方法执行virtio设备初始化的第5步骤。

### 驱动程序与设备之间的交互

驱动程序与外设通过共享的virtqueue进行通信，virtqueue保存着设备驱动的I/O请求信息和设备的I/O响应信息。virtqueue由描述符表（Descriptor Table）、可用环（Available Ring）和已用环（Used Ring）组成。在设备驱动初始化过程中，虚拟队列会被创建。

当驱动程序向设备发送I/O请求时，它会在buffer中填充命令/数据，然后将每个buffer的起始地址和大小信息放入描述符表的描述符中，并将这些描述符链接在一起，形成描述符链。描述符链的起始描述符的索引信息会放入一个称为环形队列的数据结构中，其中一类是可用环，包含设备驱动发出的I/O请求所对应的描述符索引信息，另一类是已用环，包含设备发出的I/O响应所对应的描述符索引信息。

用户进程发起的I/O操作经过层层传递到驱动程序，驱动程序将I/O请求信息放入virtqueue的可用环中，并通过通知机制通知设备。设备收到通知后，解析可用环和描述符表，取出I/O请求并在内部进行实际I/O处理。处理完成后，设备将结果作为I/O响应放入已用环中，并通过通知机制通知CPU。驱动程序解析已用环，获取I/O响应的结果，并在进一步处理后，最终返回给用户进程。

### 发出I/O请求的过程

驱动程序向设备提供新的I/O请求信息的具体步骤如下：

1. 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。

2. 将描述符链头的索引放入可用环的下一个环条目中。

3. 如果可以进行批处理（batching），则重复执行步骤1和2，通过可用环将多个I/O请求添加到描述符链中。

4. 根据添加到可用环中的描述符链头的数量，更新可用环。

5. 向设备发送通知，指示“有可用的缓冲区”。

值得注意的是，在第3和第4步中，需要进行适当的内存屏障操作（Memory Barrier），以确保设备能够看到更新后的描述符表和可用环。

描述符表用于映射缓冲区以形成描述符链，其中每个缓冲区元素代表一个I/O请求的具体内容。下面是将缓冲区信息放入描述符表的操作步骤：

对于每个缓冲区元素 `b` ，执行以下操作：

1. 获取下一个空闲描述符表条目 `d` ；

2. 将 `d.addr` 设置为 `b` 的起始物理地址；

3. 将 `d.len` 设置为 `b` 的长度；

4. 如果 `b` 是设备可写的，则将 `d.flags` 设置为 `VIRTQ\_DESC\_F\_WRITE` ，否则设置为0；

5. 如果 `b` 之后还有一个缓冲元素 `c` ：

5.1 将 `d.next` 设置为下一个空闲描述符元素的索引；

5.2 将 `d.flags` 中的 `VIRTQ\_DESC\_F\_NEXT` 位置1；

更新可用环的操作包括以下步骤：

1. 首先，驱动程序将描述符链头的索引（即描述符表条目的索引）存储到可用环中。

2. 驱动程序可能会在更新idx之前添加多个描述符链，因此需要对已添加的描述符链数量进行计数，通常称为`added`。

伪代码示例如下：

```

avail.ring[(avail.idx + added++) % qsz] = head;

```

其中，`avail.ring`是可用环数组，`avail.idx`是可用环的当前索引，`qsz`表示可用环的大小，`head`是描述符链头的索引。

3. 更新可用环的`idx`指针，以反映驱动程序添加的描述符链的数量：

```

avail.idx += added;

```

4. 一旦驱动程序更新了可用环的`idx`指针，表示描述符及其指向的缓冲区可以被设备看到。

驱动程序必须在更新`idx`之前执行适当的内存屏障操作，以确保设备看到最新的描述符和缓冲区内容。

\*\*通知设备的操作\*\*

在Qemu virt虚拟计算机中包含virtio设备时，驱动程序通常通过对特定寄存器进行写操作来发出通知，这个寄存器代表了通知的“门铃”。

### 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。

# virtio\_blk块设备驱动分析

## virtio-blk设备的关键数据结构

首先定义virtio-blk设备的结构：

```rust

pub struct VirtIOBlk<'a, H: Hal> {

header: &'static mut VirtIOHeader,

queue: VirtQueue<'a, H>,

capacity: usize,

}

```

`VirtIOHeader` 数据结构中的 `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。其中，`free\_head` 成员指向空闲描述符链表的头部。在初始时，所有描述符通过 `next` 指针依次相连，形成空闲链表。`last\_used\_idx` 成员表示设备上次已取的已用环元素的位置。`avail\_idx` 成员表示可用环的索引值。

在这里提到的 `Hal` trait 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。

## 初始化virtio-blk设备

virtio-blk设备的初始化过程与一般的virtio设备初始化过程相似。在初始化过程中，virtio-blk设备驱动程序会读取设备的配置空间，了解设备的扇区个数、扇区大小和总容量等信息。然后，需要调用 `VirtQueue::new` 成员函数来创建虚拟队列 `VirtQueue` 数据结构的实例，以便进行后续的磁盘读写操作。这个函数主要完成以下任务：

- 设定虚拟队列的描述符条目数为16；

- 计算满足描述符表、可用环和已用环所需的物理空间的大小；

- 分配物理空间，创建虚拟队列；

- 调用 `VirtIOHeader.queue\_set` 函数将虚拟队列的相关信息（如内存地址）写入virtio-blk设备的MMIO寄存器中；

- 初始化VirtQueue实例中各个成员变量的值，主要包括 `dma`、`desc`、`avail`、`used` 等。

完成这些步骤后，virtio-blk设备和设备驱动之间的虚拟队列接口就建立了，可以进行I/O数据读写操作了。

## 操作系统对接virtio-blk设备初始化过程

在操作系统中表示virtio\_blk设备的全局变量 `BLOCK\_DEVICE` 的类型是 `VirtIOBlock`。这个全局变量封装了来自 `virtio\_drivers` 模块的 `VirtIOBlk` 类型，使得操作系统内核能够通过 `BLOCK\_DEVICE` 全局变量来访问virtio\_blk设备。

`VirtIOBlock` 结构中包含了 `condvars: BTreeMap<u16, Condvar>` 条件变量结构，用于在进程等待I/O读或写操作完成之前挂起进程。每个条件变量对应着一个虚拟队列条目的编号，意味着每次I/O请求都会绑定一个条件变量，使得发出请求的线程/进程可以被挂起。

在初始化代码中，首先出现了 `VIRTIO0`，这是Qemu模拟的virtio\_blk设备中I/O寄存器的物理内存地址。`VirtIOBlk` 需要这个地址来对 `VirtIOHeader` 数据结构表示的virtio-blk I/O控制寄存器进行读写操作，以完成对某个具体的virtio-blk设备的初始化过程。

此外，代码中还涉及到了 `VirtioHal` 结构，它实现了 `virtio\_drivers` 模块定义的 `Hal` trait，提供了DMA内存分配和虚实地址映射等操作，使得 `virtio\_drivers` 模块中的 `VirtIOBlk` 类型能够得到操作系统的服务。

## virtio-blk设备的I/O操作

在操作系统的virtio-blk驱动与virtio-drivers crate中virtio-blk裸机驱动对接的过程中，需要封装virtio-blk裸机驱动的基本功能，以完成以下服务：

1. 读磁盘块并挂起发起请求的进程/线程；

2. 写磁盘块并挂起发起请求的进程/线程；

3. 处理virtio-blk设备发出的中断，并唤醒相关等待的进程/线程。

在此过程中，virtio-blk驱动程序发起的I/O请求包含操作类型（读或写）、起始扇区（块设备的最小访问单位的一个扇区的长度512字节）、内存地址以及访问长度。请求处理完成后返回的I/O响应仅包含结果状态（成功或失败），以及读操作请求的读出扇区内容。

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，因此需要三个描述符。

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。

描述符表由一系列固定长度为16字节的描述符组成，其个数与环形队列的长度相等。

```rust

struct Descriptor {

addr: Volatile<u64>,

len: Volatile<u32>,

flags: Volatile<DescFlags>,

next: Volatile<u16>,

}

```

描述符表由四个域组成：

1. \*\*addr\*\*：表示某段内存的起始地址，长度为8个字节。

2. \*\*len\*\*：表示某段内存的长度，占用4个字节，因此代表的内存段最大为4GB。

3. \*\*flags\*\*：代表内存段的读写属性等，长度为2个字节。

4. \*\*next\*\*：代表下一个内存段对应的Descriptor在描述符表中的索引，通过next字段可以将一个请求对应的多个内存段连接成链表。

可用环 `AvailRing`：

```rust

struct AvailRing {

flags: Volatile<u16>,

/// A driver MUST NOT decrement the idx.

idx: Volatile<u16>,

ring: [Volatile<u16>; 32], // actual size: queue\_size

used\_event: Volatile<u16>, // unused

}

```

可用环由以下部分组成：

1. \*\*头部\*\*：包括`flags`和`idx`域。`flags`与通知机制相关，`idx`代表最新放入IO请求的编号，从零开始单调递增。将其对队列长度取余即可得该I/O请求在可用环数组中的索引。

2. \*\*可用环数组\*\*：用来存放I/O请求占用的首个描述符在描述符表中的索引。数组长度等于可用环的长度（如果未开启event\_idx特性）。

已用环 `UsedRing`：

```rust

struct UsedRing {

flags: Volatile<u16>,

idx: Volatile<u16>,

ring: [UsedElem; 32], // actual size: queue\_size

avail\_event: Volatile<u16>, // unused

}

```

已用环包含以下部分：

1. \*\*头部\*\*：包括`flags`和`idx`域。`flags`与通知机制相关，`idx`代表最新放入I/O响应的编号，从零开始单调递增。将其对队列长度取余即可得该I/O响应在已用环数组中的索引。

2. \*\*已用环数组\*\*：用来存放I/O响应占用的首个描述符在描述符表中的索引。数组长度等于已用环的长度（如果未开启event\_idx特性）。

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。针对写操作，具体实现如下：

1. 完整的virtio-blk的I/O写请求由三部分组成：表示I/O写请求信息的结构`BlkReq`，要传输的数据块`buf`，表示设备响应信息的结构`BlkResp`。这三部分需要三个描述符来表示。

2. 驱动程序处理：调用`VirtQueue.add`函数，从描述符表中申请三个空闲描述符，每项指向一个内存块，填写上述三部分的信息，并将三个描述符连接成一个描述符链表。

3. 驱动程序处理：调用`VirtQueue.notify`函数，写MMIO模式的`queue\_notify`寄存器，向virtio-blk设备发出通知。

4. 设备处理：virtio-blk设备收到通知后，通过比较`last\_avail`和`AvailRing`中的`idx`判断是否有新的请求待处理。如果有，则从描述符表中找到这个I/O请求对应的描述符链，获知完整的请求信息，并完成存储块的I/O写操作。

5. 设备处理：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项中，并更新idx，代表放入一个响应。如果设置了中断机制，还会产生中断来通知操作系统响应中断。

6. 驱动程序处理：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，则取出响应，并将完成响应对应的三项Descriptor回收，最后将结果返回给用户进程。

I/O读请求的处理过程与I/O写请求的处理过程几乎一样，唯一的区别在于`BlkReq`的内容不同。在写操作中，`req.type\_`是`ReqType::Out`，而在读操作中，`req.type\_`是`ReqType::In`。具体可以查看`virtio-drivers/src/blk.rs`文件中的`VirtIOBlk.read\_block`函数的实现。

基于轮询的I/O访问方式效率相对较低，因此需要实现基于中断的I/O访问方式。在支持中断的`write\_block\_nb`方法中：

相比于不支持中断的`write\_block`函数，`write\_block\_nb`函数更加简单，发出I/O请求后直接返回。`read\_block\_nb`函数的处理流程与此类似。响应中断的`ack\_interrupt`函数只完成了基本的virtio设备中断响应操作。在virtio-drivers中实现的virtio设备驱动不包含进程、条件变量等操作系统关键要素，只与操作系统内核对接，因此要完整实现基于中断的I/O访问方式，需要在操作系统内核中进行进一步的实现。

## 操作系统对接virtio-blk设备I/O处理

在操作系统中，文件系统模块与块设备驱动程序`VirtIOBlock`直接交互，而`VirtIOBlock`封装了virtio-drivers中实现的virtio\_blk设备驱动。在文件系统的介绍中，我们并未深入分析virtio\_blk设备。现在我们将介绍操作系统如何对接virtio\_blk设备驱动并完成基于中断机制的I/O处理过程。

首先，我们需要扩展文件系统对块设备驱动的I/O访问要求。这体现在`BlockDevice` trait的新定义中增加了`handle\_irq`方法。操作系统的virtio\_blk设备驱动程序中的`VirtIOBlock`实现了这个方法，并且实现了既支持轮询方式，也支持中断方式的块读写操作。

接下来，需要对操作系统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()`执行的就是`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。

# 文件系统

## 文件系统架构

## 磁盘布局与磁盘上数据结构

## 文件操作

# 系统测试

## 单元测试

## 集成测试

## 项目测试

# 项目总结与展望

## 支持更多Feature

## 实现异步运行时