



SUSTECH OPERATING SYSTEM(H)

OS Final Project: Asterinas Virtio-GPU Driver

Taojie Wang, Haibin Lai, Mengxuan Wu

SUSTech CSE, Shenzhen, Guangdong, China

Abstract

在本次好玩的 Project 中，我们小组实现了 Virtio-gpu 的驱动编写。我们针对 Virtio 结构进行了基本的学习，完成了 virtio-gpu 在 Qemu 内的基本调研。随后针对模块进行了驱动实现。我们创建了 GPU 所需的 VirtQueue，并实现了 2D 模式下所有的操作，完成了页面 framebuffer 及光标 cursor 的初始化与更新。随后，我们进行了简单的测试与实验，并简单实现了一个 syscall，成功在 userspace 完成对 GPU 的调用。另外，我们对 3D 模式进行了深度调研，发现 3D 模式要求高，支持不完善。本次项目的代码开源在 <https://github.com/HaibinLai/asterinas>。

Keywords: Rust, Operating System, GPU Driver, VirtIO

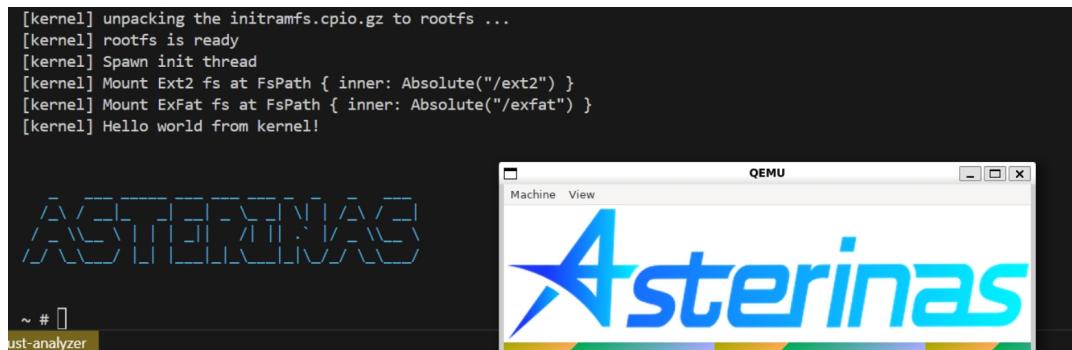


图 1. Virtio-gpu 实现效果；项目地址：Asterinas - virtio-gpu

1. 评分项总结说明

- 驱动设备的基础认知：实验课已检查设备识别和特征协商。
- 设备驱动基础功能实现：我们实现了设备初始化等基础功能。
- 设备驱动实现的特性数量和难度：我们实现了除 3D 渲染之外的所有设备功能，包括二维渲染和光标渲染。
- 用户态接口：我们实现了相关系统调用，支持用户改变显示屏颜色。
- 代码质量与注释：关键代码均添加注释。

报告内容指南：

- 设备配置空间、初始化以及操作流程：见报告 Introduction、Background 和 2D operation 实现情况 sections
- 代码仓库链接：见摘要。
- 项目基础功能实现：代码实现部分见“2D 操作实现”部分，运行结果见“实验与测试”部分。
- 设计接口：见“2D 操作实现情况接口”部分。
- 协议实现丰富度：我们实现了除 3D 渲染之外的所有设备功能，包括二维渲染和光标渲染。具体讲解见报告“实现情况”部分。此外，我们对 3D 操作进行了充分调研，见“相关工作”部分。

2. Introduction

Virtio-gpu 是一个基于 Virtio 的图形适配器。它可以在 2D 模式和 3D 模式下运行。它的职责如图2所示，在 Guest 上的 Application 程序希望渲染其 GUI 等图形化窗口。其会调用图形库帮助其渲染，例如 mesa (实现 OpenGL API 规范的开源图形库)。图形库接着将与 Guest OS 互动，调用 Kernel 内的 Virtio-gpu Driver，传输必要的图形渲染数据，随后 Virtio-gpu Driver 将与 Qemu Virtio-gpu Device 互动，该 device 将和 Host OS 内的图形库及 GPU 驱动进行交互，完成渲染后将数据返回至上层中。

在这一过程中，Virtio-gpu Driver 负责 Guest OS 与 Qemu Virtio-gpu 间的通信交互工作。在实践中，OS 向 GPU 通信的实现主要由两个接受相同格式的 VirtQueue 完成：控制队列与光标队列。控制队列负责接收 OS 向 GPU 发送的各种命令，如更新屏幕 Framebuffer，光标队列负责接收光标 (cursor) 的命令，该队列作为光标的快速更新队列 (fast track)，不会被控制队列内耗时的命令所延迟。

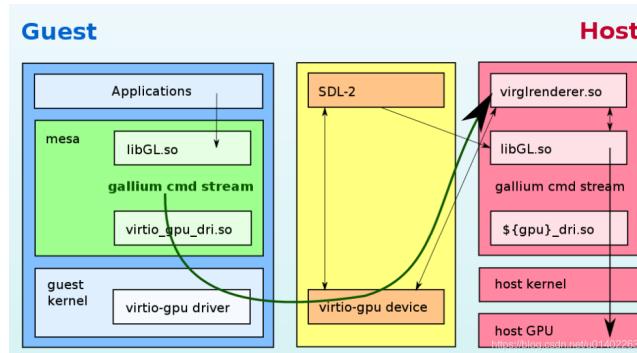


图 2. Guest Apps 至 Host GPU 图形渲染全过程

要在队列中发送命令实现 GPU 渲染，流程如下：

- Guest OS：在 host 上分配页面并调用 GPU driver。
- Guest Gpu Driver：向 queue 发送一个 header 和在环形缓冲区中的物理页面 (guest POV) 的 pointers。

- Guest Gpu Driver: 向 Host OS 发送软中断
- VM Exit
- Host QEMU: QEMU 读取 header 和 pointers。转换地址以匹配本地虚拟地址范围。
- Host QEMU: 读取 header 命令, 执行。若该命令为 3D 命令, 其会向 Host 图形库进行调用并渲染。此时 2D 指令到此结束, Qemu 不会通知 guest OS。若该命令为 3D 命令, Qemu 会向 Host GPU 发送命令, 并继续接下来的返回操作。
- Host QEMU: 收到 3D 渲染返回值后, Qemu 将值写回环形缓冲区
- Host QEMU: 向 Guest OS 发送软中断
- Guest OS: 处理 3D 命令中断, 读取环形缓冲区并处理应答

3. Background 项目背景

这里我们简单回顾在实现 virtio-gpu 驱动中我们所做的相关调研。关于 virtio queue, 半虚拟化的学习调研我们不再赘述, 我们在博客 OS Project part I Virtio, a biref summary 进行了学习记录。

3.1 Virtio-gpu 组件调研

在白皮书中, Virtio-gpu 应包含以下组件:

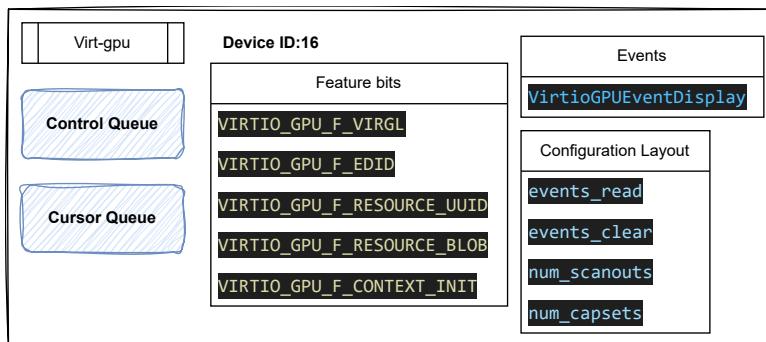


图 3. Virtio GPUConfig

我们在 introduction 部分讲解了两个队列的情况。DeviceID 为 Virtio-gpu 对该设备识别号。各 **feature bit** 含义如下:

1. VIRTIO_GPU_F_VIRGL: 是否支持 virgl 3D 模式。(该模式允许利用 Qemu 内 virgl 程序, 其将利用 host GPU 进行加速)
2. VIRTIO_GPU_F_EDID: 是否支持 EDID (外部显示设备标识数据, 显示屏的身份证, 主要用于真实显示器)
3. VIRTIO_GPU_F_RESOURCE_UUID: 用于标识和管理 GPU 资源。这样, 如果多个虚拟机使用同一物理 GPU 或者通过虚拟设备共享资源, UUID 就能确保这些虚拟机能够正确地识别和访问所需的 GPU 资源。

4. VIRTIO_GPU_F_RESOURCE_BLOB: 支持大数据块传输。该模块主要用于传输大型 3D 二进制数据块，帮助 GPU 对大批量数据进行加速渲染。

Device configuration fields (设备配置字段) 是指用于配置和管理设备行为的参数或选项。这些字段通常包含在设备的配置结构中，并由 OS 用于初始化、控制、管理设备的运行方式。

1. events_read: 向驱动程序发出待处理的事件的信号
2. events_clear: 清除驱动设备中所有待处理命令。
3. num_scanouts: 显示屏的数量。(Virtio 白皮书中允许 1-16 个屏幕。但是 Qemu 对此没有完全支持)
4. num_capsets 指定设备支持的能力集的最大数量，与 3D 模式有关。例如假设一个虚拟设备支持三种能力集：支持 2D 图形加速，支持 3D 图形加速，支持硬件视频解码，如果 num_capsets 的值为 3，表示该虚拟 GPU 设备支持这三种能力集。

3.2 Qemu virtio-gpu 实现情况调研

在 Qemu 的介绍 Virtio-gpu 中，其提到，Linux 只需要有一个 Option 选项支持，即可使用 Virtio-gpu (*Virtio-gpu requires a guest Linux kernel built with the CONFIG_DRM_VIRTIO_GPU option*)。

在我们经多维度的查找 (Qemu 文档, Qemu 源码 [github](#))，确认了在 Qemu 中对 Virtio-gpu 的实现情况支持。我们将结果放在表 1 中

唯一有缺陷的可能在 Multi-head 支持，即一个 Linux 可以打开多个图形化窗口。然而 Linux 自身构建多机位 (物理) 计算机的圈子就很小，软件支持很少，而且有缺陷。对此关于 Linux 在 Qemu Multihead 的讨论就更稀少了。无论 Linux 还是 Qemu，相关的文档都非常稀少。想在 Qemu 上开启多个屏幕在 Linux 内都非常困难。有用户曾经反馈过这个 Multihead 问题，在 Linux 上解决方案是使用 LightDM 程序库。然而经过我们的调研，这部分工程调研与工作量都非常大。具体的解决方案可以参考 LightDM 开发者的对其自身库的调整及开发过程 [sddm issue Re: \[Qemu-devel\] multihead & multiseat in qemu](#) 可以看到，这部分工程需要 Linux DRM 的支持。

其他的 Driver 需求 Qemu 都相对实现的不错，详情可在 Qemu QEMU [github](#) 源码上学习查看。

Control Queue 内有多种信息，我们将其列在下表 (3D Operation 未完全列出)，这些 Operation 在信息 Header 中记录。

表 1. Virtio-gpu 实现情况即我们的实现情况

Operation id	Device Operation on virtio white paper	Qemu impl & document	Our Implementation
5.7.6.1	Create a framebuffer and configure scanout	Y	Y
5.7.6.2	Update a framebuffer and scanout	Y	Y
5.7.6.3	Using pageflip	Y	Y
5.7.6.4	Multihead setup	Not Good	Wait for Qemu Update
5.7.6.5	Command lifecycle and fencing	Y	Y
5.7.6.6	Configure mouse cursor	Y	Y
5.7.6.7	Request header	Y	Y
5.7.6.8	Controlq	Y	Y

在 5.7.6.8 的 Controlq 中，白皮书内规定了多种命令，主要分为 2D,3D 及 Cursor 命令。我们在本次 project 中实现了 Asterinas 支持的所有 2D 及 Cursor 命令。

表 2. Virtio Control Queue 相关支持及实现情况

Control Queue Operation	2D / 3D	QEMU Support	Asterinas Support	Our Impl
VIRTIO_GPU_CMD_GET_DISPLAY_INFO	Both	Y	Y	Y
VIRTIO_GPU_CMD_GET_EDID	Both	Y	Y	Y
VIRTIO_GPU_CMD_RESOURCE_CREATE_2D	2D	Y	Y	Y
VIRTIO_GPU_CMD_RESOURCE_UNREF	2D	Y	Y	Y
VIRTIO_GPU_CMD_SET_SCANOUT	2D	Y	Y	Y
VIRTIO_GPU_CMD_RESOURCE_FLUSH	2D	Y	Y	Y
VIRTIO_GPU_CMD_TRANSFER_TO_HOST_2D	2D	Y	Y	Y
VIRTIO_GPU_CMD_RESOURCE_ATTACH_BACKING	2D	Y	Y	Y
VIRTIO_GPU_CMD_RESOURCE_DETACH_BACKING	2D	Y	Y	Y
VIRTIO_GPU_CMD_GET_CAPSET_INFO	3D	Y	Y	NO for 3D
VIRTIO_GPU_CMD_GET_CAPSET	3D	Y	Y	NO for 3D
VIRTIO_GPU_CMD_RESOURCE_ASSIGN_UUID	Both	Y	NO	NO for OS
VIRTIO_GPU_CMD_RESOURCE_CREATE_BLOB	3D	Not Good	Y	NO for 3D
VIRTIO_GPU_CMD_SET_SCANOUT_BLOB	3D	Not Good	Y	NO for 3D
VIRTIO_GPU_CMD_CTX_CREATE	3D	Y	Y	NO for 3D
VIRTIO_GPU_CMD_UPDATE_CURSOR	Cursor	Y	Y	Y
VIRTIO_GPU_CMD_MOVE_CURSOR	Cursor	Y	Y	Y

对于在白皮书中规定的 UUID，该部分在 Linux 对设备识别上有实现Linux UUID CSDN，但由于 Asterinas 对设备 UUID 没有很好的支持，我们因而没有实现该部分内容 UUID Search Result。

针对三维图形渲染，我们经过调研发现，容器中 Linux 和 qemu 版本均不满足要求。具体来说，若需要在 qemu 中添加 OpenGL 后端，需要 Linux 版本高于 16.1，qemu 版本高于 10.2。考虑到 qemu 对三维渲染的支持不成熟、本地环境受限等因素，我们将三维图形渲染留作未来工作完成。

4. 相关工作

本部分我们介绍跟本 Project 的相关工作。他们对最终的驱动代码书写没有直接关系，但是理解这些工作对我们的工程实现起到非常重要的指导作用。

4.1 GPU 渲染过程基本介绍

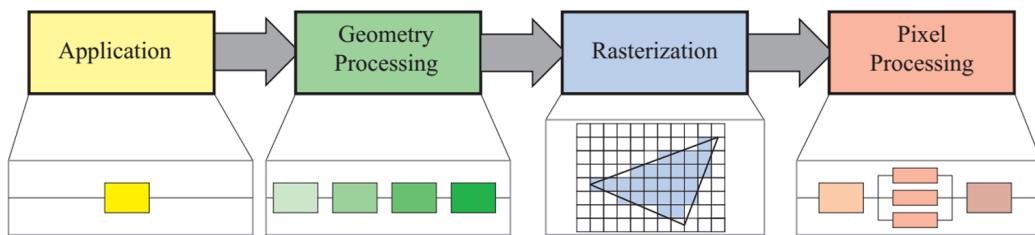


图 4. Machine Learning Training FLOPS over time

计算机依靠渲染管线来完成图形渲染。渲染管线作为图形学中伟大的发明，它将渲染过程中从图形程序到最终在 2D 屏幕生成图案的过程分为多个步骤，每个步骤单独处理一个渲染步骤（类似深度学习中的算子）。程序经过一系列渲染，最终将图案显示在屏幕上。

图6展示的光栅化图形渲染管线是最常见的实时渲染管线。其中，** 应用程序阶段 ** 在 CPU 端执行，其行为完全由开发者决定，主要工作是准备场景数据：光源、几何模型、摄像机……然后调用图形 API，将这些数据发送到显存，同时设置 GPU 状态，命令 GPU 开始渲染。从几何阶段开始，在 GPU 端执行。参考《Real-Time Rendering Fourth Edition》，将光栅化图形渲染管线的拆分为上述四个阶段：

1. 应用阶段 (Application): 本阶段由用户程序在 CPU 端完成，进行数据预处理操作，包括设置渲染状态等操作。程序将要画的画面、摄像机输出为渲染所需要的几何信息——渲染图元（点、线、三角面等），并进行粗粒度剔除等工作。随后，CPU 将向 GPU 发送 draw call 指令，该命令会指向需要被渲染的图元列表，GPU 根据渲染状态（材质、纹理、着色器）和所有输入的顶点数据来进行计算，完成渲染管线的剩余阶段。

2. 几何处理阶段 (Geometry): 本阶段在 GPU 收到 drawcall 指令后开始，GPU 将负责处理图元的变换、投影和所有其他类型几何处理任务。GPU 将计算这些图元应该在 2D 平面的具体位置。（如在 3D 场景下，该阶段需要运用基本的线性代数，将场景随摄像机视角投影到 2D 上）

3. 光栅化阶段 (Rasterization): GPU 在得到图元的具体位置后，需要将其与屏幕上离散的像素点相对应。

4. 像素处理阶段 (Pixel Processing): 在找到对应的像素点后，GPU 需要给每个像素点“上色”，对片元进行纹理采样与着色。

详细过程可参考 Unity3D 渲染管线全流程解析 unity 渲染管线

4.2 Linux 图形驱动

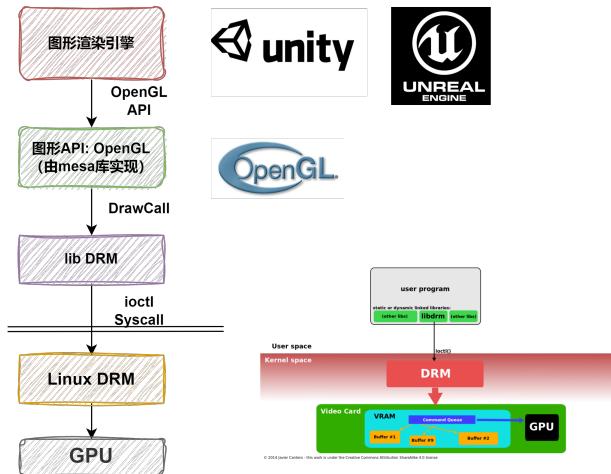


图 5. Linux 图形驱动调用栈

在 Linux 操作系统中，实际执行这一管线的系统调用栈如下图所示。图形渲染引擎会调用 OpenGL 等标准图形 API 来绘制画面。随后 OpenGL 实现库（如 mesa）将帮助其完成 Application 阶段工作，并调用 Drawcall 让 GPU 完成渲染管线的后续工作。

在 Drawcall 被调用后，在 Linux 内会由 DRM 驱动库完成调用，DRM 比 Framebuffer 更为先进，对新的显示特性和 GPU 有更好的支持和加速。随后 Kernel 即可将数据发送给 GPU 进行处理。

在本次 Project 中，我们实现的 Framebuffer 类的驱动，同时在数据传输上运用 DRM 的方法使用 DMA 加速。主要原因是，Framebuffer 实现较为清晰，在 Qemu 上即可轻松实现与使用。而 Linux DRM 的实现目标是从服务器至嵌入式开发板的多样环境下运行，而其对新显示特性及 GPU 的支持其实在 Qemu 模拟器上并没有良好的实现，因此使用 DRM 框架的意义不是很大。且 Linux DRM 的文档 Linux DRM (2012 年最后一次更新) 与 Linux DRM Virtio-gpu GPU Linux virtio-gpu ioctl 的工程过于复杂。由此，我们在较为简易的 Framebuffer 下，采用 DRM 内 DMA 传输的特性对 virtio-gpu driver 进行开发。

4.3 Qemu Backend, Mesa Backend 对 Virtio GPU 的支持情况

在 Qemu 官方对 Virtio GPU 说明中，其提到 Qemu 将 3D 渲染任务下放到 Host 图形库及 GPU 中：

QEMU 提供了一个 2D virtio-gpu 后端，以及两个加速 3D 的后端：virglrenderer ('gl' 设备标签) 和 rutabaga_gfx ('rutabaga' 设备标签)。还有一个 vhost-user 后端，它将图形堆栈运

行在一个单独的进程中，以提高隔离性。

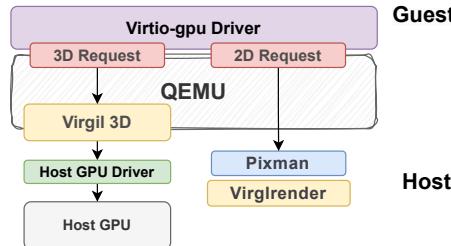


图 6. Virtio-gpu Backend 实现情况

Mesa Mesa lib，也称为 Mesa 3D 图形库，是 OpenGL, Vulkan 和其他图形 API 规范的开源软件实现。Mesa 根据规范转换特定供应商的图形硬件驱动程序。

而在 mesa 等 3D 图形库中 Mesa Virtio-gpu，其也向 Qemu virtio-gpu 驱动提供了支持。在 Qemu Guest OS 上运行的 mesa 库可以通过访问操作系统的 virtio-gpu 驱动来实现。

5. Virtio-gpu 特征协商与初始化

设备初始化的流程定义于 Virtio 白皮书的 3.1 章节部分。其中除第 4 步设备特征协商及第 7 步进行设备具体的初始化流程外的其他步骤，已在 `virtio_component_init` 方法中实现完成。

设备特征协商较为简单，代表设备特征的 64 个 bit 由 transport 层提供，driver 只需从中提取各个特征对应的 bit 即可。具体代码如下：

```

bitflags::bitflags! {
    pub struct GPUFeatures: u64{
        /// virgl 3D mode is supported.
        const VIRTIO_GPU_F_VIRGL = 1 << 0;
        /// EDID is supported.
        const VIRTIO_GPU_F_EDID = 1 << 1;
        /// assigning resources UUIDs for export to other virtio devices is
        /// supported.
        const VIRTIO_GPU_F_RESOURCE_UUID = 1 << 2;
        /// creating and using size-based blob resources is supported.
        const VIRTIO_GPU_F_RESOURCE_BLOB = 1 << 3;
        /// multiple context types and synchronization timelines supported.
        /// Requires VIRTIO_GPU_F_VIRGL.
        const VIRTIO_GPU_F_CONTEXT_INIT = 1 << 4;
    }
}

```

```
impl GPUDevice {
    pub fn negotiate_features(features: u64) -> u64 {
        let features = GPUFeatures::from_bits_truncate(features);
        early_println!("virtio_gpu_features = {:?}", features);
        features.bits()
    }
}
```

在 GPU 具体的初始化阶段，具体步骤如下：

1. 通过 transport 层的信息初始化 Config
2. 初始化控制队列和光标队列对应的两个 VirtQueue
3. 初始化上述两个 VirtQueue 对应的 Request 和 Response DMA Buffer
4. 使用上述 Config、VirtQueue、DMA Buffer 初始化设备驱动对象
5. 为设备注册控制队列和光标队列对应的中断和 Config Change 事件的中断

```
impl GPUDevice {
    pub fn init(mut transport: Box<dyn VirtioTransport>) -> Result<(), VirtioDeviceError> {
        let config_manager =
            VirtioGPUConfig::new_manager(transport.as_ref());
        early_println!("virtio_gpu_config = {:?}", config_manager.read_config());

        // Initialize virtqueues
        const CONTROL_QUEUE_INDEX: u16 = 0;
        const CURSOR_QUEUE_INDEX: u16 = 1;
        // TODO(Taojie): the size of queues?
        let control_queue = SpinLock::new(
            VirtQueue::new(CONTROL_QUEUE_INDEX, Self::QUEUE_SIZE,
                           transport.as_mut())
                .expect("create control queue failed"),
        );
        let cursor_queue = SpinLock::new(
            VirtQueue::new(CURSOR_QUEUE_INDEX, Self::QUEUE_SIZE,
                           transport.as_mut())
                .expect("create cursor queue failed"),
        );
    }
}
```

```

// Initialize DMA buffers
let control_request = {
    let vm_segment =
        → FrameAllocOptions::new().alloc_segment(1).unwrap();
    DmaStream::map(vm_segment.into(), DmaDirection::Bidirectional,
        → false).unwrap()
};

let control_response = {
    let vm_segment =
        → FrameAllocOptions::new().alloc_segment(1).unwrap();
    DmaStream::map(vm_segment.into(), DmaDirection::Bidirectional,
        → false).unwrap()
};

let cursor_request = {
    let vm_segment =
        → FrameAllocOptions::new().alloc_segment(1).unwrap();
    DmaStream::map(vm_segment.into(), DmaDirection::Bidirectional,
        → false).unwrap()
};

let cursor_response = {
    let vm_segment =
        → FrameAllocOptions::new().alloc_segment(1).unwrap();
    DmaStream::map(vm_segment.into(), DmaDirection::Bidirectional,
        → false).unwrap()
};

// Create device
let device = Arc::new(Self {
    config_manager,
    control_queue,
    cursor_queue,
    control_request,
    control_response,
    cursor_request,
    cursor_response,
    header: VirtioGpuCtrlHdr::default(),
    transport: SpinLock::new(transport),
}

```

```
});

// Interrupt handler
let clone_device = device.clone();
let handle_irq_ctl = move |_: &TrapFrame| {
    clone_device.handle_irq();
};

let clone_device = device.clone();
let handle_irq_cursor = move |_: &TrapFrame| {
    clone_device.handle_irq();
};

let clone_device = device.clone();
let handle_config_change = move |_: &TrapFrame| {
    clone_device.handle_config_change();
};

// Register irq callbacks
let mut transport = device.transport.lock();
transport
    .register_queue_callback(CONTROL_QUEUE_INDEX,
        ↵ Box::new(handle_irq_ctl), false)
    .unwrap();
transport
    .register_queue_callback(CURSOR_QUEUE_INDEX,
        ↵ Box::new(handle_irq_cursor), false)
    .unwrap();
transport
    .register_cfg_callback(Box::new(handle_config_change))
    .unwrap();

transport.finish_init();
Ok(())
}

}
```

6. Virtio-gpu 2D Operation 实现情况

在 Virtio-gpu 2D 模式下，需要对两个队列进行操作：控制队列和光标队列。这两个队列均为 virtual queues，分别负责 frame buffer 的渲染和光标的渲染。根据 virtio 白皮书，光标队列的目的是为光标在显示屏幕的更新提供“快速通道”，从而能够快速更新光标信息，而避免被其他耗时渲染操作阻塞。我们实现了白皮书中规定的以下 2D 操作：

- VIRTIO_GPU_CMD_GET_DISPLAY_INFO
- VIRTIO_GPU_CMD_RESOURCE_CREATE_2D
- VIRTIO_GPU_CMD_RESOURCE_UNREF
- VIRTIO_GPU_CMD_SET_SCANOUT
- VIRTIO_GPU_CMD_RESOURCE_FLUSH
- VIRTIO_GPU_CMD_TRANSFER_TO_HOST_2D
- VIRTIO_GPU_CMD_RESOURCE_ATTACH_BACKING
- VIRTIO_GPU_CMD_RESOURCE_DETACH_BACKING
- VIRTIO_GPU_CMD_GET_EDID
- VIRTIO_GPU_CMD_UPDATE_CURSOR
- VIRTIO_GPU_CMD_MOVE_CURSOR

下面的内容会将命令分为控制队列和光标队列两个部分，分别介绍具体的实现以及项目所使用的数据结构。

6.1 控制队列实现情况

在 virtio 架构中，设备驱动和设备是通过 virtqueue 中进行交互的。驱动在队列中添加操作相关的数据结构并且通知设备。设备在得到通知并且处理完成相应的请求之后，会将响应按照同样的方式放入队列中，并且通知设备驱动。

根据白皮书，在 virtio-gpu 的交互过程中，控制队列和光标队列具有相同的格式。每个请求和相应都包括固定的头和请求特定的数据两个部分。相应的数据结构实现如下：

Listing 1. struct VirtioGpuCtrlHdr

```

/// All requests and responses on the virt queues have a fixed header
/// using the following layout structure.
/// Reference: spec 5.7.6.7 Device Operation: Request header
#[repr(C, packed)]
#[derive(Debug, Clone, Copy, Pod)]
pub struct VirtioGpuCtrlHdr {
    /// specifies the type of the driver request (VIRTIO_GPU_CMD_*) or
    → device response (VIRTIO_GPU_RESP_*).
}

```

```

    /// On success the device will return VIRTIO_GPU_RESP_OK_NODATA in case
    → there is no payload. Otherwise the type field will indicate the kind
    → of payload.
    /// On error the device will return one of the VIRTIO_GPU_RESP_ERR_*
    → error codes.

pub type_: u32,
    /// request / response flags.

pub flags: u32,
    /// If the driver sets the VIRTIO_GPU_FLAG_FENCE bit in the request
    → flags field the device MUST:
        /// - set VIRTIO_GPU_FLAG_FENCE bit in the response,
        /// - copy the content of the fence_id field from the request to the
        → response, and
        /// - send the response only after command processing is complete.

pub fence_id: u64,
    /// Rendering context (used in 3D mode only).

pub ctx_id: u32,
    /// ring_idx indicates the value of a context-specific ring index.
    /// For more details, refer to spec.

pub ring_idx: u8,
pub padding: [u8; 3],
}

```

为了能够使得图形被渲染到显示屏上，设备驱动需要将待渲染的内容写入资源。在 virtio 的语境下，资源是独属于宿主的重要概念。在不支持共享内存的情况下，虚拟机需要通过 DMA 将数据传输到资源中。在 2D 模式下，不存在从资源向虚拟机传输的情况，虚拟机只需要向资源中传输资源即可完成渲染。

在二维模式下，资源由长、宽以及标识符构成。虚拟机必须将支撑存储依附在资源上以实现 DMA 传输。

更具体地，设备渲染的过程分为创建帧缓冲区和配置显示头两个部分。设备驱动首先需要通过 VIRTIO_GPU_CMD_RESOURCE_CREATE_2D 指令在宿主机上创建资源；然后，虚拟机从它的内存中分配帧缓冲区，并且通过 VIRTIO_GPU_CMD_RESOURCE_ATTACH_BACKING 将其依附在资源上。至此，缓冲区已经配置好，虚拟机可以将数据写入缓冲区并且通过 DMA 同步到宿主机的资源上。最后，使用命令 VIRTIO_GPU_CMD_SET_SCANOUT 来将帧缓冲区链接到显示屏展示头。

再创建帧缓冲区和配置显示头完成之后，虚拟机可以通过更新帧缓冲区和显示头来改变显示屏所展示的内容。具体来说，虚拟机首先更新帧缓冲区。然后，通过 VIRTIO_GPU_CMD_-

TRANSFER_TO_HOST_2D 命令来根据虚拟机的更新宿主机上的资源。最后，使用命令 VIRTIO_GPU_CMD_RESOURCE_FLUSH 刷新资源到显示屏上。

下面讲解上文提及命令的有关数据结构。

6.1.1 VIRTIO_GPU_CMD_RESOURCE_CREATE_2D

该命令会在宿主机上创建资源，该资源具有命令中所指定的长、宽和格式。资源标识符由虚拟机产生。

Listing 2. struct VirtioGpuResourceCreate2D

```
// VIRTIO_GPU_CMD_RESOURCE_CREATE_2D
#[repr(C, packed)]
#[derive(Debug, Clone, Copy, Pod, Default)]
pub struct VirtioGpuResourceCreate2D {
    hdr: VirtioGpuCtrlHdr,
    resource_id: u32,
    format: u32,
    width: u32,
    height: u32,
}
```

定义完成该数据结构之后，可以通过 virtqueue 完成设备驱动和设备之间的交互。该函数定义的接口如下：

```
/// From the spec:
///
/// Create a host resource using VIRTIO_GPU_CMD_RESOURCE_CREATE_2D.
/// Allocate a framebuffer from guest ram, and attach it as backing
→ storage to the resource just created, using
→ VIRTIO_GPU_CMD_RESOURCE_ATTACH_BACKING.
/// Use VIRTIO_GPU_CMD_SET_SCANOUT to link the framebuffer to a display
→ scanout.
///
/// Response type is VIRTIO_GPU_RESP_OK_NODATA.
/// This creates a 2D resource on the host with the specified width,
→ height and format. The resource ids are generated by the guest.
```

```

fn resource_create_2d(
    &self,
    resource_id: u32,
    width: u32,
    height: u32,
) -> Result<(), VirtioDeviceError>

```

按照白皮书的定义，准备请求和相应的 DMA 缓冲区以及数据结构。

```

// Prepare request data DMA buffer
let req_data_slice = {
    let req_data_slice = DmaStreamSlice::new(
        &self.control_request,
        0,
        size_of::<VirtioGpuResourceCreate2D>(),
    );
    early_println!(
        "parameters: resource_id: {}, width: {}, height: {}",
        resource_id,
        width,
        height
    );
    let req_data = VirtioGpuResourceCreate2D::new(
        resource_id,
        VirtioGpuFormat::VIRTIO_GPU_FORMAT_B8G8R8A8_UNORM,
        width,
        height,
    );
    req_data_slice.write_val(0, &req_data).unwrap();
    req_data_slice.sync().unwrap();
    req_data_slice
};

let inputs = vec![&req_data_slice];

// Prepare response DMA buffer
let resp_slice = {
    let resp_slice = DmaStreamSlice::new(

```

```

    &self.control_response,
    0,
    size_of::<VirtioGpuRespResourceCreate2D>(),
);
resp_slice
    .write_val(0, &VirtioGpuRespResourceCreate2D::default())
    .unwrap();
resp_slice.sync().unwrap();
resp_slice
};

```

将缓冲区添加到队列中。

```

// Add buffer to queue
let mut control_queue = self.control_queue.disable_irq().lock();
control_queue
    .add_dma_buf(inputs.as_slice(), &[&resp_slice])
    .expect("Add buffers to queue failed");

```

通知设备请求已经添加。

```

// Notify
if control_queue.should_notify() {
    control_queue.notify();
}

```

等待并且获取相应。

```

// Wait for response
while !control_queue.can_pop() {
    spin_loop();
}
control_queue.pop_used().expect("Pop used failed");
resp_slice.sync().unwrap();

```

```
let resp: VirtioGpuRespResourceCreate2D =
    → resp_slice.read_val(0).unwrap();
```

最后，检查反应结果。

```
// check response with type OK_NODATA
if resp.header_type() !=
    → VirtioGpuCtrlType::VIRTIO_GPU_RESP_OK_NODATA as u32 {
    return Err(VirtioDeviceError::QueueUnknownError);
}
```

上面的代码展示了一次设备驱动通过虚拟队列与 gpu 设备进行交互的完整过程。总结如下：

1. 准备 DMA 内存缓冲区，并且按照白皮书规定填充按格式填充指定数据结构。
2. 将 DMA 缓冲区放入队列并且通知队列。
3. 轮询设备获得响应。
4. 检查响应结果是否正确。若响应中包括数据，则将数据返回。

后文中提到的所有命令均遵循该程式。因此，后文不再赘述该部分内容，只介绍与命令相关的数据结构和操作部分。

6.1.2 VIRTIO_GPU_CMD_RESOURCE_ATTACH_BACKING

该命令将页帧分配给资源。完成后，会存在一组页帧作为资源的后备存储。然后，从那时起，这些页面将用于该资源的传输操作。

```
// VIRTIO_GPU_CMD_RESOURCE_ATTACH_BACKING
// Assign backing pages to a resource.
// Request data is struct virtio_gpu_resource_attach_backing, followed by
//     → struct virtio_gpu_mem_entry entries.
// Response type is VIRTIO_GPU_RESP_OK_NODATA.
#[repr(C, packed)]
#[derive(Debug, Clone, Copy, Pod)]
pub(crate) struct VirtioGpuResourceAttachBacking {
    hdr: VirtioGpuCtrlHdr,
    resource_id: u32,
    nr_entries: u32,
}
```

```
##[repr(C, packed)]
#[derive(Debug, Clone, Copy, Pod)]
pub(crate) struct VirtioGpuMemEntry {
    addr: u64,
    length: u32,
    padding: u32,
}
```

6.1.3 VIRTIO_GPU_CMD_TRANSFER_TO_HOST_2D

从虚拟机内存传输到宿主机资源。

```
// VIRTIO_GPU_CMD_TRANSFER_TO_HOST_2D
// Transfer from guest memory to host resource.
// Request data is struct virtio_gpu_transfer_to_host_2d. Response type is
// → VIRTIO_GPU_RESP_OK_NODATA.
#[repr(C, packed)]
#[derive(Debug, Clone, Copy, Pod)]
pub(crate) struct VirtioGpuTransferToHost2D {
    hdr: VirtioGpuCtrlHdr,
    r: VirtioGpuRect,
    offset: u64,
    resource_id: u32,
    padding: u32,
}
```

6.1.4 VIRTIO_GPU_CMD_RESOURCE_FLUSH

刷新扫描头的资源。

s

```
// VIRTIO_GPU_CMD_RESOURCE_FLUSH
// Flush a scanout resource.
// Request data is struct virtio_gpu_resource_flush. Response type is
// → VIRTIO_GPU_RESP_OK_NODATA.
```

```

#[repr(C, packed)]
#[derive(Debug, Clone, Copy, Pod)]
pub(crate) struct VirtioGpuResourceFlush {
    hdr: VirtioGpuCtrlHdr,
    r: VirtioGpuRect,
    resource_id: u32,
    padding: u32,
}

```

6.1.5 完整的二维图像渲染过程

在显示屏上渲染二维图形需要按照一定顺序执行上述指令。具体代码如下：

```

/// render 2d shape on screen
fn render_2d(device: Arc<GPUDevice>) -> Result<(), VirtioDeviceError> {
    // get resolution
    let (width, height) = device.resolution().expect("failed to get
    ↵ resolution");
    early_println!("[INFO] resolution: {}x{}", width, height);

    // setup framebuffer
    device.setup_framebuffer(0xbabe)?;
    let buf = device.frame_buffer.as_ref().expect("frame buffer not
    ↵ initialized");

    // write content into buffer
    for y in 0..height {
        for x in 0..width {
            let offset = (y * width + x) * 4;
            buf.write_val(offset as usize, &x).expect("error writing frame
            ↵ buffer");
            buf.write_val((offset + 1) as usize, &y).expect("error writing
            ↵ frame buffer");
            buf.write_val((offset + 2) as usize, &(x+y)).expect("error
            ↵ writing frame buffer");
        }
    }
}

```

```

// flush to screen
device.flush().expect("failed to flush");
early_println!("flushed to screen");

Ok(())
}

```

其中setup_framebuffer()函数设置帧缓冲区，具体代码如下：

```

pub fn setup_framebuffer(&self, resource_id: u32) -> Result<(), VirtioDeviceError> {
    // get display info
    let display_info = self.request_display_info()?;
    let rect = display_info.get_rect(0).unwrap();

    // create resource 2d
    self.resource_create_2d(resource_id, rect.width(), rect.height())?;

    // alloc continuous memory for framebuffer
    // Each pixel is 4 bytes (32 bits) in RGBA format.
    let size = rect.width() as usize * rect.height() as usize * 4;
    let frame_buffer_dma = self.frame_buffer.as_ref().expect("frame
        buffer not initialized");

    // attach backing storage
    self.resource_attach_backing(resource_id, frame_buffer_dma.paddr(),
        size as u32)?;

    // map frame buffer to screen
    self.set_scanout(rect, 0, resource_id)?;

    // return dma to be written
    Ok(())
}

```

从头开始渲染二维图像的完整过程可以总结为如下步骤：

1. 根据显示屏信息得到分辨率等信息。
2. 创建帧缓冲区，包括创建二维资源、依附后端存储、设置显示头。
3. 往缓冲区中写入希望渲染出来的内容。
4. 刷新显示头。

如果已经完成一次渲染，且后续需要更改渲染内容，则无需重新创建缓冲区。只需要在原有缓冲区上进行修改并且重新刷新即可。

6.2 光标队列实现情况

类似于控制队列，设备驱动对光标的控制是通过 VirtQueue 实现的。驱动在队列中添加操作请求相关的数据结构并且通知设备。但是由于 Qemu 的一些问题，目前设备会响应操作请求并向队列中放入回复，但回复数据结构的所有参数均为空白，具体可见此 issue。因此我们不得不放弃对光标队列回复内容的检查操作。

光标队列的请求有两种类型：VIRTIO_GPU_CMD_UPDATE_CURSOR 和 VIRTIO_GPU_CMD_MOVE_CURSOR。他们使用同一种请求数据结构如下：

```
pub(crate) struct VirtioGpuUpdateCursor {
    hdr: VirtioGpuCtrlHdr,
    pos: VirtioGpuCursorPos,
    resource_id: u32,
    hot_x: u32,
    hot_y: u32,
    padding: u32,
}
```

VIRTIO_GPU_CMD_UPDATE_CURSOR 将会设置 resource_id 对应的资源为光标图像，光标将被移动到 pos 位置。在鼠标按下时，以光标图像从左上角为零点，图像的 (hot_x, hot_y) 处将被认为是触发位置。

VIRTIO_GPU_CMD_MOVE_CURSOR 则只会将光标移动到 pos 位置，请求中的其余参数都将被忽略。

光标队列添加请求的代码与控制队列大部分相同，只是将使用的 VirtQueue 换为光标队列，故在此不再重复陈述。

6.3 接口

针对上述命令，我们为每个命令分别设置了一个函数调用接口。此外，我们设计了常见操作接口，如 setup_frame_buffer()，具体代码已经在上文中详细讲解。此外，我们为系统调用实现了 show_color() 接口，该接口会在 virtio-gpu 系统调用实现情况部分详细讲解。

7. Virtio-gpu 系统调用实现情况

至此我们已经完成了设备驱动的大部分内容。然而，截至目前用户还无法使用 GPU 提供的服务。为了完成这个任务，需要修改 Asterinas 内核，修改或添加系统调用，以便利用户对 GPU 的使用。

我们提出了系统调用的两种方案，分别是 ioctl 以及自创系统调用。ioctl 是 Linux 中用于操作输入输出设备的专用系统调用。尽管我们找到了 Linux 关于 virtio-gpu 的 ioctl 系统调用的源代码，但是经过调研发现，该代码缺少注释和文档，且缺少头文件。这导致用户态程序无法通过现有标准库使用 gpu 服务。此外，ioctl 涉及到大量内核的修改，包括添加 gpu 设备文件、修改文件描述符表、完成 DRM 框架等，实现比较复杂，因此我们最终并未采用该方案。

我们采用另外一种方案：自创系统调用。我们在 Asterinas 系统调用表中添加并且实现了与 gpu 有关的系统调用 show_color()，支持用户在显示屏上展示自己想要展示的颜色，从而支持用户对通过内核对 gpu 进行操作。

代码层面，首先需要声明全局变量用于在系统调用中访问 gpu 设备。

```
// global static variable
pub static GPU_DEVICE: Once<SpinLock<Arc<GPUDevice>>> = Once::new();
```

在初始化 GPU 设备的过程中，同时赋值该全局静态变量。

```
// Create device: Arc<Mutex<GPUDevice>>
let device = Arc::new(Self {
    config_manager,
    control_queue,
    cursor_queue,
    control_request,
    control_response,
    cursor_request,
    cursor_response,
    header: VirtioGpuCtrlHdr::default(),
    transport: SpinLock::new(transport),
    frame_buffer: Some(frame_buffer_dma),
});
```

```
GPU_DEVICE.call_once(|> SpinLock::new(device));
```

最后，在系统调用实现中，用全局变量 GPU_DEVICE 进行相关操作。

```
pub fn sys_show_color(color: i32, _ctx: &Context) -> Result<SyscallReturn> {
    let gpu_device = GPU_DEVICE.get().expect("GPU device not initialized");
    let reso = gpu_device
        .lock()
        .resolution()
        .expect("Failed to get resolution");
    println!("Resolution: {}x{}", reso.0, reso.1);

    // show another color on the display
    gpu_device
        .lock()
        .show_color(color)
        .expect("error showing red");

    Ok(SyscallReturn::NoReturn)
}
```

其中，show_color 是设备驱动暴露的接口。该函数会将用户指定的颜色填充进入缓冲区并且刷新至显示屏。

```
pub fn show_color(&self, color: i32) -> Result<(), VirtioDeviceError> {
    // get resolution
    let (width, height) = self.resolution().expect("failed to get
        resolution");

    // get frame buffer
    let buf = self
        .frame_buffer
        .as_ref()
        .expect("frame buffer not initialized");

    // write content into buffer
    for x in 0..height {
```

```

    for y in 0..width {
        let offset = (x * width + y) * 4;
        buf.write_val(offset as usize, &color).unwrap();
    }
}

// flush to screen
self.flush().expect("failed to flush");
early_println!("flushed to screen");
Ok(())
}

```

总而言之，我们部分牺牲了 Asterinas 与 Linux ABI 的兼容性，通过创建新的系统调用是实现了用户对于 gpu 设备的操作。另外，我们可以提供更细粒度的 gpu 控制，比如渲染用户提供的二维图案。我们将此留作未来工作。

8. 实验与测试

本节我们简单介绍对 virtio-gpu 测试。我们将实验记录在 Experiment 博客中。

要运行该驱动，我们需要一个支持图形界面的 Docker 容器和 Qemu 模拟器，从而使 Asterinas 上传输的 Framebuffer 经过 virtio-gpu 驱动能成功输出。对此，我们需要在原有镜像上配置好 X11 转发设置，其将帮助我们打通至 Docker 镜像的图形服务。随后重新编译 Qemu 使其支持图形界面，最后即可测试查看效果。

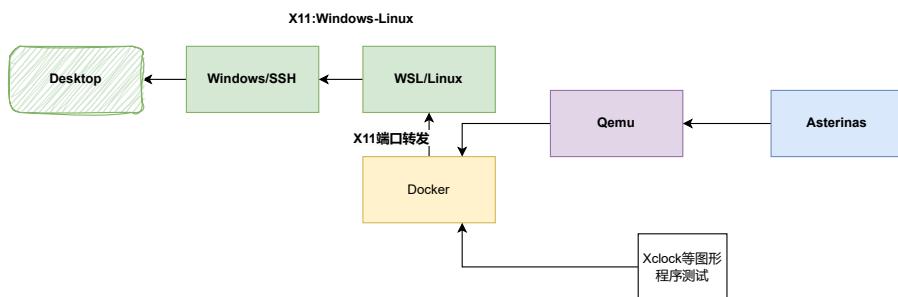


图 7. 实验环境配置

8.1 基本实验环境设置

在 list 9 中，我们将导入镜像，并在 list 4 下获取 X11 相关库。

随后我们启动 Docker X11 等图形化转发支持。若在 CentOS 中，我们在 /etc/ssh/sshd_config 中，开启以下设置：

1. AllowTcpForwarding yes
2. X11Forwarding yes

在 Ubuntu 及 WSL 中，需要开启如下设置：

1. ForwardAgent yes
2. ForwardX11 yes
3. ForwardX11Trusted yes

随后重启 sshd 服务：

```
/etc/init.d/sshd restart
```

这样在登录时加上 X11 Forwarding，使用 ssh -X user@centos 命令登录，我们即可查看图形化界面。

Listing 3. Run the image

```
# load asterinas 0.11.0
docker load -i .\asterinas.tar

# run the image with X11 forwarding
sudo docker run -it --privileged -v $(pwd)/asterinas:/root/asterinas -e DISPLAY=$DISPLAY -v
→ /tmp/.X11-unix:/tmp/.X11-unix asterinas:0.11.0

# If you try to run on linux server, use "--network=host"
# sudo docker run -it --privileged --network=host -v $(pwd)/asterinas:/root/asterinas -e
→ DISPLAY=$DISPLAY -v /tmp/.X11-unix:/tmp/.X11-unix asterinas:0.11.0
```

Listing 4. Get X11 library

```
sudo apt update
sudo apt-get install libghc-x11-dev # get x11 lib
```

Then we make a simple test on X11. As 5 displays, we provide a simple example from article/details/6601959 to test if X11 works on WSL:

将其用 X11 library 进行编译：

```
gcc hello_x11.c -o HelloX11 -lX11
```

如果其正常工作，其将如图8 打开一个屏幕并显示 “Hello World”

8.1.1 Extra: X11 Debug

如果程序发出如 Motty X11 的错误，这是 X11 转发导致的，不是 Qemu 导致的问题。解决办法是检查 Server 的 X11 转发或 Client X11 转发。

Listing 5. Get X11 library

```
#include <X11/Xlib.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void) {
    Display *d;
    Window w;
    XEvent e;
    char *msg = "Hello, World!";
    int s;

    d = XOpenDisplay(NULL);
    if (d == NULL) {
        fprintf(stderr, "Cannot open display\n");
        exit(1);
    }

    s = DefaultScreen(d);
    w = XCreateSimpleWindow(d, RootWindow(d, s), 100, 100, 500, 500, 1,
                           777215, 111111);
    printf("BlackPixel(d, s) is %d\n", (int)BlackPixel(d, s));
    printf("WhitePixel(d, s) is %d\n", (int)WhitePixel(d, s));
    XSelectInput(d, w, ExposureMask | KeyPressMask);
    XMapWindow(d, w);

    while (1) {
        XNextEvent(d, &e);
        if (e.type == Expose) {
            XFillRectangle(d, w, DefaultGC(d, s), 20, 20, 10, 10);
            XDrawString(d, w, DefaultGC(d, s), 10, 50, msg, strlen(msg));
        }
        if (e.type == KeyPress)
            break;
    }

    XCloseDisplay(d);
    return 0;
}
```

8.1.2 Qemu Recompilation

在 Host OS 支持图形界面后，我们需要我们的 Qemu 也对图形界面有支持。对此我们重新编译了 Asterinas 镜像内的 Qemu 6。

This configuration will enable GTK.

GTK is a [widget toolkit](http://en.wikipedia.org/wiki/Widget_toolkit). Each user interface

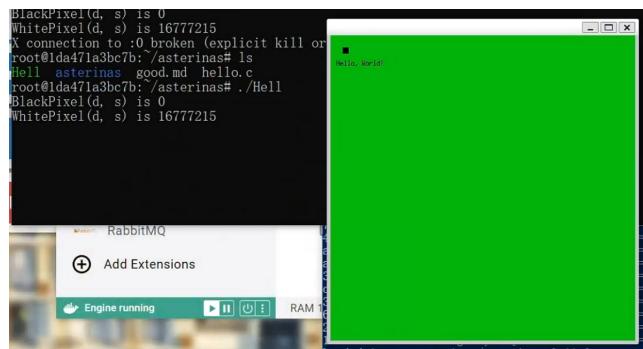


图 8. Host OS 图形界面支持测试

MoTTY X11 proxy: Unsupported authorisation protocol
Cannot open display

```
# Method:  
xhost +local:docker  
xhost +local:10
```

created by GTK consists of widgets. GTK (formerly known as GIMP Toolkit and GTK+) is a free and open-source, cross-platform widget toolkit for creating graphical user interfaces (GUIs). It is licensed under the terms of the GNU Lesser General Public License, allowing both free and proprietary software to use it. It is one of the most popular toolkits for the Wayland and X11 window systems.

User Networking (SLiRP) is the default networking backend for QEMU that provides a full TCP/IP stack within QEMU

8.2 2D FrameBuffer 及 cursor 测试

如图 10，我们在 make build 及 make run 后，即可看到 Asterinas 在 Qemu 窗口内显示出的图案。



图 9. Qemu 图形界面启动

Listing 6. Recompile QEMU

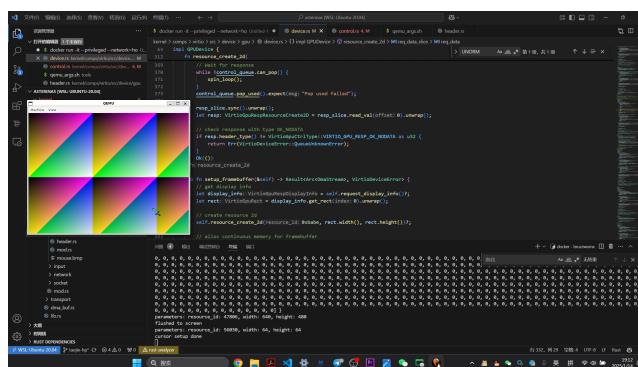
```
# get qemu source code
wget https://download.qemu.org/qemu-8.2.1.tar.xz
tar xf qemu-8.2.1.tar.xz
cd qemu-8.2.1

# ensure necessary package for qemu
sudo apt update
sudo apt install libgtk-3-dev
sudo apt install libglib2.0-dev
sudo apt install ninja-build

# configure qemu display
./configure \
--target-list=x86_64-softmmu \
--prefix=/usr/local/qemu \
--enable-slip \
--enable-gtk \
--enable-pie

# compile and build
make -j$(nproc)
make install
```

```
git clone -b
cd asterinas
make build
make run
```

**图 10.** Asterinas 图形界面启动

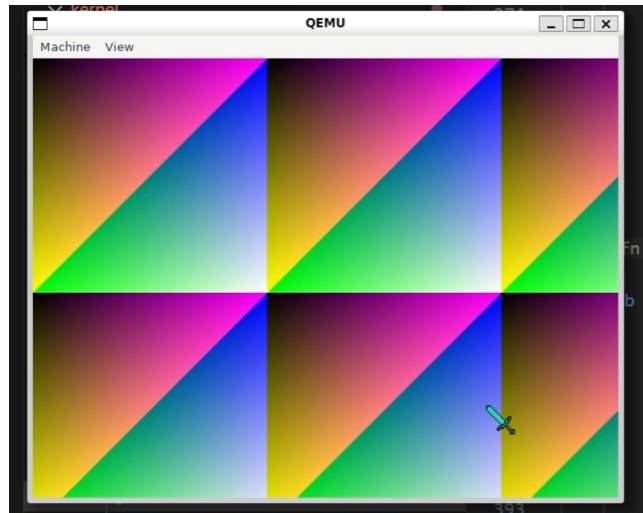


图 11. 我的圣剑!

可以看到我们的测试帧被刷新了出来。同时一个钻石剑的测试光标被成功导入，替代了 qemu 自身的光标。该帧的刷新逻辑可在 device.rs 的 test_frame_buffer 函数内找到：7。

Listing 7. Refresh argument example

```

for y in 0..height {
    for x in 0..width {
        let offset = (y * width + x) * 4;
        buf.write_val(offset as usize, &x).expect("error writing frame buffer");
        buf.write_val((offset + 1) as usize, &y).expect("error writing frame buffer");
        buf.write_val((offset + 2) as usize, &(x+y)).expect("error writing frame buffer");

    }
}

```

我们可以使用去 std 的 Rust tinybmp 库来打开我们的图片。将 Asterinas logo 转换为 bmp 图，tinybmp 打开后取出颜色矩阵并进行渲染，就获得了我们封面的 Asterinas 头图 8。

Listing 8. Refresh argument example

```

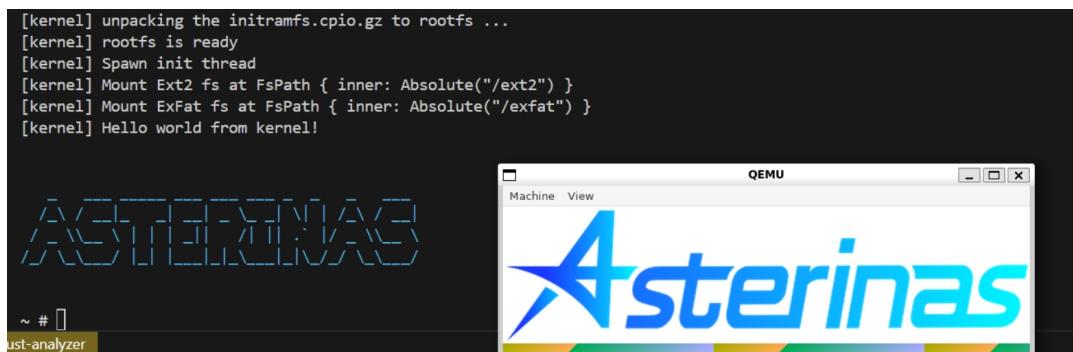
static BG_BMP_DATA: &[u8] = include_bytes!("logo_en.bmp");
// Test the functionality of gpu device and driver.
fn test_frame_buffer(device: Arc<GPUDevice>) {
    // get resolution
    let (width, height) = device.resolution().expect("failed to get resolution");

    ..... // same as test frame buffer

    // setup framebuffer
    let buf = device
        .setup_framebuffer()
        .expect("failed to setup framebuffer");

    let bmp = Bmp::<Rgb888>::from_slice(BG_BMP_DATA).unwrap();
    let raw = bmp.as_raw();
    let mut b = Vec::new();
    for i in raw.image_data().chunks(3) {
        let mut v = i.to_vec();
        b.append(&mut v);
        if i == [255, 255, 255] {
            b.push(0x0)
        } else {
            b.push(0xff)
        }
    }
    buf.write_slice(0, &b).unwrap();
    // flush to screen
    device.flush().expect("failed to flush");
    early_println!("flushed to screen");
}

```

**图 12.** Asterinas, 启动!

8.3 系统调用实现及用户态测试

上文提到为了支持用户使用 GPU 服务，我们在 Asterinas 中实现了一个新的操作系统 show_red()。由于该操作系统并不符合 Linux ABI，我们通过在用户态编写汇编程序来测试该系统调用的效果。我们将实现中 show_color 方法包装为 Syscall，编号 5070 进行演示。

用户态程序由汇编代码编写。传入的颜色为蓝色。

Listing 9. 用户态测试程序

```
# SPDX-License-Identifier: MPL-2.0

.global _start

.section .text
_start:
    call    print_asterinas
    mov     $60, %rax           # syscall number of exit
    mov     $0, %rdi            # exit code
    syscall

print_asterinas:
    mov     $5070, %rax
    mov     $0x000000ff, %rdi
    syscall
    ret

.section .rodata
```

该用户态程序会在 Asterinas 中以二进制文件的形式呈现。运行该二进制文件之后，显示屏颜色会变成用户指定的颜色，在该例子中，会变成蓝色。

运行截图如下：

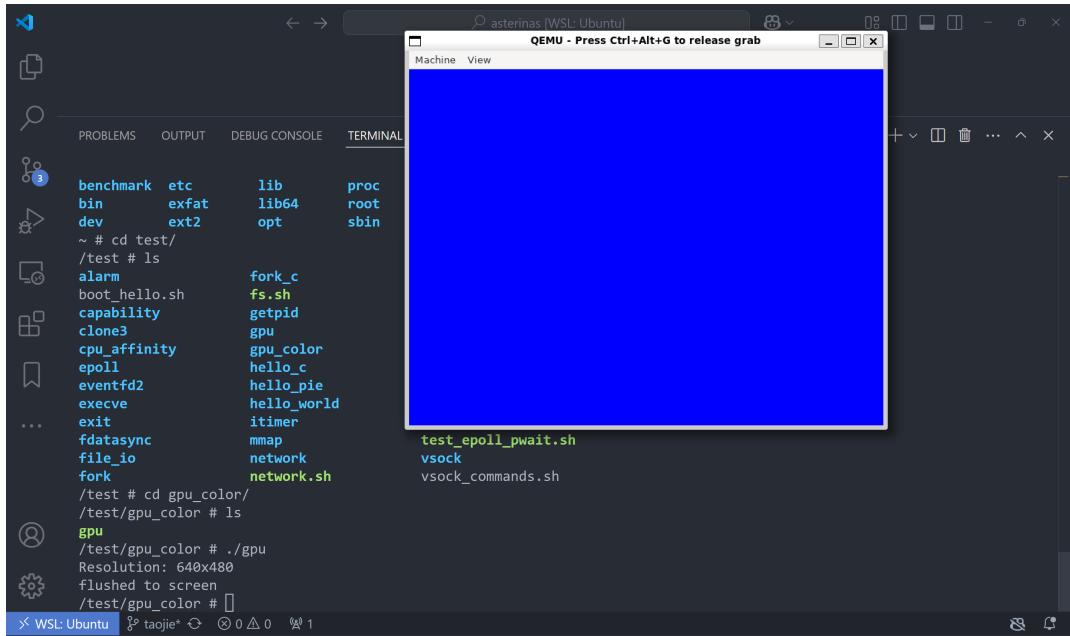


图 13

9. 结论

在本次操作系统项目中，我们成功实现了 virtio-gpu 设备驱动，实现了二维图像和光标在显示屏上的渲染。另外，我们实现了相关系统调用，支持用户态程序操作 GPU 设备。另外，我们对 3D 渲染、ioctl 等操作进行了充分调研，并将在未来继续完成。本次项目的成果计划在未来提交到 Asterinas 主分支，为 Asterinas 得到更广泛使用前进一步。

10. 相关学习链接

10.1 图形渲染

Unity - Manual: Reducing rendering work on the CPU or GPU in URP

Real-Time Rendering Resources

什么是 DrawCall? - 知乎

Unity Draw Call Batching: The Ultimate Guide [2021] | TheGamedev.Guru

猴子也能看懂的渲染管线 (Render Pipeline) - 知乎

10.2 Virtio Driver

Virtio-gpu 实现分析

基于 MMIO 的 Virtio-gpu 分析

使用 Rust 实现 VirtIO 驱动 - 杰哥的运维, 编程, 调板子小笔记

virtio-gpu 介绍

virtio-gpu_virtio gpu-CSDN 博客

virtio ——一种 Linux I/O 半虚拟化框架 [译] - 知乎

Virtio-GPU doesn't fill Response data for cursor queue (#1794) · Issues · QEMU / QEMU · GitLab

屏幕显示 (DRM) 介绍—[野火]Linux 基础与应用开发实战指南——基于 LubanCat-RK 系列板卡文档

virt queue

X11,X Window,hello world 例子 _x11 hello

10.3 Qemu

Qemu

Qemu virtio-gpu

10.4 Rust

Asterinas

Rust embedded-graphics 图形库