操作系统课程设计项目

基于Rust语言的多任务操作系统的设计与实现

张尧
yaoozhang@icloud.com

学号：2152955
仓库地址：https://github.com/Cinea4678/TJU-cinea-os
课程号码：42028702
指导教师：张慧娟
同济大学

目录

1. 项目介绍 8
2. 开发环境 9
2.1. 开发环境 9
2.2. 外部依赖 9
3. 项目设计 12
3.1. 项目整体设计 12
3.2. 项目技术选型 13
3.2.1. 编程语言 13
3.2.2. 硬件架构 15
3.2.3. 模拟器 16
3.3. 项目结构设计 16
3.4. 数据结构和类设计 17
3.4.1. 选择子 17
3.4.2. 进程相关数据 18
3.4.3. 进程 18
3.4.4. 协程 19
3.4.5. 协程执行器 19
3.4.6. 协程唤醒器 20
3.4.7. VGA 文本模式组件 20
3.4.7.1. 屏幕缓冲区 21
3.4.7.2. 屏幕输出器 21
3.4.8. 图形模式组件 21
3.4.8.1. 屏幕缓冲区 21
3.4.8.2. 屏幕绘制器 21
3.4.9. 原始时间 22
3.4.10. 进程调度器 22
3.4.10.1. 进程链表节点 22
3.4.10.2. 进程调度器 23
3.4.11. 事件队列 23
3.4.12. 堆内存分配器 24
3.4.12.1. 分配器链表节点 24
3.4.12.2. 链式分配器 24
3.4.13. 文件元数据 25
3.5. 相关算法设计 25
3.5.1. 进程调度算法设计 25
3.5.2. 堆内存分配算法设计 26

4. 项目实现 26
4.1. 编译和准备 26
4.1.1. 系统编译 26
4.1.2. 用户程序编译 29
4.1.3. 磁盘映像制作 31
4.1.3.1. 工具编写 31
4.1.3.2. 自动转换 34
4.1.3.3. QEMU启动参数 37
4.2. 引导和启动 38
4.3. 入口函数与初始化 41
4.3.1. 操作系统入口函数 41
4.3.2. 初始化 42
4.4. VGA文本模式 43
4.4.1. 相关数据结构定义 43
4.4.1.1. Color 43
4.4.1.2. ColorCode 44
4.4.1.3. ScreenChar 44
4.4.1.4. Buffer 44
4.4.1.5. Writer 45
4.4.2. 方法实现 45
4.4.2.1. 向屏幕写入一个字节 45
4.4.2.2. 向屏幕写入一个字符串 46
4.4.2.3. 清除一整行 47
4.4.2.4. 换行 47
4.5. GDT 和 TSS 48
4.5.1. 常量定义 48
4.5.2. 选择子定义 48
4.5.3. TSS 初始化准备 48
4.5.4. GDT 初始化准备 49
4.5.5. 完成 GDT 和 TSS 初始化 50
4.6. 中断 50
4.6.1. 管理PIC 51
4.6.1.1. 常量定义 51
4.6.1.2. PIC接口创建 52
4.6.2. 准备中断处理函数 52
4.6.2.1. 异常处理函数 52
4.6.2.2. 内核异常界面 54
4.6.2.3. PIC中断处理函数 56

4.6.2.4. 软件中断处理函数 58
4.6.3. 初始化中断描述符表 61
4.6.4. 启用中断 62
4.7. 内存映射 63
4.7.1. 初始化内存映射 63
4.7.2. 页帧分配器 65
4.7.2.1. 初始页帧分配器 65
4.7.2.2. 具有缓存的页帧分配器 66
4.7.3. 内存映射相关方法 68
4.7.3.1. 虚拟地址转换为物理地址 68
4.7.3.2. 页帧分配 70
4.7.3.3. 页帧释放 71
4.8. 堆内存管理 72
4.8.1. 堆内存分配器 72
4.8.1.1. 数据结构实现 73
4.8.1.2. 添加链表节点 73
4.8.1.3. 移除链表节点 75
4.8.1.4. 内存布局调整 77
4.8.1.5. 堆内存分配 78
4.8.1.6. 堆内存释放 79
4.8.1.7. GlobalAlloc 接口实现 79
4.8.2. 堆内存初始化 80
4.8.2.1. 常量定义 80
4.8.2.2. 方法实现 80
4.9. 图形模式 82
4.9.1. 进入图形模式 82
4.9.1.1. 常量定义 82
4.9.1.2. 工具方法 83
4.9.1.3. 配置BGA并进入图形模式 84
4.9.1.4. 使用LFB以获得更大的显存 85
4.9.2. 绘制与渲染 86
4.9.2.1. 物理层操作 86
4.9.2.2. 图层定义 87
4.9.2.3. 图层相关方法实现 88
4.9.2.3.1. 绘制像素点 88
4.9.2.3.2. 绘制矩形 88
4.9.2.3.3. 绘制BMP图片 89
4.9.2.3.4. 绘制TTF光栅字符 90

4.9.2.3.5. 绘制字符串 91
4.9.2.4. 图层渲染 92
4.9.3. 文本输出 93
4.9.3.1. 常量定义 93
4.9.3.2. 数据结构定义 94
4.9.3.3. 方法实现 95
4.9.3.3.1. 字符显示 95
4.9.3.3.2. 字符串显示 97
4.9.3.3.3. 换行处理 97
4.10. 时钟和时间 98
4.10.1. RTC 99
4.10.1.1. RTC时间读取 99
4.10.1.2. RTC中断控制 101
4.10.1.2.1. 打开RTC中断 102
4.10.1.2.2. 初始化RTC 104
4.10.1.2.3. RTC中断处理 105
4.10.2. PIT 108
4.10.2.1. PIT 频率设置 108
4.10.2.2. PIT 中断处理 109
4.10.2.3. PIT Uptime 111
4.10.2.4. PIT Sleep 111
4.10.3. TSC 112
4.10.3.1. 初始化和读取TSC 112
4.10.3.2. TSC Wait 113
4.11. 进程管理 114
4.11.1. 相关常量 114
4.11.2. 相关数据结构 114
4.11.3. Getter / Setter 116
4.11.4. 进程创建（spawn） 119
4.11.4.1. 初始化进程和装入程序 119
4.11.4.2. 切换到用户态并启动进程 123
4.11.5. 进程上下文切换 125
4.11.5.1. 保存现场 125
4.11.5.2. 恢复现场 126
4.11.6. 进程退出 127
4.12. 进程调度 128
4.12.1. 特性设计 128
4.12.2. 轮转算法实现 129

4.12.2.1. 数据结构定义 129
4.12.2.2. 初始化 130
4.12.2.3. 从链表中分配一个新节点 131
4.12.2.4. 归还一个节点到链表中 131
4.12.2.5. 加入一个新进程 132
4.12.2.6. 移除一个旧进程 133
4.12.2.7. 步进到下一个进程 134
4.12.2.8. `ProcessScheduler` 特性实现 134
4.12.3. 调度并切换进程 135
4.13. 系统调用 137
4.13.1. 系统调用发送 138
4.13.2. 系统调用接收 140
4.13.2.1. 中断处理函数 140
4.13.2.2. 服务例程调用 142
4.13.3. 复杂数据传递 144
4.13.3.1. 包含序列化的调用 144
4.13.3.2. 返回序列化数据的调用 145
4.14. 协程 146
4.14.1. 协程定义 146
4.14.2. 协程执行器 147
4.14.3. 协程唤醒器 150
4.14.4. 创建并运行协程 151
4.15. 键盘和鼠标 152
4.15.1. 键盘的初始化 152
4.15.2. 处理键盘中断 152
4.15.3. 处理键盘扫描码流 153
4.15.4. 键盘事件处理函数 155
4.15.5. 鼠标初始化 156
4.15.6. 处理鼠标中断 158
4.15.7. 处理鼠标扫描码流 160
4.15.8. 鼠标事件处理函数 161
4.16. 硬盘 162
4.16.1. 数据结构定义 163
4.16.1.1. FIS（帧信息结构） 163
4.16.1.2. HBA（AHCI控制器） 165
4.16.2. AHCI 初始化 173
4.16.2.1. AHCI 基址内存映射 173
4.16.2.2. AHCI 寄存器内存映射 174

4.16.2.3. 扫描端口 175
4.16.2.4. 启用HBA 177
4.16.2.5. AHCI初始化函数 178
4.16.3. AHCI端口 178
4.16.3.1. 停止/启动命令引擎 178
4.16.3.2. 重置端口基址 179
4.16.3.3. 寻找空闲命令槽 180
4.16.3.4. 从磁盘读取制定扇区 181
4.16.3.5. 向磁盘写入制定扇区 184
4.16.4. 实现Read Write Seek特性 187
4.16.4.1. 数据结构定义 187
4.16.4.2. Read 接口 188
4.16.4.3. Write 接口 191
4.16.4.4. Seek 接口 193
5. 项目展示 194
6. 项目总结 199
6.1. 项目总结 199
6.2. 过程中遇到的困难和挑战 200
6.3. 未来可以改进的方向 201
6.4. 感言和致谢 201

1. 项目介绍

本项目是一个采用Rust语言设计和实现的多任务操作系统，主要面向x86-64架构的计算机进行设计和开发。本项目实现的操作系统具有现代操作系统的绝大部分功能和特性，并允许用户使用本项目中实现的工具集进行用户程序的开发。本项目实现的操作系统还提供了基础的图形界面（GUI）和多窗口支持。

以下是本项目的特点：

- 使用现有、可靠的引导程序
- 原生64位支持
- 能够处理 PIC 8259 硬件中断
- 支持 PS/2 键盘与鼠标
- 支持串口通信
- **页式内存管理**[1]
- **链式堆内存分配**
- RTC 时钟支持
- PCI 设备交互支持
- ATA 硬盘读写支持[2]
- 基于 AHCI 协议的 SATA 硬盘读写支持
- 支持伪随机数生成
- **FAT32 文件系统支持**
- **多进程管理和调度**
- **基于事件队列的事件等待和唤醒系统**
- **同时支持抢占式和非抢占式多任务**，可以使用 async/await 运行多个协程。
- **支持窗口显示、堆内存使用的用户空间，支持ELF可执行程序**
- **完全支持中文显示**
- 800 x 600 高清真彩图形显示
- TTF 字体的读取和显示
- BMP 图片的读取和显示
- 简单的图形化 2048 游戏

[1]作者注意到，在x86-64架构中，段式内存管理已被废弃。详情请查看Intel IA-64 3a手册。
[2]由于该协议读写速度不佳，后期更换为AHCI协议。

2. 开发环境

2.1. 开发环境

本项目的开发环境如下：

1. 开发环境：
   a. macOS 与 Windows 系统
   b. QEMU v8.0.0 x86-64 模拟机
2. 开发软件：
   a. JetBrains CLion 2023.2
3. 开发语言：
   a. **Rust**（版本：nightly-2023-06-19-x86_64-pc-windows-msvc）

2.2. 外部依赖

本项目中使用的外部包（Crate）如下表所示（按字母顺序排序）：

系统内核部分（Syskrnl）与系统接口部分（Sysapi）：

名称	版本	用途
bit_field	0.10.2	提供对由比特位组成的值的操作的便捷封装。本项目中使用其来描述FAT的文件特征(flag)、AHCI中的部分特征位(flag)。
bootloader	0.29.23	制作稳定、可靠的引导程序。本项目使用其来引导计算机进入启动函数。
byteorder	1.4.3	提供编解码大字序或小字序的二进制整数的可靠实现
cinea-os-sysapi	0.1.0	封装了本项目中同时被内核和用户空间所需要的方法、类和常量
conquer-once	0.4.0	提供了一组同步初始化原语，主要用于静态变量的懒加载和一次性的初始化
crossbeam	0.8.2	用于并行编程的一系列工具，在本项目中用于协助开发非抢占式多任务功能。

embedded-graphics	0.8.0	一个二维图形库，主要用于内存受限的嵌入式设备。本项目中使用其作为图像在渲染、传输和显示过程中，像素、图形等数据结构的标准。
fatfs	0.4	一个FAT系列文件系统的Rust实现。本项目使用其作为FAT 32文件系统的底层实现以减轻开发者工作量。
futures-util	0.3.28	一组用于处理`Futures`, `Streams`, `Sinks`, `AsyncRead`和`AsyncWrite`的工具包。本项目使用其辅助实现键盘扫描码流和鼠标数据包的队列式异步处理。
lazy_static	1.4.0	用于声明懒初始化的静态变量的宏。本项目大量使用其以在Rust语言中实现需要懒加载的锁、共用变量等静态变量。
object	0.31.1	提供了跨平台处理可执行对象文件的统一接口，支持读取可重定位机器代码和可执行文件。本项目使用其读取并解析在用户空间中执行的程序。
oem_cp	2.0.0	该库可处理许多用作 OEM 代码页的 SBCS（单字节字符集）。本项目使用其来解码FAT32中的文件名为需要在Rust中使用的Unicode。
pc-keyboard	0.5.0	处理PC键盘输入并进行解码的程序。本项目使用其将键盘传入的原始字节转换为更容易处理的扫描码和ASCII字符。
pic8259	0.10.4	支持处理基本 I/O 中断的 8259 可编程中断控制器。本项目使用其来注册中断处理程序和处理中断。
postcard	1.0.6	一个专注于`#![no_std]`开发的序列化和反序列化工具。本项目大量使用此库来协助在系统调用之间传递复杂数据。
rusttype	0.9.3	RustType 是 FreeType 等库的纯 Rust 替代库。支持读取 TrueType 格式的字体和字体集合、检索字体及其字形的字形和常用属性、使用精确的分析算法（而不是基于采样）来对字形进行亚像素定位的光栅化。本项目使用其来读取和渲染汉字。

serde	1.0.174	Serde 是一个用于高效、通用地序列化和反序列化 Rust 数据结构的框架。此框架是 Rust 中进行序列化和反序列化的主要框架。
spin	0.9.8	提供了基于自旋锁的 `std::sync` 和 `std::lazy` 原语实现。本项目大量采用其实现的互斥锁和读写锁来保证静态变量的线程安全。
tinybmp	0.5.0	一个小型的 BMP 解析器，主要用于嵌入式、无 std 环境。本项目使用其来读取、解析图片文件。
volatile	0.2.3	提供封装类型 `Volatile`、`ReadOnly`、`WriteOnly`、`ReadWrite`，可封装任何可复制类型，并允许对封装值进行易失性内存访问。本项目使用其来包装显存和AHCI控制器，以避免编译器优化导致的读写设备时的未定义行为。
x86	0.52.0	x86 (amd64) 硬件编程库。包含 x86 特定的数据结构描述、数据表，以及用于调用汇编指令的便捷函数。本项目使用其来减轻开发者负担。
x86_64	0.14.10	提供 x86_64 特定的函数和数据结构，并可访问各种系统寄存器。本项目使用其来减轻开发者负担。

用户空间部分（Usrbin）：

名称	**版本**	**用途**
cinea-os-sysapi	0.1.0	封装了本项目中同时被内核和用户空间所需要的方法、类和常量
conquer-once	0.4.0	提供了一组同步初始化原语，主要用于静态变量的懒加载和一次性的初始化
embedded-graphics	0.8.0	一个二维图形库，主要用于内存受限的嵌入式设备。本项目中使用其作为图像在渲染、传输和显示过程中，像素、图形等数据结构的标准。

ryu	1.0.15	Ryū 的纯 Rust 实现，这是一种将浮点数快速转换为十进制字符串的算法。在本项目中用于在用户空间内高效转换浮点数为字符串。
lazy_static	1.4.0	用于声明懒初始化的静态变量的宏。本项目大量使用其以在Rust语言中实现需要懒加载的锁、共用变量等静态变量。
serde_json	1.0	Serde JSON 提供了高效、灵活、安全的方法，在 JSON 字符串和 Rust 数据结构之间转换数据。
spin	0.9.8	提供了基于自旋锁的 `std::sync` 和 `std::lazy` 原语实现。本项目大量采用其实现的互斥锁和读写锁来保证静态变量的线程安全。
tinyrand	0.5.0	描述基本的 RNG 和各种扩展器的特性。这里的特性和 RNG 不需要使用 Rust 标准库。本项目使用其来生成随机数。
ufmt	0.2.0	µfmt是一个比 `core::fmt` 小（6-40 倍）、快（2-9 倍）的替代方案。本项目使用其来规避 `core::fmt` 在用户空间中意外出现且难以解决的 bug。

3. 项目设计

3.1. 项目整体设计

本项目主要由 3 个 Rust Crate 组成：**系统内核**、**系统接口** 和 **用户程序**。

- 系统内核部分：包含系统内核的入口点、初始化系统的程序，以及系统内核所需要的工具类，例如硬盘交互、中断处理。
- 系统接口部分：包含被系统内核和用户程序共同需要的方法、类和常量，以及对系统调用的封装。例如，打开文件、读写文件的方法、表示时间的数据结构的封装。
- 用户程序部分：包含数个即将被编译为可执行程序，在用户空间运行的程序。

它们之间的关系如下图所示：

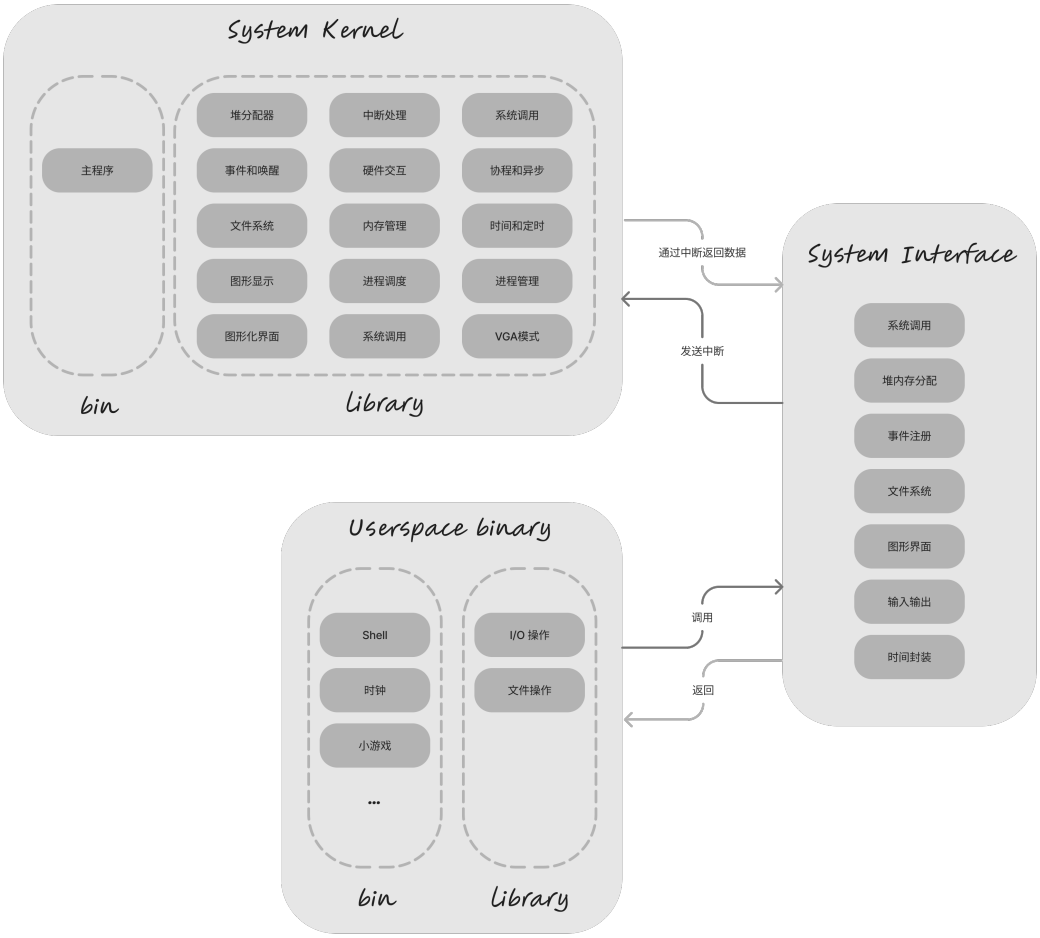

图 1: 项目架构图

在本项目中，以上三部分各自为一个 Crate ，以便单独管理。

3.2. 项目技术选型

3.2.1. 编程语言

长期以来，C语言是进行操作系统编程的经典选择。但是，C语言也有其劣势。首先，在**内存管理** 上，C语言没有自动的内存管理机制，也就是说，程序员需要手动地分配和释放内存，这增加了编程的复杂度和出错的可能性。如果内存分配或释放不当，可能会导致内存泄漏、内存碎片、空指针、悬垂指针等问题，影响程序的性能和稳定性。

此外，在 **类型安全** 上，C语言对变量的类型检查不够严格，允许隐式的类型转换和指针运算，这可能会导致数据的损失或破坏。例如，C语言可以将一个浮点数赋值给一个整

数，或者将一个指向字符的指针赋值给一个指向整数的指针，这些操作可能会造成精度的丢失或者内存的非法访问。

在 **封装性** 上，C语言是一种面向过程的语言，它没有提供面向对象的特性，如类、继承、多态等。这使得C语言难以实现数据和代码的封装和抽象，也不利于程序的模块化和复用。C语言虽然可以通过结构体和函数指针来模拟一些面向对象的功能，但是这样做比较繁琐和低效。

在 **使用外部库** 上，C语言缺乏官方的、权威的软件包管理机制，也缺乏统一的文档规则。这使得C语言很难快速找到合适的外部库及其文档；对外部库而言，每个库都需要开发者自行管理编译、链接和命名冲突，门槛相对较高。

在 **异常处理** 上，C语言没有提供异常处理的机制，也就是说，当程序遇到错误或异常时，无法通过抛出或捕获异常来进行恢复或处理。C语言只能通过返回错误码或使用全局变量来表示错误状态，这使得错误处理变得分散和混乱，也不利于程序的可读性和可维护性。

而 Rust 作为一门现代的、多范式的、编译型的新兴语言，可以提供高性能、内存安全的编程体验。具体来说，Rust 语言对于编写操作系统有以下几个优势：

首先，Rust是 **内存安全** 的。Rust 语言通过一套独特的所有权和借用规则，保证了在编译期就能检测和避免内存错误，如空指针、悬垂指针、内存泄漏、数据竞争等。这些错误是导致操作系统崩溃或者被攻击的常见原因。除此以外，Rust 语言在帮助开发者尽可能地减少不安全的代码的同时，也保留了裸指针、裸内存的操作，允许经验丰富的开发者通过经典的内存操作提高程序的运行效率。

其次，Rust 的 **类型系统** 丰富而严格。支持泛型、特征、枚举、结构体等抽象机制。这让你可以用 Rust 语言表达复杂的逻辑和数据结构，以及实现高层次的抽象和封装。这对于操作系统来说非常有帮助，因为操作系统需要处理各种各样的概念和实体，如进程、文件、设备、用户接口等。此外，Rust 提供了可空、结果等复合类型，帮助开发者处理错误和异常。Rust 的类型检查还十分严格，未经用户手动指明，Rust 编译器会拒绝不同类型之间的操作数的计算，更不会进行隐式转换。这对预防数据溢出、精度丢失等隐患很有帮助。

在 **使用外部库** 上，Rust 可以通过 `Cargo` 工具和 `crates.io` 注册表来管理项目的依赖库，从而便捷地添加来自他人的软件包，大大简化了开发过程。Rust 语言有一个庞大的社区，提供了许多高质量的库，涵盖了各种领域和功能，如网络、图形、数据库、加密、并行、测试等，这些库可以通过 `Cargo` 工具轻松地添加到项目中，无需手动下载或配置。

此外，Rust 语言提供了“无标准库”的特性，这项特性允许外部库的开发者提供可以直接被操作系统开发所使用的库，使用者无需进行任何额外的修改和适配。

在 **封装性** 上，Rust语言可以通过结构体（`struct`）和枚举类（`enum`）来定义自定义的数据类型，并通过实现（`impl`）块来为它们添加方法和关联函数。这样可以将数据和操作封装在一起，形成一个独立的单元，提高代码的可读性和可维护性。Rust语言还可以通过模块（`mod`）来组织代码，通过pub关键字来控制可见性，以及通过 `use` 关键字来引入外部的名称。这样可以实现代码的分层和复用，以及避免命名冲突。

在 **异常处理** 上，Rust语言区分了可恢复错误（recoverable error）和不可恢复错误（unrecoverable error），并分别使用 `Result<T, E>` 类型和 `panic!` 宏来处理。`Result` 类型可以让函数返回成功或失败的结果，让调用者决定如何处理错误，例如重试、记录、转换或传播。`panic!` 宏可以让程序在遇到致命错误时立即停止运行，并打印出错误信息和堆栈回溯。这样可以避免程序继续运行而导致更大的损失或不安全的状态。

综上所述，Rust语言作为一种现代的、多范式的、编译型的新兴语言，具有优秀的安全性和可靠性的同时，性能也十分优秀，是开发操作系统的优秀选择。

3.2.2. 硬件架构

本项目选择了x86_64架构来进行操作系统开发。x86_64架构是一种基于x86架构的64位指令集架构，它由AMD在1999年首次发布，后来被Intel和VIA等厂商采用。x86_64架构在开发操作系统中有以下几个好处：

- x86_64架构具有 **广泛的硬件支持**：x86_64架构是目前最为广泛使用的桌面、笔记本和服务器硬件架构之一。这意味着本项目中的操作系统可以在大量的现有硬件上运行。
- x86_64架构具有 **丰富的文档和社区资源**：由于x86架构的历史悠久和广泛使用，关于该架构的文档和资源非常丰富。无论是官方技术手册，还是社区教程和论坛，都可以为项目的开发工作提供大量的帮助。
- x86_64架构具有 **成熟的工具链**：`Rust` 编译器可以直接编译x86_64架构下的二进制文件，更有 `QEMU` 等成熟的模拟器，使用这些成熟的工具可以帮助项目更高效地开发和调试操作系统。
- x86_64架构 **支持现代特性**：x86_64架构支持一系列现代的硬件特性，如多核处理、高效的中断处理、PAE、NX bit、SSE指令集等。这为操作系统提供了更多的功能和优化的可能性。
- 学习x86_64架构可以将 **学术与实用性结合**：在学术界和业界，x86架构都有着深厚的研究基础。选择x86_64架构进行操作系统开发，不仅有助于学术研究，也有很高的实用性。

3.2.3. 模拟器

本项目采用了QEMU作为项目开发的模拟器。QEMU是一个通用的开源的机器模拟器和虚拟化器，它可以在不同的架构和平台上运行各种操作系统和程序。它具有如下的优势：

- **跨平台性**：QEMU是一个开源的模拟器，支持多种操作系统平台，包括Linux、macOS和Windows。这意味着无论使用哪种主机操作系统进行开发，都可以使用QEMU来测试项目中的操作系统。
- **全系统模拟**：QEMU可以模拟整个计算机系统，包括CPU、内存、硬盘、网络和各种外设。这使得项目可以在QEMU中运行项目的操作系统，就像在真实硬件上一样。
- **支持多种CPU架构**：QEMU支持模拟多种CPU架构，包括x86、x86_64、ARM、MIPS、RISC-V等。这为未来可能的架构迁移或者多架构支持提供了便利。
- **灵活性与可扩展性**：QEMU提供了丰富的命令行选项，可以灵活地配置虚拟机的硬件和行为。而且，由于QEMU是开源的，我们可以根据需要修改其源代码，或者为其开发插件，以满足特定的需求。
- **开发者社区的广泛接受度**：许多操作系统开发者和教程都使用或推荐QEMU，这意味着我们可以在社区中找到大量关于如何使用QEMU进行操作系统开发的资源和帮助。
- **快照与调试**：QEMU支持虚拟机的快照功能，允许我们保存和恢复虚拟机的状态。此外，QEMU还提供了丰富的调试功能，可以与GDB等调试器集成，帮助你诊断操作系统中的问题。

3.3. 项目结构设计

本项目按照如下的目录结构来组织代码：

```
├─.cargo            # 存放Cargo配置文件
├─assets            # 操作系统所使用的资源文件，如图片、字体等
├─dsk               # 挂载到操作系统的磁盘，在测试时使用
│  ├─bin            # 存放二进制程序
│  └─sys            # 系统所需的文件
│      ├─2048       # 2048程序所需的资源
│      └─ast        # 其他程序所需的资源
└─src               # 源代码目录
    ├─sysapi        # 操作系统接口库
    │  ├─.cargo
    │  └─src
    ├─syskrnl       # 操作系统内核
    │  ├─allocator  # 堆分配器模块
    │  ├─event      # 事件和唤醒模块
    │  ├─fs         # 文件系统模块
    │  ├─graphic    # 图形模块
    │  ├─gui        # GUI界面模块
```

```
    │  ├─interrupts # 中断模块
    │  ├─io         # 硬件IO模块
    │  ├─memory     # 内存模块
    │  ├─schedule   # 进程调度模块
    │  ├─syscall    # 系统调用模块
    │  ├─task       # 协程模块
    │  └─time       # 时间模块
    ├─tools         # 操作系统工具（见附录）
    └─usrbin        # 用户程序
        ├─.cargo
        └─src
            ├─bin
            └─std   # 为用户程序实现的简单std替代
```

3.4. 数据结构和类设计

本次项目中使用到了大量的数据结构和类。以下是我对本次项目中的数据结构和类的设计摘要：

3.4.1. 选择子

在x86和x86_64的体系结构中，选择子（Selectors）通常是用于描述段（Segments）的引用，它们是用于内存管理和特权级管理的数据结构。本项目中设计的选择子有以下字段：

字段	说明
`code_selector`	一个代码段选择子。代码段通常包含执行程序的机器代码。
`tss_selector`	一个任务状态段（Task State Segment, TSS）选择子。TSS在x86架构中用于存储有关任务的信息，例如任务的堆栈指针和特权级。
`data_selector`	一个数据段选择子。数据段用于存储程序的变量和数据。
`user_code_selector`	一个用户级代码段选择子。与 `code_selector` 不同的是，这个选择子通常用于非特权级的用户代码。这个字段是公有的，外部可以直接访问它。
`user_data_selector`	一个用户级数据段选择子。与 `data_selector` 不同的是，这个选择子通常用于非特权级的用户数据。这个字段也是公有的，外部可以直接访问它。

3.4.2. 进程相关数据

包含了四个字段，分别是env, dir, user, 和 file_handles:

字段	说明
`env`	存储进程的环境变量。环境变量是一些键值对，它们可以影响进程的行为或配置。例如，PATH环境变量用于指定可执行文件的搜索路径，LANG环境变量用于指定进程的语言设置等。
`dir`	存储进程的当前工作目录。工作目录是一个文件系统路径，它表示进程在执行时所处的位置。
`user`	存储进程的用户标识符。用户标识符是一个字符串，它表示进程所属的用户或用户组。
`file_handles`	存储进程打开的文件句柄。文件句柄是一个操作系统分配的整数，它表示进程对某个文件的访问权限。

3.4.3. 进程

一个名为 `Process` 的公有结构体，代表了一个操作系统中的进程。每个进程都是一个独立运行的程序实例，拥有自己的地址空间和执行上下文。以下是 `Process` 结构体的各个字段：

字段	说明
`id`	公有字段，表示进程的唯一标识符，通常叫做进程 ID 或 PID 。
`code_addr`	进程代码段的起始地址。通常，操作系统加载程序时，会把其机器代码放在特定的地址空间。
`stack_addr`	进程堆栈的起始地址。堆栈用于存储局部变量、函数调用的返回地址等。
`entry_point`	进程的入口点地址，即程序开始执行的地方。
`page_table_frame`	代表该进程的页表的物理帧。在现代操作系统中，页表用于虚拟到物理地址的映射。
`stack_frame`	一个表示中断堆栈帧的值。当进程由于某种原因（例如，硬件中断或异常）被中断时，CPU的状态会被保存在这个结构中。

registers	代表进程的寄存器状态。当操作系统切换到另一个进程时，当前进程的寄存器状态会被保存，以便后续恢复。
data	包含进程相关数据的字段，在上一节中定义。
parent	表示父进程的ID。在一些操作系统中，当一个进程创建一个子进程时，子进程会保留其父进程的ID。
allocator	该进程所使用的堆分配器。每个进程拥有自己独立的堆内存分配器，这样可以独立分配、测量和释放进程的内存，而不会互相影响。

此结构体为操作系统提供了一个完整的进程的表示，包含了进程的各种状态和属性，从地址空间布局到执行上下文，可以用于多任务切换时的现场保存和恢复。

3.4.4. 协程

一个名为 `Task` 的结构体，代表了一个异步任务。异步任务是Rust的 `async/await` 并发模型中的一个核心概念。这种模型允许开发者编写看起来像同步代码的异步代码，但在背后，它充分利用了非阻塞IO和轻量级任务切换，以实现高性能和响应性。

字段	说明
id	任务的唯一标识符，可用于跟踪、查询或管理特定任务。
future	代表一个异步操作，它在未来某个时候可能会完成。

3.4.5. 协程执行器

一个名为 `Executor` 的结构体，代表一个简单的任务执行器。在Rust的 `async/await` 并发模型中，执行器（executor）负责调度和执行 `Future` ，并确保它们得到运行的机会，直到完成。以下是Executor结构体的每个字段：

字段	说明
tasks	一个哈希表(`BTreeMap`)，用于存储所有已知的Task实例。键是TaskId，代表每个任务的唯一标识符，而值则是对应的Task。使用 `BTreeMap` 意味着任务是按TaskId排序存储的，可以快速查找、插入和删除。

task_queue	一个线程安全的队列，存储了所有待运行的TaskId。这个队列有固定的大小，它使用了为无锁并发设计的 ArrayQueue 数据结构，这个数据结构提供了高效的入队和出队操作。在设计时，使用了Rust提供的 Arc 类型，因此所有 TaskWaker 和 Executor 实例都共享同一个 task_queue 实例。
waker_cache	另一个哈希表，用于缓存任务的 Waker。在Rust的异步模型中，Waker 用于唤醒任务，当其所等待的资源（例如，网络数据或锁）变得可用时。每个任务（由 TaskId 标识）都有其对应的 Waker 。与 tasks 字段类似，waker_cache 也确保了快速的查找、插入和删除。

Executor 结构体提供了一个简单的异步任务执行器的骨架。当新任务被提交到执行器时，它的 TaskId 会被添加到 task_queue 。执行器会周期性地从队列中取出任务ID，找到对应的 Task ，并尝试执行它。如果任务因为某些资源不可用而被暂停，其 Waker 会被缓存起来，等到资源可用时再使用 Waker 来唤醒任务。

3.4.6. 协程唤醒器

TaskWaker 是用于实现 Waker 特性的结构体，以下是它的字段：

字段	说明
task_id	唤醒目标任务的唯一标识符，类型为 TaskId。当这个 TaskWaker 被触发时，它知道要唤醒哪个任务。
task_queue	一个线程安全的队列，存储了所有待运行的TaskId。这个队列有固定的大小，它使用了为无锁并发设计的 ArrayQueue 数据结构，这个数据结构提供了高效的入队和出队操作。在设计时，使用了Rust提供的 Arc 类型，因此所有 TaskWaker 和 Executor 实例都共享同一个 task_queue 实例。

当 TaskWaker 被触发（或唤醒）时，它会将其内部的 task_id 添加到 task_queue 中。执行器（前面描述的 Executor ）将周期性地检查这个队列，从中取出任务ID，并运行对应的任务。

3.4.7. VGA 文本模式组件

VGA 文本模式是计算机历史上用于在屏幕上显示文本的一种模式，在本项目正在启动且尚未完成内存初始化前，需要先在此模式下完成过渡。

3.4.7.1. 屏幕缓冲区

一个名为 `Buffer` ，代表VGA文本模式的屏幕缓冲区的结构体。

字段	说明
`chars`	一个`ScreenChar`类型的二维数组，代表屏幕上的字符。每个 `ScreenChar` 代表屏幕上的一个字符，包括其ASCII码和颜色信息。由于缓冲区位于显存上，为了避免Rust编译器优化导致未定义行为，我们采用了`Volatile`包装器封装数组。

3.4.7.2. 屏幕输出器

一个名为 `Writer` ，用于在VGA屏幕上写入文本的结构体。

字段	说明
`row_position`	当前光标在屏幕上的行位置。
`column_position`	当前光标在屏幕上的列位置。
`color_code`	当前用于写入的文本的颜色
`buffer`	一个指向屏幕缓冲区（即上述的`Buffer`）的可变引用

3.4.8. 图形模式组件

3.4.8.1. 屏幕缓冲区

一个名为 `Buffer` ，代表图形模式的屏幕缓冲区的结构体。

字段	说明
`chars`	一个`Rgb888`类型的二维数组，代表屏幕上的像素。`Rgb888` 是一个表示RGB颜色的结构体，每个颜色组件（红、绿、蓝）使用8位来表示，这是常见的24位颜色模式。由于缓冲区位于显存上，为了避免Rust编译器优化导致未定义行为，我们采用了`Volatile`包装器封装数组。

3.4.8.2. 屏幕绘制器

一个名为 `Writer` ，用于在图形模式中绘制图形的结构体。

字段	说明
data	类型为Vec<Vec<(Rgb888, bool)>>。data是一个二维向量，其中每个内部向量代表一行像素数据。(Rgb888, bool)是一个元组，其中Rgb888代表一个像素的颜色值，而bool值用于标识该像素是否应该被渲染或视为透明，以节省处理多层覆盖时的运算量。

3.4.9. 原始时间

一个名为RawTime的结构体，用于表示从CMOS中直接读取的时间。CMOS（互补金属氧化物半导体）是一个计算机上的小型存储设备，用于存储基本输入/输出系统（BIOS）设置和系统时间。

字段	说明
second	秒
minute	分钟
hour	小时
day	日期
month	月份
year	年份

3.4.10. 进程调度器

本项目由于体量不大，故目前采用了轮转算法进行进程调度。轮转算法（Round Robin）是一个简单而公平的调度算法。这种算法周期性地从任务列表中选取任务执行，当一个任务的时间片用完后，调度器将移动到下一个任务。

3.4.10.1. 进程链表节点

一个名为 RoundRollNode 的结构体，表示链表中的节点。

字段	说明
pid	表示进程的唯一标识符。
empty	表示此节点是否为空闲，也就是说是否没有关联的进程。

skip	表示是否应该跳过此节点/进程，这可能是因为该进程暂时不可运行或等待某些资源。
next	指向链表中下一个任务的指针。
prev	指向链表中上一个任务的指针。

3.4.10.2. 进程调度器

一个名为RoundRollScheduler的结构，表示进程调度器本身。

字段	说明
table	一个RoundRollNode向量，表示所有的任务。这是一个双向链表，用于轮转调度。
map	一个B树映射，其键为进程的PID，值为进程在table中的索引。这使得通过PID查找特定进程变得快速和简单。
head	一个指向链表头部的指针。
empty	一个指向链表首个空节点的指针。
cursor	一个指向当前正在执行或被调度的任务的指针。

以上两个结构体提供了轮转调度算法所需的基础数据结构和状态。使用RoundRollScheduler，操作系统或调度器可以公平地遍历所有的任务，确保每个任务都有机会得到执行，同时还能处理像任务跳过或任务结束等特殊情况。

3.4.11. 事件队列

一个名为EventQueue的结构体，用于管理和调度事件。

字段	说明
queue	一个BTreeMap数据结构，其中键是事件类型EventType，值是一个VecDeque<usize>。这意味着每种事件类型都有一个与之关联的双端队列，存储与该事件相关的数据或进程ID。

front_proc	表示当前处于前台的进程的ID。在图形用户界面中，前台进程通常是用户当前正在与之交互的进程。任何发生的事件（如键盘输入、鼠标点击等）都可能首先传递给这个前台进程。

EventQueue结构体的主要作用是管理和调度不同类型的事件。它允许存储多个事件类型，并为每种事件类型维护一个队列。此外，它还知道哪个进程处于前台，因此可以将事件优先传递给该进程。

3.4.12. 堆内存分配器

3.4.12.1. 分配器链表节点

一个名为ListNode的结构体，代表了链表中的一个节点。它表示一个内存块。

字段	说明
size	表示节点所代表的内存块的大小。
next	一个Option类型，其内部是一个静态可变引用到另一个ListNode。这使得链表节点可以连接到下一个节点。当它为None时，表示这是链表中的最后一个节点。

3.4.12.2. 链式分配器

一个名为LinkedListAllocator的结构体，表示基于链表的内存分配器。

字段	说明
head	链表的头节点，表示内存分配器中的第一个内存块。
size	表示整个链表（也就是所有可用内存块）的总大小。
allocated	表示已经被分配出去的内存的总量。

当请求一个新的内存块时，分配器会遍历链表，查找一个满足需求的节点（也就是一个大小足够的内存块）。一旦找到，它会从该节点分配所需大小的内存，并更新节点的size和next字段，以及LinkedListAllocator的allocated字段。当释放一个内存块时，分配器会将该块重新添加到链表中，并进行合并操作，将相邻的空闲块合并成一个更大的块。

3.4.13. 文件元数据

一个名为`Metadata`的结构体，用于表示文件或目录的元数据。元数据是描述其他数据的数据，对于文件系统来说，它提供了关于文件或目录的详细信息。

字段	说明
`path`	文件或目录的绝对路径。
`short_file_name`	文件的简短名称。这是为了与老式的文件系统兼容。
`file_name`	文件的完整名称。
`attributes`	包含了文件或目录的各种属性，如只读、隐藏、系统文件等。
`is_dir`	此元数据是否代表一个目录。
`is_file`	此元数据是否代表一个常规文件。
`len`	文件的大小，以字节为单位。
`created`	文件或目录的创建日期和时间。
`accessed`	文件或目录的最后访问日期。
`modified`	文件或目录的最后修改日期和时间。

3.5. 相关算法设计

3.5.1. 进程调度算法设计

经过权衡，我采用了轮转算法（Round Robin）作为我的进程调度算法。轮转算法是一种常用的进程调度算法，它的基本思想是按照进程到达的先后顺序，轮流给每个进程分配一个固定的时间片，让它们在单位时间内执行，如果时间片用完了，进程还没有结束，就把它放到就绪队列的末尾，等待下一次调度。这样可以保证每个进程都能得到公平的处理，而且响应时间较短。

轮转算法的优点是：

1. 兼顾长短作业，不会出现饥饿现象。
2. 适用于分时系统，可以提高系统的吞吐量和用户的满意度。
3. 实现简单，不需要估计进程的运行时间。

轮转算法的缺点是：

1. 平均等待时间较长，尤其是当进程的运行时间都相近时。

2. 上下文切换频繁，会增加系统的开销。
3. 不区分任务的紧急程度和重要性。

考虑到本次项目的性质，项目中的操作系统没有运行大量进程的需求，且项目中运行的进程一般在2到5个之间，没有必要使用其他复杂的算法（如优先级调度、多级反馈队列调度等）。采取轮转算法可以帮助本项目快速完成相关算法的代码实现，并且由于其简单的性质，也会降低未来出现bug的可能。

3.5.2. 堆内存分配算法设计

我采用了最坏适配算法作为我的对堆内存分配算法。这是一种在内存中寻找最大的空闲分区进行分配的方法。这种方法的优点是可以减少外部碎片，也就是分配后剩余的空闲分区的大小。

最坏适配算法的优点是：

1. 使用链表结构对内存块预排序，在申请新内存时速度很快，在O(1)内就可以完成分配。
2. 充分减少外部碎片，保证总是有比较大的剩余内存块存在。

最坏适配算法的缺点是：

1. 在释放分区时需要合并相邻块，然后在链表中根据排序找到其正确的位置。这个过程可能消耗较多时间。
2. 不相邻的小空闲分区很难被合并或再次利用，因为算法总是优先选择大的空闲分区。

但是，综合考虑，在分配速度和空间利用上，最坏适配算法仍然是一个比较适合的选择。它把分配时间更多地留到了回收阶段上，而系统实际上可以采取更多方案来降低用户感知的回收时间（例如，定期将已结束使用的内存块统一合并、放回链表等）。

4. 项目实现

4.1. 编译和准备

4.1.1. 系统编译

本项目中的系统使用`cargo`管理和编译。`cargo` 是 Rust 的构建工具和包管理器。为了满足不同项目的需求，`cargo` 提供了一系列的配置选项，这些配置选项可以在 `.cargo/config` 文件中进行设置。

以下是本项目的`cargo`配置：

```
[build]
target = "x86_64-cinea_os.json"
```

```
[unstable]
build-std = ["core", "compiler_builtins", "alloc"]
build-std-features = ["compiler-builtins-mem"]

[target.'cfg(target_os = "none")']
runner = "bootimage runner"
```

1. **编译配置**：`target`选项指定了要使用的目标规范文件。在本项目中，使用了一个自定义的目标描述文件 `x86_64-cinea_os.json`。当运行 cargo build 时，Cargo 将按照这个目标描述文件来构建项目。这对交叉编译或为特定平台/操作系统定制构建很有帮助。

   以下是`x86_64-cinea_os.json`文件的具体内容：

   ```
   {
       "llvm-target": "x86_64-unknown-none",
       "data-layout": "e-m:e-i64:64-f80:128-n8:16:32:64-S128",
       "arch": "x86_64",
       "target-endian": "little",
       "target-pointer-width": "64",
       "target-c-int-width": "32",
       "os": "none",
       "executables": true,
       "linker-flavor": "ld.lld",
       "linker": "rust-lld",
       "panic-strategy": "abort",
       "disable-redzone": true,
       "features": "-mmx,-sse,+soft-float"
   }
   ```

   `x86_64-cinea_os.json`文件描述了一个专门为 `x86_64` 架构上的裸机环境设计的自定义的目标平台。以下是其中各个字段的说明：

字段	说明
`llvm-target`	指定了 LLVM 使用的目标三元组。在此处，我们表明项目目标为 x86_64 架构，没有特定的操作系统 (unknown-none)。
`data-layout`	指定数据在内存中的布局。这是一个专门的字符串，描述了各种数据类型的大小、对齐方式等，具体含义参见相关文档。

`arch`	代表目标架构，即 `x86_64`。
`target-endian`	这定义了目标硬件的字节顺序。在这里，我们使用 little 字节顺序，这是 `x86_64` 架构的标准。
`target-pointer-width`	表示指针的宽度。对于 `x86_64` 架构，这是 `64` 位。
`target-c-int-width`	表示 C 语言中 `int` 类型的宽度，此处为 `32` 位。
`os`	操作系统字段，设置为 none，表示这是一个裸机目标。
`executables`	表明此目标支持可执行文件。
`linker-flavor`	定义了链接器信息。这里选择了 LLVM 的 `ld.lld` 链接器，用于链接裸机程序。
`panic-strategy`	在遇到 `panic` 时的策略，设置为 `abort`，表示程序应该立即中止。
`disable-redzone`	RedZone 是 x86_64 架构上的一个特性，用于优化函数调用。但在某些上下文（如操作系统开发）中，它可能导致问题。此字段禁用了 RedZone。
`features`	列出了目标硬件的特性或扩展，以及是否启用它们。在这里，我们禁用了 mmx 和 sse 指令集，而启用了软浮点运算 (soft-float)。

2. **不稳定Cargo功能**：我们在使用Rust和Cargo编译操作系统时，需要使用一些不稳定的功能来保证我们的操作系统可以被顺利编译。这些功能未必不安全，但是使用这些功能可能需要开发者格外谨慎，因此我们需要在`[unstable]`中启用它们。

   以下是这些实验性功能的使用列表：

   - **build-std**：`build-std`是一个列出了要为当前的目标平台构建的 Rust 标准库的组件的数组。由于我们的操作系统要运行在自定义的平台上，而预编译的标准库很显然只能适配当前开发机的平台，因此我们需要指示Cargo为目标平台额外重新编译Rust的标准库。在这里，我们重新编译了`core`、`compiler_builtins` 和 `alloc`三个库。`core`是Rust标准库的核心组成，`compiler_builtins`提供LLVM需要的许多底层操作，比如`memcpy`。`alloc`提供了需要堆内存分配的数据结构，如`Vec`和`String`。

   - **build-std-features**：`build-std-features`用于指定 `build-std` 中列出的库应该启用的特性。在此配置中，启用了 `compiler-builtins-mem` 特性。这项特性的作用是让

compiler_builtins提供诸如memcpy、memset等方法的实现，因为通常我们的操作系统在裸机上执行时，是不会有这些函数被提供的。

3. **特定目标平台配置**：我们需要配置Cargo，以让其在运行一些命令时，可以按照我们希望的方式执行。
   - **runner**：当使用 `cargo run` 命令时，这行配置指定了应该使用哪个运行器来执行我们的程序。在此配置中，使用了 `bootimage runner`，这意味着我们的程序是一个裸机应用或操作系统，并且需要使用 `bootimage` 工具来创建一个启动映像，然后使用某个模拟器（如 QEMU）来运行它。

4.1.2. 用户程序编译

用户程序在本项目中被组织在`userbin` Crate下的`bin/`目录中。这在Rust中表示其中的每个`rs`源文件代表一个完整的可执行程序。在编译时，我们可以手动指定所需要编译的文件，如：

```
cargo build --release --bin shell
```

为了节省时间，本项目编写了一个 `Makefile` 用于编译、裁剪和收集编译完成的二进制文件，如下：

```
.PHONY: bin

mode = release
strip = false

RUSTC = cargo rustc
RUSTFLAGS =
STRIP := $(strip)

SD = src/bin/  # Src dir

ifeq ($(mode),release)
RUSTFLAGS += -r
else
STRIP = false
endif

hello: src/bin/hello.rs
$(RUSTC) $(RUSTFLAGS) --bin hello
touch target/hello

nothing: src/bin/nothing.rs
$(RUSTC) $(RUSTFLAGS) --bin nothing
```

```
touch target/nothing

shell: src/bin/shell.rs
$(RUSTC) $(RUSTFLAGS) --bin shell
touch target/shell

infprint: src/bin/infprint.rs
$(RUSTC) $(RUSTFLAGS) --bin infprint
touch target/infprint

echo: src/bin/echo.rs
$(RUSTC) $(RUSTFLAGS) --bin echo
touch target/echo

taffy: src/bin/taffy.rs
$(RUSTC) $(RUSTFLAGS) --bin taffy
touch target/taffy

clock: src/bin/clock.rs
$(RUSTC) $(RUSTFLAGS) --bin clock
touch target/clock

2048: src/bin/2048.rs
$(RUSTC) $(RUSTFLAGS) --bin 2048
touch target/2048

bin: hello nothing shell infprint echo taffy clock 2048
basename -s .rs src/bin/*.rs | xargs -I {} \
cp target/x86_64-cinea_os/$(mode)/{} ../../dsk/bin/{}
if [ "$(STRIP)" = "true" ] && [ `arch` = "x86_64" ]; then \
strip ../../dsk/bin/*; \
fi
```

以下是对此`Makefile`的说明：

1. **.PHONY规则**：指明`bin`目标不对应文件系统上的任何文件。即使存在一个名为`bin`的文件，当执行`make bin`时，它仍然会执行`bin`目标。
2. **变量定义**：

变量	定义
`mode`	用于定义编译模式，默认为release
`strip`	定义是否对二进制进行裁剪，默认为false
`RUSTC`	用于定义编译器命令，这里使用了cargo rustc

RUSTFLAGS	Rust的编译器标志变量，默认为空
SD	源代码目录的缩写，定义为src/bin/

3. **条件判断**：
   - `ifeq ($(mode),release)`：判断mode是否为release。如果是，则添加-r标志到RUSTFLAGS，否则设置strip为false。
4. **目标定义**：每个目标对应一个Rust源文件，并用于编译该文件。例如，hello目标对应src/bin/hello.rs文件。每个目标执行两个命令：
   a. 使用RUSTC和RUSTFLAGS来编译源文件。
   b. 使用touch命令在target目录下创建一个同名文件。
5. **bin目标**：当执行make bin时，它会编译上面定义的所有Rust文件，并执行以下操作：
   - 使用basename命令从src/bin/中的所有.rs文件中获取基本名，并复制相应的编译结果到../../dsk/bin/目录。
   - 如果strip变量为true且当前架构为x86_64，则使用strip命令去除../../dsk/bin/目录下所有二进制文件的符号信息，从而减小它们的大小。要求使用x86_64架构进行裁剪的原因是在ARM等其他架构下，strip命令不能裁剪x86_64的二进制文件。

4.1.3. 磁盘映像制作

本系统支持FAT系列文件系统。不过，在系统开发中，我们通常没有条件真的接入一个物理硬盘并将其格式化为FAT32格式————这就意味着我们需要在虚拟机启动前制作一个FAT32格式的磁盘映像，并将其挂载到QEMU上。不再需要准备真实的物理磁盘，这是现代操作系统开发的便利之一。

4.1.3.1. 工具编写

我们在src/tools目录下建立一个新的Rust Crate：fs-compiler。由于Rust Cargo的config具有传染性，我们将其移动到另外的独立文件夹，并在src/tools下建立Git Submodule。

fs-compiler程序接受两个额外的参数：磁盘映像根目录所在的文件夹及存放磁盘映像的文件位置。如果参数不足，它将打印使用方法并返回。接下来，如果磁盘映像文件已经存在，会先将其删除；然后，创建一个空白的磁盘映像文件并填充40MB的0，以备使用。接下来，程序调用compile.rs模块中的方法来继续后续的转换过程。

src/compile.rs提供了递归复制文件和目录到FAT32磁盘镜像的功能。compile函数是转换过程的主要函数。以下是此函数的工作流程：

1. 打开或创建磁盘镜像文件以读/写。
2. 用fatfs库格式化磁盘镜像为FAT32格式。
3. 建一个新的fatfs::FileSystem实例以访问磁盘镜像

4. 获取磁盘镜像的根目录，并从指定的真实路径复制文件和目录
5. 卸载文件系统以确保所有的更改都被写入磁盘镜像。

复制文件的过程主要使用`copy_files_recursive`函数来递归执行。这是一个递归函数，用于将真实文件系统中的文件和目录复制到FAT32格式的磁盘镜像中。以下是该函数的工作流程：
1. 遍历指定目录的所有项。
2. 对于每一项，检查它是否是目录或文件。
3. 如果是目录，它会创建一个新的目录在磁盘镜像中，并递归复制子目录。
4. 如果是文件，它会读取文件内容并将其写入磁盘镜像中。

以下是该工具的代码实现：

`src/main.rs`

```rust
use std::fs;
use std::io::Write;

pub mod compile;

fn create_file(filename:&str) {
    let mut file = fs::File::create(filename).unwrap();

    let size = 40 << 20; // 40MB
    let buffer = vec![0; size];

    file.write_all(&buffer).unwrap();
}

fn main() {
    let args = std::env::args().collect::<Vec<String>>();

    if args.len() < 3 {
        println!("Usage: {} <pathname> <output_filename>", args[0]);
        return;
    }

    let pathname = &args[1];
    let output_filename = &args[2];

    fs::remove_file(output_filename).unwrap_or(());
    create_file(output_filename);

    compile::compile(pathname, output_filename);
}
```

src/compile.rs

```
use std::{
    fs::{self, DirEntry},
    path::{self, PathBuf},
    process::exit
};
use std::io::Write;

fn is_directory(entry: &DirEntry) -> usize {
    let file_type = entry.file_type();

    if let Ok(is_directory) = file_type.map(|ft| ft.is_dir()) {
        if is_directory {
            0
        } else {
            1
        }
    } else {
        2
    }
}

pub fn copy_files_recursive<T>(fs_dir: fatfs::Dir<T>, real_path:
&path::PathBuf)
where
    T: fatfs::ReadWriteSeek
{
    for entry in fs::read_dir(real_path).unwrap() {
        if let Ok(entry) = entry {
            match is_directory(&entry){
                0=>{
                    let mut sub_path=PathBuf::from(real_path);
                    sub_path.push(entry.file_name());
                    // println!("{:?}->{:?}", entry.file_name(), sub_path);

                    let sub_dir =
fs_dir.create_dir(entry.file_name().to_str().unwrap()).unwrap();

                    copy_files_recursive(sub_dir, &sub_path);
                },
                1=>{
                    let mut sub_path=PathBuf::from(real_path);
                    sub_path.push(entry.file_name());
                    let file_content = fs::read(&sub_path).unwrap();
                    // println!("{:?}->{:?}", entry.file_name(), sub_path);

                    let mut sub_file =
```

```
fs_dir.create_file(entry.file_name().to_str().unwrap()).unwrap();

                    sub_file.truncate().unwrap();
                    sub_file.write_all(file_content.as_slice()).unwrap();
                },
                _=>{}
            }
        }
    }
}

pub fn compile(pathname: &str, output_filename: &str) {
    println!("Compiling {} to {}", pathname, output_filename);

    if let Ok(mut img_file) = std::fs::OpenOptions::new()
        .read(true)
        .write(true)
        .create(true)
        .open(output_filename)
    {
        fatfs::format_volume(&mut img_file,
fatfs::FormatVolumeOptions::new().fat_type(fatfs::FatType::Fat32)).unwrap();
        let fs = fatfs::FileSystem::new(&img_file,
fatfs::FsOptions::new()).unwrap();
        let root_dir = fs.root_dir();
        let root_path = path::PathBuf::from(pathname);

        copy_files_recursive(root_dir, &root_path);

        fs.unmount().unwrap();
    } else {
        println!("fatal error: cannot open output file '{}'",
output_filename);
        exit(1);
    }
}
```

4.1.3.2. 自动转换

完成工具的编写后，我们还需要进一步实现自动转换的过程。Bootimage工具提供了自定义启动命令的选项，我们可以借助此能力来将自动转换过程放置在虚拟机启动前：

`cargo.toml`（节选）

```
[package.metadata.bootimage]
run-command = ["python", "start.py", "{}"]
```

在这里，我们自定义了run-command参数，这样，Bootimage工具就会使用我们的工具来完成虚拟机启动的工作，我们可以在此机会中完成磁盘的转换和挂载。start.py的工作流程如下所示：

1. **初始化**:
   a. 定义一些全局变量，如FS、FS_SOURCE、ALWAYS_FETCH_TOOLS等。
   b. 检查当前运行的Python版本，如果不是Python 3，则尝试使用python3重新启动这个脚本。
2. **定义get_latest_modified_time函数**:
   a. 此函数接受一个目录路径，并递归查找该目录及其子目录中的所有文件。
   b. 返回最近修改的文件的修改时间。
3. **设置环境变量**:
   a. 根据当前平台（Windows或其他）设置EXE_PREFIX和EXE_SUFFIX。
   b. 获取命令行参数sys.argv。这是为了提供启动映像的位置。
4. **检查文件系统编译器是否需要编译**:
   a. 如果设置为始终获取工具或文件系统编译器不存在，则执行一系列命令来获取和编译文件系统编译器。这涉及到从GitHub克隆工具、复制目录、更改工作目录、用cargo构建工具等。
5. **检查文件系统是否需要重新编译**:
   a. 检查文件系统磁盘映像是否存在。
   b. 使用get_latest_modified_time函数比较文件系统源目录和磁盘映像的修改时间。
   c. 根据上述两个步骤的结果，确定是否需要重新编译文件系统。
   d. 如果需要，使用文件系统编译器来编译文件系统。
6. **启动QEMU**:
   a. 使用os.system命令启动QEMU模拟器，以模拟x86_64架构。
   b. 传递多个参数，如驱动器格式、文件、串行接口、内存大小、监视器端口等。

以下是start.py的内容：

start.py

```
import os
import sys

FS = "datadisk.img"
FS_SOURCE = "dsk"
ALWAYS_FETCH_TOOLS = False
ALWAYS_RECOMPILE_TOOLS = False
ALWAYS_RECOMPILE = False

if sys.version_info.major < 3:
    print("Trying to call Python 3...")
```

```
    os.system("python3 " + __file__)
    exit(0)

import shutil
import platform


def get_latest_modified_time(directory):
    latest_time = 0

    for root, dirs, files in os.walk(directory):
        for file in files:
            file_path = os.path.join(root, file)
            modified_time = os.path.getmtime(file_path)
            if modified_time > latest_time:
                latest_time = modified_time

    return latest_time


PWD = os.path.dirname(os.path.abspath(__file__))
EXE_PREFIX = "" if platform.system() == "Windows" else "./"
EXE_SUFFIX = ".exe" if platform.system() == "Windows" else ""

argv = sys.argv
if len(argv) < 2:
    print("Please give me an additional argument: boot image's position.")
    exit(1)

BOOT_IMAGE = argv[1]

print("Checking need for compile FS Helper...")
FS_COMPLIER = EXE_PREFIX + "fs-compiler" + EXE_SUFFIX
if ALWAYS_FETCH_TOOLS:
    os.system("git submodule update --remote")
if ALWAYS_RECOMPILE_TOOLS or not os.path.exists(FS_COMPLIER):
    os.makedirs("../tmp-compiling", exist_ok=True)
    TOOL_DIR = "../tmp-compiling"

    if os.path.exists("../tmp-compiling"):
        shutil.rmtree("../tmp-compiling", ignore_errors=True)

    if not os.path.exists("src/tools/fs-compiler"):
        print("Cloning fs-compiler...")
        os.system("git clone git@github.com:Cinea4678/TJU-cinea-os-
tools.git ../tmp-compiling")
    else:
```

```
        shutil.copytree("src/tools/fs-compiler", "../tmp-compiling/fs-compiler", symlinks=True)

    os.chdir(TOOL_DIR + "/fs-compiler")
    os.system("cargo build --release")
    os.chdir(PWD)
    shutil.copy(TOOL_DIR + "/fs-compiler/target/release/" + FS_COMPLIER, FS_COMPLIER)
    shutil.rmtree("../tmp-compiling", ignore_errors=True)

print("Checking need for recompile the file system...")
re_compile = False
if not os.path.exists(FS):
    print("File System not exist yet...")
    re_compile = True
elif get_latest_modified_time(FS_SOURCE) > os.path.getmtime(FS):
    print("File System is older than source dir...")
    re_compile = True
if ALWAYS_RECOMPILE or re_compile:
    print("Recompiling the file system...")
    os.system(FS_COMPLIER + " " + FS_SOURCE + " " + FS)
else:
    print("File System is already newest.")

print("Starting QEMU...", flush=True)
os.system(f"qemu-system-x86_64 -drive format=raw,file={BOOT_IMAGE} -serial stdio -m 1G -monitor telnet:localhost:4444,server,nowait -drive id=data_disk,format=raw,file=datadisk.img,if=none -device ahci,id=ahci -device ide-hd,drive=data_disk,bus=ahci.0")
```

4.1.3.3. QEMU启动参数

完整QEMU启动参数如下：

```
qemu-system-x86_64 -drive format=raw,file=${BOOT_IMAGE} -serial stdio -m 1G -monitor telnet:localhost:4444,server,nowait -drive id=data_disk,format=raw,file=datadisk.img,if=none -device ahci,id=ahci -device ide-hd,drive=data_disk,bus=ahci.0
```

各参数的含义和作用如下所示：

参数	说明
`qemu-system-x86_64`	QEMU的命令行工具，用于模拟x86_64架构的机器。

`-drive format=raw,file=BOOT_IMAGE`	挂载一个磁盘映像到虚拟机，这个磁盘映像采用原始格式，路径位于BOOT_IMAGE变量。这里挂载的是系统启动映像, 内含本项目操作系统的代码。
`-serial stdio`	将QEMU的串行控制台重定向到主机的标准输入/输出。这样做的目的是允许我们在虚拟机内使用串口通信在控制台上打印调试信息。
`-m 1G`	为模拟机器分配1GB的内存。
`-monitor telnet:localhost:4444,server,nowait`	定义了一个QEMU监控控制台，可以用于动态控制和监视模拟的机器。监控台使用telnet协议，在本机的4444端口上开启监听。这样做的目的是允许我们使用telnet协议连接到QEMU监控台，便于我们调试操作系统、监控其状态。
`-drive id=data_disk,format=raw,file=datadisk.img,if=none -device ahci,id=ahci -device ide-hd,drive=data_disk,bus=ahci.0`	挂载另一个磁盘映像到虚拟机，并将它挂载到AHCI 0号端口。这里挂载的是数据盘，内含操作系统需要的各种数据。

4.2. 引导和启动

传统的操作系统需要手动制作引导程序，幸运的是，Rust社区提供了制作引导程序的工具，我们可以使用它便捷、快速地制作安全、可靠、经过广泛验证的可引导镜像。

Bootloader 包就是其中之一。它采用了三阶段式的引导方案来将操作系统加载到内存中，从入口点运行操作系统。

1. **第一阶段**：这一阶段中Bootloader扮演了一个传统的首阶段引导加载器，也被称为主引导记录 (MBR)。它的主要任务是为第二阶段的引导加载器或操作系统内核准备环境。以下是这一阶段所完成的工作的简要总结：
   a. 初始化环境:
      - 初始化所有段寄存器为0。
      - 初始化栈指针为 0x7c00。
      - 显示引导开始的消息：“Booting (first stage)...”。
   b. 启用 `A20` 线:

- 通过 I/O 端口 92 启用 `A20` 线。这是为了让计算机能够访问超过1MB的内存。

c. 进入保护模式:
   - 清除中断。
   - 加载全局描述表 (GDT)。
   - 设置`CR0`寄存器的保护模式位。
   - 进行一个远跳，确保进入到保护模式。

d. 检查 `INT 13h` 扩展:
   - 检查计算机是否支持扩展的磁盘读取功能。这是为了能够读取超过64KB的数据。

e. 从磁盘加载其余的引导加载器:
   - 使用 `INT 13h` 功能 `42h` 从磁盘加载第二阶段的引导加载器或操作系统内核。
   - 如果加载失败，会显示错误信息。

f. 跳转到第二阶段:
   - 代码执行跳转到第二阶段的引导加载器或内核的开始位置。

2. **第二阶段**：这一阶段中Bootloader提供了加载内核的代码，配置视频模式，并再次进入保护模式，为第三阶段或操作系统内核的启动做准备。以下是这一阶段所完成的工作的简要总结：

a. 显示开始消息:
   - 显示 “Booting (second stage)...” 以表示第二阶段已开始执行。

b. 设置目标操作模式:
   - 将处理器从传统模式切换为“只有长模式目标”。这允许固件启用特定于模式的优化，如预期每个CPU只进入长模式并保持在其中。

c. 从磁盘加载内核:
   - 定义了一个读取和复制内核的过程，该过程：
     i. 定义了从磁盘读取内核的位置和目的位置。
     ii. 使用`INT 13h`功能`42h`从磁盘加载内核。
     iii. 如果加载失败，会显示错误信息。
     iv. 将内核复制到2MiB（`0x2000000`）的地址。

     +读取并复制下一个内核块，直到所有块都被加载。

d. 创建内存映射:
   - 使用`e820`调用来获取系统的内存布局，并保存它在内存映射中。

e. 视频模式配置:
   - 配置视频模式以准备显示操作系统的图形用户界面或命令行界面。

f. 再次进入保护模式:
   - 关闭所有中断。
   - 重新加载全局描述表 (GDT)。
   - 设置`CR0`寄存器的保护模式位。
   - 最后，跳转到第三阶段的代码。

3. **第三阶段**：这一阶段中Bootloader准备让CPU进入64位长模式并开始执行操作系统的入口代码，这需要检查CPU的支持、设置适当的页表和启用分页。以下是这一阶段所完成的工作的简要总结：
   a. 设置数据段：
      - 初始化数据段、额外段和堆栈段为0x10。
   b. 输出启动信息：
      - 使用`vga_println`输出第三阶段的启动信息 “Booting (third stage)...” 以表示第三阶段已开始执行。
   c. 检查CPU：
      - 使用`check_cpuid`来检查CPU是否支持CPUID指令。如果不支持，它将显示错误信息并停止。
      - 使用`check_long_mode`来检查CPU是否支持长模式（64位模式）。如果不支持，它会显示错误信息并停止。
   d. 准备进入长模式：
      - 使用`set_up_page_tables`设置页表，该页表包括标识映射的引导加载程序、递归映射P4页表和将内核映射到4MB。
      - 为VGA设置映射。
   e. 启用分页：
      - 执行缓存回写并添加内存围栏以确保数据的一致性。
      - 加载`P4`表到`cr3`寄存器。
      - 在`cr4`寄存器中启用PAE（物理地址扩展）标志。
      - 设置`EFER MSR`中的长模式位。
      - 在`cr0`寄存器中启用分页。
   f. 加载64位GDT：
      - 使用`lgdt`指令加载定义的64位全局描述符表（GDT）。
   g. 跳转到长模式：
      - 将代码段设置为64位模式，并跳转到操作系统代码入口点。

操作系统的入口点由Bootloader提供的entry_point!宏来定义。这个宏本身并不复杂，它完成的工作主要是用一个新的函数来包裹用户定义的操作系统入口函数，以确保用户提供的操作系统入口函数签名正确，并且可以顺利被引导程序找到。

如读者对以上过程的具体代码实现感兴趣，可以在https://github.com/rust-osdev/bootloader/releases/tag/v0.9.23找到源代码。

4.3. 入口函数与初始化

4.3.1. 操作系统入口函数

`kernel_main`是操作系统内核的主入口函数，该函数仅需寥寥数行代码就能完成工作。它的工作流程如下：

1. **启动消息**：通过 `println!` 宏输出 "Launching Cinea's OS...", 表示操作系统的启动。
2. **初始化**：`cinea_os::init(boot_info);` 调用操作系统的初始化函数并传递引导信息。这一步通常会设置硬件、内存管理和其他关键子系统。
3. **输出信息**：通过两次 `println!` 宏输出操作系统的版本信息和作者联系方式。接着，还输出了系统的启动时间。
4. **加载子进程**：
   - 将外部的 `shell` 程序加载为一个字节数组。
   - 通过一个循环确保子进程只启动一次。它利用了一个 `flag` 标志来确保不会多次无限循环地启动 shell。
   - 引发0x81号软件中断，以保存当前进程的上下文状态。
   - 启动 `shell` 子进程。
5. **创建协程执行器**：内核进程将继续运行。它首先创建一个新的协程执行器 `executor`。
6. **添加任务**：
   - 两个协程被添加到执行器：`key_presses_handler()` 和 `mouse_handler()`，这些函数是键盘和鼠标的处理函数。
   - 这些协程将在 `executor.run();` 中被执行器并发地运行。

以下是入口函数的代码实现：

```
fn kernel_main(boot_info: &'static BootInfo) -> ! {
    println!("Launching Cinea's OS...\n");

    cinea_os::init(boot_info);

    println!("Cinea OS v1.0-dev by Cinea (Zhang Yao)
cineazhan@icloud.com");
    println!("System Uptime: {:.5} s\n", syskrnl::time::uptime());

    let subp = include_bytes!("../dsk/bin/shell");
    let args: Vec<&str> = vec![];
    let mut flag = 0;
    loop {
        unsafe { int!(0x81) };
        debugln!("I come back with flag=={}", flag);
        #[allow(unused)]
```

```
        if flag > 0 { break; } else { flag = 1; }
        syskrnl::proc::Process::spawn(subp, args.as_ptr() as usize, 0,
0).unwrap();
    }

    // 0号进程继续它的工作
    let mut executor = Executor::new();
    executor.spawn(Task::new(key_presses_handler()));
    executor.spawn(Task::new(mouse_handler()));
    executor.run();
}
```

4.3.2. 初始化

初始化通过函数`init`完成，它按照特定的顺序配置和启动了不同的子系统，使得操作系统可以正常运行和响应用户输入。以下是这个函数的详细工作流程：

1. **加载GDT（全局描述符表）**：`syskrnl::gdt::init();` 为处理器初始化了全局描述符表。GDT通常用于定义代码、数据和堆栈段的内存界限和权限。
2. **初始化中断和异常处理**：
   - `syskrnl::interrupts::init_idt();` 初始化了中断描述符表（IDT），这是处理硬件中断和软件异常的关键组件。
   - `syskrnl::interrupts::pics::PICS.lock().initialize();` 初始化了可编程中断控制器（PICs）。PIC是用来管理外部中断源的硬件设备。
   - `x86_64::instructions::interrupts::enable();` 启用了中断，从这里开始，处理器可以响应中断请求。
3. **初始化内存**：`syskrnl::memory::init(bootinfo);` 初始化内存管理子系统。它可能涉及设置页表、堆等，并基于传入的 `bootinfo` 配置内存布局。
4. **启用I/O设备**：以下一系列的初始化函数负责设置不同的I/O设备。
   - `syskrnl::io::ahci::init();` 初始化AHCI，这是一个硬盘/SSD通信协议。
   - `syskrnl::time::init();` 设置时间和时钟子系统。
   - `syskrnl::task::keyboard::init();` 初始化键盘设备，准备好接受键盘输入。
   - `syskrnl::io::mouse::init();` 初始化鼠标设备，准备好接受鼠标输入。
5. **初始化图形用户界面**：`syskrnl::gui::init();` 启动图形用户界面子系统。

以下是这个函数的具体实现：

```
pub fn init(bootinfo: &'static BootInfo) {
    // 加载GDT
    syskrnl::gdt::init();
```

```
    // 加载中断和异常处理
    syskrnl::interrupts::init_idt();
    unsafe { syskrnl::interrupts::pics::PICS.lock().initialize() };
    x86_64::instructions::interrupts::enable();

    syskrnl::vga_buffer::print_something();

    // 加载内存
    println!("\n\nInitializing the memory...\n");
    syskrnl::memory::init(bootinfo);

    // 启用各类IO设备
    syskrnl::io::ahci::init();
    syskrnl::time::init();
    syskrnl::task::keyboard::init();
    syskrnl::io::mouse::init();

    // 最后，再启动gui
    syskrnl::gui::init();
}
```

4.4. VGA文本模式

本项目的操作系统在完成初始化前，运行在VGA文本模式上。因此，本项目实现了一系列在VGA文本模式下输出文字的相关工具。

4.4.1. 相关数据结构定义

4.4.1.1. Color

VGA文本模式提供了16种颜色，这个枚举描述了这16种VGA颜色。

```
/// VGA标准颜色
#[allow(dead_code)]
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
#[repr(u8)]
pub enum Color {
    Black = 0,
    Blue = 1,
    Green = 2,
    Cyan = 3,
    Red = 4,
    Magenta = 5,
    Brown = 6,
```

```
    LightGray = 7,
    DarkGray = 8,
    LightBlue = 9,
    LightGreen = 10,
    LightCyan = 11,
    LightRed = 12,
    Pink = 13,
    Yellow = 14,
    White = 15,
}
```

4.4.1.2. ColorCode

VGA文本模式下的颜色代码，由前景色和背景色组成一个8位无符号整数。

```
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
#[repr(transparent)]
struct ColorCode(u8);

impl ColorCode {
    fn new(foreground: Color, background: Color) -> ColorCode {
        ColorCode((background as u8) << 4 | (foreground as u8))
    }
}
```

4.4.1.3. ScreenChar

这个结构体代表在VGA文本模式下的单个字符。

```
/// 提交到内存中的VGA字符
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
#[repr(C)]
struct ScreenChar {
    ascii_character: u8,
    color_code: ColorCode,
}
```

4.4.1.4. Buffer

这个结构体代表整个VGA文本模式屏幕。

```
/// VGA屏幕
#[repr(transparent)]
struct Buffer {
    chars: [[Volatile<ScreenChar>; BUFFER_WIDTH]; BUFFER_HEIGHT],
}
```

4.4.1.5. Writer

这个结构体提供了在VGA文本模式屏幕上写字符的功能。

```
/// 输出器
pub struct Writer {
    row_position: usize,
    column_position: usize,
    color_code: ColorCode,
    buffer: &'static mut Buffer,
}
```

4.4.2. 方法实现

4.4.2.1. 向屏幕写入一个字节

`Writer`结构体提供了一个`write_byte`方法，允许我们向屏幕写入一个字节。以下是该方法的工作流程：

- `write_byte`方法接受一个`u8`类型的参数`byte`，它是我们想要写入屏幕的ASCII字符。
- 使用`match`语句对传入的`byte`进行模式匹配，来决定如何处理这个字符：
  - `0x08`（ASCII码为`\b`的退格符）：调用`self.backspace()`方法，用于删除上一个字符。
  - `b'\t'`（ASCII码为制表符）：调用`self.horizontal_tab()`方法，进行水平制表位移。
  - `b'\n'`（ASCII码为换行符）：调用`self.new_line()`方法，进行换行操作。
  - `b'\r'`（ASCII码为回车符）：调用`self.carriage_return()`方法，将光标移至当前行的开头。
  - 对于所有其他字符，执行以下步骤：
    1. 检查当前列的位置是否超过`BUFFER_WIDTH`。如果是，调用`new_line()`方法并移到下一行。
    2. 获取当前的行和列位置，以及当前的颜色代码。
    3. 使用`write`方法将字符写入屏幕的当前位置。该方法是在`Volatile`封装类型上定义的，确保字符确实被写入了屏幕缓冲区，并且编译器不会优化掉这个写入操作。
    4. 增加列的位置。
    5. 调用`move_cursor()`方法，更新屏幕上光标的位置。

这个`write_byte`方法为`Writer`提供了一个基本的字符输出功能，可以处理常见的控制字符，并确保文本在屏幕上正确地呈现。以下是其代码实现：

```
pub fn write_byte(&mut self, byte: u8) {
    match byte {
        0x08 => self.backspace(), // \b
```

```
        b'\t' => self.horizontal_tab(),
        b'\n' => self.new_line(),
        b'\r' => self.carriage_return(),
        byte => {
            if self.column_position >= BUFFER_WIDTH {
                self.new_line();
            }

            let row = self.row_position.clone();
            let col = self.column_position.clone();
            let color_code = self.color_code.clone();

            self.buffer.chars[row][col].write(ScreenChar {
                ascii_character: byte,
                color_code,
            });

            self.column_position += 1;
            self.move_cursor();
        }
    }
}
```

4.4.2.2. 向屏幕写入一个字符串

`Writer`结构体提供了一个`write_byte`方法，它接收一个字符串切片 `s`，并将其写入屏幕。以下是其工作流程：

- 使用`for`循环遍历字符串的每一个字节。
- 对于每一个字节，使用`match`语句检查它是否是可打印的ASCII字符或某些特定的控制字符（如换行符、回车符、制表符和退格符）。如果是，直接调用`write_byte`方法将其写入屏幕。对于其他不匹配的字节，将其写入为`0xfe`（这表示它是一个未知字符）。

以下是其代码实现：

```
pub fn write_string(&mut self, s: &str) {
    for byte in s.bytes() {
        match byte {
            0x20..=0x7e | b'\n' | b'\r' | b'\t' | 0x08 =>
self.write_byte(byte),
            _ => self.write_byte(0xfe),
        }
    }
}
```

4.4.2.3. 清除一整行

Writer结构体提供了一个write_byte方法，它接受一个行号row，并清除该行上的所有字符，即将该行的所有字符设置为空格字符。以下是其工作流程：

- 使用for循环遍历该行的所有列，并将每个位置上的字符设置为一个空格字符（ASCII值为b' '），颜色代码为当前的颜色代码。

以下是其代码实现：

```
fn clear_row(&mut self, row: usize) {
    let blank = ScreenChar {
        ascii_character: b' ',
        color_code: self.color_code,
    };
    for col in 0..BUFFER_WIDTH {
        self.buffer.chars[row.clone()][col].write(blank);
    }
}
```

4.4.2.4. 换行

Writer结构体提供了一个new_line方法，该方法用于实现换行功能。以下是其工作流程：

- 首先，将row_position增加1，表示进入下一行，然后将column_position设置为0，表示从新行的开始位置开始写入。
- 接下来，检查row_position是否超出了屏幕的高度（即是否超过了BUFFER_HEIGHT）。如果超出，需要执行滚屏操作：
  - 使用两个for循环遍历屏幕的每一个位置，将每一个字符上移一行。这实现了屏幕内容的上滚功能。
  - 清除最后一行的内容，因为它现在已经变成了新的空行。
  - 将row_position设置为BUFFER_HEIGHT - 1，确保我们在屏幕的最后一行上写入内容。

以下是其代码实现：

```
fn new_line(&mut self) {
    self.row_position += 1;
    self.column_position = 0;

    if self.row_position >= BUFFER_HEIGHT {
        // 向上滚屏
        for row in 0..BUFFER_HEIGHT - 1 {
            for col in 0..BUFFER_WIDTH {
                self.buffer.chars[row.clone()]
[col.clone()].write(self.buffer.chars[row.clone() + 1]
```

```
[col.clone()].read());
            }
        }
        self.clear_row(BUFFER_HEIGHT - 1);
        self.row_position = BUFFER_HEIGHT - 1;
    }
}
```

4.5. GDT 和 TSS

4.5.1. 常量定义

```
const STACK_SIZE: usize = 1024 * 8;
pub const DOUBLE_FAULT_IST_INDEX: u16 = 0;
pub const PAGE_FAULT_IST_INDEX: u16 = 1;
pub const GENERAL_PROTECTION_FAULT_IST_INDEX: u16 = 2;
```

- `STACK_SIZE`定义了每个中断或特权栈的大小。
- 中断向量表中的索引:
  - `DOUBLE_FAULT_IST_INDEX`
  - `PAGE_FAULT_IST_INDEX`
  - `GENERAL_PROTECTION_FAULT_IST_INDEX`

4.5.2. 选择子定义

`Selectors` 结构体用于存储代码、TSS和数据的段选择子以及用户态的代码和数据段选择子。

```
/// 选择子
#[allow(dead_code)]
pub struct Selectors {
    code_selector: SegmentSelector,
    tss_selector: SegmentSelector,
    data_selector: SegmentSelector,
    pub user_code_selector: SegmentSelector,
    pub user_data_selector: SegmentSelector,
}
```

4.5.3. TSS 初始化准备

TSS（Task State Segment, 任务状态段）是一种用于保存进程或线程的状态信息的数据结构，它可以帮助操作系统实现多任务的切换和管理。在项目中，我们利用TSS中的内层堆栈指针区域来实现特权级转换时的栈切换，以及发生异常时的栈切换。

在准备过程中，我们使用了`lazy_static!`宏来定义一个延迟初始化的静态`TSS`变量，并为TSS中的各个中断堆栈表项和特权堆栈分配了栈内存。为了简化开发，所有的栈都是固定大小的，并在编译时分配。

以下是其代码实现：

```
lazy_static! {
  static ref TSS: TaskStateSegment = {
      let mut tss = TaskStateSegment::new();
      tss.privilege_stack_table[0] = {
          static mut STACK: [u8; STACK_SIZE] = [0; STACK_SIZE];
          VirtAddr::from_ptr(unsafe { &STACK }) + STACK_SIZE
      };
      tss.interrupt_stack_table[DOUBLE_FAULT_IST_INDEX as usize] = {
          static mut STACK: [u8; STACK_SIZE] = [0; STACK_SIZE];
          VirtAddr::from_ptr(unsafe { &STACK }) + STACK_SIZE
      };
      tss.interrupt_stack_table[PAGE_FAULT_IST_INDEX as usize] = {
          static mut STACK: [u8; STACK_SIZE] = [0; STACK_SIZE];
          VirtAddr::from_ptr(unsafe { &STACK }) + STACK_SIZE
      };
      tss.interrupt_stack_table[GENERAL_PROTECTION_FAULT_IST_INDEX as
usize] = {
          static mut STACK: [u8; STACK_SIZE] = [0; STACK_SIZE];
          VirtAddr::from_ptr(unsafe { &STACK }) + STACK_SIZE
      };
      tss
  };
}
```

4.5.4. GDT 初始化准备

GDT是一种用于定义段的属性和位置的数据结构，它可以帮助操作系统实现保护模式下的内存管理和任务切换。GDT中的每个元素叫做段描述符，它包含了段的基址、界限、类型、特权级、粒度等信息。GDT的位置和大小由一个特殊的寄存器GDTR来保存，操作系统可以通过LGDT指令来加载GDT的地址到GDTR中。CPU在访问内存时，会根据GDTR中的内容来找到GDT，并根据段寄存器中的段选择子来定位到相应的段描述符，然后根据段描述符中的信息来计算出物理地址和访问权限。

在准备过程中，我们使用了`lazy_static!`宏来定义一个延迟初始化的静态`GDT`变量，并为内核和用户态代码、数据段以及TSS分配了描述符并添加到GDT中。

以下是其代码实现：

```
lazy_static! {
  pub static ref GDT: (GlobalDescriptorTable, Selectors) = {
      let mut gdt = GlobalDescriptorTable::new();
      let tss_selector = gdt.add_entry(Descriptor::tss_segment(&TSS));
      let code_selector = gdt.add_entry(Descriptor::kernel_code_segment());
      let data_selector = gdt.add_entry(Descriptor::kernel_data_segment());
      let user_code_selector =
gdt.add_entry(Descriptor::user_code_segment());
      let user_data_selector =
gdt.add_entry(Descriptor::user_data_segment());
      (
          gdt,
          Selectors {
              code_selector,
              tss_selector,
              data_selector,
              user_code_selector,
              user_data_selector,
          },
      )
  };
}
```

4.5.5. 完成 GDT 和 TSS 初始化

在这一阶段中，我们主要将GDT加载到内存，并使用x86_64指令设置代码段（CS）和数据段（DS）的选择器，并加载TSS。

以下是其代码实现：

```
pub fn init() {
    use x86_64::instructions::segmentation::CS;
    use x86_64::instructions::tables::load_tss;
    GDT.0.load();
    unsafe {
        CS::set_reg(GDT.1.code_selector);
        DS::set_reg(GDT.1.data_selector);
        load_tss(GDT.1.tss_selector);
    }
}
```

4.6. 中断

中断是操作系统开发中的一个重要概念，它指的是当 CPU 在执行程序的过程中，遇到了一些需要紧急处理的事件，就暂时中止当前的程序，转而去执行相应的事件处理程序，

待处理完成后再返回原来的程序继续执行。中断可以提高 CPU 的利用率，实现多任务的切换和管理，以及响应外部设备的请求。

中断可以分为两种：硬中断和软中断。硬中断是由外部设备（如键盘、鼠标、磁盘、网络等）产生的异步信号，通常需要操作系统在硬件层面进行响应和处理。软中断是由当前运行的程序产生的同步信号，通常是通过一条特殊的指令（如 int 或 syscall）来触发，用于请求操作系统提供一些服务（如文件操作、进程管理、内存分配等）。

为了处理中断，我们需要做一些准备工作，主要包括以下几个方面：

- 设计一个合理的中断向量表（Interrupt Vector Table），用于存放不同类型的中断对应的处理程序的入口地址。中断向量表通常位于内存的固定位置，方便 CPU 快速查找。
- 编写一个高效的中断服务程序（Interrupt Service Routine），用于执行具体的事件处理逻辑。中断服务程序应该尽可能地简短和快速，以减少对正常程序运行的影响。
- 保护好当前程序的现场（Context），即 CPU 的寄存器和栈等状态信息。在发生中断时，我们需要将这些信息保存到内存中，以便在返回原程序时恢复。在恢复现场时，我们也需要注意清除一些标志位或寄存器，以避免重复或错误地响应同一个中断。
- 管理好中断的优先级和屏蔽。由于可能同时发生多个中断，我们需要根据不同中断的重要性和紧急性来确定哪个中断先处理，哪个后处理。我们也可以通过设置一些屏蔽位来暂时屏蔽掉一些不想处理或者不必要处理的中断。

4.6.1. 管理PIC

可编程中断控制器（PIC）是一种用于管理和分发来自外部设备的中断请求（IRQ）的芯片，它可以帮助CPU实现多任务的切换和响应。在x86架构上，最常见的PIC是8259A，它是Intel 8259的改进版本，于1976年推出，专为Intel 8085和Intel 8086微处理器而设计。后来的x86处理器，如8088、80186、80286等，也可以使用8259A作为中断控制器。

8259A PIC有两种工作模式：级联模式和单片模式。在级联模式下，可以使用两片8259A PIC来扩展中断线的数量，一片作为主PIC，另一片作为从PIC，主PIC的IRQ2引脚连接到从PIC的INT引脚，从而实现两片PIC之间的通信。这样，主PIC可以处理IRQ0-IRQ7共8个中断线，从PIC可以处理IRQ8-IRQ15共8个中断线，总共可以管理16个中断线。在单片模式下，只使用一片8259A PIC来处理IRQ0-IRQ7共8个中断线。我们默认本项目中使用到的PIC均为级联模式。

4.6.1.1. 常量定义

```
pub const PIC_1_OFFSET: u8 = 32;
pub const PIC_2_OFFSET: u8 = PIC_1_OFFSET + 8;
```

- `PIC_1_OFFSET`和`PIC_2_OFFSET`: 这两个常量定义了两个PIC的中断向量偏移。这些偏移是为了避免与CPU的异常中断向量重叠。常规的x86架构设置主PIC的偏移为32，从PIC的偏移为40（即32 + 8）。

4.6.1.2. PIC接口创建

使用汇编代码和PIC通信是很复杂的，幸好Rust社区提供了`pic8259`包来为8259A PIC控制器提供接口封装。`pic8259::ChianedPics`是级联模式下的PIC的控制接口，我们只需要创建这样的一个接口，就可以便捷地实现对PIC的控制。

创建这样的一个控制接口只需要传入主PIC和从PIC的中断向量偏移即可：

```
pub static PICS: spin::Mutex<ChainedPics> = spin::Mutex::new(unsafe
{ ChainedPics::new(PIC_1_OFFSET, PIC_2_OFFSET) });
```

创建好接口后，我们先不进行初始化操作，我们将在配置好中断向量表后再进行初始化。

4.6.2. 准备中断处理函数

中断处理函数是一种用于响应和处理中断事件的特殊函数，它通常由硬件或软件触发，需要在尽可能短的时间内完成。中断处理函数的作用是保存当前程序的状态，执行相应的事件处理逻辑，然后恢复原来程序的继续执行。中断处理函数的编写需要遵循一些原则和注意事项，例如不能嵌套、不能有返回值和形参、不能调用阻塞或延时的函数等。

处理普通中断和异常的函数具有类似这样的签名：

```
extern "x86-interrupt" fn breakpoint_handler(stack_frame:
InterruptStackFrame) {}
```

其中，`stack_frame`是中断栈帧，存储了中断发生时程序运行的上下文信息。我们可以通过将栈帧保存、替换，实现上下文保存和切换的效果。

4.6.2.1. 异常处理函数

这些中断处理函数用于处理特定的CPU异常。这些异常是由于CPU在执行过程中遇到了错误或者特定的情况而产生的。

当CPU遇到特定的异常时，对应的异常处理函数会被调用；当处理一个异常时又发生另一个异常，双重异常处理函数就会被调用。在大多数异常处理函数中，我们会通过`println!`和`qemu_print`打印相关的信息。`println!`打印错误信息到屏幕，而`qemu_print`将消息发送到QEMU，进而在宿主机控制台上输出错误信息。

以下是部分异常处理函数的实现:

```
/// 调试异常处理函数
extern "x86-interrupt" fn breakpoint_handler(_stack_frame:
InterruptStackFrame) {
    println!("EXCEPTION: BREAKPOINT\n{:#?}", _stack_frame);
    qemu_print(format!("EXCEPTION: BREAKPOINT\n{:#?}\n",
_stack_frame).as_str());
}

/// 双重异常处理函数
extern "x86-interrupt" fn double_fault_handler(_stack_frame:
InterruptStackFrame, _error_code: u64) -> ! {
    println!("EXCEPTION: DOUBLE FAULT\n{:#?}", _stack_frame);
    qemu_print(format!("EXCEPTION: DOUBLE FAULT\n{:#?}\n",
_stack_frame).as_str());
    loop {}
}

/// 一般保护异常处理函数
extern "x86-interrupt" fn general_protection_fault_handler(stack_frame:
InterruptStackFrame, error_code: u64) {
    debugln!(
        "EXCEPTION: GENERAL PROTECTION FAULT\nStack Frame: {:#?}\nError:
{:?}\n",
        stack_frame,
        error_code
    );

    let panic_desc = format!("Stack Frame: {:#?}\nError: {:?}\n",
stack_frame, error_code);
    let panic_info = panic::PanicInfo::new("一般保护异常 General Protection",
panic_desc.as_str());

    panic::handle_panic(&panic_info);
}

/// 页错异常处理函数
extern "x86-interrupt" fn page_fault_handler(stack_frame:
InterruptStackFrame, _error_code: PageFaultErrorCode) {
    use x86_64::registers::control::Cr2;

    qemu_print(format!("EXCEPTION: PAGE FAULT\n").as_str());
    qemu_print(format!("Accessed Address: {:?}\n", Cr2::read()).as_str());
    qemu_print(format!("{:#?}\n", stack_frame).as_str());

    let panic_desc = format!("Accessed Address: {:?}\n{:#?}\n", Cr2:
```

```
:read(), stack_frame);
    let panic_info = panic::PanicInfo::new("页错异常 Page Fault",
panic_desc.as_str());

    panic::handle_panic(&panic_info);
}
```

4.6.2.2. 内核异常界面

在Windows中，有被称为“蓝屏”的错误界面。本着用户友好的原则，本项目也设计和制作了“蓝屏”界面，当页错异常、一般保护异常触发时，会进入到“蓝屏”界面中。

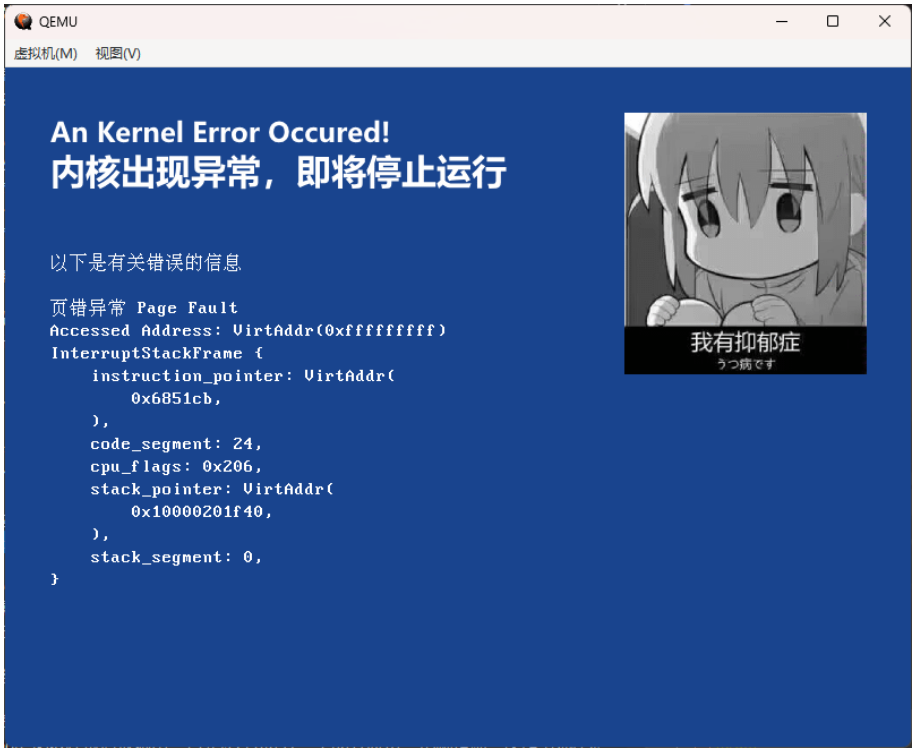

图 2: 内核异常界面示意图

内核异常界面定义了一个简单的结构体：**PanicInfo**。`PanicInfo`是一个简单的结构，用于存储有关恐慌情况的信息。它包含两个字段：`title`（标题）和`description`（描述）。它提供了`new`方法来创建一个新的`PanicInfo`实例，并且还有两个getter方法：`get_title`和`get_description`，用于获取标题和描述。

```
pub struct PanicInfo<'a> {
    title: &'a str,
    description: &'a str,
}

impl<'a> PanicInfo<'a> {
```

```
    pub fn new(title: &'a str, description: &'a str) -> PanicInfo<'a> {
        PanicInfo { title, description }
    }

    pub fn get_title(&self) -> &str {
        self.title
    }

    pub fn get_description(&self) -> &str {
        self.description
    }
}
```

此外，内核异常界面定义了一个函数：**handle_panic**。这是当系统遇到一个不能恢复的错误时调用的函数。它接受一个`PanicInfo`作为参数。它的工作流程主要如下所示：

- 函数首先通过操作`GL`锁来更改屏幕上的图层可见性[3]。它开启了前两层，并关闭了其它所有图层。
- 接下来，它显示一个蓝屏图片。这张图片位于`assets/BlueScreen_small.bmp`目录下。
- 在这个蓝屏上，使用`TextWriter`来写入关于错误的信息。
- 最后，使用`GD.lock().render`方法来渲染错误信息到屏幕上。
- `hlt_loop`函数在最后被调用，它会让CPU进入halt状态。也就是说，操作系统在显示完错误信息后，将停止所有操作。

代码实现如下所示：

```
pub fn handle_panic(info: &PanicInfo) -> ! {
    let p_lock = GL.read();
    for layer in 0..2 {
        p_lock[layer].lock().enable = true;
    }
    for layer in 2..p_lock.len() {
        p_lock[layer].lock().enable = false;
    }

    let mut lock = p_lock[0].lock();
    lock.display_img(0, 0, include_bytes!("../../../assets/
BlueScreen_small.bmp"));
    drop(lock);
    let p_lock = GL.read();
    let mut lock = p_lock[1].lock();
    for r in lock.data.iter_mut() {
```

[3]有关`GL`锁和图层可见性的含义，请参见“图形模式显示”一节。下略。

```
        for p in r.iter_mut() {
            p.0 = rgb888!(0);
            p.1 = false;
        }
    }
    drop(lock);

    let mut writer = TextWriter {
        text_area_height: 570,
        text_area_width: 700,
        text_area_pos: (22, 40),
        y_position: 0,
        line_position: 7,
        line_height: 16,
        line_gap: 4,
        max_line: 28,
        color: rgb888!(0xffffffu32),
        layer: 1,
    };

    writer.write_string("以下是有关错误的信息");
    writer.write_string("\n\n");
    writer.write_string(info.get_title());
    writer.write_string("\n");
    writer.write_string(info.get_description());

    GD.lock().render(0, 0, 600, 800);

    hlt_loop();
}
```

4.6.2.3. PIC中断处理函数

PIC中断可以分为两种：硬中断和软中断。硬中断是由外部设备（如键盘、鼠标、磁盘、网络等）产生的异步信号，通常需要操作系统在硬件层面进行响应和处理。软中断是由当前运行的程序产生的同步信号，通常是通过一条特殊的指令（如 int 或 syscall）来触发，用于请求操作系统提供一些服务（如文件操作、进程管理、内存分配等）。

硬中断的中断号码是固定且确定的：

```
                    ____________                          ____________
Real Time Clock --> |            |   Timer -------------> |            |
ACPI -------------> |            |   Keyboard-----------> |            |      _____
Available --------> | Secondary  |----------------------> | Primary    |     |     |
Available --------> | Interrupt  |   Serial Port 2 -----> | Interrupt  |---> | CPU |
Mouse ------------> | Controller |   Serial Port 1 -----> | Controller |     |_____|
```

```
Co-Processor -----> |            |   Parallel Port 2/3 -> |            |
Primary ATA ------> |            |   Floppy disk -------> |            |
Secondary ATA ----> |____________|   Parallel Port 1----> |____________|
```

我们可以直接硬编码16个硬中断到中断向量表中。由于PIC中断产生和应答的格式都相对统一，我们可以用一个叫做`irq_handler!`的宏来将中断处理函数抽象起来：

```
macro_rules! irq_handler {
    ($handler:ident, $irq:expr) => {
        pub extern "x86-interrupt" fn $handler(_stack_frame:
InterruptStackFrame) {
            let handlers = IRQ_HANDLERS.lock();
            handlers[$irq]();
            unsafe {

pics::PICS.lock().notify_end_of_interrupt(interrupt_index($irq));
            }
        }
    };
}
```

这个宏允许我们快速生成PIC中断请求的处理函数。以下是这个宏的工作流程：

1. **定义一个中断处理函数**:
   - 使用`pub extern "x86-interrupt"`来表示这是一个中断处理函数。
   - 函数的名称由`$handler`宏参数给出。
   - 这个函数接受一个`InterruptStackFrame`参数，这是x86架构中中断发生时保存的寄存器的状态。
2. **锁定`IRQ_HANDLERS`并调用处理函数**:
   - 使用`let handlers = IRQ_HANDLERS.lock();`来获得`IRQ_HANDLERS`的锁，确保在此时间内只有一个处理函数能够访问这个数组。
   - 使用`handlers[$irq]();`来调用特定的中断处理函数。这是一个函数指针数组，`$irq`指定了哪个函数应该被调用。
3. **通知PIC（可编程中断控制器）**：
   - 使用`pics::PICS.lock().notify_end_of_interrupt(interrupt_index($irq));`来通知PIC中断已经被处理完毕。这是必要的，因为PIC需要知道何时可以开始发送新的中断请求。

所有PIC中断处理函数存储于一个数组中：

```
pub static ref IRQ_HANDLERS: Mutex<[fn(); 16]> =
Mutex::new([default_irq_handler; 16]);
```

其中，default_irq_handler的定义如下：

```
/// 默认IRQ处理器
fn default_irq_handler() {}
```

我们还定义了一个便捷的函数interrupt_index，用于将IRQ的序号转换为中断号码：

```
/// 将IRQ转换为中断号码
fn interrupt_index(irq: u8) -> u8 {
    pics::PIC_1_OFFSET + irq
}
```

以下是使用上述宏和函数快速生成硬PIC中断的处理函数：

```
irq_handler!(irq0_handler, 0);
irq_handler!(irq1_handler, 1);
irq_handler!(irq2_handler, 2);
irq_handler!(irq3_handler, 3);
irq_handler!(irq4_handler, 4);
irq_handler!(irq5_handler, 5);
irq_handler!(irq6_handler, 6);
irq_handler!(irq7_handler, 7);
irq_handler!(irq8_handler, 8);
irq_handler!(irq9_handler, 9);
irq_handler!(irq10_handler, 10);
irq_handler!(irq11_handler, 11);
irq_handler!(irq12_handler, 12);
irq_handler!(irq13_handler, 13);
irq_handler!(irq14_handler, 14);
irq_handler!(irq15_handler, 15);
```

这样，外部模块只需要更改IRQ_HANDLERS数组中的值，就可以修改该中断对应的处理函数了。

4.6.2.4. 软件中断处理函数

软件中断是一种由软件程序触发的中断，它通常通过系统调用或软中断指令来向CPU发起中断请求。软件中断的目的是向操作系统请求一些服务或功能，例如文件操作、进程管理、内存分配等。软件中断与硬件中断的区别是，软件中断不是由外部设备产生的，而是由软件程序主动发起的，一般用于系统调用、进程切换、异常处理等任务。软件中断需要在程序中进行调用，其响应速度和实时性相对较差，但是具有灵活性和可控性高的特点。

软件中断的实现方式有两种：一种是使用汇编指令int，另一种是使用软件中断指令。使用汇编指令int的方式是传统的系统调用实现方式，在x86架构的CPU中，为了实现用户态

到内核态的切换，Linux使用软件中断指令“`int 0x80`”来触发异常，切换CPU特权级，实现系统调用。使用软件中断指令的方式是现代的系统调用实现方式，在x86_64架构的CPU中，为了提高系统调用的效率和兼容性，Linux使用软件中断指令“`syscall`”来触发异常，切换CPU特权级，实现系统调用。

在本项目中，我们采用中断指令“`int 0x80`”来触发软件中断。此外，我们也使用了`0x81`和`0x82`来处理部分特殊任务。

外部程序触发软件中断时，需要携带一些参数以说明软件中断的目的。为了实现这些额外参数的携带和传递，以及对程序运行上下文的保护，我们需要使用汇编语言对普通的中断处理函数进行包装：

```
macro_rules! wrap {
    ($fn: ident => $w:ident) => {
        #[naked]
        pub unsafe extern "sysv64" fn $w() {
            asm!(
                "push rax",
                "push rcx",
                "push rdx",
                "push rbx",
                "push rbp",
                "push rsi",
                "push rdi",
                "push r8",
                "push r9",
                "push r10",
                "push r11",
                "push r12",
                "push r13",
                "push r14",
                "push r15",
                "mov rsi, rsp", // Arg #2: register list
                "mov rdi, rsp", // Arg #1: interupt frame
                "add rdi, 15 * 8",
                "call {}",
                "pop r15",
                "pop r14",
                "pop r13",
                "pop r12",
                "pop r11",
                "pop r10",
                "pop r9",
                "pop r8",
```

```
                "pop rdi",
                "pop rsi",
                "pop rbp",
                "pop rbx",
                "pop rdx",
                "pop rcx",
                "pop rax",
                "iretq",
                sym $fn,
                options(noreturn)
            );
        }
    };
}

wrap!(syscall_handler => wrapped_syscall_handler);
wrap!(clock_handler => wrapped_clock_handler);
wrap!(save_context => wrapped_save_context);
wrap!(proc_wait => wrapped_proc_wait);
```

这个`wrap!`宏的目的是为了生成一组汇编指令，这些指令可以保存所有的通用寄存器的状态，然后调用一个Rust函数，并在返回时恢复这些寄存器的状态。这种模式常常被用于中断处理和系统调用的处理函数，因为这些函数可能在任何时候被调用，不应该更改正在执行的任务的状态。

这个宏有两个参数：

- `$fn`: 要调用的原始Rust函数的名称。
- `$w`: 生成的"封装"函数的名称，这个函数包含汇编代码来保存和恢复寄存器的状态。

这个宏的主要工作流程如下：

1. **定义一个naked函数**:
   - `#[naked]`属性表示该函数不应生成通常的函数开头和结尾的代码，这通常包括保存和恢复调用者保存的寄存器，设置栈帧等。这就是我们通过手动汇编来做的事情。
   - 函数类型为`sysv64`，这是x86_64架构下的System V ABI。
2. **汇编指令**:
   - 使用`asm!`宏来包含汇编代码。
   - 首先，所有的通用寄存器都被推入栈中。
   - 然后，两个参数（`rdi`和`rsi`）被设置为指向保存的寄存器的位置和中断帧的位置。
   - 使用`call`指令调用原始的Rust函数。
   - 函数返回后，寄存器的状态从栈中恢复。
   - 最后，使用`iretq`指令从中断返回。

最后，我们为四个函数调用wrap!宏，分别为它们生成了封装后的软中断处理函数。

本项目中使用了三个软件中断，他们分别是：

中断号码	中断处理函数名	作用
0x80	syscall_handler	处理系统调用
0x81	save_context	立即保存进程上下文
0x82	proc_wait	让进程等待某一事件

4.6.3. 初始化中断描述符表

中断描述符表 (Interrupt Descriptor Table, IDT)，或中断向量表，是一个非常关键的部分。它用于描述如何处理各种中断。它的主要作用是当CPU收到中断时，由它指示正确的函数来处理该中断。

在准备过程中，我们使用了lazy_static!宏来定义一个延迟初始化的静态InterruptDescriptorTable对象，配置多种中断的处理函数，例如断点、双重故障、页面错误和通用保护故障。接下来，我们使用interrupt_index函数为指定的中断号设置处理函数，例如PIT定时器和多个IRQ。对可能需要转换函数指针类型的，转换函数指针类型；对只能在用户空间（Ring 3）调用的，进行特殊标记。

最后，当中断描述符表就绪后，我们使用init_idt() 函数来加载上述的中断描述符表。

以下是具体的实现代码：

```
lazy_static! {
    static ref IDT: InterruptDescriptorTable = {
        let mut idt = InterruptDescriptorTable::new();
        idt.breakpoint
            .set_handler_fn(breakpoint_handler);
        unsafe{
          idt.double_fault
              .set_handler_fn(double_fault_handler)
              .set_stack_index(syskrnl::gdt::DOUBLE_FAULT_IST_INDEX);
          idt.page_fault
              .set_handler_fn(page_fault_handler)
              .set_stack_index(syskrnl::gdt::PAGE_FAULT_IST_INDEX);
          idt.general_protection_fault
              .set_handler_fn(general_protection_fault_handler)
         .set_stack_index(syskrnl::gdt::GENERAL_PROTECTION_FAULT_IST_INDEX);
          // PIT刷新中断
```

```
            idt[interrupt_index(0) as usize]
                .set_handler_fn(core::mem::transmute(wrapped_clock_handler as
*mut fn()));
            // 系统调用接口
            idt[0x80]
                .set_handler_fn(core::mem::transmute(wrapped_syscall_handler
as *mut fn()))
                .set_privilege_level(x86_64::PrivilegeLevel::Ring3);
            // 特殊调用接口：保存进程状态
            idt[0x81]
                .set_handler_fn(core::mem::transmute(wrapped_save_context as
*mut fn()))
                .set_privilege_level(x86_64::PrivilegeLevel::Ring3);
            // 特殊调用接口：进程等待事件
            idt[0x82]
                .set_handler_fn(core::mem::transmute(wrapped_proc_wait as
*mut fn()))
                .set_privilege_level(x86_64::PrivilegeLevel::Ring3);
        }
        idt[interrupt_index(1) as usize].set_handler_fn(irq1_handler);
        idt[interrupt_index(2) as usize].set_handler_fn(irq2_handler);
        idt[interrupt_index(3) as usize].set_handler_fn(irq3_handler);
        idt[interrupt_index(4) as usize].set_handler_fn(irq4_handler);
        idt[interrupt_index(5) as usize].set_handler_fn(irq5_handler);
        idt[interrupt_index(6) as usize].set_handler_fn(irq6_handler);
        idt[interrupt_index(7) as usize].set_handler_fn(irq7_handler);
        idt[interrupt_index(8) as usize].set_handler_fn(irq8_handler);
        idt[interrupt_index(9) as usize].set_handler_fn(irq9_handler);
        idt[interrupt_index(10) as usize].set_handler_fn(irq10_handler);
        idt[interrupt_index(11) as usize].set_handler_fn(irq11_handler);
        idt[interrupt_index(12) as usize].set_handler_fn(irq12_handler);
        idt[interrupt_index(13) as usize].set_handler_fn(irq13_handler);
        idt[interrupt_index(14) as usize].set_handler_fn(irq14_handler);
        idt[interrupt_index(15) as usize].set_handler_fn(irq15_handler);
        idt
    };
}

pub fn init_idt() {
    IDT.load();
}
```

4.6.4. 启用中断

在上一步中，我们已经填写了中断描述符表并将它加载到CPU上。在这一步中，我们将初始化主从PIC，并通知CPU开始处理中断：

```
unsafe { syskrnl::interrupts::pics::PICS.lock().initialize() };
x86_64::instructions::interrupts::enable();
```

至此，我们已经完全完成了中断的初始化工作。操作系统现在可以正常接收中断，并根据中断向量表处理中断。

4.7. 内存映射

页表是一种用于实现虚拟内存和物理内存之间的映射关系的数据结构，它可以帮助操作系统实现内存的管理和保护。页表的基本思想是将虚拟内存和物理内存都划分为固定大小的单元，称为页（page）和页框（page frame）。每个页和页框都有一个唯一的编号，称为页号（page number）和页框号（page frame number）。页表中的每个条目，称为页表项（page table entry），记录了一个页号和一个页框号之间的对应关系，以及一些控制位，如有效位、修改位、访问位、保护位等。有效位表示该页表项是否有效，修改位表示该页是否被修改过，访问位表示该页是否被访问过，保护位表示该页的访问权限等。

直接管理页表同样需要大量汇编代码来完成工作，幸运的是，Rust社区同样提供了名为x86_64的包，其中就包含了对页表相关操作的Rust接口。之后，本项目将会默认使用此接口管理页表。

4.7.1. 初始化内存映射

本项目设置了一个 `memory::init` 函数来负责操作系统的内存初始化工作，并启用某些I/O设备的内存映射。以下是这个函数的工作流程：

1. **禁用中断**：使用 `interrupts::without_interrupts` 来确保内存初始化过程中不会被中断打断。这是一个常见的做法，因为在关键部分的初始化阶段，接收到中断可能导致未定义的行为。
2. **计算可用内存大小**：
   - 遍历 `bootinfo.memory_map` 中提供的内存区域。
   - 计算每个区域的大小，并累加到 `memory_size`。
   - 打印可用的内存大小（以KB为单位）。
3. **全局保存页表、物理内存偏移量**：
   - 设置全局 `PHYS_MEM_OFFSET` 为 `bootinfo` 中的物理内存偏移量。
   - 使用 `bootinfo.memory_map` 替换全局 `MEMORY_MAP`。
   - 设置全局 `MAPPER` 为新的偏移页表。

4. **初始化堆内存**：初始化堆。这是为动态内存分配准备的，如 `Box`, `Vec`, `Rc` 等。相关函数的定义在“堆内存”一节中给出。
5. **进入宽屏模式**：配置显存和屏幕分辨率。放在这里执行是因为调整显示模式后，需要动态获得显存地址并为其在页表上分配虚拟地址。相关函数的定义在“图形模式显示”一节中给出。
6. **创建AHCI内存映射**：
   - 为AHCI BAR（基地址寄存器）创建内存映射
   - 为AHCI协议创建内存映射

以下是该函数的代码实现：

```
pub fn init(bootinfo: &'static BootInfo) {
    interrupts::without_interrupts(|| {
        let mut memory_size = 0;

        for region in bootinfo.memory_map.iter() {
            let start_addr = region.range.start_addr();
            let end_addr = region.range.end_addr();
            memory_size += end_addr - start_addr;
            // println!("MEM [{:#016X}-{:#016X}] {:?}", start_addr,
end_addr - 1, region.region_type);
        }
        println!("Memory: {} KB", memory_size >> 10);
        MEMORY_SIZE.store(memory_size, Ordering::Relaxed);

        unsafe { PHYS_MEM_OFFSET = bootinfo.physical_memory_offset };
        unsafe { MEMORY_MAP.replace(&bootinfo.memory_map) };
        unsafe { MAPPER.replace(OffsetPageTable::new(active_page_table(),
VirtAddr::new(PHYS_MEM_OFFSET))) };

        let mut mapper = mapper();
        let mut frame_allocator = unsafe
{ BootInfoFrameAllocator::init(&bootinfo.memory_map) };

        syskrnl::allocator::init_heap(mapper, &mut
frame_allocator).expect("heap initialization failed");

        syskrnl::graphic::enter_wide_mode(&mut mapper, &mut
frame_allocator); // 因为需要分配显存，就放在这里了

        syskrnl::io::ahci::create_abar_memory_mapping(&mut mapper, &mut
frame_allocator);
        syskrnl::io::ahci::create_ahci_memory_mapping(&mut mapper, &mut
frame_allocator);
```

```
    });
}
```

4.7.2. 页帧分配器

页帧分配器是一种用于管理内核中的物理内存，并提供分配和释放页帧的功能的机制。页帧是物理内存的固定大小的单元，通常等于虚拟内存页的大小。页帧分配器可以帮助内核有效和灵活地使用物理内存，并支持各种内存管理技术，如分页、交换、缓存等。

在设计操作系统时，如何设计并实现一个好的页帧分配器是重中之重，因为页帧分配器的性能直接影响到所有需要分配内存的场景。在本项目中，我设计了两种页帧分配器：第一种不依赖堆内存，在系统刚刚启动后使用；第二种需要使用堆内存来缓存可用的页帧，但是速度比第一种快数千倍。

4.7.2.1. 初始页帧分配器

我定义了`BootInfoFrameAllocator`结构体来描述这个初始页帧分配器。它有一个成员`memory_map`，这是一个指向 `MemoryMap` 的静态引用，它描述了物理内存的布局和状态：

```rust
/// 帧分配器，返回BootLoader的内存映射中的可用帧
pub struct BootInfoFrameAllocator {
    memory_map: &'static MemoryMap,
}
```

它提供了一个初始化函数`init`，这个函数用于创建一个 `BootInfoFrameAllocator` 实例。由于它依赖外部提供的内存映射，所以标记为 `unsafe`。调用者必须确保传递给它的内存映射是有效的。

```rust
impl BootInfoFrameAllocator {
    /// 使用传递的内存映射创建一个帧分配器
    ///
    /// 函数不安全，因为调用者必须保证memory_map的正确性
    pub unsafe fn init(memory_map: &'static MemoryMap) -> Self {
        BootInfoFrameAllocator { memory_map }
    }
}
```

它还提供了一个方法`usable_frames`，用于返回一个包含当前可用帧的迭代器。它的工作流程如下所示：

1. 返回一个迭代器，该迭代器生成可用的物理帧。
2. 首先，它从内存映射中筛选出类型为 `Usable` 的区域。
3. 接着，它将这些区域映射为地址范围。

4. 使用 `step_by(4096)` 方法，该函数按照4KB（一个帧的大小）的步长遍历每个地址范围，这样我们得到了每个物理帧的起始地址。
5. 最后，它使用这些起始地址来创建 `PhysFrame` 实例。

```
impl BootInfoFrameAllocator {
    /// 返回一个可用帧的迭代器
    fn usable_frames(&self) -> impl Iterator<Item = PhysFrame> {
        // 获取内存中的可用区域
        let regions = self.memory_map.iter();
        let usable_regions = regions.filter(|r| r.region_type ==
MemoryRegionType::Usable);
        // 将这些区域映射到他们的地址范围内
        let addr_ranges = usable_regions.map(|r|
r.range.start_addr()..r.range.end_addr());
        // 转换为帧起始位置的迭代器
        let frame_addresses = addr_ranges.flat_map(|r| r.step_by(4096));
        // 通过帧起始位置创建PhysFrame类
        frame_addresses.map(|addr|
PhysFrame::containing_address(PhysAddr::new(addr)))
    }
}
```

最后，我们为这个类型实现特性`FrameAllocator<Size4KiB>`的接口`allocate_frame`。每次调用时，它会返回下一个可用的帧。为了确保帧的唯一性，它使用 `ALLOCATED_FRAMES`（一个原子计数器）来跟踪已分配的帧数。

```
unsafe impl static ALLOCATED_FRAMES: AtomicUsize = AtomicUsize::new(0);

FrameAllocator<Size4KiB> for BootInfoFrameAllocator {
    fn allocate_frame(&mut self) -> Option<PhysFrame<Size4KiB>> {
        let next = ALLOCATED_FRAMES.fetch_add(1, Ordering::SeqCst);

        self.usable_frames().nth(next)
    }
}
```

必须指出，这个实现的效率是非常低的，因为它必须在每次分配页帧时，都从头开始遍历一遍`self.usable_frames()`。但是，它不需要额外的堆内存，这使得它可以在操作系统刚刚启动时为我们提供页帧分配的能力，进而为后面更换其他分配器打下基础。

4.7.2.2. 具有缓存的页帧分配器

我定义了 `HeapedBootInfoFrameAllocator` 结构体来描述这个具有缓存的页帧分配器。与先前的 `BootInfoFrameAllocator` 不同，它在堆上缓存了所有可用的物理帧，这意味着其分配帧的速度更快，但初始化时的开销会更大。

它的结构体中新增了一个名为 cached_frames 的成员，这是一个 Vec<PhysFrame>。这个向量用于存储所有可用的物理帧。

```
/// 帧分配器，返回BootLoader的内存映射中的可用帧
pub struct HeapedBootInfoFrameAllocator {
    memory_map: &'static MemoryMap,
    cached_frames: Vec<PhysFrame>,
}
```

在这一版的 init 函数中，与之前的版本不同，它现在会将所有可用的物理帧加载到 cached_frames 向量中。这会消耗更多的初始化时间，但在后续的帧分配中，性能将得到改善。

```
impl HeapedBootInfoFrameAllocator {
    /// 使用传递的内存映射创建一个帧分配器
    ///
    /// 函数不安全，因为调用者必须保证memory_map的正确性
    pub unsafe fn init(memory_map: &'static MemoryMap) -> Self {
        let mut alloc = HeapedBootInfoFrameAllocator {
            memory_map,
            cached_frames: vec![],
        };

        alloc.cached_frames = alloc.usable_frames().collect();

        alloc
    }
}
```

这一版的 usable_frames 函数并无特别的改进。

在实现特性FrameAllocator<Size4KiB>的接口allocate_frame时，我们直接使用 get 方法从向量 cached_frames 中获取帧，这比之前直接查询内存映射要快得多；然后，我们使用 cloned 方法克隆找到的 PhysFrame，这样原始向量中的帧的所有权就不会被移动出去。

```
unsafe impl FrameAllocator<Size4KiB> for HeapedBootInfoFrameAllocator {
    fn allocate_frame(&mut self) -> Option<PhysFrame<Size4KiB>> {
        let next = ALLOCATED_FRAMES.fetch_add(1, Ordering::SeqCst);
        //debug!("Allocate frame {} / {}", next,
self.usable_frames().count());

        self.cached_frames.get(next).cloned()
```

```
    }
}
```

这种实现方式在初始化时会有一些开销，但在后续的物理帧分配中，由于所有的帧都已经被缓存在一个向量中，所以分配速度会更快。

4.7.3. 内存映射相关方法

以下列出的是一些和内存映射相关的函数方法的实现。

4.7.3.1. 虚拟地址转换为物理地址

在进行设备I/O时，当涉及到内存相关的操作时，设备们往往需要的是物理地址，而不是页表内的虚拟地址。例如，使用AHCI协议的`DMA READ EX`命令从硬盘读取数据时，我们需要向AHCI控制器给出缓冲区的地址。这里的地址就必须是物理地址，而不是程序使用的虚拟地址。因此，我们需要本函数来完成这一转换过程。

我们定义了一套函数来转换虚拟地址为物理地址。外层函数`translate_addr`接受一个`u64`整数为参数，并返回`u64`。之所以不使用`x86_64`包提供的虚拟地址和物理地址包装，是考虑到绝大多数时候在使用本函数时，我们往往拥有和需要的是64位的整数式地址，再使用包装类型包装一次有多此一举之嫌。

外层函数检查传入虚拟地址的合法性，并将虚拟地址和当前的物理内存偏移量（在初始化时获得）传入内层函数`translate_addr_inner`。从内层函数取得结果后，外层函数解包结果、检查成功与否，并返回给调用者。

内层函数的工作流程如下：

1. 函数首先从 `CR3` 寄存器中读取当前活动的L4页表。
2. 然后，它遍历多级页表，从L4开始，通过L3、L2直到L1，来找到对应的物理帧。
   - 它使用每个级别的索引来查找页表中的条目。
   - 页表的条目可以是一个指向下一个级别的页框或一个实际的物理帧的引用。
   - 如果在任何时候页帧不在页表中（也就是说，没有映射），则返回 `None`。
   - 如果遇到一个大页帧（HugeFrame），则抛出`panic`，因为目前无法支持这种类型的页帧。
3. 最后，函数计算并返回物理地址。这是通过将物理帧的起始地址与虚拟地址的页偏移量相加来完成的。

这两个函数的代码实现如下：

```rust
/// 将给定的虚拟地址转换为映射的物理地址，或者None（如果不存在的话）
///
/// 这个函数是危险的。调用者必须保证完整的物理地址已经被映射到虚拟地址上，
/// 且在Physical Memory Offset所申明的位置上。
pub unsafe fn translate_addr(addr: u64) -> Option<u64> {
    if let Some(res) = translate_addr_inner(VirtAddr::new(addr), unsafe
{ VirtAddr::new(PHYS_MEM_OFFSET) }) {
        Some(res.as_u64())
    } else {
        None
    }
}

/// 私有函数
///
/// 虽然没有给出unsafe，但是它是危险的。
fn translate_addr_inner(addr: VirtAddr, physical_memory_offset: VirtAddr) -
> Option<PhysAddr> {
    use x86_64::structures::paging::page_table::FrameError;

    // 从CR3读当前活跃的L4页帧
    let (level_4_table_frame, _) = Cr3::read();

    let table_indexes = [addr.p4_index(), addr.p3_index(), addr.p2_index(),
addr.p1_index()];
    let mut frame = level_4_table_frame;

    // 遍历多级页表
    for &index in &table_indexes {
        // 将帧转换为页表引用
        let virt = physical_memory_offset + frame.start_address().as_u64();
        let table_ptr: *const PageTable = virt.as_ptr();
        let table = unsafe { &*table_ptr };

        // 从页表读取位址并更新帧
        let entry = &table[index];
        frame = match entry.frame() {
            Ok(frame) => frame,
            Err(FrameError::FrameNotPresent) => return None,
            Err(FrameError::HugeFrame) => panic!("Not Supported
HugeFrame"),
        };
    }

    // 计算物理地址
    Some(frame.start_address() + u64::from(addr.page_offset()))
}
```

4.7.3.2. 页帧分配

在操作系统开发中，分配页帧是一个非常常见的任务。很多情景下，我们都需要从内存中分配页帧，例如当一个进程要创建时，操作系统需要为该文件在虚拟内存中分配一段连续的地址空间，并为每个地址对应的页分配一个空闲的页帧。这样，新的进程就可以被加载到这个页帧中，并在其上运行。

为此，我实现了一个 `alloc_pages` 函数，这个函数被用于在虚拟内存中分配连续的页面，并映射它们到物理帧上。以下是这个函数的工作流程：

1. 创建一个新的帧分配器，用于分配物理内存帧。
2. 定义新建页面的属性标志。这里我们开启了以下三个标志：`PRESENT`表示该页面在物理内存中，`WRITABLE`允许页面被写入，`USER_ACCESSIBLE`允许用户模式代码访问此页面。
3. 计算页面范围
   a. 根据给定的地址和大小，计算所需的开始页面和结束页面。
   b. 使用`Page::range_inclusive`方法来创建一个页面范围。
4. 遍历页面并进行映射
   a. 对于`pages`中的每个页面，它尝试分配一个物理帧，并将虚拟页面映射到这个物理帧上。如果分配帧或映射失败，它会记录一条错误消息并返回错误。
5. 错误处理
   - 若成功映射了所有页面，函数返回`Ok(())`，表示操作成功。
   - 若在过程中遇到任何错误，函数返回`Err(())`。

总之，此函数的目的是为给定的虚拟地址范围分配连续的页面，并确保这些页面在物理内存中并且已经被正确映射。如果在此过程中遇到任何问题，它将返回错误。以下是它的代码实现：

```
pub fn alloc_pages(mapper: &mut OffsetPageTable, addr: u64, size: usize) ->
Result<(), ()> {
    let mut frame_allocator = syskrnl::memory::heaped_frame_allocator();
    let flags = PageTableFlags::PRESENT | PageTableFlags::WRITABLE |
PageTableFlags::USER_ACCESSIBLE;
    let pages = {
        let start_page = Page::containing_address(VirtAddr::new(addr));
        let end_page = Page::containing_address(VirtAddr::new(addr + (size
as u64) - 1));
        Page::range_inclusive(start_page, end_page)
    };
    for page in pages {
        //debugln!("Alloc page {:?}", page);
        if let Some(frame) = frame_allocator.allocate_frame() {
```

```
            //debugln!("Alloc frame {:?}", frame);
            unsafe {
                if let Ok(mapping) = mapper.map_to(page, frame, flags, &mut
frame_allocator) {
                    //debugln!("Mapped {:?} to {:?}", page, frame);
                    mapping.flush();
                } else {
                    debugln!("Could not map {:?} to {:?}", page, frame);
                    return Err(());
                }
            }
        } else {
            debugln!("Could not allocate frame for {:?}", page);
            return Err(());
        }
    }
    Ok(())
}
```

4.7.3.3. 页帧释放

当一个进程终止，为它分配的内存区域不再被需要，或它的部分内存因为交换即将被移出内存，我们就需要释放已经为它映射的页帧，这样，我们就可以把宝贵的内存区域用于其他地方。

为此，我实现了一个 `dealloc_pages` 函数，它的功能与`alloc_pages`函数相反。这个函数旨在为给定的虚拟地址范围解除页面映射，并释放对应的物理帧，实现虚拟内存的回收。以下是这个函数的工作流程：

1. 获取系统内核中的memory mapper，该mapper负责管理虚拟页面和物理帧之间的映射。
2. 计算页面范围
   - 和`alloc_pages`函数类似，这部分代码计算给定的地址和大小所对应的开始页面和结束页面，并创建一个页面范围。
3. 遍历页面并进行解映射
   - 对于`pages`中的每个页面，函数使用`mapper.unmap(page)`方法尝试解除页面映射。这会移除虚拟地址到物理地址的映射，并通常也会释放对应的物理帧。
   - 若解除映射成功，它会刷新页面映射，以确保硬件内存管理单元 (MMU) 的相关缓存更新。
   - 若解映射失败，会打印一个调试信息，表示解映射失败。

总之，此函数的目的是回收虚拟地址范围中的页面，并释放相关的物理帧。通过mapper.unmap(page)方法，该操作将从页面表中移除相应的页面映射，并确保硬件更新了其页面映射缓存。以下是具体的代码实现：

```
pub fn dealloc_pages(addr: u64, size: usize) {
    let mapper = syskrnl::memory::mapper();
    let pages: PageRangeInclusive<Size4KiB> = {
        let start_page = Page::containing_address(VirtAddr::new(addr));
        let end_page = Page::containing_address(VirtAddr::new(addr + (size
as u64) - 1));
        Page::range_inclusive(start_page, end_page)
    };
    for page in pages {
        if let Ok((_frame, mapping)) = mapper.unmap(page) {
            mapping.flush();
        } else {
            //debug!("Could not unmap {:?}", page);
        }
    }
}
```

4.8. 堆内存管理

堆内存的概念是指一种用于存储程序中的动态数据的内存区域，它可以让程序员自由地分配和释放内存空间，实现对内存的灵活管理。堆内存与栈内存相对应，栈内存是用于存储程序中的静态数据的内存区域，它由系统自动分配和回收。

堆内存的大小是不固定的，可以根据程序的需要动态地增加或减少；堆内存是不连续的，由操作系统维护一个空闲内存链表来记录可用的内存块；堆内存是由程序员主动申请和释放的。堆内存是操作系统中的重要组成，提供堆内存的支持是重要且有意义的。

4.8.1. 堆内存分配器

堆内存分配器是一种用于管理程序中的堆内存，并提供分配和释放内存块的功能的机制。堆内存是一种用于存储动态数据的内存区域，如由new操作符创建的对象，由malloc函数创建的数组，或由strcat函数创建的字符串。堆内存分配器可以帮助程序灵活和高效地使用堆内存，并避免内存泄漏或碎片。

操作系统需要实现一个堆内存管理器，来管理系统中的堆内存。在Rust语言中，标准库规定了一个裸机程序必须提供一个实现了以下特性的全局分配器以获得使用堆内存的能力：

```rust
pub unsafe trait GlobalAlloc {
    unsafe fn alloc(&self, layout: Layout) -> *mut u8;
    unsafe fn dealloc(&self, ptr: *mut u8, layout: Layout);
}
```

4.8.1.1. 数据结构实现

我们采用链表结构来管理空闲的堆内存。在实现时，我们定义了一个简单的链表结构`ListNode`以及一个基于该链表结构的`LinkedListAllocator`分配器：

```rust
struct ListNode {
    size: usize,
    next: Option<&'static mut ListNode>,
}

impl ListNode {
    const fn new(size: usize) -> Self {
        ListNode { size, next: None }
    }

    fn start_addr(&self) -> usize {
        self as *const Self as usize
    }

    fn end_addr(&self) -> usize {
        self.start_addr() + self.size
    }
}

pub struct LinkedListAllocator {
    head: ListNode,
    size: usize,
    allocated: usize,
}
```

可以注意到，我们的链表节点有一个特点：它是直接位于它所代表的内存地址上的。采用这样的设计，可以帮助我们在不使用堆内存的前提下就管理全部的内存区域（传统的链表需要堆内存来创建和存储节点）。

4.8.1.2. 添加链表节点

分配器需要一个方法来向其中添加空闲的链表节点。这既可以用于释放已经不再需要的节点，又可以被用于初始化和可分配内存区域的增长。我们定义了`add_free_region`方法，该方法将一个指定的内存区域添加到`LinkedListAllocator`的空闲链表中。以下是它的工作流程：

1. **校验空闲区域对齐**:
   - 使用assert_eq!确保给定的地址已正确对齐，与ListNode的对齐方式相匹配。这是因为我们需要在这片区域内存储链表节点。
   - 使用assert!确保提供的大小至少与ListNode结构的大小相同。这是必要的，因为我们需要至少这么多空间来存储链表节点。
2. **合并相邻空闲区域**:
   - 此部分代码遍历现有链表，查找与给定内存区域相邻的空闲区域。
   - 如果找到相邻的空闲区域，它会合并给定的内存区域与找到的区域，并更新起始地址和大小。然后从链表中移除找到的相邻区域。
3. **插入新的空闲区域**:
   - 使用给定的大小创建一个新的ListNode节点。
   - 然后，遍历链表，找到一个合适的位置插入新节点，确保链表是按区域大小从大到小排序的。
   - 创建并初始化新节点的内存区域。
   - 将新节点插入链表。

总体而言，add_free_region方法的目的是将一个指定的内存区域（地址和大小）添加到空闲链表中，并在必要时与相邻的空闲区域合并。这种设计有助于减少内存碎片，并确保链表始终保持有序。它的代码实现如下所示:

```
/// 将指定的内存区域增加到链表中
unsafe fn add_free_region(&mut self, addr: usize, size: usize) {
    // 确保这个空闲区域和链表是适配的
    assert_eq!(align_up(addr, mem::align_of::<ListNode>()), addr);
    assert!(size >= mem::size_of::<ListNode>());

    // 先寻找左右两边是否也为空闲

    let (mut left_found, mut right_found) = (false, false);
    let (mut addr, mut size) = (addr, size);
    let mut current = &mut self.head;
    while let Some(ref mut region) = current.next {
        if !left_found && region.end_addr() == addr {
            left_found = true;
            addr = region.start_addr();
            size += region.size;
            current.next = region.next.take();
        } else if !right_found && region.start_addr() == addr + size {
            right_found = true;
            size += region.size;
            current.next = region.next.take();
        } else if left_found && right_found {
```

```
            break;
        } else {
            current = current.next.as_mut().unwrap();
        }
    }

    // 创建一个新的链表节点，并把它加入到合适位置（按照地址从大到小排序）
    let mut node = ListNode::new(size);
    let mut current = &mut self.head;
    while let Some(ref mut region) = current.next {
        if region.size <= node.size {
            break;
        }
        current = current.next.as_mut().unwrap();
    }
    node.next = current.next.take();
    let node_ptr = addr as *mut ListNode;
    node_ptr.write(node);
    current.next = Some(&mut *node_ptr);
}
```

4.8.1.3. 移除链表节点

同样的，分配器需要一个方法来向其中查找并移除空闲的链表节点。这可以用于申请特定的内存区域。我们定义了 `find_region` 方法和 `alloc_from_region` 方法，它们分别负责查找恰当的链表节点，和将节点从链表中取出。

`find_region` 方法的工作流程如下：

1. 遍历链表，查找一个足够大的空闲区域。
2. 对于每一个区域，调用`alloc_from_region`来确定是否可以从该区域中分配所需大小和对齐的内存。
3. 如果找到了合适的区域，那么从链表中移除这个区域，并返回该区域的节点以及分配的起始地址。
4. 如果找不到合适的区域，则返回`None`。

`alloc_from_region` 方法的工作流程如下：

1. 使用`align_up`函数确保分配的起始地址满足给定的对齐要求。
2. 计算分配结束的地址。如果计算结果溢出，返回错误。
3. 如果分配结束地址超出了区域的结束地址，表示该区域不够大，返回错误。
4. 计算剩余空间的大小（该区域结束地址与分配结束地址之间的空间）。如果这个剩余空间小于`ListNode`的大小，并且大于0，则说明剩余空间不足以容纳一个新的链表节点，返回错误。

5. 如果以上所有检查都通过，则返回分配的起始地址。

这两个方法结合使用，提供了在链表分配器中进行内存分配的核心功能。find_region首先确定是否存在一个合适的空闲区域，然后alloc_from_region确保这个区域可以满足分配的大小和对齐要求。如果分配成功，这个区域将从链表中移除，反之，分配器会继续查找其他合适的空闲区域。

以下是它们的代码实现：

```
/// 寻找一个满足给定大小的空闲区域，并把它从链表中移除
///
/// 返回列表节点和可用区域的起始地址
fn find_region(&mut self, size: usize, align: usize) -> Option<(&'static
mut ListNode, usize)> {
    let mut current = &mut self.head;
    // 寻找足够的空间
    while let Some(ref mut region) = current.next {
        if let Ok(alloc_start) = Self::alloc_from_region(&region, size,
align) {
            // 找到了，返回
            let next = region.next.take();
            let ret = Some((current.next.take().unwrap(), alloc_start));
            current.next = next;
            return ret;
        } else {
            // 这个区域不够大，继续往下找
            current = current.next.as_mut().unwrap();
        }
    }

    // 找不到啊找不到，因为你找不到
    None
}

/// 尝试使用给定区域进行具有给定大小和对齐方式的分配
///
/// 成功时返回分配起始地址
fn alloc_from_region(region: &ListNode, size: usize, align: usize) ->
Result<usize, ()> {
    let alloc_start = align_up(region.start_addr(), align);
    let alloc_end = alloc_start.checked_add(size).ok_or(())?;

    if alloc_end > region.end_addr() {
        // 这个区域不够大
        return Err(());
    }
```

```
    let excess_size = region.end_addr() - alloc_end;
    if excess_size > 0 && excess_size < mem::size_of::<ListNode>() {
        // 剩余空间不足以容纳一个ListNode
        return Err(());
    }

    Ok(alloc_start)
}
```

4.8.1.4. 内存布局调整

链表中的节点具有特定的内存布局；因此，如果用户申请的内存区域不满足这个内存布局（例如太小，或者和它不对齐），我们需要一个工具函数来将用户申请的内存向上调整到正确的内存布局，再提交给分配器的工具函数来查找合适的区域。

我设计了函数`size_align`来负责这一工作。它的目的是对给定的内存布局`layout`进行调整，确保其能够容纳一个`ListNode`。此调整涉及两个方面：大小和对齐方式。以下是它的工作内容：

1. **调整对齐方式**：
   - 使用`align_to`方法将`layout`的对齐方式调整为`ListNode`的对齐方式。这确保了分配的内存块对齐方式满足`ListNode`的要求。
   - 使用`pad_to_align`方法对布局进行调整，确保其大小与调整后的对齐方式相匹配。
2. **调整大小**：
   - 使用`max`函数确保调整后的布局大小至少与`ListNode`结构的大小相同。

这个方法的主要作用是确保在进行分配时，分配的内存块不仅可以满足给定的布局要求，而且可以满足存储一个链表节点的要求。这样在将来，如果需要，这块内存可以被重用作为一个链表节点。代码实现如下所示：

```
/// 调整给出的布局，使得其内存区域也能满足存储一个链表节点的需求
///
/// 返回调整后的布局大小和对齐方式
fn size_align(layout: Layout) -> (usize, usize) {
    let layout = layout
        .align_to(mem::align_of::<ListNode>())
        .expect("adjusting alignment failed")
        .pad_to_align();
    let size = layout.size().max(mem::size_of::<ListNode>());
    (size, layout.align())
}
```

4.8.1.5. 堆内存分配

除了实现之前的添加、删除节点的方法外，我们还需要为`GlobalAlloc`特性提供相关方法的实现。`LinkedListAllocator`中的`alloc`方法便是在实现此接口，它被用于根据提供的`layout`从链表分配器中分配内存。

它的工作流程如下所示：

1. **调整布局**：
   - 使用`LinkedListAllocator::size_align`方法调整提供的布局。这步骤确保分配的内存块可以容纳`ListNode`结构。
2. **查找合适的空闲区域**：
   - 使用`find_region`方法根据调整后的大小和对齐方式查找一个合适的空闲区域。
3. **如果找到了合适的空闲区域**：
   a. 计算分配结束的地址。
   b. 计算从分配结束地址到该空闲区域结束地址的剩余空间大小。
   c. 如果剩余空间大于0，则使用`add_free_region`方法将其加入到空闲链表中。
   d. 更新已分配的内存大小统计。
   e. 返回分配的起始地址。
4. **如果没有找到合适的空闲区域**：
   - 返回一个空指针。

通过这种方式，`LinkedListAllocator`可以为请求的内存布局分配内存，并根据需要调整和更新其内部的空闲区域链表。具体的代码实现如下所示：

```
pub unsafe fn alloc(&mut self, layout: Layout) -> *mut u8 {
    // 进行布局调整
    let (size, align) = LinkedListAllocator::size_align(layout);

    if let Some((region, alloc_start)) = self.find_region(size, align) {
        // 找到了，进行分配
        let alloc_end = alloc_start.checked_add(size).expect("overflow");
        let excess_size = region.end_addr() - alloc_end;
        if excess_size > 0 {
            // 有剩余空间，把它加入到链表中
            self.add_free_region(alloc_end, excess_size);
        }
        self.allocated += layout.size();
        alloc_start as *mut u8
    } else {
        // 没找到，返回空指针
        core::ptr::null_mut()
```

```
    }
}
```

4.8.1.6. 堆内存释放

`LinkedListAllocator`中的`dealloc`方法用于实现`GlobalAlloc`的接口，它被用于根据提供的指针`ptr`和`layout`释放先前分配的内存。

它的工作流程如下所示：

1. **调整布局**：使用`LinkedListAllocator::size_align`方法调整提供的布局。这个步骤返回布局的大小（`size`）和对齐方式，但在这个函数中，我们只关心调整后的`size`。
2. **将内存区域添加回链表**：使用`add_free_region`方法，将以`ptr`为起始地址，大小为`size`的内存区域添加回链表中的空闲区域。这一步确保了该内存块再次可用于分配。
3. **更新已分配的内存统计**：更新已分配内存的统计数据，将其减少`layout`的大小。

它的代码实现如下所示：

```rust
pub unsafe fn dealloc(&mut self, ptr: *mut u8, layout: Layout) {
    let (size, _) = LinkedListAllocator::size_align(layout);

    self.add_free_region(ptr as usize, size);
    self.allocated -= layout.size();
}
```

4.8.1.7. GlobalAlloc 接口实现

至此，已经不需太多的工作量就可以完成对`GlobalAlloc`接口的实现了。以下是相关的实现代码：

```rust
unsafe impl GlobalAlloc for Locked<LinkedListAllocator> {
    unsafe fn alloc(&self, layout: Layout) -> *mut u8 {
        self.lock().alloc(layout)
    }

    unsafe fn dealloc(&self, ptr: *mut u8, layout: Layout) {
        self.lock().dealloc(ptr, layout)
    }
}
```

值得一提的是，`Locked`是本项目设计的一个互斥锁结构。在这里，它的作用是保证`LinkedListAllocator`能够被互斥使用，以避免可能的读写冲突。`Locked`的实现如下所示：

```rust
#[derive(Debug)]
pub struct Locked<A> {
    inner: spin::Mutex<A>,
}

impl<A> Locked<A> {
    pub const fn new(inner: A) -> Self {
        Locked {
            inner: spin::Mutex::new(inner),
        }
    }

    pub fn lock(&self) -> spin::MutexGuard<A> {
        self.inner.lock()
    }
}
```

最后，我们将`LinkedListAllocator`设置为我们的全局堆内存分配器：

```rust
#[global_allocator]
pub static ALLOCATOR: Locked<LinkedListAllocator> =
Locked::new(LinkedListAllocator::new());
```

4.8.2. 堆内存初始化

4.8.2.1. 常量定义

```rust
pub const HEAP_START: usize = 0x_0001_0000_0000;
pub const HEAP_SIZE: usize = 40 * 1024 * 1024; // 40 MiB
```

上述两个常量：

- **HEAP_START**：代表堆内存在虚拟地址上的起始位置。
- **HEAP_SIZE**：代表堆内存的大小。本项目中的操作系统内核不需要过多的内存，因此，通过实验，选择了40MiB作为初始堆内存的一个比较合适的大小。

4.8.2.2. 方法实现

我们设置了一个函数`init_heap`，用于初始化堆内存。

以下是这个函数的工作流程：

1. **计算页范围**：
   - 这一步计算堆开始和结束的页。使用`VirtAddr::new`和`Page::containing_address`来计算这个范围，并创建一个表示这个范围的`page_range`迭代器。

2. **页映射**:
   - 对于page_range中的每一个页，执行以下操作:
     a. 使用提供的frame_allocator为这个页分配一个物理帧。
     b. 设置页映射的标志为PRESENT和WRITABLE，表示该页现在是存在的并且是可写的。
     c. 使用提供的mapper来映射这个页到分配的物理帧。
     d. 刷新映射，使其生效。
3. **初始化堆分配器**:
   - 使用先前提到的ALLOCATOR的init方法初始化堆分配器，设置其开始地址为HEAP_START和大小为HEAP_SIZE。
4. **返回成功**:
   - 如果上述所有步骤都成功完成，返回Ok(())。

这个函数设置了初始化了堆内存和堆分配器，使得我们的操作系统可以在后续进行动态内存分配。以下是其具体的代码实现:

```
pub fn init_heap(mapper: &mut impl Mapper<Size4KiB>, frame_allocator: &mut
impl FrameAllocator<Size4KiB>) -> Result<(), MapToError<Size4KiB>> {
    syskrnl::proc::init_process_addr((HEAP_START + HEAP_SIZE) as u64);

    let page_range = {
        let heap_start = VirtAddr::new(HEAP_START as u64);
        let heap_end = heap_start + HEAP_SIZE - 1u64;
        let heap_start_page = Page::containing_address(heap_start);
        let heap_end_page = Page::containing_address(heap_end);
        Page::range_inclusive(heap_start_page, heap_end_page)
    };

    for page in page_range {
        let frame =
frame_allocator.allocate_frame().ok_or(MapToError::FrameAllocationFailed)?;
        let flags = PageTableFlags::PRESENT | PageTableFlags::WRITABLE;
        unsafe { mapper.map_to(page, frame, flags,
frame_allocator)?.flush() };
    }

    unsafe {
        ALLOCATOR.lock().init(HEAP_START, HEAP_SIZE);
    }

    Ok(())
}
```

4.9. 图形模式

4.9.1. 进入图形模式

传统的操作系统开发中使用 `VESA BIOS Extensions`（VBE）来让操作系统开发者在不依赖于特定显卡驱动的情况下，进入图形模式，并实现丰富多彩的图形界面效果。

VESA编程的基本步骤如下：

- 通过中断0x10，AX=0x4F00来获取VESA信息，包括版本号，显卡厂商，支持的模式列表等。
- 从模式列表中选择一个合适的图形模式。
- 通过中断0x10，AX=0x4F02，BX=模式号来设置图形模式。
- 通过中断0x10，AX=0x4F01，CX=模式号来获取模式信息，包括分辨率，色彩深度，每行字节数，显存地址等。
- 根据模式信息，直接向显存地址写入像素数据，实现画点，画线等图形操作。

不过，由于本项目开发时运行于模拟机内，因此我们可以考虑采用虚拟机提供的API来定制更加自由的图形模式。BGA（`Bochs Graphics Adaptor`）就是这样的一套接口。它是一种在Bochs虚拟机中使用的图形编程接口，它可以让操作系统开发者在不使用VBE BIOS的情况下，直接设置高分辨率和高色彩深度的图形模式。

BGA相比VESA的优势主要如下：

- BGA可以直接设置图形模式，而 **不需要通过BIOS中断调用**。这样可以避免使用BIOS调用时伴随的风险。
- BGA可以自由设置显示设置，而不需从给定的配置中选择。例如，我可以自由地设置屏幕的宽度和高度，以及色彩的深度。
- BGA **可以在长模式和保护模式中调用**，而不需要切换到实模式或者虚拟8086模式。VESA BIOS扩展只能在实模式下使用，如果要在保护模式下使用，需要使用特殊的技巧和工具。这就意味着，在系统启动后，我们便难以再使用VESA调整显示设置了。

BGA通过两个16位的IO端口来访问，一个是索引端口，一个是数据端口。通过这两个端口，可以向BGA写入或读取不同的寄存器，用来控制图形模式的参数，如分辨率，色彩深度，虚拟屏幕大小，偏移量等。BGA还支持使用线性帧缓冲区（LFB）或者分段模式来访问显存地址，实现画点，画线等图形操作。

4.9.1.1. 常量定义

```
const VBE_DISPI_IOPORT_INDEX: u16 = 0x01CE;
const VBE_DISPI_IOPORT_DATA: u16 = 0x01CF;
```

BGA通过两个16位的IO端口来访问，一个是索引端口，一个是数据端口。这里定义了索引端口和数据端口的编号。

```
#[allow(dead_code)]
#[repr(u16)]
enum VbeDispiIndex {
    Id = 0,
    Xres,
    Yres,
    Bpp,
    Enable,
    Bank,
    VirtWidth,
    VirtHeight,
    XOffset,
    YOffset,
}
```

BGA 提供了诸多可供读取和设置的条目，读取和写入这些设置项需要使用特定的序号。

```
#[allow(dead_code)]
#[repr(u16)]
enum VbeDispiBpp {
    _4 = 4,
    _8 = 8,
    _24 = 24,
    _32 = 32,
    // 省略了很多用不到的深度
}
```

BGA 同样提供了大量可供选择的像素深度。

4.9.1.2. 工具方法

我们定义了一个函数`bga_write_register`，该函数用于写入一个特定的VBE（VESA BIOS Extensions）寄存器。该函数采用了两个参数：一个是要写入的寄存器的索引（`index`），另一个是要写入的值（`value`）。

函数的工作流程：

1. 定义要写入的寄存器的索引。
2. 通过`outw`函数将该索引写入VBE的索引寄存器。
3. 定义要写入该寄存器的值。
4. 通过`outw`函数将该值写入VBE的数据寄存器。

以下是具体的代码实现：

```
unsafe fn bga_write_register(index: u16, value: u16) {
    outw(VBE_DISPI_IOPORT_INDEX, index);
    outw(VBE_DISPI_IOPORT_DATA, value);
}
```

4.9.1.3. 配置BGA并进入图形模式

我们定义了一个名为`bga_enter_graphic_mode`的函数，该函数用于将BGA切换到图形模式。

以下是该函数的工作流程：

1. **禁用VBE**：
   - 使用前面定义的`bga_write_register`函数禁用 VBE 。
   - 禁用VBE的原因是我们即将修改`VBE_DISPI_INDEX_XRES`、`VBE_DISPI_INDEX_YRES`、`VBE_DISPI_INDEX_BPP`三个寄存器的内容。根据BGA的要求，更改这三个寄存器时必须先禁用VBE。
2. **设置显示模式**：
   - 设置了VBE的分辨率和每像素的位数（BPP - Bits Per Pixel）。具体的分辨率和BPP的值在父模块中定义。
3. **启用VBE**：
   - 重新启用VBE，启动模式是`0x41`，这代表着我们将会使用LFB（Linear Frame Buffer）。
4. **获取LFB地址**：
   - 首先，使用函数`pci_find_device_by_device_id`查找具有特定设备ID的PCI设备。
   - 接着，使用`pci_config_read_u32`函数读取该PCI设备的配置，并从中获取LFB的地址。这个地址存储在PCI设备的配置空间中的`0x10`偏移量处。
5. **初始化显存**：
   - 使用内存页面映射和页帧分配器，以及刚刚获取的LFB地址，调用`create_graphic_memory_mapping`函数初始化显存映射。

这样，我们的操作系统就可以开始在图形模式下渲染内容了。代码实现如下：

```
pub unsafe fn bga_enter_graphic_mode(mapper: &mut OffsetPageTable,
frame_allocator: &mut impl FrameAllocator<Size4KiB>) {
    // 禁用VBE
    bga_write_register(VbeDispiIndex::Enable as u16, 0);

    // 设置显示模式
    bga_write_register(VbeDispiIndex::Xres as u16, super::WIDTH as u16);
    bga_write_register(VbeDispiIndex::Yres as u16, super::HEIGHT as u16);
```

```
    bga_write_register(VbeDispiIndex::Bpp as u16, VbeDispiBpp::_32 as u16);

    // 启用VBE
    bga_write_register(VbeDispiIndex::Enable as u16, 0x41);

    // 获取LFB地址
    let device = pci_find_device_by_device_id(0x1111, 0x1234);
    let address = pci_config_read_u32(device.0, device.1, device.2, 0x10);

    // 初始化显存
    create_graphic_memory_mapping(mapper, frame_allocator, address as u64);
}
```

4.9.1.4. 使用LFB以获得更大的显存

LFB是Linear Frame Buffer（线性帧缓冲区）的缩写，它是一种在计算机图形编程中使用的技术，可以让操作系统直接访问显存地址，而不需要通过分段模式或者BIOS中断调用。LFB的作用是提高图形性能和兼容性，同时也简化了图形编程的难度。

在早期的计算机图形系统中，显存是分成多个段（bank）的，每个段只能存储一部分的像素数据。为了访问不同的段，操作系统需要通过BIOS中断或者特殊的IO端口来切换当前的段号。这样的方式不仅效率低下，而且编程复杂，还存在一定的兼容性问题。

为了解决这些问题，VESA BIOS扩展（VBE）提出了一种使用线性帧缓冲区（LFB）的方案，可以让操作系统在实模式下使用高分辨率和高色彩深度的图形模式。LFB是一种将所有的显存映射到一个连续的线性地址空间的技术，可以让操作系统直接读写显存，而不需要切换段号。LFB大大提高了图形编程的效率和灵活性，可以让操作系统实现更丰富和更美观的图形界面效果。LFB也促进了图形硬件和软件的发展，为后来的3D加速器和OpenGL等图形标准奠定了基础。

如上一节中所述，我们启用LFB，需要在启用BGA时传输参数`0x41`。之后，BGA会将LFB的起始地址写入PCI设备`0x1111,0x1234`中的`0x10`偏移处。不过，给出的LFB地址是物理地址，为了在操作系统中使用，我们还需要将它映射到虚拟地址空间中。

以下是这个操作的工作流程：

1. **常数定义**：

```
const NEEDED_PAGE_NUM: usize = 469;
pub const START_VIRT_ADDR: u64 = 0xC000_0000;
```

- `NEEDED_PAGE_NUM`表示需要映射的页面数量，这里是469页。
- `START_VIRT_ADDR`是显存映射的开始的虚拟地址，这里是`0xC000_0000`。

2. **设置显存映射**:
   - 这是一个循环，会循环NEEDED_PAGE_NUM次，用于映射每一页显存。页的权限为PRESENT和WRITABLE，意味着页面是存在的，并且可写。

以下是具体的代码实现:

```
/// 初始化显存
pub fn create_graphic_memory_mapping(mapper: &mut OffsetPageTable,
frame_allocator: &mut impl FrameAllocator<Size4KiB>, start_physic_addr:
u64) {
    use x86_64::structures::paging::PageTableFlags as Flags;

    for i in 0..NEEDED_PAGE_NUM {
        let page =
Page::<Size4KiB>::containing_address(VirtAddr::new(START_VIRT_ADDR + 0x1000
* i as u64));
        let frame =
PhysFrame::containing_address(PhysAddr::new(start_physic_addr + 0x1000 * i
as u64));
        let flags = Flags::PRESENT | Flags::WRITABLE;

        let map_to_result = unsafe { mapper.map_to(page, frame, flags,
frame_allocator) };
        map_to_result.expect("Map_to_GraphicMemory Failed").flush();
    }
}
```

4.9.2. 绘制与渲染

在图形模式中绘制图形和渲染是较为复杂的工作。我们需要处理图层、覆盖、透明等诸多细节。因此，本项目定义了多个结构和方法，协助我们完成这一任务。

4.9.2.1. 物理层操作

在这一部分中，我们主要阐述对物理层的显存的操作的相关封装与实现。

首先，我设计了一个Buffer结构体:

```
#[repr(transparent)]
pub struct Buffer {
    chars: [[Volatile<Rgb888>; WIDTH]; HEIGHT],
}
```

`#[repr(transparent)]` 属性确保了 `Buffer` 结构体的内存布局与其中的char字段完全相同，这在进行底层的显存操作时很重要。`chars`: 是一个二维数组，代表屏幕的每一个像素。每个像素使用 `Rgb888` 类型来描述，这是 `embedded_graphics` 包提供的工具类，它提

供了像素操作的相关方法，并使用和VBE一致的内存布局存储像素的数据。Volatile 是一个用于表示某些数据应该避免被编译器优化的类型。这在低级系统编程中很常见，尤其是当数据代表硬件映射的内存时。如果不使用 Volatile ， 编译器优化就可能会更改或删除对这些数据的读写，这可能会导致不可预测的行为。

接下来，我设计了一个名为 PhysicalWriter 的结构：

```
pub struct PhysicalWriter(&'static mut Buffer);
```

它持有一个 &'static mut Buffer 引用，意味着它引用的是一个静态生命周期的可变 Buffer。PhysicalWriter 被用于直接写入屏幕的物理缓冲区。

接下来，我们在一个lazy_static宏内实例化 PhysicalWriter：

```
lazy_static! {
    // 底层的显存
    pub static ref GD: Mutex<PhysicalWriter> = {
        Mutex::new(PhysicalWriter(unsafe {&mut
*(Page::<Size4KiB>::containing_address(VirtAddr::new(0xC000_0000))
        .start_address().as_mut_ptr() as *mut Buffer) }))
    };
}
```

在这里，我们使用 Mutex 来包装 PhysicalWriter。这意味着在多线程环境中，对这块物理显存的访问是互斥的。这确保了任何时候只有一个线程可以写入显存。此外，在初始化部分中使用了 unsafe 代码。这是因为它直接使用了一个固定的虚拟地址 0xC000_0000 来获取显存的位置，并强制转换这个地址到 *mut Buffer 指针。这个地址在上一部分中已经映射到了LFB上。

4.9.2.2. 图层定义

我们使用一个结构体Writer来表示每个图层，并通过实现这个图层的相关方法，来为该图层提供便捷的绘制方式（例如，画矩形等）。

```
#[derive(Clone, Debug)]
pub struct Writer {
    pub data: Vec<Vec<(Rgb888, bool)>>,
    pub enable: bool,
}
```

data是一个二维向量，每个元素都是一个 (Rgb888, bool) 的元组。其中 Rgb888 代表像素颜色，而 bool 表示该像素是否被启用（如果未被启用，说明该点是透明的）。enable 是一个布尔字段，用于决定是否显示本图层中的数据。

将多个图层放入一个向量中，并加以排列，就可以便捷地实现多层叠加显示。

```rust
lazy_static! {
    // 多层叠加显示
    pub static ref GL: RwLock<Vec<Mutex<Writer>>> = {
        let mut v:Vec<Mutex<Writer>> = vec![];
        v.reserve(5);
        for _ in 0..5{
            v.push(Mutex::new(Writer::new()));
        }
        RwLock::new(v)
    };
}
```

4.9.2.3. 图层相关方法实现

4.9.2.3.1. 绘制像素点

我们设计了`display_pixel`方法和`display_pixel_safe`方法。前者不进行安全和越界检查，用于在关键路径中减少冗余分支。

二者的工作流程如下：

1. 若调用的是 `display_pixel_safe` 方法，则首先检查给定的坐标是否在屏幕的有效范围内。
2. 使用坐标访问 `chars` 二维数组，并调用 `write` 方法来设置像素的颜色。

具体的代码实现如下：

```rust
pub unsafe fn display_pixel(&mut self, x: usize, y: usize, color: Rgb888) {
    self.0.chars[x][y].write(color);
}

pub fn display_pixel_safe(&mut self, x: usize, y: usize, color: Rgb888) {
    if x < HEIGHT && y < WIDTH {
        self.0.chars[x][y].write(color);
    }
}
```

4.9.2.3.2. 绘制矩形

我设计了 `display_rect` 方法，它负责在屏幕上绘制一个矩形。

函数的工作流程如下：

- 使用 `min` 函数确定矩形的终止坐标，以确保它不超出屏幕的边界。
  - `x_end` 和 `y_end` 分别表示 x 和 y 的终止坐标。

- 使用两个嵌套的 `for` 循环来遍历矩形内的每个像素，并使用 `display_pixel` 方法设置每个像素的颜色。由于 `display_pixel` 是一个 `unsafe` 方法，所以这里的调用也被标记为 `unsafe`。

具体的代码实现如下：

```
pub fn display_rect(&mut self, x: usize, y: usize, w: usize, h: usize,
color: Rgb888) {
    let x_end = min(x + h, HEIGHT);
    let y_end = min(y + w, WIDTH);
    for i in x..x_end {
        for j in y..y_end {
            unsafe {
                self.display_pixel(i, j, color);
            };
        }
    }
}
```

4.9.2.3.3. 绘制BMP图片

BMP是一种常见的无压缩图像格式，其未压缩的特性可以降低在解析图片时的运算量。本项目中设计了`display_img`方法来显示BMP图像。

函数的工作流程如下：

- 使用 `tinybmp` 库中提供的方法尝试从 `bmp_data` 字节切片中解析 BMP 图像。
- 如果解析成功 (`Ok(bmp)`)：
  - 使用 `bmp.pixels()` 遍历图像中的每个像素。
  - 对于每个像素 (`Pixel(position, color)`)，使用 `display_pixel` 方法将其绘制到屏幕上，偏移由给定的 `x` 和 `y` 参数确定。
- 如果解析失败 (`Err(error)`)：
  - 打印相关的错误信息。

具体的代码实现如下：

```
pub fn display_img(&mut self, x: usize, y: usize, bmp_data: &[u8]) {
    match Bmp::<Rgb888>::from_slice(bmp_data) {
        Ok(bmp) => {
            for Pixel(position, color) in bmp.pixels() {
                unsafe {
                    self.display_pixel(x + position.y as usize, y +
position.x as usize, color);
                };
```

```
            }
        }
        Err(error) => {
            qemu_print(format!("{:?}\n", error).as_str());
        }
    }
}
```

4.9.2.3.4. 绘制TTF光栅字符

字符显示也是操作系统的重要功能。当前程序处于图形模式中，必须由我们手动处理字体及其渲染。字体的显示通常比像素和矩形要复杂得多，因为它涉及到处理矢量图形、字形等内容。此处，我们设计了`display_font`方法来显示字体。

函数的工作流程如下：

- 尝试获取字形的确切边界框。如果这失败，将使用一个默认的方形边界框。
- 调整字形的垂直位置，这是为了让g、j等字母能够正确地位于其基线上。
- 使用 `glyph.positioned(point(0.0, 0.0))` 来设置字形的位置。
- 使用 `glyph.draw` 方法绘制字形。这个方法需要一个闭包，它为字形中的每个点提供x, y坐标和透明度v。这个闭包在每个点上都被调用。
  - 使用 `alpha_mix` 函数混合前景色和背景色，根据给定的值v生成实际的像素颜色。
  - 使用 `display_pixel_safe` 方法将计算出的颜色绘制到屏幕上。

具体的代码实现如下：

```
pub fn display_font(
    &mut self,
    glyph: ScaledGlyph,
    x_pos: usize,
    y_pos: usize,
    size: f32,
    line_height: usize,
    fg_color: Rgb888,
    bg_color: Rgb888,
) {
    let bbox = glyph.exact_bounding_box().unwrap_or(Rect {
        min: point(0.0, 0.0),
        max: point(size, size),
    });

    let x_offset = (line_height as f32 + bbox.min.y) as usize;
    //qemu_print(format!("{:?},{:?},{:?}\n",ch,bbox,x_offset).as_str());
```

```
    let glyph = glyph.positioned(point(0.0, 0.0));
    glyph.draw(|y, x, v| {
        let (color, _) = alpha_mix(fg_color, v, bg_color, 1.0);
        self.display_pixel_safe(x_offset + x_pos + x as usize, y_pos + y as
usize + bbox.min.x as usize, color);
    })
}
```

4.9.2.3.5. 绘制字符串

之前我们已经提供了绘制TTF字符的方法，但是在实际使用中，提供一个字符串就能完成绘制更为方便。为此，我们设计并实现了`display_font_string`方法。

函数的工作流程如下：

- 对于字符串中的每个字符：
  - 检查当前的 `y_pos` 是否超过了屏幕的宽度。如果超过，则立即返回，避免写入屏幕边界之外。
  - 使用 `get_font(ch, size)` 函数为当前字符获取字形 (`glyph`) 及其横向指标 (`hm`)。
  - 使用 `display_font` 方法将当前字形渲染到屏幕上。
  - 更新 `y_pos` 以包括当前字形的宽度 (`hm.advance_width`) 和额外的1个像素间距，准备为下一个字符渲染。

具体的代码实现如下：

```
/// 此方法不检查换行
pub unsafe fn display_font_string(
    &mut self,
    s: &str,
    x_pos: usize,
    y_pos: usize,
    size: f32,
    line_height: usize,
    fg_color: Rgb888,
    bg_color: Rgb888,
) {
    let mut y_pos = y_pos;
    for ch in s.chars() {
        if y_pos >= WIDTH {
            return;
        }
        let (glyph, hm) = get_font(ch, size);
        self.display_font(glyph, x_pos, y_pos, size, line_height, fg_color,
bg_color);
        y_pos += hm.advance_width as usize + 1usize;
```

```
    }
}
```

4.9.2.4. 图层渲染

我设计了render方法来完成图层渲染的工作。render方法属于PhysicalWriter，它是一个渲染函数，用于绘制屏幕上的一部分。这个方法的目的是，给定一个屏幕上的矩形区域，将该区域的内容从多层（由GL表示）渲染到实际的物理屏幕上。我们可以将这看作为一种基础的“图层合成”技术。

函数的工作流程如下：

- 首先，检查提供的坐标是否都在屏幕范围内，如果不在则直接返回。
- 获取GL的读取锁，GL代表一个包含多层图形的列表。
- 对于最顶部的层（最后一个图层），创建一个新的graph，该graph是一个二维数组，代表屏幕上每个像素的颜色和一个布尔值，表示该像素是否是透明的。
- 遍历其它的层，从顶部层到底部层。如果某层被禁用，就跳过它。
  - 对于每个像素位置，如果当前graph中的像素是透明的，并且当前层中的像素不透明，那么将当前层的像素复制到graph。
- 最后，对于每个像素位置，如果graph中的像素是透明的，那么从底层复制像素到graph。
- 遍历矩形区域的每个像素，并使用PhysicalWriter的方法将graph中的像素写入实际的显存区域。

具体的代码实现如下：

```
impl PhysicalWriter {
    pub fn render(&mut self, sx: usize, sy: usize, ex: usize, ey: usize) {
        if sx < HEIGHT && sy < WIDTH && ex <= HEIGHT && ey <= WIDTH {
            if GL.read().len() == 0 {
                return;
            }
            let p_lock = GL.read();
            let lock = p_lock[p_lock.len() - 1].lock();
            let mut graph: Box<Vec<Vec<(Rgb888, bool)>>> = if lock.enable {
                Box::new(lock.data.clone())
            } else {
                let mut g = Box::new(lock.data.clone());
                for r in g.iter_mut() {
                    for p in r.iter_mut() {
                        p.1 = false;
```

```
                    }
                }
                g
            };
            drop(lock);
            for layer in (1..p_lock.len() - 1).rev() {
                let lock = p_lock[layer].lock();
                if !lock.enable {
                    continue;
                }
                let tomix = &lock.data;
                for x in sx..ex {
                    for y in sy..ey {
                        if !graph[x][y].1 && tomix[x][y].1 {
                            graph[x][y] = tomix[x][y]
                        }
                    }
                }
            }
            let tomix = &p_lock[0].lock().data;
            for x in sx..ex {
                for y in sy..ey {
                    graph[x][y].0 = if graph[x][y].1 { graph[x][y].0 } else
{ tomix[x][y].0 };
                }
            }
            for x in sx..ex {
                for y in sy..ey {
                    self.0.chars[x][y].write(graph[x][y].0);
                }
            }
        }
    }
}
```

这个方法实现了一个简单的font-to-back图层合成策略。开始时，它先考虑最顶部的图层。然后，它遍历每个更低的图层，只替换之前图层中透明的像素。这样，更高层的不透明像素始终位于更低层的像素之上。

4.9.3. 文本输出

实现了图形模式下的绘制和渲染后，标准输出的实现就应该提上日程。此处，我们实现了一个简单的标准输出（和VGA文本输出类似），它可以在特定图层上显示文本。

4.9.3.1. 常量定义

```rust
const TEXT_AREA_HEIGHT: usize = super::HEIGHT - 30;
const TEXT_AREA_WIDTH: usize = super::WIDTH;
const TEXT_AREA_POS: (usize, usize) = (22, 0);
const TEXT_SIZE: f32 = 16.0;
const TEXT_HEIGHT: usize = TEXT_SIZE as usize;
const TEXT_COLOR: Rgb888 = rgb888!(0xdddddddu32);
const TAB_SIZE: usize = 4 * 16;
```

- **TEXT_AREA_HEIGHT** 和 **TEXT_AREA_WIDTH** 定义了文本区域的大小。
- **TEXT_AREA_POS** 定义了文本区域在屏幕上的起始位置。
- **TEXT_SIZE** 定义了文本的大小。
- **TEXT_HEIGHT** 定义了文本的行高。
- **TEXT_COLOR** 定义了文本的颜色。
- **TAB_SIZE** 定义了制表符的大小。

4.9.3.2. 数据结构定义

```rust
pub struct TextWriter {
    pub(crate) text_area_height: usize,
    pub(crate) text_area_width: usize,
    pub(crate) text_area_pos: (usize, usize),
    pub(crate) y_position: usize,
    pub(crate) line_position: usize,
    pub(crate) line_height: usize,
    pub(crate) line_gap: usize,
    pub(crate) max_line: usize,
    pub(crate) color: Rgb888,
    pub(crate) layer: usize,
}
```

TextWriter 结构体: `TextWriter` 是一个公开的结构体，它表示文本输出器。

其字段如下：

- `text_area_height` 和 `text_area_width`：代表文本输出区域的高度和宽度。
- `text_area_pos`：表示文本输出区域在屏幕上的位置。
- `y_position` 和 `line_position`：表示当前输出文本的垂直和水平位置。
- `line_height`：表示文本行的高度。
- `line_gap`：表示文本行之间的间隔。
- `max_line`：表示文本区域内可以显示的最大文本行数。
- `color`：代表文本的颜色。
- `layer`：表示输出文本的图层。

我们随后对 `TextWriter` 进行了实例化：

```rust
lazy_static! {
    pub static ref TEXT_WRITER: Mutex<TextWriter> = {
        Mutex::new(TextWriter {
            text_area_height: TEXT_AREA_HEIGHT,
            text_area_width: TEXT_AREA_WIDTH,
            text_area_pos: TEXT_AREA_POS,
            y_position: 0,
            line_position: 0,
            line_height: TEXT_HEIGHT,
            line_gap: 4,
            max_line: TEXT_AREA_HEIGHT / (TEXT_HEIGHT + 4),
            color: TEXT_COLOR,
            layer: 1,
        })
    };
}
```

这里我们使用 `lazy_static!` 宏定义了一个名为 `TEXT_WRITER` 的全局可变引用。这个宏确保在整个程序运行期间只有一个 `TextWriter` 实例被创建，并且这个实例是线程安全的，因为它被包裹在一个 `Mutex` 内。

4.9.3.3. 方法实现

4.9.3.3.1. 字符显示

我们设计了一个名为`write_char`函数，该函数的目的是在屏幕上渲染一个字符。这个函数是`TextWriter`结构体的方法，接收一个字符作为参数，并根据该字符来决定如何显示。

函数的工作流程如下：

1. **匹配字符**：
   - 对给定的字符进行模式匹配。如果字符是't'（制表符），它调用`self.horizontal_tab()`。这可能是使输出位置向右移动一个制表符宽度的方法。
   - 如果字符是'n'（换行符），它调用`self.new_line()`。这将导致输出位置移动到下一行的开始。
   - 对于其他任何字符，执行下面的代码。
2. **获取字形和度量信息**：
   - 使用`get_font(ch, TEXT_SIZE)`函数从字体系统中获取字符的字形和相关的度量信息。
3. **检查文本宽度**：
   - 如果当前位置加上即将绘制的字符的宽度超出了文本区域的宽度，它将调用`self.new_line()`以移动到下一行。

4. **锁定和绘制**:
   - 使用`GL.read()`获取一个锁，这是为了确保在同一时间只有一个线程可以访问并绘制到指定的图层。
   - 使用`lock.display_font`在屏幕上渲染该字符。
5. **更新位置**:
   - 更新`self.y_position`以包括已绘制的字符的宽度，再加上一个额外的像素。这确保了每个字符之间有适当间隔。

具体的代码实现如下:

```
fn _write_char(&mut self, ch: char) {
    match ch {
        '\t' => self.horizontal_tab(),
        '\n' => self.new_line(),
        ch => {
            let (glyph, hm) = get_font(ch, TEXT_SIZE);
            if self.y_position + hm.advance_width as usize >
self.text_area_width {
                self.new_line();
            }

            let p_lock = GL.read();
            let mut lock = p_lock[self.layer].lock();
            lock.display_font(
                glyph,
                (self.line_height + self.line_gap) * self.line_position +
self.text_area_pos.0,
                self.y_position + self.text_area_pos.1,
                TEXT_SIZE,
                self.line_height,
                self.color,
            );

            drop(lock);

            self.y_position += hm.advance_width as usize + 1usize;
        }
    }
}

/// 提供外部调用的版本，内部勿调用
pub fn write_char(&mut self, ch: char) {
    self._write_char(ch);
}
```

4.9.3.3.2. 字符串显示

我们设置了一个公开方法write_string，它接收一个字符串引用s并为该字符串中的每个字符调用私有方法_write_char。

函数的工作流程如下：

- 首先遍历输入字符串s中的每个字符。
- 对于s中的每个字符ch，它调用之前定义的_write_char方法。

具体的代码实现如下：

```
pub fn write_string(&mut self, s: &str) {
    for ch in s.chars() {
        self._write_char(ch);
    }
}
```

write_string方法提供了一个简单的方式来在屏幕上显示一个字符串。它依赖于_write_char方法来处理每个字符的实际渲染逻辑，这确保了字符串中的每个字符都按照预期被处理并显示。例如，当_write_char遇到换行符时，它会将文本位置移动到下一行，这样write_string可以正确地渲染多行文本。

4.9.3.3.3. 换行处理

我们设计了 new_line 方法，它的目的是处理换行的情况。

函数的工作流程如下：

1. **回车**（Carriage Return）：
   - 将水平位置设置为0，这表示文本的起始位置在行的开始。
2. **增加新行**:
   - 如果当前的行位置（self.line_position）还未到达最大行数（self.max_line），则简单地增加self.line_position，表示进入到下一行。
   - 如果当前行已经是文本区域的最后一行，那么需要进行滚动操作：
     a. 锁定当前的层，以保证在同一时间只有一个线程可以访问和操作。
     b. 循环将文本数据向上滚动一个行的高度，也就是将每行的数据复制到上一行。这样做会导致第一行的数据被覆盖，实现文本的滚动效果。
     c. 接下来，循环将文本区域的最后一行清空（设置为默认颜色和未使用状态）。这样做是为了确保滚动后的新空行没有旧数据。
     d. 最后，释放互斥锁。

总而言之，new_line方法的工作是处理文本换行。如果文本还没有到达文本区域的底部，它只会简单地将当前的行位置增加。但如果文本已经在文本区域的底部，它将滚动整个文本区域，使得新的文本可以继续在底部被写入。该函数的具体实现如下：

```
fn new_line(&mut self) {
    // 1. 回车
    self.y_position = 0;
    // 2. 加新行
    if self.line_position + 1 < self.max_line {
        self.line_position += 1;
    } else {
        let p_lock = GL.read();
        let mut lock = p_lock[self.layer].lock();
        for x in self.text_area_pos.0..self.text_area_pos.0 +
(self.max_line - 1) * (self.line_height + self.line_gap) {
            for y in self.text_area_pos.1..self.text_area_pos.1 +
self.text_area_width {
                lock.data[x][y] = lock.data[x + (self.line_height +
self.line_gap)][y];
            }
        }
        for x in self.text_area_pos.0 + (self.max_line - 1) *
(self.line_height + self.line_gap)..self.text_area_pos.0 +
self.text_area_height {
            for y in self.text_area_pos.1..self.text_area_pos.1 +
self.text_area_width {
                lock.data[x][y] = (DEFAULT_RGB888, false);
            }
        }

        drop(lock);
    }
}
```

4.10. 时钟和时间

时间在操作系统开发中是一个很重要的主题，它是操作系统的核心概念之一，它影响着操作系统的功能，性能和安全性。

时间是操作系统管理和调度进程和线程的基础，它可以让操作系统实现多任务和并发的功能，提高系统的效率和响应性。操作系统通过使用时钟中断，定时器，优先级等机制来控制进程和线程的执行顺序和时间片。

除此以外，大量的程序都需要运行在时间上——例如等待一段时间后再运行下一段程序。总之，时间在操作系统开发中的重要性是不可忽视的，它涉及到操作系统的各个方面和层次。

4.10.1. RTC

RTC（Real Time Clock）是计算机中的一个重要元件，它可以测量和记录时间的流逝。RTC可以在主设备的电源关闭或不可用的情况下，继续保持时间的运行。这样可以让设备在恢复电源后，不会丢失时间信息。此外，RTC可以提供稳定和精确的时间信号，用于管理和调度设备的各种功能，如进程，线程，中断，定时器等。

在本系统中，我使用RTC来获得当前的时间和日期，并实现操作系统中的动态时间显示功能。

4.10.1.1. RTC时间读取

我设计了结构 `RawTime` 来存储从 RTC 中获得的时间：

```
#[derive(Debug, Clone, Copy, Eq, PartialEq, Default)]
pub struct RawTime {
    pub second: u8,
    pub minute: u8,
    pub hour: u8,
    pub day: u8,
    pub month: u8,
    pub year: u32,
}
```

有了存储时间的结构体之后，我们就可以着手完成获取时间的函数了。该函数的主要目的是从实时时钟（RTC）读取当前的时间。

该函数的工作流程如下：

1. **第一次获取时间**：
   - 使用 `while get_update_in_progress_flag() > 0 {}` 循环确保 RTC 不在更新过程中，以此避免读取到的时间不准确。
   - 使用 `get_rtc_register(register)` 函数读取 RTC 的指定寄存器的值，并赋值给对应的 `RawTime` 字段。
2. **第二次获取时间**：
   - 为了确保读取的时间是准确的，代码再次从 RTC 读取时间值。
   - 使用循环来不断读取时间，直到读取的两次时间值完全相同，从而确保读取的时间是准确的。

3. **处理时间格式**:
   - RTC 可以以二进制编码的十进制（BCD）格式或二进制格式存储时间值。`register_b` 的第三位决定了使用的格式。
   - 如果 RTC 使用 BCD 格式，那么我们需要将其转换为普通的二进制格式。使用位运算和算数操作来实现这一转换。
4. **转换为24小时制**:
   - 如果 RTC 使用的是12小时制，并且当前时间是下午，那么将小时值转换为24小时制。
5. **计算完整的4位数年份**:
   - RTC 通常只返回两位数的年份，例如 “23” 而不是 “2023”。我们通过加上当前世纪的值来得到四位数的年份。
6. 最后，该函数返回读取和转换后的 `RawTime` 结构。

总的来说，这个函数的主要作用是从实时时钟读取当前的日期和时间，并对其进行必要的格式转换，然后返回这个时间。以下是这个函数的具体实现：

```
pub fn read_rtc() -> RawTime {
    let mut time = RawTime::default();

    // 第一次获取时间
    while get_update_in_progress_flag() > 0 {}
    time.second = get_rtc_register(0x00);
    time.minute = get_rtc_register(0x02);
    time.hour = get_rtc_register(0x04);
    time.day = get_rtc_register(0x07);
    time.month = get_rtc_register(0x08);
    time.year = get_rtc_register(0x09) as u32;

    // 第二次获取时间
    loop {
        let last_time = time;
        while get_update_in_progress_flag() > 0 {}
        time.second = get_rtc_register(0x00);
        time.minute = get_rtc_register(0x02);
        time.hour = get_rtc_register(0x04);
        time.day = get_rtc_register(0x07);
        time.month = get_rtc_register(0x08);
        time.year = get_rtc_register(0x09) as u32;
        if time == last_time {
            break;
        }
    }
```

```
    // 处理时间格式
    let register_b = get_rtc_register(0x0B);
    if (register_b & 0x04) == 0 {
        time.second = (time.second & 0x0F) + ((time.second / 16) * 10);
        time.minute = (time.minute & 0x0F) + ((time.minute / 16) * 10);
        time.hour = ((time.hour & 0x0F) + (((time.hour & 0x70) / 16) * 10))
| (time.hour & 0x80);
        time.day = (time.day & 0x0F) + ((time.day / 16) * 10);
        time.month = (time.month & 0x0F) + ((time.month / 16) * 10);
        time.year = (time.year & 0x0F) + ((time.year / 16) * 10);
    }

    // 如有必要，转换为24小时制
    if (register_b & 0x02) == 0 && (time.hour & 0x80) > 0 {
        time.hour = ((time.hour & 0x7F) + 12) % 24;
    }

    // 计算完整的4位数年份
    time.year += CURRENT_YEAR - (CURRENT_YEAR % 100);

    time
}
```

有了 `read_rtc()` 函数之后，我们就可以通过调用这个函数来为操作系统提供获得当前时间的方法了：

```
const TIME_ZONE: u8 = 8;

/// 获取RTC时间
pub fn raw_time() -> RawTime {
    let mut tm = read_rtc();
    tm.hour += TIME_ZONE;
    if tm.hour >= 24 {
        tm.hour -= 24;
        tm.day += 1;
    }
    tm
}
```

4.10.1.2. RTC中断控制

除了提供时间，RTC也可以发送中断。RTC可以发送周期性的中断，用于管理和调度设备的各种功能；RTC可以发送闹钟中断，用于提醒用户或执行某些任务；RTC可以发送更新中断，用于提醒操作系统其存储的时间有更新。

本项目主要使用RTC的周期中断和更新中断。

4.10.1.2.1. 打开RTC中断

我们首先准备如下两个工具函数:

```
/// 禁用NMI（非屏蔽中断）
fn disable_nmi() {
    unsafe {
        let prev = inb(CmosPort::Address as u16);
        outb(CmosPort::Address as u16, prev | 0x80);
    }
}

/// 启用NMI（非屏蔽中断）
fn enable_nmi() {
    unsafe {
        let prev = inb(CmosPort::Address as u16);
        outb(CmosPort::Address as u16, prev & 0x7F);
    }
}
```

RTC的NMI（非屏蔽中断）是一种特殊的中断，它可以让RTC在发生某些紧急或异常的情况时，向CPU发送一个不能被屏蔽或忽略的信号，强制CPU执行一个预定义的中断处理程序。但是，当正在打开RTC中断时，NMI中断必须被关闭。

接下来, 我们需要另一个函数来通知RTC先前发生的中断已经处理完毕, 并刷新其状态:

```
/// 通知CMOS中断已经处理结束
pub fn notify_end_of_interrupt() {
    unsafe {
        outb(CmosPort::Address as u16, 0x0C);
        inb(CmosPort::Data as u16);
    }
}
```

为了提供对不同中断类型的统一封装，我们还需要设计一个 `Interrupt` 枚举:

```
#[repr(u8)]
enum Interrupt {
    /// 周期模式
    Periodic = 1 << 6,
    /// 闹钟模式
    Alarm = 1 << 5,
    /// 更新模式
    Update = 1 << 4,
}
```

枚举包含三个值，代表三种不同的中断类型：周期模式、闹钟模式和更新模式。枚举通过 `<<` 操作符为每个中断类型分配一个特定的位。例如，`Periodic = 1 << 6` 表示周期模式中断的位位于第6位。这些位是RTC标准规定的。

现在，我们可以开始实现启用RTC中断的工具函数了。`enable_interrupt`函数用于在CMOS 中启用特定类型的中断，该函数接收一个 `Interrupt` 枚举值作为参数，表示需要启用的中断类型。以下是这个函数的工作流程：

- 通过`interrupts::without_interrupts`完成关中断。这是为了避免在修改中断设置时发生任何冲突。
- 通过`disable_nmi()`禁用NMI。
- 完成I/O操作：
  - 在 `unsafe` 块内，函数首先选择 CMOS 的 0x0B 地址，这通常是 RTC 中中断使能和控制相关的位置。
  - 然后，使用 `inb` 函数读取当前的中断设置。
  - 之后，使用 `outb` 函数再次选择 0x0B 地址。
  - 使用 `outb` 函数，将原先的中断设置与传入的中断枚举值进行或操作，并写回到 RTC 的数据寄存器。这启用了指定的中断。
- 通过`enable_nmi()`启用NMI，调用 `notify_end_of_interrupt()` 函数通知 RTC 中断已经处理完毕。

以下是具体的代码实现：

```
/// 启用CMOS中断
fn enable_interrupt(interrupt: Interrupt) {
    interrupts::without_interrupts(|| {
        disable_nmi();
        unsafe {
            outb(CmosPort::Address as u16, 0x0B);
            let prev = inb(CmosPort::Data as u16);
            outb(CmosPort::Address as u16, 0x0B);
            outb(CmosPort::Data as u16, prev | interrupt as u8);
        }
        enable_nmi();
        notify_end_of_interrupt();
    });
}
```

之后，我们就能为每种中断封装一个启用函数了：

```
/// 启用周期中断
pub fn enable_periodic_interrupt() {
```

```
    enable_interrupt(Interrupt::Periodic);
}

/// 启用闹钟中断
pub fn enable_alarm_interrupt() {
    enable_interrupt(Interrupt::Alarm);
}

/// 启用更新中断
pub fn enable_update_interrupt() {
    enable_interrupt(Interrupt::Update);
}
```

4.10.1.2.2. 初始化RTC

在初始化RTC之前，我们还要准备一个用于设置 RTC 周期性中断的频率。这是通过修改 RTC 寄存器的特定位来完成的。函数接收一个 `u8` 类型的参数 `rate`，参数 `rate` 必须介于 3 到 15 之间。最终设置的频率是 `32768 >> (rate - 1)` Hz。

这个函数的工作流程如下：

- 用 `interrupts::without_interrupts` 包围了函数主体，确保设置过程不会被其他中断打断。
- 用 `disable_nmi()` 和 `enable_nmi()` 在修改 RTC 设置之前禁用和之后启用非屏蔽中断（NMI）。
- 在 `unsafe` 块内：
  - 使用 `outb` 函数选择 RTC 的 0x0A 地址，这是周期中断频率相关的地址。
  - 使用 `inb` 函数读取当前的设置。
  - 再次使用 `outb` 函数选择 0x0A 地址。
  - 使用 `outb` 函数，将原先的设置 (保留高 4 位，即 `prev & 0xF0`) 与新的 `rate` 值进行或操作，并写回到 RTC 的数据寄存器。这设置了新的周期中断频率。
- 最后，调用 `notify_end_of_interrupt()` 函数通知 RTC 中断已经处理完毕。

该函数的实现如下：

```
/// 设置周期中断的频率
///
/// `rate`必须介于3到15之间
///
/// 效果是: `32768 >> (rate - 1)`
pub fn set_periodic_interrupt_rate(rate: u8) {
    interrupts::without_interrupts(|| {
        disable_nmi();
```

```
        unsafe {
            outb(CmosPort::Address as u16, 0x0A);
            let prev = inb(CmosPort::Data as u16);
            outb(CmosPort::Address as u16, 0x0A);
            outb(CmosPort::Data as u16, (prev & 0xF0) | rate);
        }
        enable_nmi();
        notify_end_of_interrupt();
    });
}
```

现在我们可以执行初始化了。我们定义了一个名为 `init` 的函数，其目的是初始化与 RTC (实时时钟) 相关的设置和中断处理程序。

以下是这个函数的工作流程：

1. 设置 IRQ 8 的中断处理程序为 `rtc_interrupt_handler`。IRQ 8 与 CMOS RTC 相关联。这意味着，当 RTC 产生一个中断时，`rtc_interrupt_handler` 函数将被调用来处理它。
2. 使用之前定义的 `set_periodic_interrupt_rate` 函数设置周期性中断的频率。值 `15` 是最大的可接受值，意味着中断频率将被设置为最低 (`32768 >> (15 - 1)`)，也就是 2Hz。这是为了每半秒更新一次系统的时间显示（以实现时间中央冒号闪烁的效果）。
3. 启用 CMOS 的周期性中断。
4. 启用 CMOS 的更新中断。

总之，init 函数的目的是为 CMOS RTC 设置一个中断处理程序，将周期中断的频率设置为最低，并启用周期和更新中断。这为操作系统或其他上下文提供了 RTC 功能和中断处理。以下是它的代码实现：

```
pub fn init() {
    set_irq_handler(8, rtc_interrupt_handler);
    set_periodic_interrupt_rate(15); // 拉满
    enable_periodic_interrupt();
    enable_update_interrupt();
}
```

4.10.1.2.3. RTC中断处理

我们定义了名为 `rtc_interrupt_handler` 的 RTC（实时时钟）中断处理程序。当 RTC 产生一个中断并且此处理程序已被设置为相应的 IRQ 处理程序时，这个函数将被调用。

以下是这个函数的工作流程：

1. **获取状态**:
   - 通过使用 `outb` 函数向 `CmosPort::Address` 写入 `0x0C`，来选择 CMOS 中断状态寄存器。
   - 使用 `inb` 函数从 `CmosPort::Data` 读取当前的中断状态。
2. **周期中断检查**:
   - 使用位与操作 `status & Interrupt::Periodic as u8` 来检查是否周期中断已被触发。
   - 如果触发了周期中断，它调用了 `syskrnl::clock::half_sec_handler()` 函数。这是半秒事件的处理函数。
3. **更新中断检查**:
   - 使用位与操作 `status & Interrupt::Update as u8` 来检查是否更新中断已被触发。
   - 如果触发了更新中断，它会更新 `LAST_RTC_UPDATE` 的值。这是全局原子变量，用于存储最后一次 RTC 更新的系统滴答时钟值。这允许其他系统部分知道 RTC 最后一次更新的时间。
4. **中断结束通知**:
   - 在处理完中断后，调用 `notify_end_of_interrupt()` 函数来通知 CMOS 中断已经处理完毕。

以下是它的代码实现:

```rust
/// RTC中断处理函数
pub fn rtc_interrupt_handler() {
    let status = unsafe {
        outb(CmosPort::Address as u16, 0x0C);
        inb(CmosPort::Data as u16)
    };
    if status & Interrupt::Periodic as u8 > 0 {
        // 周期中断
        syskrnl::clock::half_sec_handler();
    } else if status & Interrupt::Update as u8 > 0 {
        // 更新中断
        LAST_RTC_UPDATE.store(super::ticks(), Ordering::Relaxed);
    }
    notify_end_of_interrupt();
}
```

在上文提到了一个名为“半秒事件”的事件。这个事件每半秒钟触发一次，用于每半秒更新一次系统的时间显示（以实现时间中央冒号闪烁的效果）。

以下是这个事件的处理函数的工作流程:

1. **翻转 EVEN_ODD_FLAG**:
   - 使用 `fetch_not` 方法翻转 `EVEN_ODD_FLAG` 的值。这个标志是用来追踪时间的奇偶性的（例如，确定已经过了整秒或半秒）。
   - `eof` 存储了翻转前的 `EVEN_ODD_FLAG` 的值。
2. **更新状态栏时间**:
   - 调用 `syskrnl::gui::status_bar::update_status_bar_time(eof)` 函数以根据 `EVEN_ODD_FLAG` 的前一个值更新 GUI 的状态栏时间。如果当前过了整秒，那么时间中央的冒号将被隐藏。
3. **处理 GUI 时间更新事件**:
   - 如果 `EVEN_ODD_FLAG` 在翻转前是假，即代表着已经过去整秒，进入该块。
   - 该块首先锁定 `GUI_TIME_UPDATE_EVENT_NEEDER`，这是一个包含所有需要时间更新的事件编号的列表。
   - 对于该列表中的每个事件编号 (eid):
     - 使用 `EVENT_QUEUE` 尝试唤醒与该事件编号关联的进程或线程，并为其发送一个 `GUI_EVENT_TIME_UPDATE` 事件。
     - 如果 `wakeup_with_ret` 成功找到一个进程（返回 Some(pid)），则使用 `SCHEDULER` 唤醒该进程。

以下是这个事件的处理函数的具体实现:

```
lazy_static! {
    pub static ref GUI_TIME_UPDATE_EVENT_NEEDER: Mutex<BTreeSet<usize>> =
Mutex::new(BTreeSet::new());
}

static EVEN_ODD_FLAG: AtomicBool = AtomicBool::new(false);

/// 半秒事件处理器
pub fn half_sec_handler() {
    let eof = EVEN_ODD_FLAG.fetch_not(Ordering::Relaxed);
    syskrnl::gui::status_bar::update_status_bar_time(eof);
    if !eof {
        // debugln!("Half sec: {:?}", GUI_TIME_UPDATE_EVENT_NEEDER.lock());
        for eid in GUI_TIME_UPDATE_EVENT_NEEDER.lock().iter() {
            if let Some(pid) = EVENT_QUEUE
                .lock()
                .wakeup_with_ret(*eid,
gui_event_make_ret(GUI_EVENT_TIME_UPDATE, 0, 0, 0))
            {
                SCHEDULER.lock().wakeup(pid);
            }
        }
```

```
    }
}
```

4.10.2. PIT

PIT是Programmable Interval Timer（可编程间隔定时器）的缩写，它是一种用于产生周期性的时钟中断信号的硬件设备，可以用于操作系统的管理和调度，如多任务，定时器，性能分析等。PIT的重要性和作用有以下几点：

- PIT可以提供一个稳定和精确的时钟源，用于测量和记录时间的流逝，以及同步不同的事件和操作。
- PIT可以根据操作系统的需求，设置不同的计数值和工作模式，来调整时钟中断的频率和形式。
- PIT可以与其他外设或模块协作，如DMA，ADC，DAC等，来实现一些高级的功能，如数据传输，模拟转换，声音播放等。

PIT通常使用一个晶体振荡器来产生一个固定的频率信号，通常是1.193182 MHz。这个频率可以被分频或倍频，来得到不同的时钟周期。

4.10.2.1. PIT 频率设置

首先，我们定义相关的常量：

```
/// `PIT_FREQUENCY`的值是x86架构默认的
pub const PIT_FREQUENCY: f64 = 3_579_545.0 / 3.0; // 1_193_181.666 Hz
/// `PIT_DIVIDER`的值算个半魔数，用来得到大概1ms的PIT频率
const PIT_DIVIDER: usize = 1193;
/// 两次PIT中断之间的时间间隔
const PIT_INTERVAL: f64 = (PIT_DIVIDER as f64) / PIT_FREQUENCY;
/// 每秒Tick数
pub const PIT_PER_SECOND: usize = 1000;
```

- 常量 **PIT_FREQUENCY** 表示PIT的基础频率。在x86架构中，PIT的标准频率大约是1.193181666 MHz。
- 常量 **PIT_DIVIDER** 是一个用于配置PIT的分频值。它的目的是调整PIT的输出频率，使其接近于每毫秒一次的中断频率。在这里，分频值是1193。
- 常量 **PIT_INTERVAL** 代表了两次连续的PIT中断之间的时间间隔。通过将`PIT_DIVIDER`值除以`PIT_FREQUENCY`，我们可以得到每次中断之间的精确时间间隔。
- 常量 **PIT_PER_SECOND** 定义了每秒的Tick数。我们设计的目标是使PIT每毫秒产生一次Tick，即每秒1000次。

接下来，我们为PIT设备设置分频器。我们定义了一个函数 `set_pit_frequency_divider`，其主要目的是为PIT设置分频器值，从而调整PIT的输出频率。它的工作流程如下：

1. 使用 `interrupts::without_interrupts` 确保在执行时中断是关闭的，以避免可能的竞争条件或中断干扰。
2. 将分频器转换为一个小端字节数组。
3. 定义PIT的工作模式为6，代表方波生成器。
4. 定义了PIT的访问模式为3，代表既要写入低字节也要写入高字节。
5. 设置PIT的命令字节。
6. 将分频器的低字节和高字节发送到PIT的数据端口（0x40）。

以下是它的代码实现：

```
/// 设置PIT频率分频器
///
/// `divider` - 分频器
/// `channel` - 通道（听说有4个?）
pub fn set_pit_frequency_divider(divider: u16, channel: u8) {
    // 关中断
    interrupts::without_interrupts(|| {
        let bytes = divider.to_le_bytes();
        let operating_mode = 6; // 方波生成器
        let access_mode = 3; // 低字节和高字节都要写入
        unsafe {
            // 选择通道和模式
            x86::io::outb(0x43, (channel << 6) | (access_mode << 4) |
operating_mode);
            // 写入分频器
            x86::io::outb(0x40, bytes[0]);
            x86::io::outb(0x40, bytes[1]);
        }
    })
}
```

4.10.2.2. PIT 中断处理

本系统中借助PIT实现的功能包括屏幕刷新、进程调度等。由于进程调度会在后面章节中提及，因此本部分将不叙述和进程调度的部分。`RENDER.store(render_flag, Ordering::Relaxed);`

我们定义了一个函数 `pit_interrupt_handler`，当PIT产生中断时，此处理程序会被调用，以执行与定时任务相关的功能。以下是它的工作流程：

1. 原子地增加PIT_TICKS的值，这意味着PIT产生了另一个tick。这个原子变量将被用于计算时间间隔、系统启动时间等。
2. 首先检查是否可以进行渲染（由原子量RENDER_OK确定），然后它检查是否需要在本次tick渲染。每40ms（1/25秒）渲染一次。
   - 若需要在本次tick渲染，则将RENDER标志设置为7。
4. 从RENDER加载当前的渲染标志。
5. 接下来检查每个渲染标志，并尝试获取与每个渲染任务相关的锁，以安全地进行渲染，防止死锁。
   - 尝试锁定鼠标光标，并更新鼠标状态。然后清除与鼠标相关的渲染标志。
   - 尝试锁定窗口管理器，并进行渲染。然后清除与窗口管理器相关的渲染标志。
   - 尝试锁定图形设备（GD），并在给定的坐标区域进行渲染。然后清除与GD相关的渲染标志。
6. 最后，更新渲染标志。

总结：此PIT中断处理程序主要关注渲染任务。当每次PIT中断发生时，它会增加一个tick计数。如果允许渲染，它将每40个ticks设置一系列的渲染标志。然后，处理程序检查这些标志，并尝试执行相关的渲染任务，这涉及锁定相关资源以避免并发问题。以下是这个程序的代码实现：

```
/// PIT中断处理程序
pub fn pit_interrupt_handler() {
    let time = PIT_TICKS.fetch_add(1, Ordering::Relaxed);

    // 每1/25秒渲染一次
    if RENDER_OK.load(Ordering::Relaxed) && time % 40 == 0 {
        RENDER.store(7, Ordering::Relaxed);
    }

    let mut render_flag = RENDER.load(Ordering::Relaxed);

    if render_flag > 0 {
        without_interrupts(|| {
            if let Some(mut mouse_lock) = MOUSE_CURSOR.try_lock() {
                mouse_lock.update_mouse();
                render_flag &= !4;
            }
            if let Some(mut window_lock) = WINDOW_MANAGER.try_lock() {
                window_lock.render();
                render_flag &= !2;
            }
            if let Some(mut gd_lock) = GD.try_lock() {
```

```
                gd_lock.render(0, 0, 600, 800);
                render_flag &= !1;
            }
        });
        RENDER.store(render_flag, Ordering::Relaxed);
    }
}
```

4.10.2.3. PIT Uptime

我们可以借助PIT中断触发时累计的ticks来计算系统的启动时间（Uptime）。

```
pub fn get_ticks() -> usize {
    PIT_TICKS.load(Ordering::Relaxed)
}

pub fn get_uptime() -> f64 {
    (get_ticks() as f64) * PIT_INTERVAL
}
```

4.10.2.4. PIT Sleep

我们可以借助PIT实现较大时间尺度（毫秒、秒）的Sleep功能。

首先，我们实现一个即将在`Sleep`函数中使用的`halt`函数。这个`halt`函数和普通的halt操作不同，它需要检查中断禁用情况，并保证PIT中断出现时，程序能顺利恢复。

函数的工作流程如下：

1. 检查当前中断是否被禁用。如果中断被禁用，设置`disabled`变量为`true`，标志`halt`结束后需要重新禁用中断。
2. 调用`interrupts::enable_and_hlt()`方法，首先启用中断（如果它们之前被禁用），然后执行`HLT`指令来暂停CPU。这意味着CPU将进入低功率模式，直到下一个中断发生。
3. 如果之前中断是被禁用的（即`disabled`为`true`），那么在执行`HLT`指令并从暂停状态唤醒后，再次禁用中断。

函数的代码实现如下：

```
/// Halt
pub fn halt() {
    let disabled = !interrupts::are_enabled();
    interrupts::enable_and_hlt();
    if disabled {
        interrupts::disable();
```

```
    }
}
```

接下来，我们来实现sleep函数。这是一个简单的函数，用于暂停当前线程或任务的执行指定的秒数。这个实现是基于前面提供的uptime和halt函数。

它首先调用uptime函数并加上指定的秒数seconds，从而得到唤醒的时间。然后，它将通过循环不断地检查当前的系统运行时间（通过uptime），并与预定的wakeup_time进行比较。只要当前时间小于唤醒时间，它就会调用halt函数，暂停CPU的执行，直到下一个中断发生。这种方法将不断地在检查时间和暂停CPU之间切换，直到达到预定的唤醒时间为止。

总结：sleep函数提供了一种方法，使当前线程或任务在指定的时间内暂停执行。这个实现相对简单，并利用了uptime来获取当前时间以及halt来暂停CPU。由于它基于uptime的轮询，所以它是低精度的，因为实际的休眠时间可能会比请求的时间稍微长一些。

以下是代码实现：

```
/// 等待指定的秒数（低精度）
pub fn sleep(seconds: f64) {
    let wakeup_time = uptime() + seconds;
    while uptime() < wakeup_time {
        halt();
    }
}
```

4.10.3. TSC

TSC是Time Stamp Counter（时间戳计数器）的缩写，它是一种用于测量和记录CPU的时钟周期数的寄存器，可以用于操作系统的管理和调度，如性能分析，定时器，同步等。TSC可以提供一个高精度和高分辨率的时间源，用于测量和记录操作系统和应用程序的执行时间，以及同步不同的事件和操作。

4.10.3.1. 初始化和读取TSC

TSC是一个64位的寄存器，它在每个CPU核心上都有一个副本。它在CPU上电或复位时开始计数，并且在每个时钟周期内递增一次。TSC可以通过RDTSC指令来读取其低32位和高32位的值。借助这个指令，我们可以读取TSC存储的时间。

我们首先定义了一个原子量CLOCKS_PER_NANOSECOND，用于保存每纳秒的时钟周期数：

```
pub static CLOCKS_PER_NANOSECOND: AtomicU64 = AtomicU64::new(0);
```

接下来，我们定义一个函数，用于读取TSC当前的值：

```
pub fn rdtsc() -> u64 {
    unsafe {
        arch::x86_64::_mm_lfence(); // 确保读取到的是最新的值
        arch::x86_64::_rdtsc()
    }
}
```

- `arch::x86_64::_mm_lfence()`执行了一个加载围栏指令，确保所有之前的读操作在继续执行之前都已完成，这样就可以确保读取TSC的操作获得的是最新的值。
- `arch::x86_64::_rdtsc()`返回当前TSC的值。

接下来，我们初始化TSC模块并测量每纳秒TSC的递增数量：

```
/// 初始化并记录TSC每纳秒的递增数
pub fn init() {
    let calibration_time = 250_000; // 0.25 seconds
    let a = rdtsc();
    super::sleep(calibration_time as f64 / 1e6);
    let b = rdtsc();
    CLOCKS_PER_NANOSECOND.store((b - a) / calibration_time,
Ordering::Relaxed);
}
```

首先，我们定义校准时间为250,000微秒（0.25秒）。接下来，读取并存储开始时的TSC值。然后，我们使当前线程休眠指定的时间，即0.25秒。然后，再次读取TSC的值。现在，将两次值之差除以校准时间，就得到了每纳秒TSC的递增数。这个值将被用于高精度计时。

4.10.3.2. TSC Wait

接下来，我们使用TSC完成高精度wait功能。它的作用和`Sleep`相同，但它可以用于以微秒和纳秒级别进行等待。我定义了一个公共函数`nanowait`，它接受一个`u64`类型的参数`nanoseconds`，表示需要等待的纳秒数。

它的工作流程如下所示：

1. 使用`rdtsc`函数获取TSC的当前值，并将其存储在`start`变量中。
2. 读取`CLOCKS_PER_NANOSECOND`（即每纳秒的TSC递增数）并与输入的`nanoseconds`相乘。这计算出了等待时间对应的TSC增量，存储在`delta`中。
3. 不断地循环检查TSC的当前值，并与`start`相减来得到已经过去的TSC增量。
   - 如果这个差值小于我们预期的`delta`，则继续执行循环。

- 在循环中，使用`hint::spin_loop()`提供一个提示给CPU，告诉它当前正在进行的操作是一个短暂的忙等待循环，从而允许CPU进行更有效的功率管理。

总结：`nanowait`函数使用TSC来等待精确的纳秒数，它依赖于TSC的高分辨率来提供准确的等待时间。这种方法比传统的休眠或延迟函数更加精确，尤其是在需要微秒或纳秒级精度的情况下。然而，这也意味着它会忙等待，这在某些情况下可能不是功率效率最高的方法，但是由于使用了`hint::spin_loop()`，它可以更有效地管理CPU功率。

它的代码实现如下所示：

```
pub fn nanowait(nanoseconds: u64) {
    let start = rdtsc();
    let delta = nanoseconds *
CLOCKS_PER_NANOSECOND.load(Ordering::Relaxed);
    while rdtsc() - start < delta {
        hint::spin_loop(); // 短途睡眠
    }
}
```

4.11. 进程管理

4.11.1. 相关常量

```
const MAX_PROCS: usize = 16;
const MAX_PROC_SIZE: usize = 10 << 20;
const MAX_FILE_HANDLES: usize = 64;
```

- **MAX_PROCS**：最大同时存在的进程数目；
- **MAX_PROC_SIZE**：每个进程所能使用的最大栈内存；
- **MAX_FILE_HANDLES**：每个进程所能打开的最多文件数量。

4.11.2. 相关数据结构

在这一部分中，我实现了一系列数据结构以保存进程的相关数据，及上下文信息。

`ProcessData` 用于保存进程运行的相关数据。

```
#[derive(Clone, Debug)]
pub struct ProcessData {
    env: BTreeMap<String, String>,
    dir: String,
    user: Option<String>,
    file_handles: Arc<Mutex<BTreeMap<usize, OpenFileHandle>>>,
}
```

`Registers` 用于保存进程运行上下文中的寄存器数据。

```
#[repr(align(8), C)]
#[derive(Debug, Clone, Copy, Default)]
pub struct Registers {
    pub r15: usize,
    pub r14: usize,
    pub r13: usize,
    pub r12: usize,
    pub r11: usize,
    pub r10: usize,
    pub r9: usize,
    pub r8: usize,
    pub rdi: usize,
    pub rsi: usize,
    pub rbp: usize,
    pub rbx: usize,
    pub rdx: usize,
    pub rcx: usize,
    pub rax: usize,
}
```

`Process` 用于保存进程。

```
#[derive(Clone, Debug)]
pub struct Process {
    pub id: usize,
    code_addr: u64,
    stack_addr: u64,
    entry_point: u64,
    page_table_frame: PhysFrame,
    stack_frame: InterruptStackFrameValue,
    registers: Registers,
    data: ProcessData,
    #[allow(unused)]
    parent: usize,
    allocator: Arc<Locked<LinkedListAllocator>>,
}
```

所有进程保存在 `PROCESS_TABLE` 中：

```
pub static ref PROCESS_TABLE: RwLock<[Box<Process>; MAX_PROCS]> = {
    let table: [Box<Process>; MAX_PROCS] = [(); MAX_PROCS].map(|_|
Box::new(Process::new(0)));
    RwLock::new(table)
};
```

4.11.3. Getter / Setter

为了方便操作系统内核读取和写入当前进程的相关数据，我实现了一系列Getter 和Setter，专门用于获取和设置当前进程的相关数据。

以下是他们的实现代码：

```rust
/// 获取当前进程PID
pub fn id() -> usize {
    PID.load(Ordering::SeqCst)
}

/// 设置当前进程PID
pub fn set_id(id: usize) {
    PID.store(id, Ordering::SeqCst);
}

/// 获取当前进程的环境变量
pub fn env(key: &str) -> Option<String> {
    let table = PROCESS_TABLE.read();
    let process = &table[id()];
    process.data.env.get(key).cloned()
}

/// 获取当前进程的环境变量
pub fn envs() -> BTreeMap<String, String> {
    let table = PROCESS_TABLE.read();
    let process = &table[id()];
    process.data.env.clone()
}

/// 获取当前进程的工作目录
pub fn dir() -> String {
    let table = PROCESS_TABLE.read();
    let process = &table[id()];
    process.data.dir.clone()
}

/// 获取当前进程的用户名
pub fn user() -> Option<String> {
    let table = PROCESS_TABLE.read();
    let process = &table[id()];
    process.data.user.clone()
}

/// 设置当前进程的环境变量
pub fn set_env(key: &str, val: &str) {
```

```rust
    let mut table = PROCESS_TABLE.write();
    let proc = &mut table[id()];
    proc.data.env.insert(key.into(), val.into());
}

/// 设置当前进程的工作目录
pub fn set_dir(dir: &str) {
    let mut table = PROCESS_TABLE.write();
    let proc = &mut table[id()];
    proc.data.dir = dir.into();
}

/// 设置当前进程的用户名
pub fn set_user(user: &str) {
    let mut table = PROCESS_TABLE.write();
    let proc = &mut table[id()];
    proc.data.user = Some(user.into())
}

/// 获取当前进程的代码地址
pub fn code_addr() -> u64 {
    let table = PROCESS_TABLE.read();
    let process = &table[id()];
    process.code_addr
}

/// 设置当前进程的代码地址
pub fn set_code_addr(addr: u64) {
    let mut table = PROCESS_TABLE.write();
    let proc = &mut table[id()];
    proc.code_addr = addr;
}

/// 偏移地址转换实际地址
pub fn ptr_from_addr(addr: u64) -> *mut u8 {
    let base = code_addr();
    if addr < base {
        (base + addr) as *mut u8
    } else {
        addr as *mut u8
    }
}

/// 获取当前进程的寄存器
pub fn registers() -> Registers {
    let table = PROCESS_TABLE.read();
    let process = &table[id()];
```

```rust
    process.registers
}

/// 设置当前进程的寄存器
pub fn set_registers(regs: Registers) {
    let mut table = PROCESS_TABLE.write();
    let proc = &mut table[id()];
    proc.registers = regs
}

/// 获取当前进程的栈帧
pub fn stack_frame() -> InterruptStackFrameValue {
    let table = PROCESS_TABLE.read();
    let proc = &table[id()];
    proc.stack_frame
}

/// 设置当前进程的栈帧
pub fn set_stack_frame(stack_frame: InterruptStackFrameValue) {
    let mut table = PROCESS_TABLE.write();
    let proc = &mut table[id()];
    proc.stack_frame = stack_frame;
}

pub unsafe fn page_table_frame() -> PhysFrame {
    let table = PROCESS_TABLE.read();
    let proc = &table[id()];
    proc.page_table_frame
}

pub unsafe fn set_page_table_frame(frame: PhysFrame) {
    let mut table = PROCESS_TABLE.write();
    let proc = &mut table[id()];
    proc.page_table_frame = frame
}

/// 获取当前进程的堆分配器
pub fn heap_allocator() -> Arc<Locked<LinkedListAllocator>> {
    let table = PROCESS_TABLE.read();
    let proc = &table[id()];
    proc.allocator.clone()
}

pub fn file_handles() -> Arc<Mutex<BTreeMap<usize, OpenFileHandle>>> {
    let table = PROCESS_TABLE.read();
    let proc = &table[id()];
    proc.data.file_handles.clone()
```

```
}

pub unsafe fn page_table() -> &'static mut PageTable {
    syskrnl::memory::create_page_table(page_table_frame())
}
```

4.11.4. 进程创建（spawn）

4.11.4.1. 初始化进程和装入程序

我设计了一个`create`函数来完成这个操作。这个函数的目的是为给定的二进制数据创建一个新的进程，并将程序装入到内存中。

以下是其工作流程：

1. **页表和内存管理**:
   - 新的页面被分配给新进程的页表。
   - 为新的进程创建一个新的页表，并将活动的内核页表赋值给新的页表。这是为了使新进程和内核共享某些内存区域。
   - 创建映射器（mapper）以管理页表的地址转换。
2. **代码和栈的设置**:
   - 计算新进程代码的地址和栈的地址。进程的大小被限制为 `MAX_PROC_SIZE`，并且栈紧随代码段之后。
3. **处理二进制格式**:
   - 检查给定的二进制是否为 ELF 格式（常见的可执行格式）或其他二进制格式（例如 BIN）。
   - 如果是 ELF 格式，解析它，并将代码复制到先前计算的地址。
   - 如果是 BIN 格式，代码将被直接复制。
4. **从父进程复制**:
   - 当新进程被创建时，它从当前的父进程（调用这个 `create` 函数的进程）克隆了一些数据。
5. **堆的初始化**:
   - 这个新进程有一个堆，它的大小被定义为 `DEFAULT_HEAP_SIZE`。
   - 为这个堆分配内存，并将其初始化。
6. **进程的创建和插入**:
   - 从 `PID_POOL` 中取出一个新的进程 ID。
   - 使用先前计算和设置的所有信息来创建一个新的 `Process` 对象。
   - 在 `PROCESS_TABLE` 中加入新的进程。
   - 返回新进程的 ID。

7. **错误处理**:
   - 如果在上述任何步骤中出现错误，函数会返回一个错误。

总体来说，这个函数的目的是为新的程序代码创建一个新的进程，并设置其内存、代码、数据和堆，然后将其加入到进程表中。以下是它的代码实现：

```
fn create(bin: &[u8]) -> Result<usize, ()> {
    let page_table_frame =
syskrnl::memory::heaped_frame_allocator().allocate_frame().expect("frame
alloc failed");
    let page_table = unsafe
{ syskrnl::memory::create_page_table(page_table_frame) };
    let kernel_page_table = unsafe
{ syskrnl::memory::active_page_table() };

    for (user_page, kernel_page) in
page_table.iter_mut().zip(kernel_page_table.iter()) {
        *user_page = kernel_page.clone();
    }

    let phys_mem_offset = unsafe { syskrnl::memory::PHYS_MEM_OFFSET };
    let mut mapper = unsafe { OffsetPageTable::new(page_table,
VirtAddr::new(phys_mem_offset)) };
    let _kernel_mapper = unsafe { OffsetPageTable::new(kernel_page_table,
VirtAddr::new(phys_mem_offset)) };

    // 特别地，打开用户页表的内核使用权限
    unsafe { fix_page_fault_in_userspace(&mut mapper) };

    let proc_size = MAX_PROC_SIZE as u64;
    let kernel_code_addr = CODE_ADDR.fetch_add(proc_size,
Ordering::SeqCst);
    let code_addr = kernel_code_addr;
    let stack_addr = code_addr + proc_size - 4096;
    // 紧跟在程序段后面
    debugln!("code_addr:  {:#x}", kernel_code_addr);
    debugln!("stack_addr: {:#x}", stack_addr);

    let mut entry_point = 0;
    let code_ptr = kernel_code_addr as *mut u8;
    let _code_size = bin.len();
    if bin[0..4] == ELF_MAGIC {
        // 进程代码是ELF格式的
        if let Ok(obj) = object::File::parse(bin) {
            // 先在用户页表上分配
            alloc_pages(&mut mapper, code_addr, proc_size as
```

```
usize).expect("proc mem alloc 754");
            // // 接下来，把用户页表的地址映射到内核页表上，并在内核页表上分配
            // let user_code_phys_frame =
mapper.translate_addr(VirtAddr::new(code_addr)).expect("Map fail 12341");
            // alloc_pages_to_known_phys(&mut kernel_mapper,
kernel_code_addr, proc_size as usize, user_code_phys_frame.as_u64(),
true).expect("proc mem alloc 564");

            entry_point = obj.entry();
            debugln!("entry_point:{:#x}", entry_point);
            for segment in obj.segments() {
                let addr = segment.address() as usize;
                if let Ok(data) = segment.data() {
                    debugln!(
                        "before flight? codeaddr,addr,datalen is {:#x},
{:#x},{}",
                        code_ptr as usize + addr,
                        addr,
                        data.len()
                    );
                    for (i, b) in data.iter().enumerate() {
                        unsafe {
                            //debugln!("code:       {:#x}",
code_ptr.add(addr + i) as usize);
                            //debugln!("WRITE: from {:p} to {:p}", b,
code_ptr.add(addr + i));
                            core::ptr::write(code_ptr.add(addr + i), *b)
                        }
                    }
                }
            }
        }
    } else if bin[0..4] == BIN_MAGIC {
        for (i, b) in bin.iter().skip(4).enumerate() {
            unsafe {
                core::ptr::write(code_ptr.add(i), *b);
            }
        }
    } else {
        // 文件头错误
        return Err(());
    }

    // 父进程
    let parent = {
        let table = PROCESS_TABLE.read();
        table[id()].clone()
```

```
    };

    let data = parent.data.clone();
    let registers = parent.registers;
    let stack_frame = parent.stack_frame;
    let parent = parent.id;

    // 初始化进程的堆分配器
    let mut allocator = LinkedListAllocator::new();
    let heap_addr = PROC_HEAP_ADDR.fetch_add(DEFAULT_HEAP_SIZE,
Ordering::SeqCst);

    // 先在用户页表上分配
    alloc_pages(&mut mapper, heap_addr as u64,
DEFAULT_HEAP_SIZE).expect("proc heap mem alloc failed 8520");
    // // 再映射到内核页表上
    // let heap_frame = mapper.translate_addr(VirtAddr::new(heap_addr as
u64)).expect("map fail 7897");
    // alloc_pages_to_known_phys(&mut kernel_mapper, heap_addr as u64,
DEFAULT_HEAP_SIZE, heap_frame.as_u64(), true).expect("proc heap mem alloc
failed 3652");

    unsafe { allocator.init(heap_addr, DEFAULT_HEAP_SIZE) };
    let allocator = Arc::new(Locked::new(allocator));

    if let Some(id) = PID_POOL.lock().pop_first() {
        let proc = Process {
            id,
            code_addr,
            stack_addr,
            data,
            registers,
            stack_frame,
            entry_point,
            parent,
            allocator,
            page_table_frame,
        };

        let mut table = PROCESS_TABLE.write();
        table[id] = Box::new(proc);

        Ok(id)
    } else {
        Err(())
    }
}
```

4.11.4.2. 切换到用户态并启动进程

在这一阶段中，我设计了exec方法，这个方法的目标是在用户空间中执行这一个新的进程。

以下是它的工作流程：

1. **初始化**:
   - 获取活动页表和页表映射器。这将允许函数对虚拟到物理的地址映射进行修改。
2. **处理传入参数**:
   - `args_ptr`, `args_len`, `args_cap` 定义了一个指针-长度对的向量。这里，它们被重建为 Rust 的 `Vec` 类型。
   - 使用这些指针-长度对，一个新的 `args` 向量被创建，其中每个元素是一个字符串引用（指向输入参数）。
   - 创建用于存放参数的新堆内存，并复制参数到这块内存上。确保新内存的对齐。
3. **调度与准备切换**:
   - 将新进程添加到调度器中。
   - 设置一个标志以指示需要进行任务切换。
4. **切换页表并跳转到新进程**:
   - 设置新进程的页表为活动页表。
   - 使用 `asm!` 宏执行汇编代码进行以下操作：
     - 关闭中断。
     - 准备栈和其它必要的寄存器来切换到新的上下文。
     - 使用 `iretq` 指令返回，这实际上会启动新的进程。

总体来说，这个函数首先准备了新进程的上下文（包括它的参数和页表），然后通过汇编代码执行实际的上下文切换，从而在用户模式下运行新进程的代码。以下是它的代码实现：

```
// 切换到用户空间并执行程序
fn exec(&mut self, args_ptr: usize, args_len: usize, args_cap: usize) {
    //syskrnl::allocator::alloc_pages(heap_addr, 1).expect("proc heap
alloc");
    let page_table = unsafe { page_table() };
    let phys_mem_offset = unsafe { syskrnl::memory::PHYS_MEM_OFFSET };
    let _mapper = unsafe { OffsetPageTable::new(page_table,
VirtAddr::new(phys_mem_offset)) };

    // 处理参数
    // 重建指针-长度对
    let ptr_len_pair: Vec<(usize, usize)> = unsafe
```

```
{ Vec::from_raw_parts(args_ptr as *mut (usize, usize), args_len,
args_cap) };
    // 重建参数数组
    let args: Vec<&str> = ptr_len_pair
        .iter()
        .map(|pair| unsafe {
            let slice = core::slice::from_raw_parts(pair.0 as *const u8,
pair.1);
            //debugln!("{:?}",slice);
            core::str::from_utf8(slice).expect("utf8 fail 8547")
        })
        .collect();
    // 在子进程分配用于存放参数的堆内存
    let mut addr = unsafe {
        self.allocator
            .lock()
            .alloc(core::alloc::Layout::from_size_align(1024,
1).expect("Layout problem 8741"))
    } as u64;
    // 将参数复制到这些内存上
    let vec: Vec<&str> = args
        .iter()
        .map(|arg| {
            let ptr = addr as *mut u8;
            addr += arg.len() as u64;
            unsafe {
                let s = core::slice::from_raw_parts_mut(ptr, arg.len());
                s.copy_from_slice(arg.as_bytes());
                //debugln!("{:?}",s);
                core::str::from_utf8_unchecked(s)
            }
        })
        .collect();
    let align = core::mem::align_of::<&str>() as u64;
    addr += align - (addr % align);
    let args = vec.as_slice();
    let ptr = addr as *mut &str;
    let args: &[&str] = unsafe {
        let s = core::slice::from_raw_parts_mut(ptr, args.len());
        s.copy_from_slice(args);
        s
    };
    let args_ptr = args.as_ptr() as u64;

    SCHEDULER.lock().add(self.clone(), 0);
    // if self.id != 1 {  // 不需要进入环三
    //     return;
```

```
    // }
    syskrnl::interrupts::SCHEDULE.store(true, Ordering::SeqCst);

    debugln!("LAUNCH");
    set_id(self.id);

    unsafe {
        let (_, flags) = Cr3::read();
        Cr3::write(self.page_table_frame, flags);

        asm!(
        "cli",         // 关中断
        "push {:r}",   // Stack segment (SS)
        "push {:r}",   // Stack pointer (RSP)
        "push 0x200", // RFLAGS with interrupts enabled
        "push {:r}",   // Code segment (CS)
        "push {:r}",   // Instruction pointer (RIP)
        "iretq",
        in(reg) syskrnl::gdt::GDT.1.user_data_selector.0,
        in(reg) self.stack_addr,
        in(reg) syskrnl::gdt::GDT.1.user_code_selector.0,
        in(reg) self.code_addr + self.entry_point,
        in("rdi") args_ptr,
        in("rsi") args.len(),
        );
    }
}
```

4.11.5. 进程上下文切换

保存进程上下文需要在中断处理程序中进行，因为只有在那里，我们才能访问到中断栈帧和寄存器信息。

4.11.5.1. 保存现场

在之前，我们定义了`wrap!`包装宏，这个宏可以捕捉进程上下文的寄存器信息，并传递到中断处理程序中。保存现场的核心就是把寄存器信息和栈帧保存下来。本系统中的中断`0x81`就可以完成这个工作，以下是其实现：

```
wrap!(save_context => wrapped_save_context);

extern "sysv64" fn save_context(stack_frame: &mut InterruptStackFrame,
regs: &mut Registers) {
    syskrnl::proc::set_stack_frame(**stack_frame);
    syskrnl::proc::set_registers(*regs);
```

```
    unsafe
{ pics::PICS.lock().notify_end_of_interrupt(interrupt_index(0x81) as u8) };
}
```

4.11.5.2. 恢复现场

恢复现场和保存现场所做的工作是一样的。我们定义了一个便捷的方法`switch_context_to`来完成这项工作。它的目的是切换到另一个进程的上下文。

以下是这个函数的工作流程:

1. **更换进程 ID**:
   - 使用 `syskrnl::proc::set_id(pid)` 来更换当前的进程 ID。
2. **获取上下文**:
   - 使用 `syskrnl::proc::stack_frame()` 获取目标进程的栈帧。
3. **更改页表**:
   - 使用 `Cr3::read()` 获取当前的 CR3 寄存器内容，该寄存器存储活动页表帧。
   - 使用 `syskrnl::proc::page_table_frame()` 获取目标进程的页表帧。
   - 通过 `Cr3::write()` 将 CR3 寄存器更换为目标进程的页表，这样下一条指令执行时，物理到虚拟地址的映射就会使用目标进程的页表。
4. **更换当前栈帧和寄存器状态**:
   - 使用 `core::ptr::write_volatile` 进行写操作，确保编译器不会优化掉这些写操作。这是因为这些写操作对于上下文切换至关重要。
   - 更新 `stack_frame` 为目标进程的栈帧。
   - 更新 `regs` 为目标进程的寄存器状态。

以下是其代码实现:

```
unsafe fn switch_context_to(pid: usize, stack_frame: &mut
InterruptStackFrame, regs: &mut Registers) {
    syskrnl::proc::set_id(pid);
    let sf = syskrnl::proc::stack_frame();
    //stack_frame.as_mut().write(sf);
    let (_, flags) = Cr3::read();
    Cr3::write(syskrnl::proc::page_table_frame(), flags);
    core::ptr::write_volatile(stack_frame.as_mut().extract_inner() as *mut
InterruptStackFrameValue, sf); // FIXME
    core::ptr::write_volatile(regs, syskrnl::proc::registers());
}
```

4.11.6. 进程退出

当进程即将退出时，我们需要进行一系列操作，来将进程的相关内存释放，并回收其堆内存。我们定义了一个函数 `exit`，用于表示一个进程的退出。

以下是这个函数的工作流程：

1. **读取进程表**：
   - 使用 `PROCESS_TABLE.read()` 获取当前系统中所有进程的只读引用。
2. **获取当前进程**：
   - 通过 `&table[id()]` 获取当前进程的引用。这里，`id()` 函数可能返回当前执行的进程的ID。
3. **释放进程所占内存**：
   - 使用 `syskrnl::allocator::dealloc_pages(proc.code_addr, MAX_PROC_SIZE);` 释放进程所使用的内存。我们已知每个进程最多可以使用 `MAX_PROC_SIZE` 的内存，并从 `proc.code_addr` 开始。
4. **回收进程ID**：
   - 使用 `PID_POOL.lock().insert(id());` 将当前进程的ID放回PID池中，这样它可以被重新使用。
5. **通知调度器进程已经退出**：
   - 使用 `SCHEDULER.lock().terminate(proc);` 通知系统调度器当前进程已经结束，并询问下一个应该运行的进程ID。
6. **返回下一个进程ID**：
   - 最后，函数返回下一个要执行的进程的ID。

以下是这个函数的代码实现：

```rust
/// 进程退出
pub fn exit() -> usize {
    let table = PROCESS_TABLE.read();
    let proc = &table[id()];
    syskrnl::allocator::dealloc_pages(proc.code_addr, MAX_PROC_SIZE);
    PID_POOL.lock().insert(id());
    let next_pid = SCHEDULER.lock().terminate(proc);
    debugln!("EXIT:{} -> {}", id(), next_pid);
    next_pid
}
```

4.12. 进程调度

进程调度在操作系统开发中是一个重要的主题，因为进程调度是操作系统管理和分配处理机资源的核心功能，它直接影响着操作系统的性能，效率和用户体验。

- 进程调度可以提高处理机的利用率，使得处理机不会因为等待某个进程而闲置，而是尽可能地为多个进程提供服务。这样可以提高系统的吞吐量和响应速度。
- 进程调度可以实现多任务和并发的功能，使得多个进程可以在同一时间段内交替执行，或者在多核处理机上同时执行。这样可以提高系统的功能性和并行性。

4.12.1. 特性设计

由于我们的操作系统可能还存在后续的迭代，因此我没有直接将调度器直接写死，而是采取了“实现特性”的方式，让它能够随时被更换。以下是我设计的 `ProcessScheduler` 特性：

```rust
/// 进程调度器
pub trait ProcessScheduler: Send + Debug {
    /// 注册新进程
    /// 返回 - PID
    fn add(&mut self, process: Process, priority: u32) -> usize;

    /// 取消进程
    fn terminate(&mut self, process: &Process) -> usize;

    /// 时间到轮换
    fn timeup(&mut self) -> usize;

    /// 主动退出运行
    fn giveup(&mut self) -> usize;

    /// 进程等待
    fn wait(&mut self) -> usize;

    /// 进程唤醒
    fn wakeup(&mut self, process: usize) -> usize;
}
```

这样，我们就能够将我们的进程调度器直接放入代码了：

```rust
lazy_static! {
    pub static ref SCHEDULER: Mutex<Box<dyn ProcessScheduler + 'static +
Send>> = { Mutex::new(Box::new(RoundRollScheduler::new())) };
}
```

4.12.2. 轮转算法实现

4.12.2.1. 数据结构定义

我们定义了 `RoundRollNode` 和 `RoundRollScheduler` 两个数据结构，分别表示链表中的节点及链表本身：

```rust
#[derive(Debug)]
struct RoundRollNode {
    /// pid
    pid: usize,

    /// 是否空闲
    empty: bool,

    /// 是否可跳过
    skip: bool,

    /// 指向下一任务的指针
    next: usize,

    /// 指向上一任务的指针
    prev: usize,
}

/// 轮转算法
#[derive(Debug)]
pub struct RoundRollScheduler {
    /// 进程表
    table: Vec<RoundRollNode>,

    /// PID对应表
    map: BTreeMap<usize, usize>,

    /// 头指针
    head: usize,

    /// 空节点指针
    empty: usize,

    /// 当前节点
    cursor: usize,
}
```

4.12.2.2. 初始化

`RoundRollScheduler` 的 `new` 方法可以被用于创建一个新的`RoundRollScheduler`实例。下面是这个方法的工作流程:

- 创建一个`RoundRollScheduler`结构体实例`s`。
- `table`: 初始化为一个包含一个`RoundRollNode`的向量。其中，`RoundRollNode::new()`方法返回一个新的`RoundRollNode`实例，其中`pid`, `next`, `prev`都默认为0，`empty`和`skip`为false。
- `map`: 初始化为一个空的`BTreeMap`。
- `head`, `empty`, `cursor`: 所有这些都被初始化为0，表示`table`中的第一个节点是当前活跃的节点、空闲节点和当前指针节点。
- 使用`s.map.insert(0, 0);`将一个0-0映射插入到`map`中。这意味着进程ID为0的进程在`table`的0位置。这是内核进程。

以下是其代码实现:

```rust
impl RoundRollNode {
    pub fn new() -> Self {
        RoundRollNode {
            pid: 0,
            empty: false,
            skip: false,
            next: 0,
            prev: 0,
        }
    }
}

impl RoundRollScheduler {
    pub fn new() -> Self {
        let mut s = RoundRollScheduler {
            table: vec![RoundRollNode::new()],
            map: BTreeMap::new(),
            head: 0,
            empty: 0,
            cursor: 0,
        };
        s.map.insert(0, 0); // 插入0-0映射
        s
    }
}
```

4.12.2.3. 从链表中分配一个新节点

我们定义了alloc方法来从链表中分配一个新节点。它的工作流程如下：

1. 检查当前的空闲节点是否确实是空闲的。
2. **如果当前空闲节点确实是空闲的**：
   - 将当前的空节点ID存储在id中。
   - 更新空节点指针self.empty，将其指向当前空节点的下一个节点。
   - 返回先前的空节点ID id。
3. **如果当前空闲节点不是空闲的**：
   - 在表末尾推入一个新的RoundRollNode。
   - 返回新节点的索引，也就是 self.table 的最后一位。

以下是其代码实现：

```rust
/// IN 调度一个新节点
fn alloc(&mut self) -> usize {
    if self.table[self.empty].empty {
        let id = self.empty;
        self.empty = self.table[id].next;
        id
    } else {
        self.table.push(RoundRollNode::new());
        self.table.len() - 1
    }
}
```

4.12.2.4. 归还一个节点到链表中

我们定义了dealloc方法来归还一个已经使用过的节点。它的工作流程如下：

1. **检查是否正在归还头结点**：
   - 如果node指向的是当前的头节点（即self.head == node），则将头指针self.head更新为当前头节点的下一个节点。
2. **调整前一个和后一个节点的指针**：
   - 获取节点node的前一个和后一个节点的索引，存储在(prev, next)中。
   - 更新前一个节点的next指针，使其指向node的下一个节点。
   - 更新后一个节点的prev指针，使其指向node的前一个节点。
   - 这两步操作从链表中移除了节点node。
3. **设置节点为empty**：
   - 将节点node的empty标志设置为true，表示此节点现在是空闲的。

4. **将节点添加到空闲链表**：
   - 将节点node的next指针设置为当前的空节点。
   - 更新self.empty指针，使其指向节点node，因此node现在是新的空闲节点。

它的代码实现如下：

```
/// IN 归还一个节点
fn dealloc(&mut self, node: usize) {
    if self.head == node {
        self.head = self.table[node].next;
    }
    let (prev, next) = (self.table[node].prev, self.table[node].next);
    self.table[prev].next = next;
    self.table[next].prev = prev;
    self.table[node].empty = true;
    self.table[node].next = self.empty;
    self.empty = node;
}
```

4.12.2.5. 加入一个新进程

我设计了一个方法 add，用于在RoundRollScheduler中添加一个新的进程。以下是它的工作流程：

1. **分配新节点**：
   - 使用self.alloc()方法分配一个新节点，返回其索引node。
2. **设置进程ID**：
   - 将新节点的pid设置为传入的process_id。
3. **插入新节点到链表**：
   - 首先，获取当前头节点的前一个节点索引prev。
   - 更新prev节点的next指针，使其指向新的节点node。
   - 更新头节点的prev指针，使其指向新的节点node。
   - 为新的节点node设置prev和next指针，使其分别指向prev节点和头节点。
4. **更新进程映射**：
   - 在self.map中插入一个新的条目，将process_id映射到新节点的索引node。

通过这个 add 方法，新的进程被添加到调度表的末尾，这也意味着在所有现有的进程之后运行它。这是轮转调度算法的一个典型特性，新进程被加入队列的尾部并等待其轮次来运行。它的代码实现如下：

```rust
/// 加入新进程
pub fn add(&mut self, process_id: usize) {
    let node = self.alloc();
    self.table[node].pid = process_id;
    let prev = self.table[self.head].prev;
    self.table[prev].next = node;
    self.table[self.head].prev = node;
    self.table[node].prev = prev;
    self.table[node].next = self.head;
    self.map.insert(process_id, node);
}
```

4.12.2.6. 移除一个旧进程

我设计了一个方法 `remove`，用于从 `RoundRollScheduler` 中移除一个已存在的进程。以下是它的工作流程：

1. **查找进程**：
   - 尝试从`map`中获取与给定`process_id`关联的节点索引。如果找到，`Some(node)`将被匹配，其中`node`是这个进程的节点索引。
2. **检查当前节点**：
   - 如果`node`是当前正在执行的进程（即`self.cursor == node`），那么调用`self.step()`。`step`方法是一个移动游标到下一个进程的方法。
3. **归还节点**：
   - 使用`self.dealloc(node)`方法归还此节点，这将从链表中移除此节点，并将其标记为空闲，以供后续重用。
4. **从映射中移除进程**：
   - 从`map`中移除与`process_id`关联的条目。

通过这个 `remove` 方法，指定的进程将被从调度器中移除，其占用的节点将被归还，以便将来重用。如果要移除的进程正处于执行状态，调度器可能会切换到另一个进程。它的代码实现如下：

```rust
/// 移除旧进程
pub fn remove(&mut self, process_id: usize) {
    if let Some(node) = self.map.get(&process_id) {
        let node = *node;
        if self.cursor == node {
            self.step();
        }
        self.dealloc(node);
```

```
        self.map.remove(&process_id).unwrap();
    }
    // debugln!("Remove Process: {}, now:{}, snapshot:
{:?}",process_id,self.now(),self.table)
}
```

4.12.2.7. 步进到下一个进程

我设计了一个方法 `step`，用于在 `RoundRollScheduler` 中步进到下一个进程。以下是它的工作流程：

1. **更新游标**：
   - 首先，`self.cursor`被更新为它所指向的节点的`next`指针。这意味着`cursor`现在指向链表中的下一个节点，或者说下一个进程。
2. **跳过可跳过的进程**：
   - 接下来，有一个`while`循环，它检查新的`cursor`指向的进程是否被标记为`skip`。如果是，这意味着这个进程应该被跳过，不应该被执行。
   - 在这种情况下，`cursor`再次被更新为它所指向的节点的`next`指针，即指向链表中的下一个节点。
   - 循环将继续这个过程，直到找到一个没有被标记为`skip`的进程，或者遍历完所有进程。
3. **返回当前进程**：
   - `self.now()`被调用并返回。`now`方法返回当前游标`cursor`指向的进程的ID。

总的来说，`step`方法将调度器从当前进程移动到下一个应该执行的进程。它会跳过任何被标记为`skip`的进程，直到找到下一个应该执行的进程，然后返回其ID。它的代码实现如下：

```
/// 向后进一步
pub fn step(&mut self) -> usize {
    self.cursor = self.table[self.cursor].next;
    while self.table[self.cursor].skip {
        self.cursor = self.table[self.cursor].next;
    }
    self.now()
}
```

4.12.2.8. `ProcessScheduler` 特性实现

特性实现的过程相对简单，只需要综合运用上述的几个方法即可。以下是相关的实现代码：

```rust
impl ProcessScheduler for RoundRollScheduler {
    fn add(&mut self, process: Process, _priority: u32) -> usize {
        self.add(process.id);
        self.cursor = *self.map.get(&process.id).unwrap();
        self.now()
    }

    fn terminate(&mut self, process: &Process) -> usize {
        self.remove(process.id);
        self.now()
    }

    fn timeup(&mut self) -> usize {
        self.step()
    }

    fn giveup(&mut self) -> usize {
        self.step()
    }

    fn wait(&mut self) -> usize {
        self.table[self.cursor].skip = true;
        self.step()
    }

    fn wakeup(&mut self, process: usize) -> usize {
        // debugln!("Process Wakeup:{}",process);
        if let Some(node) = self.map.get(&process) {
            self.table[*node].skip = false;
        }
        self.now()
    }
}
```

4.12.3. 调度并切换进程

接下来，我们只需在PIT中断的中断处理程序中调用调度器就可以了。函数`clock_handler`是一个时钟中断处理程序，当时钟中断发生时，这个函数会被调用。它主要进行了一些与调度、事件处理和中断相关的操作。以下是它的工作流程：

1. **调用IRQ处理程序**：
   - 通过一个锁定的全局变量`IRQ_HANDLERS`来访问IRQ处理程序的列表，并执行PIT中断对应处理程序：`handlers[0]()`。
2. **检查是否有进程需要唤醒**：

- `time::check_wakeup()`会检查是否有进程达到了唤醒的时间。如果有，它返回这个进程的PID。随后，这个进程会被调度程序唤醒。这会在事件和唤醒一节详细叙述。

3. **检查是否需要调度**：
   - 如果`SCHEDULE`标志被设置，并且自上次调度以来已经过去了超过10个ticks，那么可能需要进行一次新的进程调度。
   - 如果`NO_SCHEDULE`标志被设置，说明当前不应该调度新的进程。但如果自上次调度以来已经过去了超过1000个ticks，那么将强制进行调度。
   - 使用`SCHEDULER.lock().timeup()`来获取下一个应该被调度的进程的PID。
   - 如果下一个要被调度的进程与当前进程不同，会保存当前进程的状态并进行上下文切换，切换到新的进程。
   - 检查是否有与下一个进程关联的事件数据，如果有，更新`regs.rax`。这会在事件和唤醒一节详细叙述。
4. **通知中断结束**：
   - 通知PIC（可编程中断控制器）时钟中断已经被处理完毕。

这个函数的主要作用是，当时钟中断发生时，根据需要执行相应的IRQ处理、唤醒进程、调度新的进程或进行其他与中断相关的操作。它确保了操作系统可以定时进行进程调度，以及处理与时钟中断相关的其他任务。以下是这个函数的具体实现：

```
pub static SCHEDULE: AtomicBool = AtomicBool::new(false);
static LAST_SCHEDULE: AtomicUsize = AtomicUsize::new(0);
pub static NO_SCHEDULE: AtomicBool = AtomicBool::new(false);

wrap!(clock_handler => wrapped_clock_handler);

/// 时钟中断处理程序
extern "sysv64" fn clock_handler(stack_frame: &mut InterruptStackFrame,
regs: &mut Registers) {
    // 先把时钟发过去
    let handlers = IRQ_HANDLERS.lock();
    handlers[0]();

    if let Some(pid) = time::check_wakeup() {
        SCHEDULER.lock().wakeup(pid);
    }

    if SCHEDULE.load(Ordering::SeqCst) && ticks() -
LAST_SCHEDULE.load(Ordering::SeqCst) > 10 {
        let mut schedule = || {
            if NO_SCHEDULE.load(Ordering::SeqCst) {
                if ticks() - LAST_SCHEDULE.load(Ordering::SeqCst) > 1000 {
                    // 强行恢复调度
```

```
                    NO_SCHEDULE.store(false, Ordering::SeqCst);
                } else {
                    return;
                }
            }

            let next_pid = SCHEDULER.lock().timeup();

            if next_pid != syskrnl::proc::id() {
                syskrnl::proc::set_stack_frame(**stack_frame);
                syskrnl::proc::set_registers(*regs);

                unsafe {
                    switch_context_to(next_pid, stack_frame, regs);
                }
            }

            let lock = syskrnl::event::EVENT_DATA.lock();
            if lock.contains_key(&next_pid) {
                regs.rax = *lock.get(&next_pid).unwrap();
            }
            drop(lock);

            LAST_SCHEDULE.store(ticks(), Ordering::SeqCst);
        };

        schedule();
    }

    unsafe { pics::PICS.lock().notify_end_of_interrupt(interrupt_index(0)
as u8) };
}
```

4.13. 系统调用

在Linux中，系统调用是一种让用户空间的进程可以请求内核提供服务的机制。系统调用的实现原理大致如下：

- 用户空间的进程通过调用C库函数或者直接使用汇编指令，发出一个特定的软件中断，如`int $0x80`、`sysenter`或`syscall`，来触发系统调用。
- CPU收到中断信号后，从用户态切换到内核态，并保存当前的寄存器和环境状态，然后跳转到内核中预先定义好的中断处理程序，如`system_call()`。

- system_call()函数根据用户空间传递过来的系统调用号，从一张系统调用表sys_call_table中找到对应的服务例程的地址，并调用该例程。系统调用号是每个系统调用的唯一标识，通常存放在eax寄存器中。
- 服务例程执行相应的功能，如读写文件、创建进程、管理内存等，并将返回值存放在eax寄存器中。服务例程结束后，返回到system_call()函数。
- system_call()函数恢复之前保存的寄存器和环境状态，并使用一条特定的指令，如iret、sysexit或sysret，从内核态返回到用户态，继续执行用户空间的进程。

本项目中实现的系统调用的过程大致相同。

4.13.1. 系统调用发送

本项目提供了一个名为syscall!的宏来帮助程序发送系统调用。这个宏会根据接收到的参数的数量，调用不同的系统调用辅助函数来发送系统调用。

具体来说，这个宏名为syscall，有多个变种，可以传递0到4个参数。根据传入的参数数量，该宏会调用不同的辅助函数(syscall0, syscall1, ... syscall4)。每个变种都将参数转换为usize类型，并传递给相应的函数。

对辅助函数而言：
- 这些辅助函数使用了asm!宏，用于内联汇编。它们允许Rust代码直接使用汇编语言。
- 每个函数都有一个n参数，表示系统调用的编号，以及最多4个其他参数。
- 使用int 0x80指令来执行x86的系统调用。此指令是一个软件中断，会触发中断80h，该中断在本项目中用于系统调用。
- 各个参数使用了不同的寄存器，例如rdi, rsi, rdx和r8。这是x86-64 本项目的系统调用传参约定。
- 调用完成后，函数将返回值放在rax寄存器中。

这套代码的目的是提供一个方便的系统调用机制，允许Rust代码通过一个简化的接口执行系统调用。通过使用syscall!宏，开发者可以简洁地执行一个系统调用，而不必关心背后复杂的细节和寄存器管理。

以下是这些代码的具体实现：

```
#[macro_export]
macro_rules! syscall {
    ($n:expr) => {
        $crate::syscall::syscall0($n as usize)
    };
    ($n:expr, $a1:expr) => {
        $crate::syscall::syscall1($n as usize, $a1 as usize)
```

```
    };
    ($n:expr, $a1:expr, $a2:expr) => {
        $crate::syscall::syscall2($n as usize, $a1 as usize, $a2 as usize)
    };
    ($n:expr, $a1:expr, $a2:expr, $a3:expr) => {
        $crate::syscall::syscall3($n as usize, $a1 as usize, $a2 as usize,
$a3 as usize)
    };
    ($n:expr, $a1:expr, $a2:expr, $a3:expr, $a4:expr) => {
        $crate::syscall::syscall4(
            $n as usize,
            $a1 as usize,
            $a2 as usize,
            $a3 as usize,
            $a4 as usize,
        )
    };
}

/***
 * 发送系统调用
 */

#[doc(hidden)]
pub unsafe fn syscall0(n: usize) -> usize {
    let res: usize;
    asm!(
    "int 0x80", in("rax") n,
    lateout("rax") res
    );
    res
}

#[doc(hidden)]
pub unsafe fn syscall1(n: usize, arg1: usize) -> usize {
    let res: usize;
    asm!(
    "int 0x80", in("rax") n,
    in("rdi") arg1,
    lateout("rax") res
    );
    res
}

#[doc(hidden)]
pub unsafe fn syscall2(n: usize, arg1: usize, arg2: usize) -> usize {
    let res: usize;
```

```
    asm!(
    "int 0x80", in("rax") n,
    in("rdi") arg1, in("rsi") arg2,
    lateout("rax") res
    );
    res
}

#[doc(hidden)]
pub unsafe fn syscall3(n: usize, arg1: usize, arg2: usize, arg3: usize) ->
usize {
    let res: usize;
    asm!(
    "int 0x80", in("rax") n,
    in("rdi") arg1, in("rsi") arg2, in("rdx") arg3,
    lateout("rax") res
    );
    res
}

#[doc(hidden)]
pub unsafe fn syscall4(n: usize, arg1: usize, arg2: usize, arg3: usize,
arg4: usize) -> usize {
    let res: usize;
    asm!(
    "int 0x80", in("rax") n,
    in("rdi") arg1, in("rsi") arg2, in("rdx") arg3, in("r8") arg4,
    lateout("rax") res
    );
    res
}
```

4.13.2. 系统调用接收

系统调用接收的过程分为中断处理函数函数处理、`dispatcher`函数解包并调用服务例程、例程处理系统调用三个步骤。

4.13.2.1. 中断处理函数

我们定义了名为`syscall_handler`的系统调用处理函数。当一个系统调用被触发时（通过之前提到的`int 0x80`指令），这个函数将被执行。

这个函数的工作流程如下所示：

1. **从寄存器中获取参数**:

- 使用System V ABI约定，系统调用的参数存储在不同的寄存器中：`rdi`, `rsi`, `rdx`, `r8` 和 `rax`。此代码从这些寄存器中读取参数，并保存在局部变量中。

2. **处理系统调用**:
   - 判断系统调用号n是否等于`SPAWN`。如果是，则保存当前的堆栈帧和寄存器状态。这是因为`SPAWN`系统调用是用于启动新进程的，因此需要保存父进程的状态。
   - 调用`syskrnl::syscall::dispatcher`函数来实际处理这个系统调用，并将返回值保存在`res`中。
   - 如果系统调用号n是`EXIT`，这意味着当前进程已经结束执行，操作系统需要切换到另一个进程。它通过`switch_context_to`函数切换进程上下文。
   - 如果不是`EXIT`系统调用，则简单地将返回值`res`存储到`rax`寄存器中，这样调用方可以获取系统调用的结果。
3. **结束中断处理**:
   - 使用`pics::PICS.lock().notify_end_of_interrupt(0x80)`来通知中断控制器中断处理已经完成。这将允许其他中断被处理。

总的来说，这个`syscall_handler`函数是一个典型的系统调用处理流程。它从寄存器中读取系统调用的参数，处理系统调用，并在完成后更新寄存器状态或进行上下文切换（如果需要的话）。这个函数的代码实现如下所示：

```
wrap!(syscall_handler => wrapped_syscall_handler);

extern "sysv64" fn syscall_handler(stack_frame: &mut InterruptStackFrame,
regs: &mut Registers) {
    // The registers order follow the System V ABI convention
    let n = regs.rax;
    let arg1 = regs.rdi;
    let arg2 = regs.rsi;
    let arg3 = regs.rdx;
    let arg4 = regs.r8;

    if n == cinea_os_sysapi::call::SPAWN {
        // 保存现场
        syskrnl::proc::set_stack_frame(**stack_frame);
        syskrnl::proc::set_registers(*regs);
    }

    let res = syskrnl::syscall::dispatcher(n, arg1, arg2, arg3, arg4);

    if n == cinea_os_sysapi::call::EXIT {
        // 恢复现场
        debugln!("恢复现场");
        debugln!("额外信息: {:?}", SCHEDULER.lock());
```

```
        let next_pid = res;
        unsafe {
            switch_context_to(next_pid, stack_frame, regs);
        }
    } else {
        regs.rax = res;
    }

    unsafe { pics::PICS.lock().notify_end_of_interrupt(0x80) };
}
```

4.13.2.2. 服务例程调用

我们创建了dispatcher函数来分发系统调用，它根据给定的系统调用编号(syscall_id)来确定应该执行哪个系统调用，并使用传入的参数调用相应的服务。

这个函数的工作流程如下：

1. **中断禁用**:
   - interrupts::without_interrupts确保系统调用处理过程中不会被其他中断打断。这提供了一个原子执行环境，确保系统调用的完整执行。
2. **系统调用的匹配**:
   - 使用match语句来根据syscall_id确定哪个系统调用应该被执行。
   - 对于每个匹配的syscall_id，相应的服务函数被调用，传入所需的参数。
3. **系统调用的具体处理**:
   - 例如，当syscall_id为EXIT时，service::exit函数被调用，并将arg1作为退出码传入。
   - 如果syscall_id为SPAWN，则使用四个参数调用service::spawn函数。
   - 对于某些系统调用，例如SLEEP, FREE, NO_SCHE, 和CON_SCHE，它们在完成后返回0作为默认的返回值。
   - unimplemented!()宏被用于那些还未实现的系统调用，它将触发一个未实现的错误。
   - 如果遇到未知的syscall_id，函数将触发一个panic，表示发生了一个严重错误。

综上所述，这个dispatcher函数是一个中心分发点，用于将系统调用请求转发给相应的服务函数。这是操作系统设计中的常见模式，可以清晰、高效地处理多种系统调用。这个函数的代码实现如下：

```
pub fn dispatcher(syscall_id: usize, arg1: usize, arg2: usize, arg3: usize,
arg4: usize) -> usize {
    interrupts::without_interrupts(|| match syscall_id {
```

```
        EXIT => service::exit(ExitCode::from(arg1)),
        SPAWN => service::spawn(arg1, arg2, arg3, arg4) as usize,
        INFO => service::info(arg1),
        DUP => unimplemented!(),
        DELETE => unimplemented!(),
        STOP => unimplemented!(),
        SLEEP => {
            service::sleep(f64::from_bits(arg1 as u64));
            0
        }
        LOG => service::log(arg1, arg2),
        ALLOC => service::alloc(arg1, arg2),
        FREE => {
            service::free(arg1, arg2, arg3);
            0
        }
        PANIC => service::panic(arg1),
        NO_SCHE => {
            service::stop_schedule();
            0
        }
        CON_SCHE => {
            service::restart_schedule();
            0
        }
        TEST_SERDE => service::test_serde(arg1),
        LIST => service::list(arg1),
        OPEN => service::open(arg1),
        WRITE_ALL => service::write_all(arg1),
        READ => service::read(arg1),
        WRITE_PATH => service::write_path(arg1),
        READ_PATH => service::read_path(arg1),
        SPAWN_FROM_PATH => service::spawn_from_path(arg1),
        CREATE_WINDOW => service::create_window(arg1),
        DISPLAY_FONT_STRING => service::display_font_string(arg1),
        LOAD_FONT => service::load_font(arg1),
        DESTROY_WINDOW => service::destroy_window(),
        REGISTER_TIMER => service::register_timer(arg1),
        GUI_SUBSCRIBE_TIME_UPDATE => service::gui_time_update_register(),
        READ_TIME => service::read_time(),
        GUI_SUBSCRIBE_KEYBOARD => service::gui_time_update_register(),
        _ => panic!("unknown syscall id: {}", syscall_id),
    })
}
```

4.13.3. 复杂数据传递

如读者所见，迄今为止我们只能在系统调用见通过寄存器传递一些简单的64位整数。不过，如果我们把数据保存在公共的内存上，然后将数据的指针作为64位的参数传递，就可以传递更加复杂的数据了。

本项目利用Serde框架和Postcard库来实现在系统调用之间数据的高效传递。数据被序列化后再传递，可以避免由于调用号不匹配等引发的严重未定义行为；这是因为如果期望的数据格式和实际获得的数据不匹配，Serde会检测到不一致并抛出错误。

4.13.3.1. 包含序列化的调用

我们设计了一个函数，用于在发送调用前将复杂数据序列化。此函数`syscall_serialized`为给定的数据项`data`进行序列化，并返回一个表示序列化数据的内存地址的整数。

以下是这个函数的工作流程：

1. **序列化**：
   - 使用`postcard::to_allocvec`方法将`data`序列化到一个`Vec<u8>`。
2. **获取序列化数据的内存地址**：
   - 使用`into_raw_parts`方法将`vecdata`转化为其三个原始组成部分：指向数据的指针，容量和长度。
   - 创建一个`addr_slice`，该切片包含这三个部分的信息。这可以帮助用户更远地定位整个序列化的数据结构。
   - 使用`into_raw_parts`方法获取`addr_slice`的指针。
3. **返回结果**：
   - 返回`final_ptr`指针值作为`usize`类型。

以下是这个函数的具体实现：

```rust
pub fn syscall_serialized<T>(data: &T) -> usize where T: Serialize {
    let vecdata = postcard::to_allocvec(data).unwrap();
    let addr = vecdata.into_raw_parts();
    let addr_slice = vec![addr.0 as usize, addr.1, addr.2];
    let final_ptr = addr_slice.into_raw_parts().0;

    final_ptr as usize
}
```

此外，我们还设计了一个宏，来进一步减少进行这种系统调用的步骤：

```
#[macro_export]
macro_rules! syscall_with_serialize {
    ($call:expr,$obj:expr) => {
        {
            let _encoded = $crate::call::syscall_serialized(&$obj);
            unsafe { $crate::syscall!($call, _encoded) };
        }
    };
}
```

4.13.3.2. 返回序列化数据的调用

这一步，我们设计了两个函数来完成返回数据的反序列化。

首先是`syscall_deserialized_prepare`函数。它的工作流程如下：

- 使用`Vec::from_raw_parts`函数，从传入的`ptr`恢复`addr_slice`。
- 使用`addr_slice`中的三个值（指针，长度，容量）来恢复序列化的数据的`Vec<u8>`。

接下来是`syscall_deserialized`函数。它的工作流程如下：

- 使用`postcard::from_bytes`方法，从传入的`vec_data`中反序列化数据。

这两个函数的组合是为了从一个原始指针恢复出序列化的数据，并将这些数据反序列化回其原始格式。这与`syscall_serialized`是相反的操作。它们的代码实现如下所示：

```
pub fn syscall_deserialized_prepare(ptr: usize) -> Vec<u8> {
    let addr_slice = unsafe { Vec::from_raw_parts(ptr as *mut usize, 3, 3)
 };
    unsafe { Vec::from_raw_parts(addr_slice[0] as *mut u8, addr_slice[1],
addr_slice[2]) }
}

pub fn syscall_deserialized<'de, T>(vec_data: &'de Vec<u8>) -> Result<T,
postcard::Error> where T: Deserialize<'de> {
    postcard::from_bytes(vec_data.as_slice())
}
```

同样的，我设计了一个宏来减少调用代码量：

```
#[macro_export]
macro_rules! syscall_with_deserialize {
    ($($arg:tt)*) => {
        {
            let _ret = unsafe { $crate::syscall!($($arg)*) };
            let _ret_vec_data =
```

```
$crate::call::syscall_deserialized_prepare(_ret);
            $crate::call::syscall_deserialized(&_ret_vec_data)
        }
    };
}
```

以及，对同时需要序列化参数和反序列化结果的调用，我也设计了一个宏：

```
#[macro_export]
macro_rules! syscall_with_serdeser {
    ($call:expr,$obj:expr) => {
        {
            let _encoded = $crate::call::syscall_serialized(&$obj);
            let _ret = unsafe { $crate::syscall!($call, _encoded) };
            let _ret_vec_data =
$crate::call::syscall_deserialized_prepare(_ret);
            $crate::call::syscall_deserialized(&_ret_vec_data)
        }
    };
}
```

4.14. 协程

操作系统开发中，协程的作用是提高系统的并发性能和编程的简便性。协程是一种比线程更加轻量级的存在，它可以在用户空间内实现多任务的切换和调度，而不需要涉及到内核态的上下文切换。协程可以看做是一种用户级线程的实现，它可以利用I/O阻塞的时间来执行其他任务，从而提高系统的吞吐量和响应速度。协程还可以按照同步的思维来写异步的代码，避免了回调地狱和复杂的状态管理。

4.14.1. 协程定义

首先，我们定义协程的编号：

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord)]
struct TaskId(u64);

impl TaskId {
    fn new() -> Self {
        static NEXT_ID: AtomicU64 = AtomicU64::new(0);
        TaskId(NEXT_ID.fetch_add(1, Ordering::Relaxed))
    }
}
```

可以看到，我们使用了静态的原子变量 NEXT_ID 来保证每次分配的ID都是唯一的。每次调用 new 方法时，都会递增 NEXT_ID。

接下来，我们定义协程本身：

```
pub struct Task {
    id: TaskId,
    future: Pin<Box<dyn Future<Output = ()>>>,
}

impl Task {
    pub fn new(future: impl Future<Output = ()> + 'static) -> Self {
        Self {
            id: TaskId::new(),
            future: Box::pin(future),
        }
    }

    fn poll(&mut self, context: &mut Context) -> Poll<()> {
        self.future.as_mut().poll(context)
    }
}
```

在协程的结构体中，每个协程都有一个 `TaskId` 作为标识；`future` 字段则包含了一个将被执行的异步任务。这里用到了 `Pin<Box<dyn Future<Output = ()>>>`，表示一个固定的、堆上分配的、返回 `()` 类型的异步任务。

协程还提供了new和poll方法。poll方法允许外部调用者尝试执行任务的 `future`，并返回其执行状态（已完成或未完成）；而这个函数的context 参数提供了任务执行所需的上下文信息，例如 `Waker`。

总的来说，这段代码描述了一个简单的任务调度器的构建块，其中任务由其ID标识并关联一个要执行的异步操作。当操作系统或任务调度器想要执行一个任务时，它可以使用 `poll` 方法来推进任务的执行。

4.14.2. 协程执行器

接下来，我们定义协程的执行器。它是一个异步代码调度器的核心部分。

```
pub struct Executor {
    tasks: BTreeMap<TaskId, Task>,
    task_queue: Arc<ArrayQueue<TaskId>>,
    waker_cache: BTreeMap<TaskId, Waker>,
}
```

这里，我们定义了 `Executor` 结构体。它拥有以下字段：

- **tasks**：使用BTreeMap来存储所有任务，任务ID作为键，实际任务Task为值。

- **task_queue**：一个固定大小的队列，用于存储尚未完成的任务ID。这里使用`Arc<ArrayQueue<TaskId>>`，`Arc`允许在多个地方共享这个队列，而不担心生命周期问题。
- **waker_cache**：缓存任务的唤醒器 (`Waker`)。当任务被暂停并稍后重新唤醒时，这些唤醒器是必要的。

```
pub fn new() -> Self {
    Executor {
        tasks: BTreeMap::new(),
        task_queue: Arc::new(ArrayQueue::new(100)),
        waker_cache: BTreeMap::new(),
    }
}
```

这是`Executor`的`new`方法。它创建一个新的`Executor`实例，初始化其字段。

```
pub fn spawn(&mut self, task: Task) {
    let task_id = task.id;
    if self.tasks.insert(task.id, task).is_some() {
        panic!("同名任务已经存在！");
    }
    self.task_queue.push(task_id).expect("队列满");
}
```

这是`Executor`的`spawn`方法。它将一个新的任务`Task`添加到`tasks`字典，并将其ID加入到`task_queue`；如果一个与给定ID相同的任务已经存在，则会触发一个panic。

```
fn run_ready_tasks(&mut self) {
    // 解构self以防止所有权检查报错
    let Self {
        tasks,
        task_queue,
        waker_cache,
    } = self;

    while let Some(task_id) = task_queue.pop() {
        let task = match tasks.get_mut(&task_id) {
            Some(task) => task,
            None => continue,
        };
        let waker = waker_cache.entry(task_id).or_insert_with(||
TaskWaker::new(task_id, task_queue.clone()));
        let mut context = Context::from_waker(waker);
        match task.poll(&mut context) {
```

```
            Poll::Ready(()) => {
                tasks.remove(&task_id);
                waker_cache.remove(&task_id);
            }
            Poll::Pending => {}
        }
    }
}
```

这是Executor的run_ready_tasks方法。它的工作流程如下：

- 从task_queue中取出任务ID并运行它们，直到队列为空或任务暂时不能运行。
- 使用waker_cache来获取或创建任务的唤醒器。
- 运行任务并检查其状态：
  - 如果任务已完成（Poll::Ready），则从tasks和waker_cache中移除它。
  - 如果任务还在运行中（Poll::Pending），则将其保留在队列中，以便稍后再次轮询。

```
fn sleep_if_idle(&self) {
    use x86_64::instructions::interrupts::{disable, enable,
enable_and_hlt};
    // 关中断
    disable();

    if self.task_queue.is_empty() {
        enable_and_hlt();
    } else {
        enable();
    }
}
```

这是Executor的sleep_if_idle方法。它的工作流程如下：

- 使用x86_64架构特定的指令来处理系统休眠。
- 如果task_queue为空，说明系统当前没有要运行的任务，所以系统会进入休眠状态，等待中断来唤醒它。
- 否则，如果有任务在队列中等待，它将重新启用中断并继续执行。

```
pub fn run(&mut self) -> ! {
    loop {
        self.run_ready_tasks();
        self.sleep_if_idle();
    }
}
```

这是Executor的run方法。它是Executor的主循环。它不断运行就绪的任务并在系统空闲时休眠。

总的来说，这段代码描述了一个协程调度器的基础实现。在此基础上，协程可以并行执行，并在等待I/O时出让CPU给其他协程。

4.14.3. 协程唤醒器

唤醒器是异步编程中的另一个关键组件。当一个协程（或Future）不能立即完成并需要稍后再次尝试时，Waker用于通知执行器唤醒和重新轮询协程。

我们定义唤醒器如下：

```
struct TaskWaker {
    task_id: TaskId,
    task_queue: Arc<ArrayQueue<TaskId>>,
}
```

TaskWaker结构体有如下两个字段：

- task_id: 要被唤醒的任务的ID。
- task_queue: 存储待运行任务ID的队列。

接下来，我们定义TaskWaker的方法：

```
impl TaskWaker {
    fn wake_task(&self) {
        self.task_queue.push(self.task_id).expect("任务队列满");
    }

    fn new(task_id: TaskId, task_queue: Arc<ArrayQueue<TaskId>>) -> Waker {
        Waker::from(Arc::new(TaskWaker { task_id, task_queue }))
    }
}
```

这两个方法：

- new: 这个方法创建一个新的TaskWaker并从它生成一个Waker。这样做是因为Waker需要一个实现了Wake trait的对象。
- wake_task: 这个方法将任务ID推送到task_queue，从而通知执行器在稍后的某个时间点再次尝试运行这个任务。

接下来，我们为TaskWaker实现特性Wake的方法：

```
impl Wake for TaskWaker {
    fn wake(self: Arc<Self>) {
```

```
        self.wake_task();
    }

    fn wake_by_ref(self: &Arc<Self>) {
        self.wake_task();
    }
}
```

- `Wake` trait定义了唤醒一个任务的方法，这通常是通过通知执行器该任务已准备好运行。
- `wake`: 这个方法在`TaskWaker`被丢弃时调用，并唤醒与之关联的任务。
- `wake_by_ref`: 这个方法在不丢弃`TaskWaker`的情况下唤醒任务，它是为那些只有引用的场景设计的。

这个`TaskWaker`的实现简单而有效。当任务不能立即完成并需要在未来的某个时刻被重新轮询时，它会被放入`task_queue`中。执行器会定期检查这个队列并运行其中的任务。

4.14.4. 创建并运行协程

在操作系统内核的主函数中，当完成初始化并启动1号进程（Shell）后，便开始创建并执行了几个用于处理I/O的协程：

```
let mut executor = Executor::new();
executor.spawn(Task::new(key_presses_handler()));
executor.spawn(Task::new(mouse_handler()));
executor.run();
```

这段代码显示了如何使用前面定义的`Executor`和`Task`进行异步任务的调度和运行。下面是具体的步骤：

1. **初始化执行器**:
   - 在这里，我们创建了一个新的`Executor`实例。此时，执行器的协程队列是空的，因此它没有协程要运行。
2. **添加协程到执行器**:
   - 这里，我们创建了两个新的协程，并将它们添加到执行器的协程队列中。
     - 第一个协程是处理键盘按键事件的`key_presses_handler()`函数。我们假设这是一个返回`Future`的异步函数，当按键事件发生时，它可能会被挂起，并在事件处理完毕后再次被唤醒。
     - 第二个协程是处理鼠标事件的`mouse_handler()`函数。与键盘处理函数类似，它也可能会因为等待事件而被挂起。
   - 使用`spawn`方法，这些协程将被添加到执行器的协程队列中，并在适当的时候被执行。
3. **启动执行器**:

- 使用run方法，执行器开始其主循环，不断地运行就绪的协程。它会持续运行，轮询协程，直到所有协程都完成。由于run方法的定义中有一个无限循环（loop），它实际上永远不会返回，而是会在没有协程要运行时进入休眠状态。

4.15. 键盘和鼠标

键盘和鼠标在操作系统开发中，作为用户与系统之间的输入设备，通过产生中断信号或系统调用来通知内核执行相应的操作。键盘和鼠标的驱动程序是操作系统内核的一部分，负责接收和处理这些信号，并将它们转换为可识别的字符或坐标。操作系统还需要提供一些接口和库，让用户空间的程序可以方便地使用键盘和鼠标的功能，如读取输入、移动光标、绘制图形等。

4.15.1. 键盘的初始化

PS/2 键盘作为设备本身并不需要额外的启用操作，就已经可以在被按下时发出IRQ 1中断了。因此，我们需要做的只有将其注册到中断服务例程表中：

```
pub fn init() {
    interrupts::without_interrupts(|| {
        syskrnl::interrupts::set_irq_handler(1,
keyboard_interrupt_handler);
    });
}
```

4.15.2. 处理键盘中断

首先，我准备了一个并发队列来缓冲接收到的键盘扫描码：

```
static SCANCODE_QUEUE: OnceCell<ArrayQueue<u8>> = OnceCell::uninit();
```

- SCANCODE_QUEUE是一个静态变量，意味着它在整个程序的生命周期中只会被初始化一次。
- OnceCell是一个容器类型，用于保证内容只被设置/初始化一次。它特别适用于静态变量，因为Rust通常不允许静态变量具有运行时的初始化过程。
- ArrayQueue<u8>是一个固定大小的并发队列，其中元素类型是u8。
- OnceCell::uninit()表示此单元尚未被初始化。

当内核准备好接受键盘输入时，这个扫描码队列会被初始化：

```
SCANCODE_QUEUE
            .try_init_once(|| ArrayQueue::new(100))
            .expect("ScancodeStream::new 只应当被调用一次哦");
```

当键盘被按下，中断被触发时，我们可以从I/O端口读取键盘的扫描码：

```
/// 键盘中断处理函数
fn keyboard_interrupt_handler() {
    let scancode: u8 = unsafe { inb(0x60) };
    add_scancode(scancode);
}
```

add_scancode函数会把扫描码插入到扫描码缓冲队列中：

```
pub(crate) fn add_scancode(scancode: u8) {
    if let Ok(queue) = SCANCODE_QUEUE.try_get() {
        if let Err(_) = queue.push(scancode) {
            debugln!("警告：键盘扫描码队列已满；正在丢弃键盘输入");
        } else {
            WAKER.wake();
        }
    } else {
        debugln!("警告：键盘扫描码队列尚未初始化");
    }
}
```

这个函数接受一个u8参数，即键盘的扫描码。它的工作流程如下所示：

1. 尝试从SCANCODE_QUEUE中获取队列的引用。try_get方法尝试获取OnceCell的内容，如果已经初始化过，则返回Ok(T)；否则返回Err。
2. 如果成功获取了队列的引用，接下来这行代码会尝试将scancode压入队列。如果队列已满或出现其他错误，push方法会返回一个错误。
3. 如果队列已满，将打印一个警告消息，说明扫描码队列已满，并正在丢弃当前的键盘输入。
4. 如果scancode成功地被添加到队列中，则调用WAKER的wake方法。WAKER是一个用于唤醒等待扫描码处理任务的对象：

   ```
   static WAKER: AtomicWaker = AtomicWaker::new();
   ```

5. 如果从SCANCODE_QUEUE中获取队列失败（即队列尚未初始化），则打印另一个警告消息。

4.15.3. 处理键盘扫描码流

我们为键盘扫描码队列定义了一个结构：ScancodeStream：

```
pub struct ScancodeStream {
    _private: (),
}
```

它有一个私有字段_private，此字段的存在是为了阻止其他模块或库直接构造此结构体的实例。这是因为它的构造函数是有特殊作用的：

```
impl ScancodeStream {
    pub fn new() -> Self {
        SCANCODE_QUEUE
            .try_init_once(|| ArrayQueue::new(100))
            .expect("ScancodeStream::new 只应当被调用一次哦");
        ScancodeStream { _private: () }
    }
}
```

- 这为ScancodeStream提供了一个new函数，该函数尝试初始化SCANCODE_QUEUE。此函数使用OnceCell的try_init_once方法来确保队列只初始化一次，并为其分配100个元素的空间。
- 如果尝试多次调用new函数，expect将触发一个panic，并输出错误消息。

接下来，我们为ScancodeStream实现Stream trait：

```
impl Stream for ScancodeStream {
    type Item = u8;

    fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>) ->
Poll<Option<Self::Item>> {
        let queue = SCANCODE_QUEUE.try_get().expect("未初始化");

        if let Some(scancode) = queue.pop() {
            return Poll::Ready(Some(scancode));
        }

        WAKER.register(&cx.waker());
        match queue.pop() {
            Some(scancode) => {
                WAKER.take();
                Poll::Ready(Some(scancode))
            }
            None => Poll::Pending,
        }
    }
}
```

- Stream trait定义了一个异步流，可以一个接一个地提供项目。
- type Item = u8;表明流产生的元素类型为u8，即扫描码。
- poll_next函数是Stream trait的核心，它尝试从流中获取下一个项目。它的工作流程如下：
  - 尝试获取队列的引用。

- 尝试从队列中弹出一个扫描码。
- 如果队列为空，该方法会使用全局的WAKER来注册当前任务的waker。
- 然后，它再次尝试从队列中弹出一个扫描码，以确保在注册waker期间没有错过新的输入。如果成功弹出，它会删除waker并立即返回结果：
- 如果队列仍然为空，它返回Poll::Pending，表示没有可用的数据，但当有新数据到来时，当前任务的waker会被唤醒。

ScancodeStream结构体为键盘扫描码提供了异步流的抽象。使用这种结构，我们可以在Rust的异步框架（如tokio或async-std）中方便地处理扫描码，这样当有新的键盘输入到达时，我们的异步任务会被唤醒并处理这些输入。

4.15.4. 键盘事件处理函数

接下来实现在内核中运行的协程，键盘事件处理函数：

```
pub async fn key_presses_handler() {
    let mut scancodes = ScancodeStream::new();
    let mut keyboard = Keyboard::new(layouts::Us104Key, ScancodeSet1,
HandleControl::MapLettersToUnicode);

    while let Some(scancode) = scancodes.next().await {
        if let Ok(Some(key_event)) = keyboard.add_byte(scancode) {
            if let Some(key) = keyboard.process_keyevent(key_event) {
                // debugln!("{:?}",key);
                match key {
                    DecodedKey::Unicode(character) =>
key_event_handler(character),
                    DecodedKey::RawKey(key) => undk_handler(key),
                }
            }
        }
    }
}
```

可以看到，这是一个异步函数，意味着它可以被暂停和恢复执行，不会阻塞整个操作系统或程序。

在此函数内，首先创建了ScancodeStream和Keyboard两个实例：

- ScancodeStream 提供了一个从键盘获取扫描码的异步流。
- Keyboard 则提供了将扫描码解码为按键的方法。

函数的核心逻辑是一个循环，这个循环会不断地从ScancodeStream获取扫描码，然后使用Keyboard实例来处理它们，并在处理完后调用相应的处理函数。

undk_handler函数用于处理不能被转换成ASCII字符的键盘事件：

```
lazy_static! {
    pub static ref GUI_UNDK_KEY_EVENT_SUBSCRIBER: Mutex<Vec<usize>> =
{ Mutex::new(Vec::new()) };
}

fn undk_handler(ch: KeyCode) {
    let c = ch as u16;
    for eid in GUI_TIME_UPDATE_EVENT_NEEDER.lock().iter() {
        if let Some(pid) = event::EVENT_QUEUE
            .lock()
            .wakeup_with_ret(*eid, gui_event_make_ret(GUI_EVENT_UNDK_KEY,
c, 0, 0))
        {
            SCHEDULER.lock().wakeup(pid);
        }
    }
}
```

当收到特定的键事件时，它将该事件广播给所有在GUI_UNDK_KEY_EVENT_SUBSCRIBER列表中注册的事件监听器。

key_event_handler函数用于处理字符事件：

```
fn key_event_handler(ch: char) {
    if let Some(pid) =
event::EVENT_QUEUE.lock().wakeup_with_ret(KEYBOARD_INPUT, ch as u32 as
usize) {
        SCHEDULER.lock().wakeup(pid);
    }
}
```

当收到特定的字符时，它将该事件发送给在EVENT_QUEUE中注册的监听器。

4.15.5. 鼠标初始化

和键盘不同，PS/2 鼠标默认不会发送中断，它们需要通过I/O操作来被启用，并在状态改变时发送中断。

我们首先实现一个用于向鼠标设备发送命令的函数send_command：

```
const ACK: u8 = 0xFA;

fn send_command(cmd: u8) {
    unsafe {
        outb(0x64, 0xD4);
        outb(0x60, cmd);
```

```
    }
    if cmd != 0xFF {
        while unsafe { inb(0x60) } != ACK {}
    } else {
        while unsafe { inb(0x60) } != 0xAA {}
    }
}
```

ACK是一个常量，其值为0xFA。通常在PS/2协议中，当设备（如鼠标或键盘）接收到一个命令并处理时，它会返回一个ACK(acknowledgment)信号表示已经接收到命令。

这个函数负责将命令cmd发送到鼠标设备，并等待ACK或其他响应（如果发送的是复位指令0xFF，不会响应ACK，而是0xAA）。它使用outb函数（一个通常用于端口I/O操作的低级函数）向特定的端口发送命令。

接下来我们实现鼠标初始化函数。这个函数的主要工作流程如下：

1. 为端口0x64和0x60创建两个新的Port对象。这两个端口是PS/2控制器的标准端口，用于与鼠标和键盘通信。
2. 使用while循环等待端口0x64的状态字节满足条件：第二个比特为0、第一个比特为1。这实际上是等待控制器准备好写入命令。
3. 通过端口0x64发送0xA8命令以开启鼠标。
4. 发送0x20命令到端口0x64，这个命令的意图是读取当前的设置。
5. 在读取状态之前，需要确保数据可以从端口0x60读取（即端口0x64的最低有效位为1）。
6. 读取状态，并发送0x60命令到端口0x64以设置命令字节。接着，通过修改状态（使其第二比特设置为1）并写回到端口0x60，完成配置。
7. 使用send_command函数发送几个鼠标的标准初始化命令：0xF6（设备重置），0xF4（开启数据报告），0xF3（设置鼠标速率）和40（这通常是设置鼠标的采样率，这里设置为40Hz）。
8. 最后，禁用中断并设置IRQ 12（通常是鼠标的IRQ）的中断处理程序为mouse_interrupt_handler。

总的来说，这段代码主要是为了初始化和配置鼠标设备，使其能够与操作系统正常通信。此外，它还为鼠标中断设置了处理程序，以便操作系统可以在鼠标活动时收到通知。以下是它的代码实现：

```
pub fn init() {
    // 开启鼠标设备
    let mut port_64 = Port::<u8>::new(0x64);
    let mut port_60 = Port::<u8>::new(0x60);

    // 等待端口0x64状态字节的第二个比特为0、第一个比特为1
```

```
    while unsafe { port_64.read() } & 0x2 != 0 {}

    unsafe {
        port_64.write(0xA8); // 开启鼠标
        port_64.write(0x20); // 读取当前设置
        while { port_64.read() } & 0x1 == 0 {}
        let status = port_60.read();
        port_64.write(0x60); // 设置命令字节
        port_60.write(status | 0x2); // 设置第二个比特
    }

    send_command(0xF6);
    send_command(0xF4);
    send_command(0xF3);
    send_command(40);

    interrupts::without_interrupts(|| {
        syskrnl::interrupts::set_irq_handler(12, mouse_interrupt_handler);
    });
}
```

4.15.6. 处理鼠标中断

和键盘处理不同的是，鼠标的数据包解包较为复杂。根据鼠标数据包的规则，我们可以定义数据包的结构体，并为其编写解包逻辑。结构体定义如下所示：

```
pub struct MousePackage {
    middle_btn: bool,
    right_btn: bool,
    left_btn: bool,
    delta_x: i32,
    delta_y: i32,
}
```

这个结构体用于描述鼠标的状态：

- `middle_btn`, `right_btn`, `left_btn`: 这三个字段表示鼠标的中键、右键和左键的状态。如果某个按钮被按下，则相应的字段为`true`，否则为`false`。
- `delta_x`, `delta_y`: 这两个字段表示鼠标在X和Y轴上的移动量。

接下来，我们实现解包的代码：

```
impl MousePackage {
    pub fn new(byte_1: u8, byte_2: u8, byte_3: u8) -> Option<Self> {
        if byte_1 & 0xC0 > 0 || byte_1 & 0x8 == 0 {
            None
```

```
        } else {
            let mut delta_x = byte_2 as i32;
            if byte_1 & 0x10 > 0 {
                delta_x = delta_x as i8 as i32;
            }
            let mut delta_y = byte_3 as i32;
            if byte_1 & 0x20 > 0 {
                delta_y = delta_y as i8 as i32;
            }
            delta_y *= -1; // 坐标体系不一致，需要反转
            let middle_btn = byte_1 & 0x4 > 0;
            let right_btn = byte_1 & 0x2 > 0;
            let left_btn = byte_1 & 0x1 > 0;
            Some(MousePackage {
                middle_btn,
                right_btn,
                left_btn,
                delta_x,
                delta_y,
            })
        }
    }
}
```

这是一个`MousePackage`的方法，用于从三个字节中创建一个新的`MousePackage`对象。PS/2鼠标通常每次发送3个字节的数据包，分别表示鼠标的状态和移动量。

- `byte_1` 包含了鼠标的按键状态和移动方向标志。如果`byte_1` 的某些位设置不正确，函数将返回`None`。否则，它将解析这3个字节，并从中提取按钮状态和移动量。
- 我们还对移动量进行了补码转换，以处理负的移动量，并使得结果能够正确地表示在各个方向上的移动。
- 最后，构造并返回一个包含所有这些信息的`MousePackage`对象。

我们还为结构体设计了一系列Getter：

```
pub fn middle_btn(&self) -> bool {
    self.middle_btn
}
pub fn right_btn(&self) -> bool {
    self.right_btn
}
pub fn left_btn(&self) -> bool {
    self.left_btn
}
pub fn delta_x(&self) -> i32 {
    self.delta_x
```

```
}
pub fn delta_y(&self) -> i32 {
    self.delta_y
}
```

接下来，我们可以处理鼠标中断了：

```
/// 鼠标中断处理函数
pub(crate) fn mouse_interrupt_handler() {
    let b1 = unsafe { inb(0x60) };
    let b2 = unsafe { inb(0x60) };
    let b3 = unsafe { inb(0x60) };
    if let Some(package) = MousePackage::new(b1, b2, b3) {
        add_package(package);
    }
}
```

函数在`0x60`端口上连续读取三次，获得了鼠标数据包的三个字节。然后，函数将数据包解包，并插入数据包队列。

4.15.7. 处理鼠标扫描码流

这一部分的代码逻辑和键盘相同，因此不再赘述，直接给出代码：

```
static MOUSE_PACKAGE_QUEUE: OnceCell<ArrayQueue<MousePackage>> =
OnceCell::uninit();
static WAKER: AtomicWaker = AtomicWaker::new();

pub(crate) fn add_package(package: MousePackage) {
    if let Ok(queue) = MOUSE_PACKAGE_QUEUE.try_get() {
        if let Err(_) = queue.push(package) {
            debugln!("警告: 鼠标数据包队列已满; 正在丢弃数据包");
        } else {
            WAKER.wake();
        }
    } else {
        debugln!("警告: 鼠标数据包队列尚未初始化");
    }
}

pub struct PackageStream {
    _private: (),
}

impl PackageStream {
    pub fn new() -> Self {
```

```
        MOUSE_PACKAGE_QUEUE
            .try_init_once(|| ArrayQueue::new(100))
            .expect("PackageStream::new 只应当被调用一次哦");
        PackageStream { _private: () }
    }
}

impl Stream for PackageStream {
    type Item = MousePackage;

    fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>) ->
Poll<Option<Self::Item>> {
        let queue = MOUSE_PACKAGE_QUEUE.try_get().expect("未初始化");

        if let Some(package) = queue.pop() {
            return Poll::Ready(Some(package));
        }

        WAKER.register(&cx.waker());
        match queue.pop() {
            Some(package) => {
                WAKER.take();
                Poll::Ready(Some(package))
            }
            None => Poll::Pending,
        }
    }
}
```

4.15.8. 鼠标事件处理函数

接下来实现在内核中运行的协程，鼠标事件处理函数：

```
pub async fn mouse_handler() {
    let mut packages = PackageStream::new();

    let mut btn_down = false;

    while let Some(package) = packages.next().await {
        MOUSE_CURSOR.lock().handle_change(package.delta_x,
package.delta_y);
        if btn_down && !package.left_btn {
            btn_down = false;
            MOUSE_CURSOR.lock().handle_click();
        } else if !btn_down && package.left_btn {
            btn_down = true;
```

```
        }
    }
}
```

这段代码的工作流程如下：

1. **创建PackageStream对象**
2. **处理鼠标事件**
   - 这是一个异步while循环，它等待PackageStream生成下一个数据包。每次有一个新的数据包时，它都会处理这个数据包。
3. **移动鼠标光标**
   - 调用MOUSE_CURSOR对象的handle_change函数，根据鼠标的移动量(delta_x, delta_y)更新光标的位置。
4. **处理鼠标点击**
   - btn_down是一个布尔变量，用于跟踪左键是否被按下。
   - 如果btn_down为true，并且当前数据包表示左键没有被按下（即释放），则handle_click函数被调用来处理鼠标点击事件，并将btn_down设置为false。
   - 如果btn_down为false，并且当前数据包表示左键被按下，那么将btn_down设置为true。这表示鼠标左键已经被按下，但还未释放。
   - 这个函数的设计是这样的：只有在鼠标左键从按下状态变为释放状态时才会处理点击事件，这是大多数图形界面中常见的鼠标点击行为。

总体来说，mouse_handler函数异步地处理来自PackageStream的鼠标数据包，更新鼠标光标的位置，并处理鼠标点击事件。

4.16. 硬盘

硬盘在操作系统中作为一种持久化的存储设备，可以保存操作系统本身、用户数据和其他应用程序。硬盘的性能和容量直接影响了操作系统的运行效率和功能。操作系统开发中需要考虑如何管理和优化硬盘的读写操作，如何设计合适的文件系统和磁盘驱动程序，以及如何利用硬盘实现虚拟内存等技术。

在操作系统开发中，常见的硬盘交互协议有以下几种：

- **ATA**（Advanced Technology Attachment）：一种并行传输协议，用于连接主板和硬盘等存储设备。ATA协议定义了数据线、控制线、电源线等接口规范，以及数据传输的命令和模式。ATA协议最初是为机械硬盘设计的，后来也被用于固态硬盘和光驱等设备。ATA协议的速度和可靠性受到并行传输的限制，因此逐渐被串行传输协议所取代。
- **SATA**（Serial ATA）：一种串行传输协议，用于替代ATA协议。SATA协议相比ATA协议，具有更高的速度、更低的功耗、更小的接口尺寸、更好的热散性等优点。SATA协议也是为机械硬盘设计的，但也可以用于固态硬盘和光驱等设备。

- **AHCI**（Advanced Host Controller Interface）：一种软件接口标准，用于让操作系统和SATA控制器之间进行通信。AHCI标准定义了一套寄存器和命令结构，以支持SATA设备的高级功能，如热插拔、本地命令队列、本地错误恢复等。AHCI标准可以提高SATA设备的性能和稳定性，但也需要操作系统的支持。目前大多数操作系统都支持AHCI标准。
- **NVMe**（Non-Volatile Memory Express）：一种基于PCIe总线的高性能存储协议，专门为固态硬盘设计。NVMe协议利用PCIe总线的高带宽和低延迟，以及固态硬盘的高并行度和随机访问能力，实现了比SATA/AHCI更高的性能和效率。NVMe协议也定义了一套寄存器和命令结构，以支持固态硬盘的高级功能，如电源管理、温度监控、安全擦除等。

在本项目中，我先后实现了ATA协议和AHCI协议。ATA协议实现简单，但是效率较低，因此我废弃了本项目中的ATA支持，转而采用AHCI协议。AHCI协议的实现较ATA更难，但是可以更高效、更现代地进行硬盘操作。

本文主要向读者介绍AHCI协议的实现。

4.16.1. 数据结构定义

AHCI协议中使用到了大量的数据结构。

4.16.1.1. FIS（帧信息结构）

FIS是SATA协议中用于Host和device之间信息传输的机制，每个FIS的格式都是固定的，并且对应唯一的ID，SATA Spec定义了14类FIS ID。FIS的全称是Frame Information Structure，意思是帧信息结构。FIS可以看作是传统的任务文件的数据集合，或者是ATA命令的封装。FIS可以通过不同的方式在Host和device之间传输，例如PIO、DMA、NCQ等。FIS分为八种类型，常见的有H2D（Host to Device）、D2H（Device to Host）、DMA Setup和Data。

从实际需要的角度出发，本项目中仅实现了`FIS_H2D`。这种类型的FIS由Host发送给device，用于发送ATA命令或控制信息。

```
#[repr(C)]
#[derive(Debug)]
pub struct FisRegH2d {
    // DWORD 0
    fis_type: u8, // FIS_TYPE_REG_H2D

    // This is a byte with mixed fields, so I will use a private u8 and
provide getters and setters for its parts.
    _pmport_rsv0_c: u8,
```

```
    command: u8,  // Command register
    featurel: u8, // Feature register, 7:0

    // DWORD 1
    lba0: u8,   // LBA low register, 7:0
    lba1: u8,   // LBA mid register, 15:8
    lba2: u8,   // LBA high register, 23:16
    device: u8, // Device register

    // DWORD 2
    lba3: u8,     // LBA register, 31:24
    lba4: u8,     // LBA register, 39:32
    lba5: u8,     // LBA register, 47:40
    featureh: u8, // Feature register, 15:8

    // DWORD 3
    countl: u8,  // Count register, 7:0
    counth: u8,  // Count register, 15:8
    icc: u8,     // Isochronous command completion
    control: u8, // Control register

    // DWORD 4
    rsv1: [u8; 4], // Reserved
}

impl FisRegH2d {
    // Getter and Setter for pmport (4 bits)
    pub fn pmport(&self) -> u8 {
        self._pmport_rsv0_c & 0x0F
    }

    pub fn set_pmport(&mut self, value: u8) {
        self._pmport_rsv0_c = (self._pmport_rsv0_c & 0xF0) | (value &
0x0F);
    }

    // Getter and Setter for rsv0 (3 bits)
    pub fn rsv0(&self) -> u8 {
        (self._pmport_rsv0_c >> 4) & 0x07
    }

    pub fn set_rsv0(&mut self, value: u8) {
        self._pmport_rsv0_c = (self._pmport_rsv0_c & 0x8F) | ((value &
0x07) << 4);
    }
```

```
    // Getter and Setter for c (1 bit)
    pub fn c(&self) -> u8 {
        (self._pmport_rsv0_c >> 7) & 0x01
    }

    pub fn set_c(&mut self, value: u8) {
        self._pmport_rsv0_c = (self._pmport_rsv0_c & 0x7F) | ((value &
0x01) << 7);
    }
}
```

4.16.1.2. HBA（AHCI控制器）

从实际需要的角度出发，本项目中实现了HBA_MEM、HBA_PORT、HBA_CMD_HEADER、HBA_CMD_TABLE、HBA_PRDT_ENTRY。

1. **HBA_MEM**：HBA内存寄存器（HBA memory registers）。HBA内存寄存器用于控制和监测AHCI控制器和SATA设备之间的数据和命令传输。HBA内存寄存器可以分为两部分：通用寄存器和端口寄存器。通用寄存器用于控制整个控制器的行为和特性，如支持的最大端口数、最大命令槽数、是否支持本地命令队列、热插拔、电源管理等。端口寄存器用于管理每个端口连接的设备的状态和功能，如启动、停止、DMA传输、中断类型和原因、命令列表基地址等。操作系统可以通过系统内存和内存映射寄存器来访问HBA内存寄存器，从而实现对SATA设备的操作和管理。

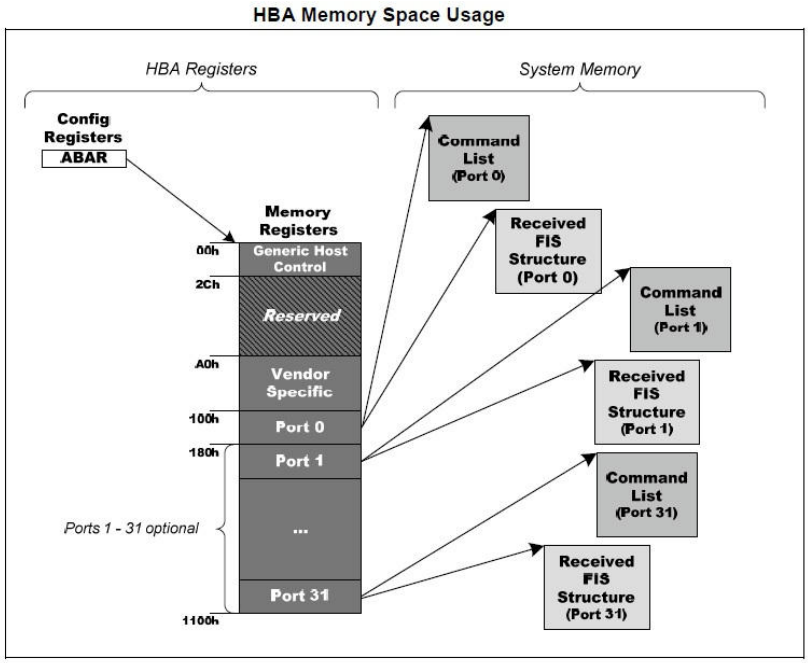

图 3: HBA内存寄存器的内存分布

```rust
#[repr(C)]
#[derive(Debug)]
pub struct HbaMem {
    // 0x00 - 0x2B, Generic Host Control
    pub cap: Volatile<u32>,     // 0x00, Host capability
    pub ghc: Volatile<u32>,     // 0x04, Global host control
    pub is: Volatile<u32>,      // 0x08, Interrupt status
    pub pi: Volatile<u32>,      // 0x0C, Port implemented
    pub vs: Volatile<u32>,      // 0x10, Version
    pub ccc_ctl: Volatile<u32>, // 0x14, Command completion coalescing
control
    pub ccc_pts: Volatile<u32>, // 0x18, Command completion coalescing
ports
    pub em_loc: Volatile<u32>,  // 0x1C, Enclosure management location
    pub em_ctl: Volatile<u32>,  // 0x20, Enclosure management control
    pub cap2: Volatile<u32>,    // 0x24, Host capabilities extended
    pub bohc: Volatile<u32>,    // 0x28, BIOS/OS handoff control and
status

    // 0x2C - 0x9F, Reserved
    pub rsv: [Volatile<u8>; 0xA0 - 0x2C],

    // 0xA0 - 0xFF, Vendor specific registers
    pub vendor: [Volatile<u8>; 0x100 - 0xA0],

    // 0x100 - 0x10FF, Port control registers
    pub ports: [HbaPort; 32], // 1 ~ 32
}
```

2. **HBA_PORT**：HBA端口寄存器（HBA port registers）。用于存储上一部分中提到的“端口寄存器”部分。

```rust
#[repr(C)]
#[derive(Debug)]
pub struct HbaPort {
    pub clb: Volatile<u64>,         // 0x00, command list base address,
1K-byte aligned
    pub fb: Volatile<u64>,          // 0x08, FIS base address, 256-byte
aligned
    pub is: Volatile<u32>,          // 0x10, interrupt status
    pub ie: Volatile<u32>,          // 0x14, interrupt enable
    pub cmd: Volatile<u32>,         // 0x18, command and status
    pub rsv0: Volatile<u32>,        // 0x1C, Reserved
    pub tfd: Volatile<u32>,         // 0x20, task file data
    pub sig: Volatile<u32>,         // 0x24, signature
    pub ssts: Volatile<u32>,        // 0x28, SATA status (SCR0:SStatus)
```

```
    pub sctl: Volatile<u32>,        // 0x2C, SATA control (SCR2:
SControl)
    pub serr: Volatile<u32>,        // 0x30, SATA error (SCR1:SError)
    pub sact: Volatile<u32>,        // 0x34, SATA active (SCR3:SActive)
    pub ci: Volatile<u32>,          // 0x38, command issue
    pub sntf: Volatile<u32>,        // 0x3C, SATA notification (SCR4:
SNotification)
    pub fbs: Volatile<u32>,         // 0x40, FIS-based switch control
    pub rsv1: [Volatile<u32>; 11],  // 0x44 ~ 0x6F, Reserved
    pub vendor: [Volatile<u32>; 4], // 0x70 ~ 0x7F, vendor specific
}
```

3. **HBA_CMD_HEADER**：HBA命令头。Host通过命令列表向Device发送命令。命令列表由1到32个命令头组成，每一个称为一个槽（Slot）。每个命令头描述了一个ATA或ATAPI命令，包括一个命令FIS、一个ATAPI命令缓冲区和一系列物理区域描述符表，指定数据有效载荷的地址和大小。

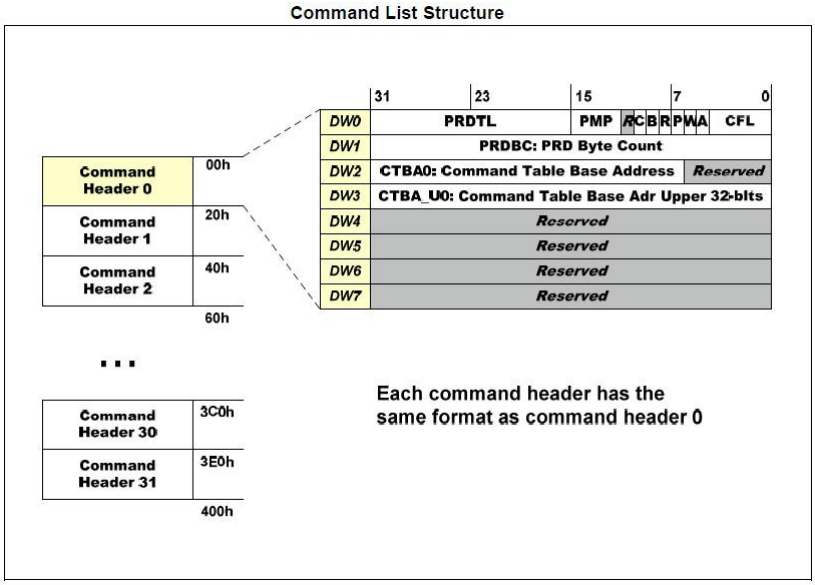

图 4: HBA命令列表的内存分布

```
#[repr(C)]
#[derive(Debug)]
pub struct HbaCmdHeader {
    // DW0
    dw0: u16,       // Combines cfl, a, w, p, r, b, c, rsv0, and pmp
into two bytes
    pub prdtl: u16, // Physical region descriptor table length in
entries

    // DW1
    pub prdbc: Volatile<u32>, // Physical region descriptor byte count
```

```
transferred

    // DW2, 3
    pub ctba: u64, // Command table descriptor base address

    // DW4 - 7
    pub rsv1: [u32; 4], // Reserved
}

impl HbaCmdHeader {
    /// CFL: Command FIS length in DWORDS, 2 ~ 16
    pub fn cfl(&self) -> u8 {
        (self.dw0 & 0b0001_1111) as u8
    }

    pub fn set_cfl(&mut self, value: u8) {
        self.dw0 = (self.dw0 & !0b0001_1111) | ((value as u16) &
0b0001_1111);
    }

    /// A: ATAPI
    pub fn a(&self) -> bool {
        (self.dw0 & 0b0010_0000) != 0
    }

    pub fn set_a(&mut self, value: bool) {
        if value {
            self.dw0 |= 0b0010_0000;
        } else {
            self.dw0 &= !0b0010_0000;
        }
    }

    /// W: Write, 1: H2D, 0: D2H
    pub fn w(&self) -> bool {
        (self.dw0 & 0b0100_0000) != 0
    }

    pub fn set_w(&mut self, value: bool) {
        if value {
            self.dw0 |= 0b0100_0000;
        } else {
            self.dw0 &= !0b0100_0000;
        }
    }

    // P: Prefetchable
```

```
    pub fn p(&self) -> bool {
        (self.dw0 & 0b1000_0000) != 0
    }

    pub fn set_p(&mut self, value: bool) {
        if value {
            self.dw0 |= 0b1000_0000;
        } else {
            self.dw0 &= !0b1000_0000;
        }
    }

    // R: Reset
    pub fn r(&self) -> bool {
        (self.dw0 & 0b1_0000_0000) != 0
    }

    pub fn set_r(&mut self, value: bool) {
        if value {
            self.dw0 |= 0b1_0000_0000;
        } else {
            self.dw0 &= !0b1_0000_0000;
        }
    }

    // B: BIST
    pub fn b(&self) -> bool {
        (self.dw0 & 0b10_0000_0000) != 0
    }

    pub fn set_b(&mut self, value: bool) {
        if value {
            self.dw0 |= 0b10_0000_0000;
        } else {
            self.dw0 &= !0b10_0000_0000;
        }
    }

    // C: Clear busy upon R_OK
    pub fn c(&self) -> bool {
        (self.dw0 & 0b100_0000_0000) != 0
    }

    pub fn set_c(&mut self, value: bool) {
        if value {
            self.dw0 |= 0b100_0000_0000;
        } else {
```

```
            self.dw0 &= !0b100_0000_0000;
        }
    }

    // RSV0: Reserved
    pub fn rsv0(&self) -> bool {
        (self.dw0 & 0b1000_0000_0000) != 0
    }

    pub fn set_rsv0(&mut self, value: bool) {
        if value {
            self.dw0 |= 0b1000_0000_0000;
        } else {
            self.dw0 &= !0b1000_0000_0000;
        }
    }

    // PMP: Port multiplier port
    pub fn pmp(&self) -> u8 {
        ((self.dw0 >> 12) & 0b1111) as u8
    }

    pub fn set_pmp(&mut self, value: u8) {
        self.dw0 = (self.dw0 & !(0b1111 << 12)) | (((value as u16) &
0b1111) << 12);
    }
}
```

4. **HBA_CMD_TABLE**：HBA命令表。如上所述，一个命令表包含了一个ATA命令FIS、一个ATAPI命令缓冲区和一系列PRDT（物理区域描述符表），指定了数据有效载荷的地址和大小。一个命令表可以有0到65535个PRDT条目。实际的PRDT条目数是在命令头中设置的（HBA_CMD_HEADER.prdtl）。AHCI控制器作为PCI总线主设备，直接在设备和系统内存之间传输数据有效载荷。

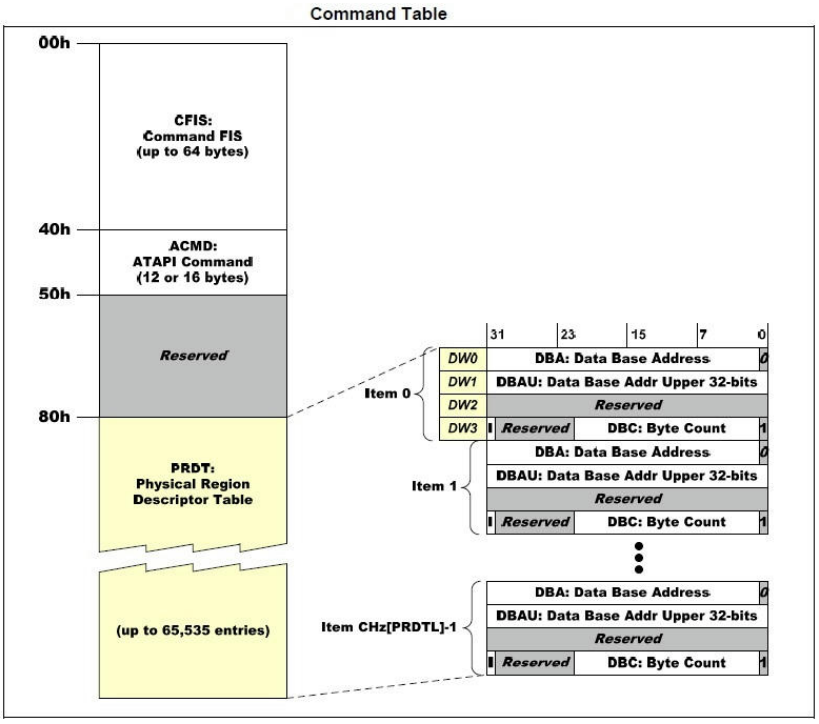

图 5: HBA命令表的内存分布

```
#[repr(C)]
#[derive(Debug)]
pub struct HbaCmdTbl {
    // 0x00
    pub cfis: [u8; 64], // Command FIS

    // 0x40
    pub acmd: [u8; 16], // ATAPI command, 12 or 16 byte

    // 0x50
    pub rsv: [u8; 48], // Reserved

    // 0x80
    prdt_entry: [HbaPrdtEntry; 0], // Physical region descriptor table
entries, 0 ~ 65535
}

impl HbaCmdTbl {
    pub fn with_entries(n: usize) -> Box<Self> {
        let layout = Layout::from_size_align(
            mem::size_of::<Self>() + n * mem::size_of::<HbaPrdtEntry>(),
            mem::align_of::<HbaPrdtEntry>(),
        )
        .unwrap();

        unsafe {
```

```
            let raw = alloc::alloc::alloc_zeroed(layout);
            let hba = &mut *(raw as *mut HbaCmdTbl);

            Box::from_raw(hba)
        }
    }

    pub fn pred_entries(&mut self, len: usize) -> &[HbaPrdtEntry] {
        unsafe { slice::from_raw_parts(self.prdt_entry.as_ptr(), len) }
    }

    pub fn prdt_entries_mut(&mut self, len: usize) -> &mut
[HbaPrdtEntry] {
        unsafe { slice::from_raw_parts_mut(self.prdt_entry.as_mut_ptr(),
len) }
    }
}
```

5. **HBA_PRDT_ENTRY**：HBA PRDT条目。

```
#[repr(C)]
#[derive(Debug)]
pub struct HbaPrdtEntry {
    pub dba: u64,
    // Data base address
    pub rsv0: u32, // Reserved

    // DW3
    pub dw3: u32, // Byte count/Interrupt on completion
}

impl HbaPrdtEntry {
    /// Byte count, 4M max
    pub fn dbc(&self) -> u32 {
        self.dw3 & 0b00111111_11111111_11111111
    }

    pub fn set_dbc(&mut self, value: u32) {
        self.dw3 = (self.dw3 & !0b00111111_11111111_11111111) | (value &
0b00111111_11111111_11111111)
    }

    /// Interrupt on completion
    pub fn i(&self) -> bool {
        (self.dw3 >> 31) > 0
    }
```

```
    pub fn set_i(&mut self, value: bool) {
        if value {
            self.dw3 |= 1 << 31
        } else {
            self.dw3 &= !(1 << 31)
        }
    }
}
```

4.16.2. AHCI 初始化

4.16.2.1. AHCI 基址内存映射

主机通过系统内存和内存映射寄存器与AHCI控制器进行通信。AHCI控制器的最后一个PCI基地址寄存器（`BAR[5]`，头部偏移量`0x24`）指向AHCI基地址内存，它被称为ABAR（AHCI Base Memory Register，AHCI基地址内存寄存器）。所有的AHCI寄存器和内存都可以通过ABAR来定位。

在获得基地址后，我们还需要创建页表映射，以在操作系统中使用虚拟地址操作ABAR中的内容。为此，我实现了一个用于初始化ABAR内存映射的函数，它的工作流程如下所示：

1. **常量定义**:
   - 定义了一个常量`ABAR_ADDR`，它是一个64位整数，表示虚拟地址。这是本项目计划将AHCI控制器的ABAR映射到的地址。
2. **寻找AHCI控制器**:
   - 使用了一个函数（`pci_find_device_by_class_code`）来在PCI总线上查找AHCI控制器。
3. **读取ABAR地址**:
   - 从AHCI控制器的PCI配置空间中读取ABAR的物理地址。
4. **设置页表映射标志**:
   - 设置了两个页表映射标志：`PRESENT`和`WRITABLE`。这意味着这个页面是存在的，并且是可写的。
5. **映射内存**:
   - 将`page`映射到`frame`上。
   - `page`是包含之前定义的`ABAR_ADDR`的虚拟页面。
   - `frame`是包含读取到的ABAR地址的物理帧。
6. **检查映射结果**:

- 如果映射失败，则会引发一个panic。如果映射成功，会调用flush方法来确保这个映射立即生效。

函数的代码实现如下所示：

```
const ABAR_ADDR: u64 = 0xffffffff_febf1000;

pub fn create_abar_memory_mapping(mapper: &mut OffsetPageTable,
frame_allocator: &mut impl FrameAllocator<Size4KiB>) {
    let controller = io::pci::pci_find_device_by_class_code(0x01, 0x06);
    let abar_addr = io::pci::pci_config_read_u32(controller.0, controller.
1, controller.2, 0x24);

    use x86_64::structures::paging::PageTableFlags as Flags;

    let page =
Page::<Size4KiB>::containing_address(VirtAddr::new(ABAR_ADDR));
    let frame = PhysFrame::containing_address(PhysAddr::new(abar_addr as
u64));
    let flags = Flags::PRESENT | Flags::WRITABLE;

    let map_to_result = unsafe { mapper.map_to(page, frame, flags,
frame_allocator) };
    map_to_result.expect("Map_to_AHCI aBar Memory Failed").flush();
}
```

4.16.2.2. AHCI 寄存器内存映射

同样的，操作系统使用AHCI协议时，应当把相关的寄存器内容放在特定的内存区域上。因此，我们需要手动配置AHCI协议所使用的内存空间，并将它们映射到页表上。

首先，我们定义两个常量：

```
const AHCI_BASE: u64 = 0x0100_0000;
const AHCI_PHYSIC_BASE: u64 = 0x0100_0000;
```

这里定义了两个常量：`AHCI_BASE` 和 `AHCI_PHYSIC_BASE`，它们都被设置为了同一个值。显然，我们的目的是将这个物理地址空间映射到一个等效的虚拟地址空间。

```
pub fn create_ahci_memory_mapping(mapper: &mut OffsetPageTable,
frame_allocator: &mut impl FrameAllocator<Size4KiB>) {
    use x86_64::structures::paging::PageTableFlags as Flags;

    // We need 16 pages
    for i in 0..16 {
        let page =
```

```
Page::<Size4KiB>::containing_address(VirtAddr::new(AHCI_BASE + 0x1000 *
i));
        let frame =
PhysFrame::containing_address(PhysAddr::new(AHCI_PHYSIC_BASE + 0x1000 *
i));
        let flags = Flags::PRESENT | Flags::WRITABLE;

        let map_to_result = unsafe { mapper.map_to(page, frame, flags,
frame_allocator) };
        map_to_result.expect("Map_to_AHCI Memory Failed").flush();
    }
}
```

和之前的部分类似，不过这一次我们分配了16个页面。这是为了保证AHCI内存空间足够我们使用。

上述两个方法在初始化内存时被调用：

src/memory/mod.rs

```
pub fn init(bootinfo: &'static BootInfo) {
    interrupts::without_interrupts(|| {

        ......

        syskrnl::io::ahci::create_abar_memory_mapping(&mut mapper, &mut
frame_allocator);
        syskrnl::io::ahci::create_ahci_memory_mapping(&mut mapper, &mut
frame_allocator);
    });
}
```

4.16.2.3. 扫描端口

接下来，我们需要在AHCI的端口中扫描已经连接到AHCI控制器的磁盘。以下函数probe_port的目的是探测AHCI控制器的端口以确定连接到它的设备类型。

该函数的工作流程如下所示：

1. **读取pi（Ports Implemented）寄存器**
   - pi寄存器标明了哪些端口是已实现的。每个比特代表一个端口：比特0代表端口0，比特1代表端口1，以此类推。
2. **循环探测端口**
   - 首先检查端口是否已实现。使用按位与操作检查pi的最低位。如果结果大于0，表示当前端口i是已实现的。

- 若端口已实现，则使用check_type函数被用来确定连接到给定端口的设备类型。check_type的实现如下：

```
/// 检查设备类型
fn check_type(port: &HbaPort) -> u8 {
    let ssts = port.ssts.read();

    let ipm = (ssts >> 8) & 0x0F;
    let det = ssts & 0x0F;

    if det != HBA_PORT_DET_PRESENT || ipm != HBA_PORT_IPM_ACTIVE {
        return AHCI_DEV_NULL;
    }

    match port.sig.read() {
        SATA_SIG_ATAPI => AHCI_DEV_SATAPI,
        SATA_SIG_SEMB => AHCI_DEV_SEMB,
        SATA_SIG_PM => AHCI_DEV_PM,
        _ => AHCI_DEV_SATA,
    }
}
```

- 根据设备类型输出调试信息并处理。对于SATA设备，在一个全局可用端口集合中插入该端口号。
- 最后，更新pi和端口索引。pi被右移一位，将下一个端口的状态移到最低位。端口索引i增加1，为下次迭代做准备。

综上所述，这个函数对AHCI控制器的每个已实现的端口进行探测，确定连接到每个端口的设备类型，并在调试输出中打印相关信息。对于SATA设备，它还更新了一个全局的可用端口集合。该函数的实现如下：

```
fn probe_port(abar: &HbaMem) {
    // 在已实现的端口中扫描磁盘
    let mut pi = abar.pi.read();
    let mut i = 0;
    while i < 32 {
        if pi & 1 > 0 {
            let dt = check_type(&abar.ports[i]);
            match dt {
                AHCI_DEV_SATA => {
                    debugln!("SATA drive found at port {}", i);
                    AVALIABLE_PORTS.lock().insert(i);
                }
                AHCI_DEV_SATAPI => {
                    debugln!("SATAPI drive found at port {}", i);
```

```
                }
                AHCI_DEV_SEMB => {
                    debugln!("SEMB drive found at port {}", i);
                }
                AHCI_DEV_PM => {
                    debugln!("PM drive found at port {}", i);
                }
                _ => {
                    // debugln!("No drive found at port {}", i);
                }
            }
        }
        pi >>= 1;
        i += 1;
    }
}
```

4.16.2.4. 启用HBA

接下来，我们设置了一个函数`start_hba`来启动AHCI主机总线适配器（HBA）。从代码中，我们可以看到它修改了HBA控制器的全局主机控制（GHC）寄存器。

函数的工作流程如下所示：

1. **启用AHCI模式**：
   - 修改GHC寄存器以设置`HBA_GHC_AE`位，该位是AHCI启用位。当这一位被设置时，它启用AHCI模式。
2. **重置HBA**：
   - 设置`HBA_GHC_HR`位来发起HBA重置。重置是为了确保HBA处于一个已知的初始状态。
3. **等待HBA重置完成**：
   - 使用一个忙等待循环来检查`HBA_GHC_HR`位是否已清除，表明重置已完成。
4. **再次启用AHCI模式**：
   - 在HBA重置完成后，再次确保AHCI模式已启用。

函数的代码实现如下所示：

```
const HBA_GHC_HR: u32 = 0x0001; // HBA Reset
const HBA_GHC_AE: u32 = 0x8000; // AHCI Enable

fn start_hba(abar: &mut HbaMem) {
    abar.ghc.update(|x| *x |= HBA_GHC_AE);
    abar.ghc.update(|x| *x |= HBA_GHC_HR);
    while (abar.ghc.read() & HBA_GHC_HR) != 0 {}
```

```
    abar.ghc.update(|x| *x |= HBA_GHC_AE);
}
```

4.16.2.5. AHCI初始化函数

现在我们可以着手实现我们的AHCI初始化函数了：

```
pub fn init() {
    let abar = unsafe { &mut *(ABAR_ADDR as *mut HbaMem) };
    probe_port(abar);
    start_hba(abar);
}
```

4.16.3. AHCI端口

AHCI端口是我们访问每个AHCI设备的通道和桥梁。现在，我们将会为HBA_PORT类实现一系列方法，以实现对特定AHCI设备的访问功能。

4.16.3.1. 停止/启动命令引擎

在重新配置端口的命令列表、FIS和命令表的内存基址前，我们首先要停止端口的命令引擎。停止命令引擎的操作可以通过设置端口的cmd寄存器来实现。我们首先清除ST和FRE位，然后等待FR和CR位都被清除。

代码实现如下所示：

```
/// 停止命令引擎
fn stop_cmd(port: &mut HbaPort) {
    // 清除ST
    port.cmd.update(|x| *x &= !HBA_PxCMD_ST);
    // 清除FRE
    port.cmd.update(|x| *x &= !HBA_PxCMD_FRE);

    // 等待FR和CR位被清除
    while port.cmd.read() & HBA_PxCMD_FR > 0 || port.cmd.read() &
HBA_PxCMD_CR > 0 {}
}
```

同样地，我们可以操作端口的cmd寄存器来启动命令引擎。我们首先确保CR位被清除，然后设置FRE和ST位来启动命令引擎。

代码实现如下所示：

```rust
/// 启动命令引擎
fn start_cmd(port: &mut HbaPort) {
    while port.cmd.read() & HBA_PxCMD_CR > 0 {} // 等待CR位被清除

    // 置FRE和ST位
    port.cmd.update(|x| *x |= HBA_PxCMD_FRE);
    port.cmd.update(|x| *x |= HBA_PxCMD_ST);
}
```

4.16.3.2. 重置端口基址

接下来，我实现了一个函数`port_rebase`来实现端口基址重置操作。这个函数的工作流程如下：

- **停止命令引擎**：使用`stop_cmd`函数确保命令引擎停止，以避免在配置过程中产生不稳定或不期望的行为。
- **命令列表配置**：为命令列表分配内存并清除该内存。每个端口的命令列表的大小为1K（32条目，每条目32字节）。
- **FIS配置**：为FIS（Frame Information Structure）分配内存并清除该内存。FIS用于与设备交互。每个端口的FIS大小为256字节。
- **命令表配置**：对于每个命令头部，在命令列表中定义命令表的位置并清除相关内存。每个命令表的大小为256字节。
- **启动命令引擎**：使用`start_cmd`函数重新启动命令引擎。

通过重置基址操作，它确保了适当的内存空间已被分配并被清零，为后续的磁盘操作做好准备。以下是相关函数的代码实现：

```rust
pub fn port_rebase(port: &mut HbaPort, port_no: u64) {
    // 停止命令引擎
    stop_cmd(port);

    // 命令列表偏移量：1K * port_no
    // 命令列表条目大小：32
    // 命令列表条目最大数量：32
    // 命令列表最大大小：32 * 32 = 1K per Port （看懂了）
    port.clb.write(AHCI_PHYSIC_BASE + (port_no << 10));
    unsafe {
        ptr::write_bytes((port.clb.read() - AHCI_PHYSIC_BASE + AHCI_BASE)
as *mut u8, 0, 1024);
    }

    // FIS 偏移量：32K + 256 * port_no
    // FIS 条目大小：256byte per Port
```

```
    port.fb.write(AHCI_PHYSIC_BASE + (32 << 10) + (port_no << 8));
    unsafe {
        ptr::write_bytes((port.fb.read() - AHCI_PHYSIC_BASE + AHCI_BASE) as
*mut u8, 0, 256);
    }

    // 命令表偏移量: 40K + 8K * port_no
    // 命令表大小: 256 * 32 = 8K per Port
    let cmd_header = unsafe { slice::from_raw_parts_mut((port.clb.read() -
AHCI_PHYSIC_BASE + AHCI_BASE) as *mut HbaCmdHeader, 32) };
    for i in 0..32 {
        cmd_header[i].prdtl = 8; // 8 prdt entries per command table; 256
bytes per command table, 64+16+48+16*8

        // 命令表偏移量: 40K + 8K * port_no +  cmd_header_index * 256
        cmd_header[i].ctba = AHCI_PHYSIC_BASE + (40 << 10) + (port_no <<
13) + ((i as u64) << 8);
        unsafe {
            ptr::write_bytes((cmd_header[i].ctba - AHCI_PHYSIC_BASE +
AHCI_BASE) as *mut u8, 0, 256);
        }
    }

    start_cmd(port); // 重启命令引擎
}
```

4.16.3.3. 寻找空闲命令槽

当我们启动一个新命令之前，我们首先需要找到存放这个命令的命令槽。我实现了一个用于寻找可用命令槽的函数`find_cmdslot`，它的工作流程如下所示：

1. **合并活动和已启动的命令槽位**:
   - `self.sact.read()`: 从`SACT`（Serial ATA Active）寄存器中读取数据，该寄存器指示哪些命令槽位当前是活动的。
   - `self.ci.read()`: 从`CI`（Command Issue）寄存器中读取数据，该寄存器指示哪些命令槽位已经启动。
   - 通过将这两个值进行或操作，得到一个标识哪些命令槽位当前处于使用中的值。
2. **查找第一个可用的命令槽位**:
   - 从最低的槽位开始循环，检查每个命令槽位是否可用。如果找到一个空闲的命令槽位（即其对应的位为0），就返回该槽位的索引。
3. **如果所有的命令槽位都已使用**:
   - 如果所有的32个命令槽位都在使用中，返回`None`。

总的来说，这个函数提供了一个快速的方法来找到一个未使用的命令槽位，使得可以在该槽位上执行一个新的命令。如果没有可用的命令槽位，它将返回`None`。代码实现如下所示：

```
fn find_cmdslot(&self) -> Option<u32> {
    let mut slots = self.sact.read() | self.ci.read();
    for i in 0..32 {
        if (slots & 1) == 0 {
            return Some(i);
        }
        slots >>= 1;
    }
    debugln!("Cannot find free command list entry");
    None
}
```

4.16.3.4. 从磁盘读取制定扇区

我实现了一个函数`read`，用于从指定位置读取一定数量的扇区到内存。该函数的工作流程主要如下所示：

1. **检查缓冲区大小**:
   - 首先，函数检查提供的缓冲区是否足够大以容纳所需的数据。如果缓冲区太小，程序会panic。
2. **清除中断状态**:
   - 使用`self.is.write((-1i32) as u32)`清除中断状态。
3. **查找命令插槽**:
   - 函数接下来使用`self.find_cmdslot()`来寻找可用的命令插槽。
4. **准备命令头**:
   - 函数设置命令头以及其他相关参数，例如命令头的长度、数据传输方向、PRDT（Physical Region Descriptor Table）长度等。
5. **设置命令表**:
   - 在这部分，函数创建一个命令表，并为每个PRDT条目设置DBA (Data Base Address)和DBC (Data Byte Count)。
6. **准备FIS (Frame Information Structure)**:
   - 这是与硬盘通信的数据结构。函数设置其LBA地址、设备、命令以及其他必要的参数。
7. **等待端口空闲**:
   - 在执行命令之前，函数使用循环检查端口的状态，确保它空闲。如果端口长时间忙碌，函数会输出错误消息。

8. **发出命令**:
   - 通过设置ci（命令发出）标志来发出命令。
9. **等待完成**:
   - 函数在循环中等待命令完成，检查错误标志以及命令插槽的状态。
10. **检查错误**:
   - 在命令完成后，函数再次检查是否发生了错误，并相应地返回成功或失败。

代码实现如下所示:

```
pub fn read(&mut self, pos: u64, count: u32, buf: &mut [u8]) -> bool {
    if count as usize * 512 > buf.len() {
        panic!("错误的缓冲区大小")
    }

    self.is.write((-1i32) as u32);
    let mut spin = 0;
    if let Some(slot) = self.find_cmdslot() {
        let cmd_header = unsafe { &mut *((self.clb.read() -
AHCI_PHYSIC_BASE + AHCI_BASE) as *mut HbaCmdHeader).add(slot as usize) };
        cmd_header.set_cfl((mem::size_of::<FisRegH2d>() /
mem::size_of::<u32>()) as u8); // Command FIS size
        cmd_header.set_w(false); // Read from device
        cmd_header.prdtl = (((count - 1) >> 4) + 1) as u16;

        let mut cmd_table = HbaCmdTbl::with_entries(cmd_header.prdtl as
usize);

        let mut buf_addr = unsafe { translate_addr(buf.as_mut_ptr() as
u64).unwrap() };
        let mut count = count;
        let prdt_entry = cmd_table.prdt_entries_mut(cmd_header.prdtl as
usize);

        for i in 0..cmd_header.prdtl as usize {
            if i < cmd_header.prdtl as usize - 1 {
                prdt_entry[i].dba = buf_addr;
                prdt_entry[i].set_dbc(8 * 1024 - 1); // 8K bytes (this
value should always be set to 1 less than the actual value)
                prdt_entry[i].set_i(true);
                buf_addr += 8 * 1024; // 8K bytes
                count -= 16; // 16 sectors
            } else {
                prdt_entry[i].dba = buf_addr;
                prdt_entry[i].set_dbc((count << 9) - 1); // 512 bytes per
sector
```

```
                prdt_entry[i].set_i(true);
            }
        }

        let cmd_fis = unsafe { &mut *(cmd_table.cfis.as_ptr() as *mut
FisRegH2d) };
        cmd_fis.fis_type = FisType::FisTypeRegH2d as u8;
        cmd_fis.set_c(1); // Command
        cmd_fis.command = ATA_CMD_READ_DMA_EX;

        cmd_fis.lba0 = pos as u8;
        cmd_fis.lba1 = (pos >> 8) as u8;
        cmd_fis.lba2 = (pos >> 16) as u8;
        cmd_fis.lba3 = (pos >> 24) as u8;
        cmd_fis.lba4 = (pos >> 32) as u8;
        cmd_fis.lba5 = (pos >> 40) as u8;

        cmd_fis.device = 1 << 6; // LBA mode
        cmd_fis.countl = (count & 0xFF) as u8;
        cmd_fis.counth = ((count >> 8) & 0xFF) as u8;

        let cmd_table_addr = Box::into_raw(cmd_table) as u64;
        cmd_header.ctba = unsafe
{ translate_addr(cmd_table_addr).unwrap() };

        // The below loop waits until the port is no longer busy before
issuing a new command
        while (self.tfd.read() & (ATA_DEV_BUSY | ATA_DEV_DRQ) as u32) > 0
&& spin < 1000000 {
            spin += 1;
        }
        if spin == 1000000 {
            debugln!("Port is hung");
            let _ = unsafe { Box::from_raw(cmd_table_addr as *mut
HbaCmdTbl) };
            false
        } else {
            self.ci.update(|x| *x |= 1 << slot); // 签发命令

            // Wait for completion
            loop {
                // In some longer duration reads, it may be helpful to spin
on the DPS bit
                // in the PxIS port field as well (1 << 5)
                if self.ci.read() & (1 << slot) == 0 {
                    break;
                }
```

```
                if self.is.read() & HBA_PxIS_TFES > 0 {
                    debugln!("Read disk error");
                    let _ = unsafe { Box::from_raw(cmd_table_addr as *mut
HbaCmdTbl) };
                    return false;
                }
            }

            let _ = unsafe { Box::from_raw(cmd_table_addr as *mut
HbaCmdTbl) };

            // Check again
            if self.is.read() & HBA_PxIS_TFES > 0 {
                debugln!("Read disk error");
                false
            } else {
                true
            }
        }
    } else {
        false
    }
}
```

4.16.3.5. 向磁盘写入制定扇区

我实现了一个函数`write`，用于向指定位置写入从内存写入一定数量的扇区到磁盘。该函数的工作流程主要如下所示：

1. **检查缓冲区大小**:
   - 同读操作一样，首先验证缓冲区的大小是否足够以容纳要写入的数据。
2. **清除中断状态**:
   - 使用`self.is.write((-1i32) as u32)`清除中断状态。
3. **查找命令插槽**:
   - 使用`self.find_cmdslot()`寻找一个可用的命令插槽。
4. **准备命令头**:
   - 设定命令头及其他相关参数，这里的一个关键差异是`cmd_header.set_w(true)`，这表示此操作为写操作。
5. **设置命令表和PRDT**:
   - 这部分与读操作几乎完全相同，除了此时我们从提供的缓冲区写入数据。
6. **准备FIS**:
   - 准备帧信息结构，差异在于`cmd_fis.command = ATA_CMD_WRITE_DMA_EX`，表示此为DMA写操作。

7. **等待端口不忙**:
   - 同读操作一样，确保端口不忙。
8. **发出命令**:
   - 通过设置ci（命令发出）标志来发出写命令。
9. **等待完成**:
   - 在循环中等待命令完成，监测错误标志及命令插槽的状态。
10. **检查错误并返回**:
   - 在命令完成后，再次检查是否有错误，并返回操作的结果。

代码实现如下所示:

```
pub fn write(&mut self, pos: u64, count: u32, buf: &[u8]) -> bool {
    if count as usize * 512 > buf.len() {
        panic!("错误的缓冲区大小")
    }

    self.is.write((-1i32) as u32);
    let mut spin = 0;
    if let Some(slot) = self.find_cmdslot() {
        let cmd_header = unsafe { &mut *((self.clb.read() -
AHCI_PHYSIC_BASE + AHCI_BASE) as *mut HbaCmdHeader).add(slot as usize) };
        cmd_header.set_cfl((mem::size_of::<FisRegH2d>() /
mem::size_of::<u32>()) as u8); // Command FIS size
        cmd_header.set_w(true); // Write to device
        cmd_header.prdtl = (((count - 1) >> 4) + 1) as u16;

        let mut cmd_table = HbaCmdTbl::with_entries(cmd_header.prdtl as
usize);

        let mut buf_addr = unsafe { translate_addr(buf.as_ptr() as
u64).unwrap() };
        let mut count = count;
        let prdt_entry = cmd_table.prdt_entries_mut(cmd_header.prdtl as
usize);

        for i in 0..cmd_header.prdtl as usize {
            if i < cmd_header.prdtl as usize - 1 {
                prdt_entry[i].dba = buf_addr;
                prdt_entry[i].set_dbc(8 * 1024 - 1); // 8K bytes (this
value should always be set to 1 less than the actual value)
                prdt_entry[i].set_i(true);
                buf_addr += 8 * 1024; // 8K bytes
                count -= 16; // 16 sectors
            } else {
                prdt_entry[i].dba = buf_addr;
```

```
                prdt_entry[i].set_dbc((count << 9) - 1); // 512 bytes per
sector
                prdt_entry[i].set_i(true);
            }
        }

        let cmd_fis = unsafe { &mut *(cmd_table.cfis.as_ptr() as *mut
FisRegH2d) };
        cmd_fis.fis_type = FisType::FisTypeRegH2d as u8;
        cmd_fis.set_c(1); // Command
        cmd_fis.command = ATA_CMD_WRITE_DMA_EX;

        cmd_fis.lba0 = pos as u8;
        cmd_fis.lba1 = (pos >> 8) as u8;
        cmd_fis.lba2 = (pos >> 16) as u8;
        cmd_fis.lba3 = (pos >> 24) as u8;
        cmd_fis.lba4 = (pos >> 32) as u8;
        cmd_fis.lba5 = (pos >> 40) as u8;

        cmd_fis.device = 1 << 6; // LBA mode
        cmd_fis.countl = (count & 0xFF) as u8;
        cmd_fis.counth = ((count >> 8) & 0xFF) as u8;

        let cmd_table_addr = Box::into_raw(cmd_table) as u64;
        cmd_header.ctba = unsafe
{ translate_addr(cmd_table_addr).unwrap() };

        // The below loop waits until the port is no longer busy before
issuing a new command
        while (self.tfd.read() & (ATA_DEV_BUSY | ATA_DEV_DRQ) as u32) > 0
&& spin < 1000000 {
            spin += 1;
        }
        if spin == 1000000 {
            debugln!("Port is hung");
            let _ = unsafe { Box::from_raw(cmd_table_addr as *mut
HbaCmdTbl) };
            false
        } else {
            self.ci.update(|x| *x |= 1 << slot); // 签发命令

            // Wait for completion
            loop {
                // In some longer duration reads, it may be helpful to spin
on the DPS bit
                // in the PxIS port field as well (1 << 5)
                if self.ci.read() & (1 << slot) == 0 {
```

```
                    break;
                }
                if self.is.read() & HBA_PxIS_TFES > 0 {
                    debugln!("Write disk error");
                    let _ = unsafe { Box::from_raw(cmd_table_addr as *mut
HbaCmdTbl) };
                    return false;
                }
            }

            let _ = unsafe { Box::from_raw(cmd_table_addr as *mut
HbaCmdTbl) };

            // Check again
            if self.is.read() & HBA_PxIS_TFES > 0 {
                debugln!("Write disk error");
                false
            } else {
                true
            }
        }
    } else {
        false
    }
}
```

4.16.4. 实现Read Write Seek特性

由于本项目中使用的文件系统接口需要我们提供一个实现了`Read+Write+Seek`的封装，因此我将为其创建一个封装类，实现上述三个接口。

4.16.4.1. 数据结构定义

我定义了一些常量，如下所示：

```
const SECTOR_SIZE: usize = 512;
const SECTOR_BIN_SZ: usize = 9;
const SECTOR_MASK: usize = 0x1FF;
const CACHE_SECTORS: u64 = 64;
const CACHE_SIZE: usize = SECTOR_SIZE * CACHE_SECTORS as usize; // 32KB
```

- `SECTOR_SIZE`: 表示一个扇区的大小，这里设置为512字节。这与许多存储设备的常用扇区大小相吻合。
- `SECTOR_BIN_SZ`: 这是`SECTOR_SIZE`的二进制表示的大小。因为`(2^9 = 512)`，所以这里是9。

- SECTOR_MASK: 这是一个掩码，值为0x1FF（即511）。在二进制运算中，与此掩码进行AND操作可以快速获得一个数的最低9位。
- CACHE_SECTORS: 缓存中可以存储的扇区数量，这里设置为64。
- CACHE_SIZE: 缓存的总大小，计算方法是扇区大小乘以扇区数量，结果为32KB。

接下来，我定义了这个结构体：

```
pub struct AhciDeviceReader {
    pub port_no: u64,
    max_sector: u64,
    position: usize,
    cache: Vec<u8>,
    cache_valid: bool,
    cache_start: u64,
    port: usize,
}
```

- port_no: 表示AHCI控制器上的端口号。这通常与磁盘或其他AHCI设备有关。
- max_sector: 设备上的最大扇区数，用于界定读取范围。
- position: 当前读取位置。一般用于跟踪读取操作的进度。
- cache: 一个缓冲区，用于缓存读取的数据，其大小为上面定义的CACHE_SIZE。
- cache_valid: 一个布尔值，表示缓存中的数据是否有效。如果为true，则缓存中的数据是最近读取的且可以使用；如果为false，则需要从设备中重新读取数据。
- cache_start: 缓存中数据对应的起始扇区号。例如，如果缓存存储的是从扇区10开始的数据，此值则为10。
- port: AHCI端口的虚拟地址。稍后会利用它访问端口对象。

4.16.4.2. Read 接口

首先，我实现了read_exact方法。它的目的是从AHCI设备上精确地读取一定数量的数据。根据约定，使用者传入一个可变引用的缓冲区，并期望这个缓冲区在函数执行后填满。如果成功，则返回 Ok(())，否则返回一个错误。

以下是该函数的工作流程：

1. **参数检查**:
   - 如果所请求的数据量加上当前的位置超出了设备的上限，函数直接返回一个错误。
2. **获取端口**:
   - 获取HbaPort的可变引用。这部分的代码将直接与硬件通信。
3. **计算读取的扇区数**:
   - 计算根据当前位置和请求的字节数所需的扇区数。
   - read_times 表示需要从设备读取多少次才能满足所需的数据量。

4. **从缓存中读取**:
   - 如果缓存有效且数据已经在缓存中，那么直接从缓存中读取数据并返回。
5. **从设备读取数据**:
   - 对于每次从设备读取的操作，先将缓存设置为0。
   - 根据是不是最后一次读取，来确定每次读取多少扇区。
   - 使用 `port.read` 方法从设备读取数据到缓存中。
   - 更新 `cache_start`，即缓存中数据的起始扇区。
   - 根据当前的迭代次数和之前读取的数据量，决定从缓存中拷贝多少数据到输出缓冲区。
6. **更新状态**:
   - 更新 `position`，表示当前的读取位置。
   - 设置 `cache_valid` 为 `true`，表示缓存中的数据现在是有效的。
7. **返回**:
   - 如果上述所有步骤都成功，那么返回 `Ok(())`。

此函数实际上执行了两种读取方式：首先尝试从缓存中读取（如果可能的话），如果缓存中没有所需的数据，那么它将从设备读取数据。这种机制有效地减少了不必要的设备I/O，从而提高了性能。以下是该函数的代码实现：

```
fn read_exact(&mut self, buf: &mut [u8]) -> Result<(), Self::Error> {
    if self.position + buf.len() > self.max_sector as usize * SECTOR_SIZE {
        return Err(());
    };

    let port = unsafe { &mut *(self.port as *mut HbaPort) };

    // 计算总共需要读取的扇区
    let sectors = (offset(self.position) + buf.len()) / SECTOR_SIZE + 1;
    let read_times = sectors / CACHE_SECTORS as usize;
    let start_sector = sector(self.position);

    if self.cache_valid {
        if start_sector == self.cache_start && read_times == 0 {
            let next_copy = buf.len();
            buf.copy_from_slice(&self.cache.as_slice()
[offset(self.position)..offset(self.position) + next_copy]);
            self.position += next_copy;
            return Ok(());
        }
    }

    let mut buf_pos = 0;
```

```
    #[allow(unused_assignments)]
    let mut last_read = 0;

    for i in 0..=read_times {
        self.cache.fill(0);
        if i == read_times {
            // 最后一次
            if !port.read(
                start_sector + i as u64 * CACHE_SECTORS,
                (sectors % CACHE_SECTORS as usize) as u32,
                self.cache.as_mut_slice(),
            ) {
                return Err(());
            }

            self.cache_start = start_sector + i as u64 * CACHE_SECTORS;
            last_read = (sectors % CACHE_SECTORS as usize) * SECTOR_SIZE;
        } else {
            if !port.read(start_sector + i as u64 * CACHE_SECTORS,
CACHE_SECTORS as u32, self.cache.as_mut_slice()) {
                return Err(());
            }
            self.cache_start = start_sector + i as u64 * CACHE_SECTORS;
            last_read = CACHE_SIZE;
        }
        if i == 0 {
            let next_copy = (last_read -
offset(self.position)).min(buf.len() - buf_pos);
            buf.copy_from_slice(&self.cache.as_slice()
[offset(self.position)..offset(self.position) + next_copy]);
            buf_pos += next_copy;
        } else {
            let next_copy = (last_read).min(buf.len() - buf_pos);
            let _ =
&(buf[buf_pos..]).copy_from_slice(&self.cache.as_slice()[0..next_copy]);
            buf_pos += next_copy;
        }
    }

    self.position += buf.len();
    self.cache_valid = true;

    Ok(())
}
```

在实现了read_exact后，我们便可以便捷地实现read：

```
fn read(&mut self, buf: &mut [u8]) -> Result<usize, Self::Error> {
    let read_len = buf.len().min(self.max_sector as usize * SECTOR_SIZE -
self.position);
    self.read_exact(&mut buf[0..read_len])?;
    Ok(read_len)
}
```

4.16.4.3. Write 接口

和之前类似地，我们实现了一个write_all函数来向 AHCI 设备写入数据。它的目的是向AHCI设备上精确地写入一定数量的数据。根据约定，使用者传入一个引用的缓冲区，并期望这个缓冲区在函数执行后完全写入硬盘。如果成功，则返回 Ok(())，否则返回一个错误。

这个函数的工作流程如下所示：

1. **参数检查**:
   - 同样检查是否写入的数据量加上当前位置会超出设备上限。
2. **无效化缓存**:
   - 因为是写入操作，其当前所持有的缓存将被设置为无效。这意味着在下一次读取之前，缓存中的数据不应被视为是设备上最新的数据。
3. **获取端口**:
   - 使用不安全代码来获取 HbaPort 的可变引用，这与之前的读取函数一样，是用于与硬件通信。
4. **计算需要写入的扇区数**:
   - 跟 read_exact 类似，这里也计算了根据当前位置和给定的数据量所需的扇区数。
   - write_times 表示为满足所需的数据量需要写入设备多少次。
5. **向设备写入数据**:
   - 对于每一次写入操作，首先将缓存设置为0，作为临时缓冲区。
   - 根据是不是循环的第一次迭代，决定从输入缓冲区拷贝多少数据到缓存中。
   - 然后，根据是不是最后一次写入，确定每次写入多少扇区。
   - 使用 port.write 方法从缓存写入数据到设备。
   - 更新 last_write 以追踪每次迭代写入的数据量。
6. **更新位置**:
   - 更新 position，表示当前的写入位置。
7. **返回**:
   - 如果所有步骤都成功执行，返回 Ok(())。

整体上看，这个 `write_all` 函数在功能上与 `read_exact` 函数是互补的，但方向相反。这个函数关注于从输入缓冲区取数据并写入到设备，而之前的函数则关注于从设备取数据并填入输出缓冲区。它的代码实现如下所示：

```
fn write_all(&mut self, buf: &[u8]) -> Result<(), Self::Error> {
    if self.position + buf.len() > self.max_sector as usize * SECTOR_SIZE {
        return Err(());
    };

    self.cache_valid = false;

    let port = unsafe { &mut *(self.port as *mut HbaPort) };

    // 计算总共需要写入的扇区
    let sectors = (offset(self.position) + buf.len()) / SECTOR_SIZE + 1;
    let write_times = sectors / CACHE_SECTORS as usize;
    let start_sector = sector(self.position);

    let mut buf_pos = 0;
    let mut last_write = 0;

    for i in 0..=write_times {
        self.cache.fill(0);
        if i == 0 {
            let next_copy = (last_write -
offset(self.position)).min(buf.len() - buf_pos);

self.cache.copy_from_slice(&buf[offset(self.position)..offset(self.position)
+ next_copy]);
            buf_pos += next_copy;
        } else {
            let next_copy = (last_write).min(buf.len() - buf_pos);
            let _ = &(self.cache.as_mut_slice()
[buf_pos..]).copy_from_slice(&buf[0..next_copy]);
            buf_pos += next_copy;
        }
        if i == write_times {
            // 最后一次
            if !port.write(
                start_sector + i as u64 * CACHE_SECTORS,
                (sectors % CACHE_SECTORS as usize) as u32,
                self.cache.as_slice(),
            ) {
                return Err(());
            }
            last_write = (sectors % CACHE_SECTORS as usize) * SECTOR_SIZE;
        } else {
```

```
            if !port.write(start_sector + i as u64 * CACHE_SECTORS,
CACHE_SECTORS as u32, self.cache.as_slice()) {
                return Err(());
            }
            last_write = CACHE_SIZE;
        }
    }

    self.position += buf.len();

    Ok(())
}
```

在我们实现了 `write_all` 后，`write`的实现同样易如反掌：

```
fn write(&mut self, buf: &[u8]) -> Result<usize, Self::Error> {
    let write_len = buf.len().min(self.max_sector as usize * SECTOR_SIZE -
self.position);
    self.write_all(&buf[0..write_len])?;
    Ok(write_len)
}
```

4.16.4.4. Seek 接口

Seek 接口的实现同样简单：

```
impl Seek for AhciDeviceReader {
    fn seek(&mut self, pos: SeekFrom) -> Result<u64, Self::Error> {
        match pos {
            SeekFrom::Start(pos) => {
                self.set_position(pos as usize)?;
                Ok(pos)
            }
            SeekFrom::End(npos) => {
                self.set_position((self.max_sector as i64 * SECTOR_SIZE as
i64 - npos) as usize)?;
                Ok(self.position as u64)
            }
            SeekFrom::Current(rpos) => {
                self.set_position((self.position as i64 + rpos) as usize)?;
                Ok(self.position as u64)
            }
        }
    }
}
```

我们根据三种不同的`SeekFrom`实现对应的移动即可。

限于篇幅，事件和唤醒、文件系统封装、GUI界面的实现将不再描述。若读者感兴趣，可以阅读代码的相关部分。

5. 项目展示

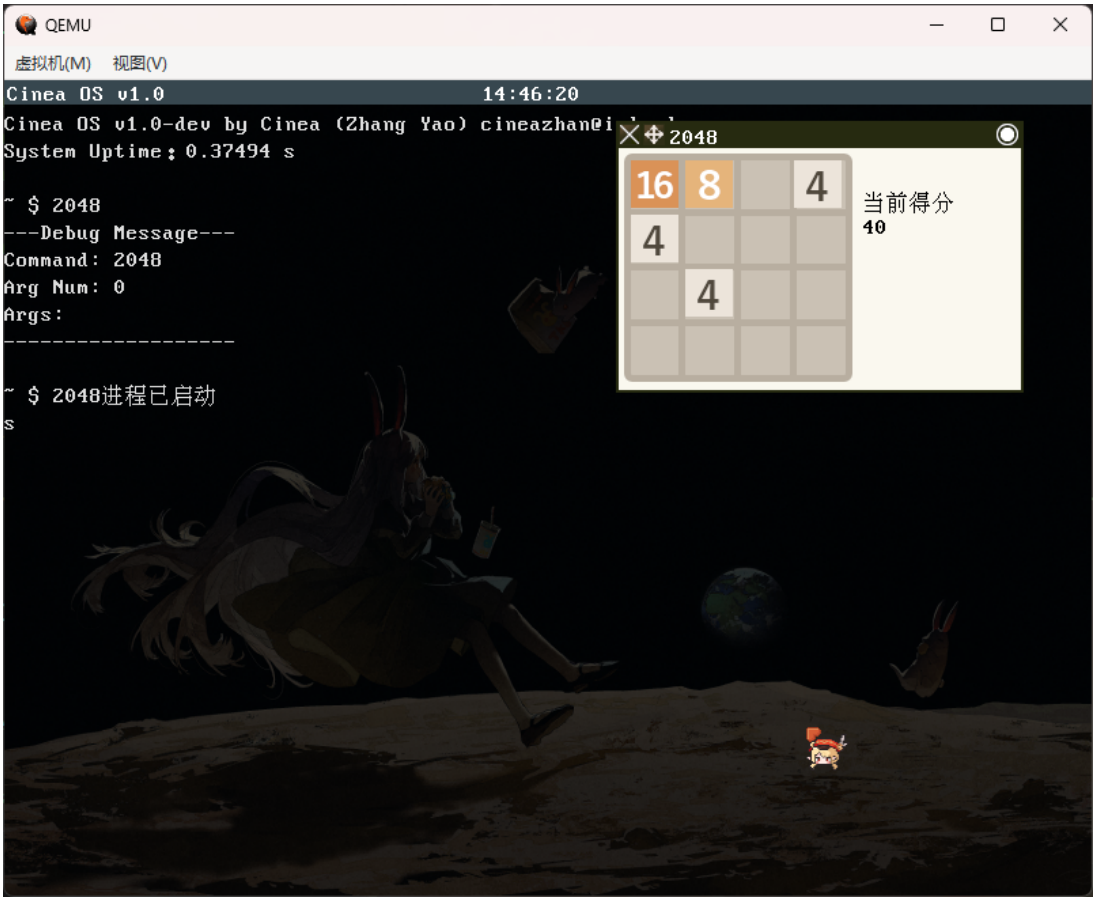

图 6: 在本项目的操作系统中运行2048游戏

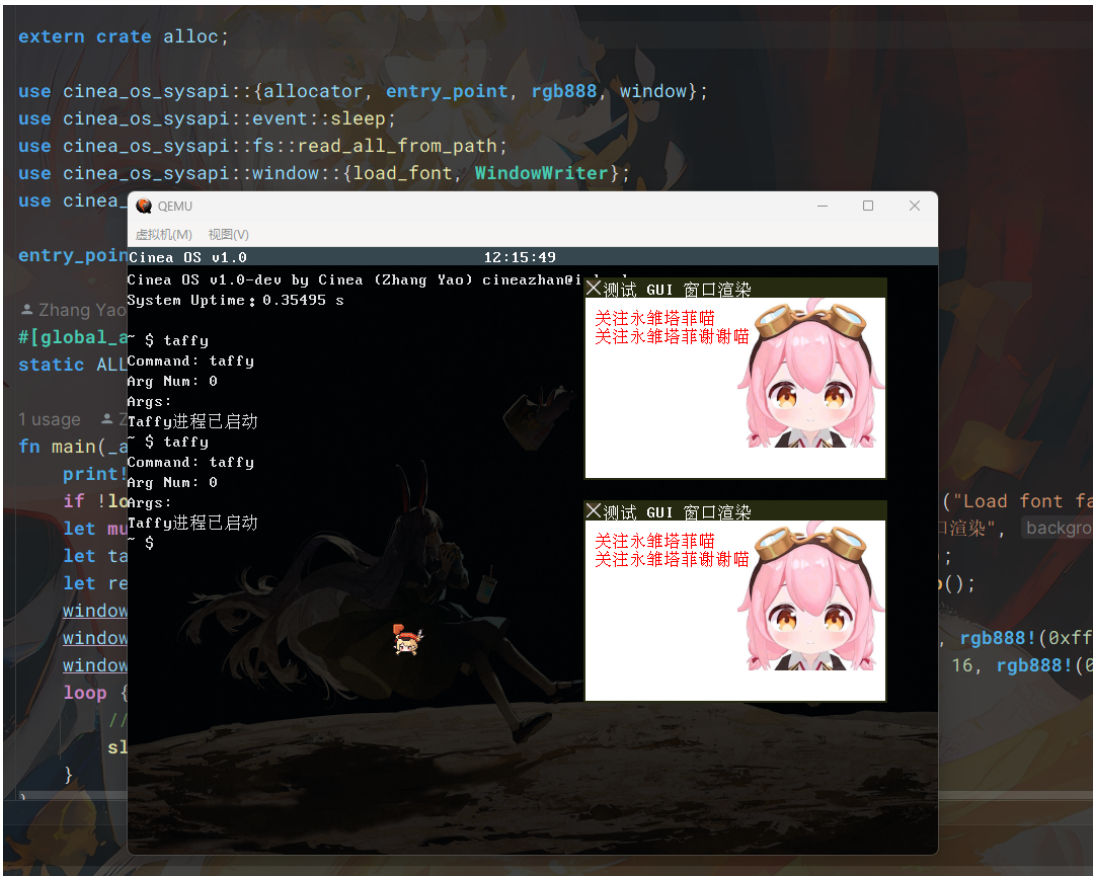

图 7: 在本项目的操作系统中同时打开和管理多窗口

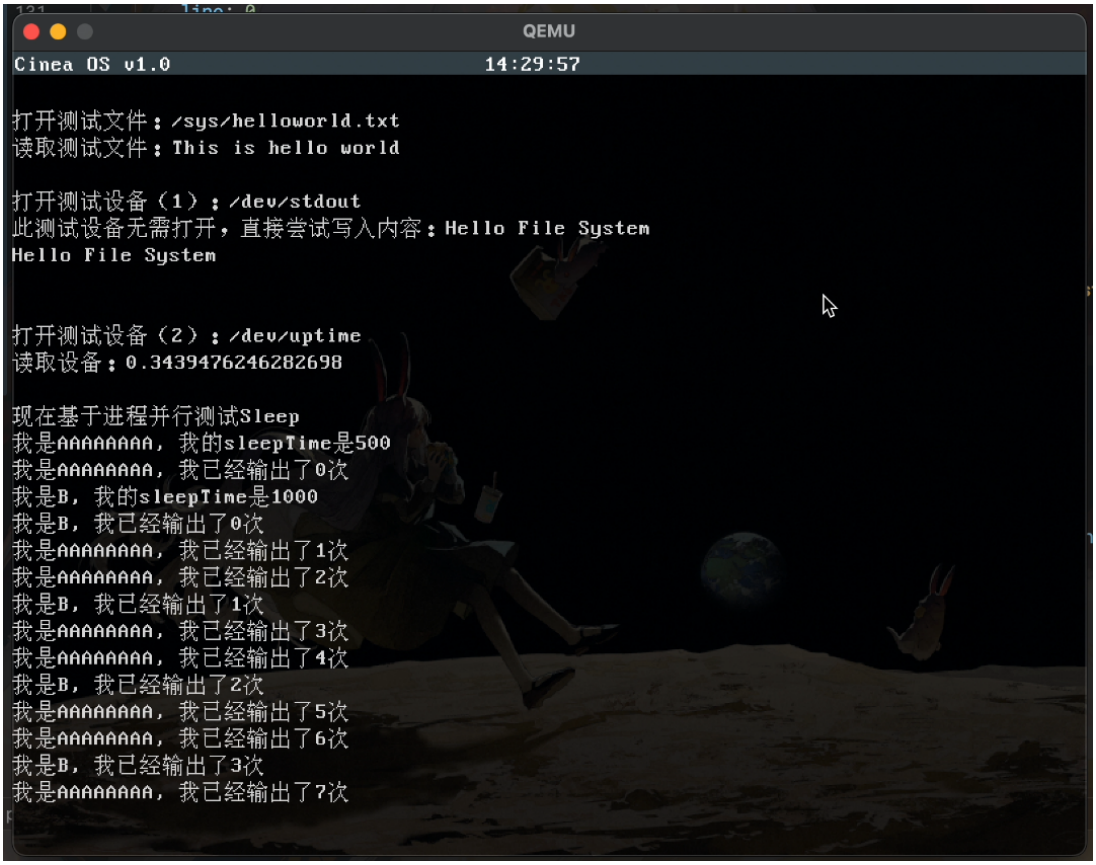

图 8: 在本项目的操作系统中两个Sleep不同时间的进程同时运行

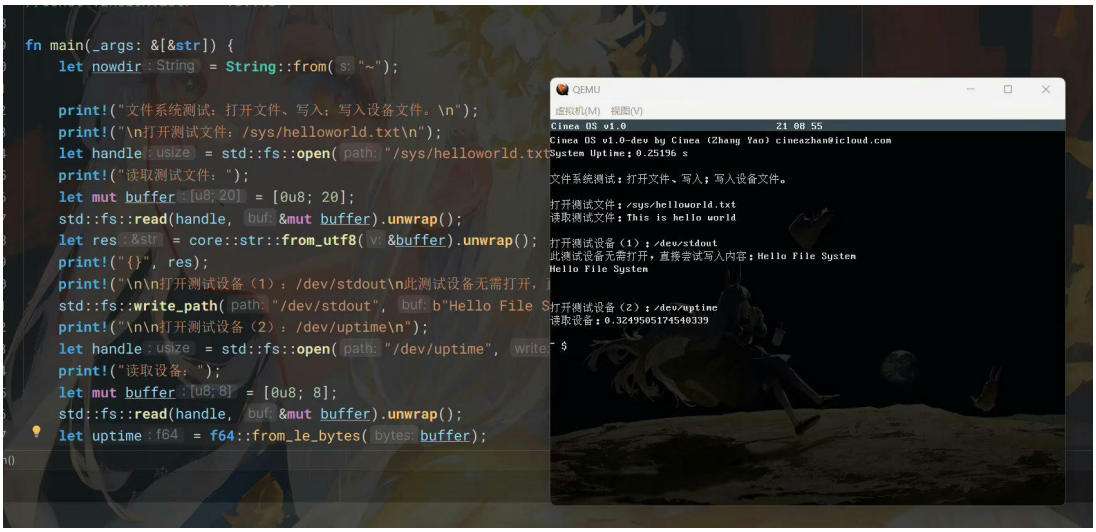

图 9: 在本项目的操作系统中成功测试文件系统和设备读写

在测试中用到的代码（部分）：

多窗口测试：

```
#![no_std]
#![no_main]

extern crate alloc;

use cinea_os_sysapi::{allocator, entry_point, ExitCode, gui, rgb888};
use cinea_os_sysapi::event::{GUI_EVENT_EXIT, wait_gui_event};
use cinea_os_sysapi::fs::read_all_from_path;
use cinea_os_sysapi::gui::{load_font, remove_window_gui, WindowWriter};
use cinea_os_sysapi::syscall::exit;
use cinea_os_userspace::print;

entry_point!(main);

#[global_allocator]
static ALLOCATOR: allocator::UserProcAllocator =
allocator::UserProcAllocator;

fn main(_args: &[&str]) {
    print!("Taffy进程已启动\n");
    if !load_font("Vonwaon", "/sys/ast/VonwaonBitmap-16px.ttf") { panic!
("Load font failed") }
    let mut window_instance: WindowWriter = gui::init_window_gui("测试 GUI
窗口渲染", rgb888!(0xffffffu32)).expect("获取窗口实例失败");
    let taffy_img = read_all_from_path("/sys/ast/taffy.bmp").unwrap();
    let resolved_taffy_img =
WindowWriter::resolve_img(taffy_img.as_slice()).unwrap();
```

```
    window_instance.display_resolved(0, 150, &resolved_taffy_img);
    window_instance.display_font_string("关注永雏塔菲喵", "Vonwaon", 10, 10,
16.0, 16, rgb888!(0xff0000u32));
    window_instance.display_font_string("关注永雏塔菲谢谢喵", "Vonwaon", 28,
10, 16.0, 16, rgb888!(0xff0000u32));
    loop {
        //print!("关注永雏塔菲喵\n");
        let (code, _arg0, _arg1, _arg2) = wait_gui_event();
        match code {
            GUI_EVENT_EXIT => {
                remove_window_gui(window_instance);
                break;
            }
            _ => {}
        }
    }
}
```

不同时间Sleep测试：启动了两个参数不同的infprint进程。

```
#![no_std]
#![no_main]

extern crate alloc;

use alloc::string::String;

use ufmt::uwrite;

use cinea_os_sysapi::{allocator, entry_point, syscall::log};
use cinea_os_userspace::std::StringWriter;

entry_point!(main);

#[global_allocator]
static ALLOCATOR: allocator::UserProcAllocator =
allocator::UserProcAllocator;

fn main(args: &[&str]) {
    let mut strout = StringWriter::new();
    if args.len() > 1 {
        let mut num = 0;
        let output = String::from(args[0]);
        let sleep_time = usize::from_str_radix(args[1], 10).unwrap();
        uwrite!(
            strout,
```

```
            "{}, 我的sleepTime是{}\n",
            output.as_str(),
            sleep_time
        )
        .unwrap();
        loop {
            uwrite!(strout, "{}, 我已经输出了{}次\n", output.as_str(),
num).unwrap();
            log(strout.value().as_bytes());
            strout.clear();
            cinea_os_sysapi::event::sleep(sleep_time);
            num += 1;
        }
    }
}
```

文件系统测试:

```
#![no_std]
#![no_main]

extern crate alloc;

use alloc::string::{String, ToString};
use alloc::vec;
use alloc::{format, vec::Vec};

use cinea_os_sysapi::{allocator, entry_point, syscall::spawn};
use cinea_os_userspace::{print, std};

entry_point!(main);

#[global_allocator]
static ALLOCATOR: allocator::UserProcAllocator =
allocator::UserProcAllocator;

//const VERSION:&str = "v0.1.0";

fn main(_args: &[&str]) {
    let nowdir = String::from("~");

    print!("文件系统测试: 打开文件、写入; 写入设备文件。\n");
    print!("\n打开测试文件: /sys/helloworld.txt\n");
    let handle = std::fs::open("/sys/helloworld.txt", false).unwrap();
    print!("读取测试文件: ");
    let mut buffer = [0u8; 20];
```

```
    std::fs::read(handle, &mut buffer).unwrap();
    let res = core::str::from_utf8(&buffer).unwrap();
    print!("{}", res);
    print!("\n\n打开测试设备 (1): /dev/stdout\n此测试设备无需打开，直接尝试写入内容:
Hello File System\n");
    std::fs::write_path("/dev/stdout", b"Hello File System\n").unwrap();
    print!("\n\n打开测试设备 (2): /dev/uptime\n");
    let handle = std::fs::open("/dev/uptime", false).unwrap();
    print!("读取设备: ");
    let mut buffer = [0u8; 8];
    std::fs::read(handle, &mut buffer).unwrap();
    let uptime = f64::from_le_bytes(buffer);
    print!(
        "{}\n\n现在基于进程并行测试Sleep\n",
        ryu::Buffer::new().format(uptime)
    );

    loop {
        //print!("{} $ ", nowdir.as_str());

        loop {}
    }
}
```

6. 项目总结

6.1. 项目总结

本项目成功地设计和实现了一个完整的多任务操作系统，专门为x86-64架构设计。基于Rust这一现代编程语言，这个操作系统不仅展示了Rust在系统编程方面的潜能，同时也保证了代码的安全性和高效性。

我的系统从底层硬件支持到上层的图形化用户界面都展现了相当的完整性。例如，操作系统原生支持64位处理，具有与现代硬件如PS/2设备、PCI设备、ATA和基于AHCI的SATA硬盘进行交互的能力。这确保了系统与广泛的硬件设备间有良好的兼容性。

内存管理和进程调度是任何操作系统的核心，我们采用了页式内存管理和链式堆内存分配技术，同时实现了多进程管理与调度，基于事件队列的事件等待和唤醒系统，确保了系统的稳定性和响应速度。

值得一提的是，本操作系统支持抢占式和非抢占式多任务，利用Rust的async/await特性，为开发者提供了一种更加直观和高效的方式来编写并发代码。

在用户界面方面，除了支持中文显示和800 x 600的高清图形显示，我们还为用户提供了图形化的工具和应用，如TTF字体的显示、BMP图片的读取，甚至还有一个简单的2048游戏，使得这个操作系统不仅仅是一个学术性项目，更有实际应用的潜力。

总的来说，本项目不仅仅展现了一个操作系统的基本功能和特性，更展现了一个成熟、稳定并具有用户友好界面的现代操作系统的面貌。在此过程中，我深切体会到了操作系统设计的挑战和乐趣，同时也为未来在这一领域的深入研究和开发打下了坚实的基础。

6.2. 过程中遇到的困难和挑战

在本项目的开发过程中，我面临了一系列具有挑战性的技术问题，具体如下：

1. **用户空间的内存问题与Rust编译器的优化**：

   在处理用户空间的过程中，我发现了与内存相关的bug。这部分问题的出现与Rust编译器对代码进行的过度优化有关。特定情况下，编译器的这种优化导致用户空间程序的执行出现错误。一个典型的例子就是使用format宏时出现的异常。

2. **切换到图形模式的问题**：

   当尝试为操作系统加入图形界面时，我遇到了切换模式的难题。特别是，系统在尝试进入实模式或V86模式时遭遇了障碍。幸运的是，通过引入Bochs Graphics Adapter (BGA) 方案，我得以绕过这个问题并成功实现了图形模式的切换。

   除此以外，刚开始我的显存只能使用最初的64KB。那时我还不知道有bank和LFB，直到后面才知道`0x200000`那个地方的显存确实最大64KB，如果需要更多空间，要么更换bank，要么用LFB。

3. **与AHCI操作相关的bug**：

   在实现基于AHCI协议的SATA硬盘读写功能时，我一直无法解决一个奇怪的bug。我无论读取什么大小、什么位置的扇区，得到的结果都是空白。经过深入研究，我发现问题出在AHCI控制器不能识别虚拟地址上，它应该接受的是物理地址。这个发现帮助我针对性地解决了问题。

4. **用户空间的调试难题**：

   最后，值得一提的是，在用户空间进行调试工作极具挑战性。与内核空间相比，用户空间提供的调试工具和方法有限（尤其是不能用format宏），这使得问题的定位和修复工作变得尤为困难。

总的来说，这些问题为我们的开发过程增添了不小的难度，但同时也提供了宝贵的学习机会，使我在面对各种技术挑战时得以不断成长和进步。

6.3. 未来可以改进的方向

在项目的开发过程中，我认识到还有一些方面存在改进的潜力，以使操作系统更加健壮、高效和安全。但是，由于时间、精力等因素，我没有把它们完全实现。具体的改进方向如下：

1. **堆内存分配与页式管理的整合**：
   当前的堆内存分配策略并没有真正结合页式内存管理。这意味着我的内存分配不是以页为单位，这在某些场景下可能会影响性能和内存使用的效率。为了优化性能和提高资源利用率，未来可以考虑将堆内存分配策略与页式管理更为紧密地结合。
2. **完全独立的进程页表**：
   尽管我为每个进程设计了独立的页表，但在实际的实施中，所有进程仍共享同一套地址分配方案。这种设计可能会在多任务环境下引发问题，尤其是关于内存隔离和保护的问题。为了增强操作系统的稳定性和安全性，我应当在后续的开发中真正实现每个进程使用其独立的页表和地址分配方案。
3. **页帧分配器的优化**：
   我当前的页帧分配器还有进一步优化的空间。为了提高分配和回收的效率，可以考虑引入如位图和链表等数据结构和技术，这不仅可以帮助我更快速地定位可用或已用的页帧，还可以提高整体的内存管理效率。

综上所述，这些建议的改进方向不仅能增强操作系统的性能和稳定性，还可以为我在未来的开发中提供新的挑战和机会。我十分期待在后续的开发中继续完善和优化这些方面，使得操作系统更加完善和高效。

6.4. 感言和致谢

设计并撰写一个操作系统是大学期间一次独特而宝贵的经历。在学习操作系统开发的过程中，从选型到着手开发，实属不易。

最初看到于老师的《Orange'S》一书，想要跟随学习，后发现难度过大，从GDT和TSS开始就几乎什么都跟不上了。最后只好放弃。之后也看过一段时间的《30天自制操作系统》，此书生动简单，作者著书时亦是一位和我年纪相差不大、爱好相似的"日本肥宅"，但无奈此书成书于二十年前，书中许多概念与工具早已过时，QEMU、64位等更是不知何物。因此，一时间，何去何从竟难以抉择。

幸而，在互联网偶遇phil-opp（Philipp Oppermann）老师的《Writing an OS in Rust》系列博客，由于我那段时间特别偏爱Rust，因此如同久旱逢甘，兴致冲冲地跟随其做了起来。作者实力不凡，不仅文章通俗易懂，更为初学者写好了大量工具，打通了阻碍初学

者体验OS开发的几座高墙————引导程序、实模式、保护模式、长模式。在phil-opp老师的指引下，我顺利地度过了异常和中断处理、VGA文本模式、内存和堆内存等几大难关。

然而，事情很难一帆风顺。phil-opp老师在2020年后便不再更新《Writing an OS in Rust》系列博客，原因是他希望能用UEFI重写这个博客系列，因此他不得不放下这一版博客的更新。但是，他的博客最终仅仅止步“协程”部分。对进程这一巨大的难点，他仅仅留下“协程的非抢占式调度为我们之后进程的抢占式调度埋下基础”这样的一句话便再未更新。如何走完剩下的路，成了埋在我心头的一根刺，让我不时心烦意乱。

这时，我在《Writing an OS in Rust》系列博客的评论区中偶然发现了一个指向vinc/moros项目的链接。这个项目继承《Writing an OS in Rust》而来，但是作者Vinc通读各大经典OS的源码，并在《Writing an OS in Rust》的基础上实现了进程、Shell等高级功能，将这个Rust系统变成了一个完整的的操作系统。我如获至宝，潜心研究moros，并在moros的基础上完成了进程、进程切换、进入用户空间和系统调用。

但是，令人遗憾的是，moros并没有走出VGA的文本模式，可是我想做的系统是图形模式，可以画窗口、设背景，显示图片。因此，我第一次尝试跳出舒适圈，尝试自己探索教程、别人的代码里没有出现过的新技术。

图形模式就是我跳出舒适圈的“第一炮”。在翻阅《30天》时，我注意到作者说可以采用VESA协议来进入图形模式。因此，我开始尝试按照书上的方法调用VESA BIOS扩展（VBE）。可是，怎么实验也不成功。后来我才知道，这是因为调用VBE只能在实模式或V86模式中完成，但我已经进入长模式了。

正当棘手之际，我偶然间在一个叫做“OSDev Wiki”的网站上认识了一种叫做Bochs Graphics Adaptor的虚拟显卡，Bochs和QEMU两个经典虚拟机都支持它。通过这个虚拟显卡的接口，就可以直接在长模式中通过I/O就切换图形模式了。这是我第一次从OSDev Wiki上获益，并且从此以后，我开始经常泡在上面查阅各种资料。这个网站上的文章内容详尽，却又不事无巨细、像官方说明书。它还会提供很多例程，对我的帮助很大。

终于，经过日复一日的开发和成长，我已经可以不再参考别人的系统，而是根据自己的想法和见解随心所欲地实现我想要的功能。具体的例子就是我的操作系统的GUI窗口。我没有询问GPT什么是最佳实践，也没有看别人的代码。我自己构想和设计出了这个功能模块的架构，并成功地在代码中将它表达了出来。虽然它的性能不尽如人意，但是已经完全符合我的预期。

这也许就是成长的感觉，从懵懂、一无所知的小白，从看手把手的教程开始，到复现别人的GitHub代码，再到根据OSDev Wiki自己设计和实现，乃至于对于一些不涉及具体硬

件的功能需求，可以完全自己独立解决。这种飞速成长的感觉，如果不是经历了这样的一次开发，又何尝能体味到呢？

回首本次项目，从5月30日开始第一行代码，到今天（8月19日），将近三个月的时间里，写下了一万多行代码。惊叹于自己完成了如此巨大的工作，更感叹自己代码能力的巨大成长。如今，项目要画上句号了，竟也有点依依不舍。

感谢学校、老师和助教给了我这样的一次开发OS的机会。感谢phil-opp和vinc优质的代码，感谢OSDev Wiki的编者们。他们都给予了我很大的帮助。最后，也感谢自己，能够一直坚持下来，完成这项工作。

2023年8月19日