



TONGJI UNIVERSITY

# MinmusOS 项目说明文档

课题名称	基于 Rust 语言的操作系统的设计与实现
副 标 题	同济大学操作系统课程设计项目说明文档
学 院	软件学院
专 业	软件工程
学生姓名	林继申
学 号	2250758
指导教师	王冬青
日 期	2024 年 8 月 27 日

装

订

线

# 基于 Rust 语言的操作系统的设计与实现

## 摘要

随着技术的不断进步，操作系统的安全性和效率成为了研究和开发的重点。传统上，操作系统多采用 C 语言开发，虽然这种语言提供了高效的系统级底层操作能力，但同时也带来了许多内存安全问题，如缓冲区溢出和空指针解引用等。为了解决这些问题，越来越多的开发者和研究者开始探索使用 Rust 语言来开发操作系统，因为 Rust 语言提供了保证内存安全的机制，而不牺牲性能。

MinmusOS 是一个基于 Rust 语言和 Intel IA-32 (x86) 架构开发的多任务操作系统，它实现了系统内核与用户空间的分离，并允许用户使用标准运行库开发用户应用程序。其核心功能涵盖从引导过程到多层次的系统管理。引导程序负责启动系统，切换到保护模式，并加载内核。内核功能包括中断管理、异常处理、设备驱动（如键盘和磁盘）、任务调度、系统调用、内存管理以及一个基于 FAT16 的文件系统。此外，内核还支持 VGA 文本模式输出和实时时钟管理。标准运行库提供数学计算、互斥同步、输出打印、随机数生成、数据排序和字符串处理等基本功能。系统还包括一些基本应用程序，如汉诺塔解决方案，展示了如何使用这些库来开发更复杂的用户空间应用程序。

在技术实现方面，MinmusOS 充分利用了 Rust 语言的核心特性——所有权和借用机制，来应对传统操作系统开发中的一些关键挑战。这些挑战主要包括内存泄漏和并发执行中的数据竞争等问题。Rust 通过其独特的所有权系统，确保了每个数据只有一个清晰的所有者，并且通过编译时的生命周期和借用检查，它可以在不牺牲性能的前提下，防止空指针解引用、双重释放等内存安全问题。整体而言，MinmusOS 通过其模块化的设计展示了 Rust 语言在系统级编程中的强大潜力和内存安全特性。

综上，MinmusOS 通过结合 Rust 语言的内存安全特性和高效的系统设计，提供了一个稳定且功能丰富的操作系统框架，展示了现代编程语言在解决传统操作系统开发挑战中的有效性和潜力。

本项目所有源代码均已在 GitHub 平台上开源发布，并遵循 MIT 许可证协议。GitHub 仓库地址：[https://github.com/MinmusLin/Minmus\\_Operating\\_System\\_in\\_Rust](https://github.com/MinmusLin/Minmus_Operating_System_in_Rust).

本仓库包含的代码和资料仅用于个人学习和研究目的，不得用于任何商业用途。请其他用户在下载或参考本仓库内容时，严格遵守学术诚信原则，不得将这些资料用于任何形式的作业提交或其他可能违反学术诚信的行为。本人对因不恰当使用仓库内容导致的任何直接或间接后果不承担责任。请在使用前务必确保您的行为符合所在学校或机构的规定，以及适用的法律法规。如有任何问题，请通过电子邮件与我联系。

联系方式：minmuslin@outlook.com

关键词：Rust 操作系统开发，Intel IA-32 (x86) 架构，操作系统内核，引导程序，标准运行库

装

订

线

# The Design and Implementation of an Operating System Based on Rust Language

## ABSTRACT

As technology continues to advance, security and efficiency have become focal points in operating system research and development. Traditionally, operating systems have been developed using C, a language that offers efficient low-level system operations but also introduces numerous memory safety issues, such as buffer overflows and null pointer dereferencing. To address these problems, an increasing number of developers and researchers are exploring the use of Rust to develop operating systems because Rust offers mechanisms to ensure memory safety without sacrificing performance.

MinmusOS is a multitasking operating system developed with Rust and based on the Intel IA-32 (x86) architecture. It features separation between the system kernel and user space, allowing users to develop applications with standard runtime libraries. Its core functionalities range from the boot process to multilevel system management. The bootloader is responsible for starting the system, switching to protected mode, and loading the kernel. Kernel functionalities include interrupt management, exception handling, device drivers (such as keyboards and disks), task scheduling, system calls, memory management, and a FAT16-based file system. Additionally, the kernel supports VGA text mode output and real-time clock management. The standard runtime libraries provide basic functionalities like mathematical computations, mutex synchronization, output printing, random number generation, data sorting, and string processing. The system also includes basic applications, such as solutions for the Towers of Hanoi, demonstrating how to use these libraries to develop more complex user-space applications.

Technically, MinmusOS fully utilizes Rust's core features—ownership and borrowing mechanisms—to address key challenges in traditional operating system development, such as memory leaks and data races during concurrent execution. Rust's unique ownership system ensures that each piece of data has one clear owner, and through compile-time life cycle and borrowing checks, it prevents issues like null pointer dereferencing and double-freeing without sacrificing performance. Overall, MinmusOS showcases Rust's strong potential and memory safety features in system-level programming through its modular design.

In summary, by integrating Rust's memory safety features with an efficient system design, MinmusOS provides a stable and feature-rich operating system framework, demonstrating the effectiveness and potential of modern programming languages in addressing traditional operating system development challenges.

All source code of this project has been open-sourced on the GitHub platform and is released under the

MIT License. The GitHub repository can be found at [https://github.com/MinmusLin/Minmus\\_Operating\\_System\\_in\\_Rust](https://github.com/MinmusLin/Minmus_Operating_System_in_Rust).

The code and materials contained in this repository are intended for personal learning and research purposes only and may not be used for any commercial purposes. Other users who download or refer to the content of this repository must strictly adhere to the principles of academic integrity and must not use these materials for any form of homework submission or other actions that may violate academic honesty. I am not responsible for any direct or indirect consequences arising from the improper use of the contents of this repository. Please ensure that your actions comply with the regulations of your school or institution, as well as applicable laws and regulations, before using this content. If you have any questions, please contact me via email.

Contact: minmuslin@outlook.com

**Key words:** Rust OS Dev, Intel IA-32 (x86) Arch, OS Kernel, Bootloader, Standard Runtime Library

## 目 录

1 引言 .....	1
1.1 项目背景 .....	1
1.2 项目简介 .....	1
1.3 基本概念 .....	2
1.3.1 操作系统 (Operating System) .....	2
1.3.2 内核 (Kernel) .....	3
1.3.3 命令行解释器 (Shell) .....	4
1.3.4 英特尔 IA-32 架构 (Intel IA-32 Architecture) .....	4
1.3.5 实模式 (Real Mode) .....	5
1.3.6 保护模式 (Protected Mode) .....	5
1.3.7 Rust 语言 .....	6
1.4 文档结构 .....	6
2 开发环境 .....	8
2.1 开发环境概述 .....	8
2.1.1 Windows 11 .....	8
2.1.2 Windows Subsystem for Linux (WSL) .....	9
2.1.3 Ubuntu 22.04 LTS .....	9
2.1.4 QEMU 模拟机 (qemu-system-i386) .....	10
2.1.5 JetBrains RustRover .....	11
2.1.6 Rust 语言 .....	11
2.2 开发环境搭建 .....	12
2.2.1 配置 Windows Subsystem for Linux (WSL) .....	12
2.2.2 配置 Ubuntu 22.04 LTS .....	13
2.2.3 配置 Rust 语言开发环境 .....	14
2.2.4 配置编译环境与 QEMU 模拟机 (qemu-system-i386) .....	16
2.3 MinmusOS 项目配置 .....	17
2.3.1 项目工作空间和编译选项配置 .....	17
2.3.2 特定硬件平台的目标配置 .....	18
2.3.3 Rust 工具链配置 .....	21
2.3.4 项目自动化构建过程 (Makefile) 配置 .....	22
2.4 运行 MinmusOS 项目 .....	25
3 系统设计 .....	26
3.1 系统设计原则 .....	26

3.1.1 封装 (Encapsulation) .....	26
3.1.2 高内聚低耦合 (High Cohesion and Low Coupling) .....	26
3.1.3 面向对象 (Object-Oriented) .....	26
3.1.4 异常处理 (Exception Handling) .....	26
3.1.5 维护性 (Maintainability) .....	27
3.1.6 模块性 (Modularity) .....	27
3.1.7 扩展性 (Scalability) .....	27
3.1.8 安全性 (Security) .....	27
3.2 系统架构设计 .....	27
3.3 系统技术选型 .....	28
3.3.1 编程语言 .....	28
3.3.2 硬件架构 .....	34
3.4 项目结构设计 .....	35
4 引导程序实现 .....	37
4.1 引导过程概述 .....	37
4.2 第一阶段引导: 实模式 .....	37
4.2.1 主引导记录 (Master Boot Record, MBR) .....	37
4.2.2 BIOS 中断 (BIOS Interrupts) .....	38
4.2.3 启动模块 (Boot Module) .....	40
4.2.4 磁盘读取器模块 (Disk Reader Module) .....	41
4.2.5 主启动程序 (Main Boot Program) .....	44
4.2.6 链接器脚本 (Linker Script) .....	46
4.3 第二阶段引导: 保护模式与内核加载 .....	47
4.3.1 保护模式 (Protected Mode) .....	47
4.3.2 全局描述符表 (Global Descriptor Table) .....	47
4.3.3 磁盘读取器模块 (Disk Reader Module) .....	49
4.3.4 全局描述符表模块 (GDT Module) .....	50
4.3.5 打印器模块 (Printer Module) .....	53
4.3.6 主启动程序 (Main Boot Program) .....	56
4.3.7 链接器脚本 (Linker Script) .....	59
5 内核功能实现 .....	62
5.1 中断 (Interrupts) .....	62
5.1.1 中断描述符表 (Interrupt Descriptor Table) .....	63
5.1.2 中断服务例程 (Interrupt Service Routines) .....	66

5.1.3 CPU 异常 (CPU Exceptions) .....	68
5.1.4 可编程中断控制器 (Programmable Interrupt Controller) .....	69
5.2 驱动程序 (Drivers) .....	70
5.2.1 可编程中断控制器驱动程序 (Programmable Interrupt Controller Driver) .....	70
5.2.2 键盘驱动程序 (Keyboard Driver) .....	73
5.2.3 高级技术附件磁盘驱动程序 (Advanced Technology Attachment Disk Driver) .....	78
5.3 多任务处理 (Multitasking) .....	83
5.3.1 上下文切换 (Context Switching) .....	84
5.3.2 CPU 调度器 (CPU Scheduler) .....	84
5.3.3 任务管理器 (Task Manager) .....	87
5.4 系统调用 (System Calls) .....	91
5.4.1 系统调用概述 .....	91
5.4.2 系统调用实现 .....	91
5.5 VGA 文本模式 (VGA Text Mode) .....	93
5.5.1 标准色彩定义 .....	93
5.5.2 printc 方法 .....	94
5.5.3 prints 方法 .....	94
5.5.4 delete 方法 .....	95
5.5.5 get_cursor_position 方法 .....	96
5.5.6 set_cursor_position 方法 .....	96
5.5.7 scroll 方法 .....	97
5.5.8 set_colors 方法和 reset_colors 方法 .....	97
5.5.9 new_line 方法 .....	98
5.5.10 clear 方法 .....	98
5.6 内存管理 (Memory Management) .....	100
5.6.1 内存分配器 (Memory Allocator) .....	100
5.6.2 全局分配器接口 (GlobalAlloc Trait) .....	101
5.6.3 分页管理器 (Paging Manager) .....	102
5.6.4 分页目录 (Page Directory) 与页表 (Page Table) .....	103
5.7 文件系统 (File System) .....	105
5.7.1 FAT16 文件系统 .....	105
5.7.2 FAT16 全局常量定义 .....	107
5.7.3 Header 数据结构 .....	107
5.7.4 Entry 数据结构 .....	107

5.7.5 FatDriver 数据结构 .....	109
5.7.6 FAT16 文件操作 .....	109
5.7.7 示例文本文件 .....	111
5.8 定时器 (Timer) .....	112
5.8.1 日期时间 (Datetime) .....	115
5.8.2 UNIX 时间戳 (UNIX Timestamp) .....	116
5.8.3 CPU 时钟周期计数 (CPU Ticks) .....	116
5.9 命令行解释器 (Shell) .....	118
5.9.1 Shell 数据结构 .....	119
5.9.2 cal 命令 .....	122
5.9.3 cat <filename> 命令 .....	122
5.9.4 clear 命令 .....	124
5.9.5 color 命令 .....	124
5.9.6 date 命令 .....	125
5.9.7 echo <text> 命令 .....	125
5.9.8 exit 命令 .....	125
5.9.9 help 命令 .....	125
5.9.10 hostname 命令 .....	126
5.9.11 kill <pid> 命令 .....	126
5.9.12 ls 命令 .....	127
5.9.13 ps 命令 .....	128
5.9.14 pwd 命令 .....	128
5.9.15 reboot 命令 .....	128
5.9.16 run <appname> 命令 .....	128
5.9.17 shutdown 命令 .....	130
5.9.18 ticks 命令 .....	130
5.9.19 timestamp 命令 .....	130
5.9.20 uname 命令 .....	130
5.9.21 whoami 命令 .....	130
5.10 主启动程序 (Main Boot Program) .....	130
5.10.1 编译器指令 .....	130
5.10.2 常量定义 .....	131
5.10.3 内核启动过程 .....	132
5.10.4 PANIC 处理器 .....	132

5.11 链接器脚本 (Linker Script) .....	135
6 标准运行库实现 .....	138
6.1 标准运行库概述 .....	138
6.2 math 库 .....	139
6.3 mutex 库 .....	141
6.3.1 互斥对象 (Mutual Exclusion Object) .....	141
6.3.2 Mutex 数据结构 .....	141
6.3.3 Mutex 库作用 .....	142
6.4 print 库 .....	143
6.4.1 Printer 数据结构 .....	143
6.4.2 print 宏 .....	145
6.4.3 println 宏 .....	145
6.5 rand 库 .....	146
6.5.1 线性同余生成器 (Linear Congruential Generator) .....	146
6.5.2 异或移位算法 (Xorshift) .....	147
6.5.3 中平方方法 (Middle Square Method) .....	148
6.5.4 斐波那契线性同余生成器 (Fibonacci Linear Congruential Generator) .....	148
6.5.5 梅森旋转算法 (Mersenne Twister) .....	149
6.5.6 时间种子生成器 (Time Seed Generator) .....	150
6.5.7 线性反馈移位寄存器 (Linear Feedback Shift Register) .....	152
6.5.8 组合生成器 (Combined Generator) .....	152
6.6 sort 库 .....	154
6.6.1 冒泡排序 (Bubble Sort) .....	154
6.6.2 选择排序 (Selection Sort) .....	154
6.6.3 插入排序 (Insertion Sort) .....	155
6.6.4 合并排序 (Merge Sort) .....	155
6.6.5 快速排序 (Quick Sort) .....	155
6.6.6 堆排序 (Heap Sort) .....	155
6.6.7 希尔排序 (Shell Sort) .....	155
6.6.8 计数排序 (Counting Sort) .....	155
6.6.9 桶排序 (Bucket Sort) .....	155
6.6.10 基数排序 (Radix Sort) .....	156
6.7 string 库 .....	156
7 应用程序实现 .....	157

---

7.1 应用程序构建脚本 .....	157
7.2 应用程序链接器脚本 .....	157
7.3 应用程序入口 .....	157
7.4 应用程序实现 .....	160
8 系统演示 .....	163
9 总结与展望 .....	184
9.1 项目总结 .....	184
9.2 项目挑战 .....	185
9.3 项目展望 .....	185
参考资料 .....	187
谢辞 .....	188

装

订

线

## 1 引言

在本节（章节 1）中，笔者将介绍 MinmusOS 的项目背景、项目简介，以及开发 MinmusOS 相关的一些基本概念，文档结构部分旨在为读者提供清晰的导览。通过详细解释操作系统的功能和概念，以及为何选择 Rust 作为开发语言，笔者希望能使读者了解 MinmusOS 的设计理念和实现方法。此外，文档也将详细讨论 MinmusOS 的技术选型和架构设计，从而揭示如何通过现代编程技术解决传统操作系统开发中的常见问题。

接下来的章节将依次展开，详细介绍每一个关键组成部分的工作原理和设计细节，从操作系统的基础结构到具体的功能实现，每一部分都将提供必要的理论背景和代码实例。通过这种结构化的展示，笔者希望不仅帮助读者理解 MinmusOS 如何运作，而且展示如何利用 Rust 语言的特性来构建安全、高效且可靠的操作系统软件。

### 1.1 项目背景

长期以来，C 语言因其接近硬件的特性和高效性，一直是开发操作系统内核的首选语言。然而，C 语言在内存安全和并发处理方面存在一些固有的问题，这些问题往往导致安全漏洞和系统不稳定。随着计算机科学的发展，需要一种更安全、更现代的语言来提高操作系统的安全性和可维护性。

Rust 语言自推出以来，以其独特的内存管理机制，提供了零成本抽象和保证内存安全的特性，逐渐成为系统级编程的有力竞争者。Rust 通过所有权、借用检查和生命周期的概念，有效避免了空指针解引用、缓冲区溢出等常见的安全问题，这些都是开发操作系统时需要特别关注的问题。

在开源社区中，已经出现了使用 Rust 开发操作系统的先例，例如 Philipp Oppermann 的 Blog-OS<sup>1</sup> 和清华大学的 rCore-OS<sup>2</sup>。这些项目不仅展示了 Rust 在操作系统开发中的实际应用，也验证了 Rust 在系统级编程中的可行性和效益。

借着在同济大学软件学院 2024 年春季课程操作系统设计的契机，笔者决定使用 Rust 语言开发一个名为 MinmusOS 的操作系统。MinmusOS 项目不仅会包含基本的操作系统功能，如进程管理、内存管理、文件管理、设备管理和安全管理，还将利用 Rust 的现代语言特性，提高系统的整体安全性和性能。

### 1.2 项目简介

MinmusOS 是一个基于 Rust 语言和 Intel IA-32 (x86) 架构开发的多任务操作系统，它实现了系统内核与用户空间的分离，并允许用户使用标准运行库开发用户应用程序。

MinmusOS 的功能：

(1) 引导程序

① 引导 MinmusOS

<sup>1</sup>[https://github.com/phil-opp/blog\\_os](https://github.com/phil-opp/blog_os)

<sup>2</sup><https://github.com/rcore-os/rCore>

② 切换到非真实模式（在 16 位实模式下使用 32 位地址）

③ 加载内核（从磁盘将内核复制到受保护的内存）

④ 加载全局描述符表

⑤ 切换到保护模式

⑥ 跳转内核

## (2) 内核

① 加载中断描述符表

② CPU 异常处理器与 PANIC 处理器

③ 可编程中断控制器（8259 PIC）驱动程序

④ PS/2 键盘驱动程序

⑤ ATA 磁盘驱动程序

⑥ CPU 轮转调度器

⑦ 任务管理器

⑧ 系统调用

⑨ VGA 文本模式

⑩ 页式内存管理

⑪ FAT16 文件系统

⑫ RTC 定时器

⑬ 命令行解释器支持 20 种可执行命令

## (3) 标准运行库

① math 数学库

② mutex 互斥库

③ print 打印库

④ rand 随机数生成库

⑤ sort 排序库

⑥ string 字符串处理库

## (4) 应用程序

① 汉诺塔解决方案应用程序

② 允许用户使用标准运行库开发用户应用程序

## 1.3 基本概念

### 1.3.1 操作系统（Operating System）

操作系统是一种控制计算机系统及其资源运行的软件。它是许多计算设备的核心软件组件，包括个人计算机、服务器、智能手机和嵌入式设备。操作系统通过提供多个硬件抽象层，使用户和应

应用程序能够通过一个简单的接口使用设备，隐藏了底层硬件的复杂性。根据操作系统的复杂程度，它们可以满足不同的需求。例如，多任务操作系统，如本项目设计与实现的系统，可以在同一台机器上同时运行多个任务。

操作系统的主要功能可能包括：

- (1) 管理内存和其他系统资源
- (2) 实施安全和访问策略
- (3) 调度和复用进程和线程
- (4) 动态启动和关闭用户程序
- (5) 提供基本的用户界面和应用程序编程接口

这意味着操作系统承担了以有组织、优化和安全的方式管理可用资源的责任。当进程运行时，操作系统必须确保不会发生意外和有害的行为，例如进程读取另一个进程的内存或占用 CPU 过长时间。正因如此，操作系统必须尽可能安全。而 Rust 编程语言可以通过其编译器在编译时保证内存安全和线程安全，从而满足这一要求。

并非所有的操作系统都提供所有这些功能。例如，单任务系统如 MS-DOS 不会调度进程，而嵌入式系统如 eCOS 可能没有用户界面，或者只能与一组静态的用户程序一起工作。

操作系统不是：

- (1) 计算机硬件
- (2) 特定的应用程序（如文字处理器、网页浏览器等）
- (3) 一套实用程序（如在许多 Unix 派生系统中使用的 GNU 工具）
- (4) 开发环境（尽管一些操作系统，如 UCSD Pascal 或 Smalltalk-80，包含解释器和 IDE）
- (5) 图形用户界面（尽管许多现代操作系统将 GUI 作为操作系统的一部分）

尽管大多数操作系统都会配备这些工具，但它们本身并不是操作系统的必要部分。一些操作系统，如 Linux，可能以几种不同的打包形式分发，这些形式被称为发行版，它们可能拥有不同的应用程序和实用程序套件，并可能以不同的方式组织系统的某些方面。尽管如此，它们都是同一基本操作系统的版本，不应被视为不同类型的的操作系统。

### 1.3.2 内核 (Kernel)

内核是操作系统的中心部分，它负责处理所有基础级别的任务，尤其是与硬件直接交互的任务。内核的角色是桥接硬件和运行在计算机上的应用程序之间的通信。用户通常不会直接与内核交互，但它是系统运行的基础。

内核的主要职责包括：

- (1) **中断处理：**内核响应由硬件产生的事件（称为中断）。例如，当按下键盘上的键时，内核的中断处理程序会获取按键编号，并将其转换成相应的字符存储在缓冲区中，供其他程序使用。
- (2) **系统调用：**系统调用是用户级程序请求操作系统服务的方式。例如，打开文件、启动其他

程序等。内核需要检查传入的参数是否有效，然后执行内部操作以完成请求。

(3) **资源管理**: 内核管理计算机的资源，如 CPU、内存和输入/输出设备。它负责资源的分配和回收，确保系统的高效运行。

(4) **进程调度**: 内核负责进程和线程的调度。它决定哪个程序在何时使用 CPU，以及如何在多个程序间公平地分配处理器时间。

(5) **抽象层**: 内核通常定义一些基础的抽象概念，如文件、进程、套接字和目录等，这些都对应于它记住的关于上次操作的内部状态，使程序可以更高效地执行一系列操作。

用户程序通常不直接发出系统调用（除了某些汇编程序），而是使用标准库来处理与内核交互所需的参数格式化和系统调用的生成。

总之，内核是操作系统最关键的组成部分，它提供了一个稳定和安全的环境，使得上层的应用程序能够不用关心硬件的复杂性而进行运行。

### 1.3.3 命令行解释器（Shell）

Shell 是一个特殊的程序，通常集成在操作系统发行版中，为用户提供与计算机交互的界面。Shell 的表现形式可能因系统而异（命令行、文件浏览器等），但其基本概念始终如一：

(1) **程序启动**: 允许用户选择要启动的程序，并可选地给它提供特定于会话的参数。

(2) **文件操作**: 允许对本地存储执行简单操作，如列出目录内容、在系统中移动和复制文件等。

为了完成这些操作，Shell 需要发出多个系统调用。Shell 也可被其他程序用来启动程序。

现代的 Shell 还具有各种额外功能，包括：

(1) **自动补全**: 通过按 TAB 键（或任何首选键），正在输入的单词将被自动补全为有效的 Shell 命令、文件、目录等。多次按自动完成键可以在其他补全可能性之间切换。

(2) **字符插入**: 用户可以使用箭头键在输入的内容中移动。在句子中间键入新字符时，字符将被“插入”。

(3) **Shell 历史记录**: 通过使用上下箭头键，用户可以浏览之前的输入。

(4) **滚动**: 当控制台的行数超过其高度时，将输出保存在缓冲区中，并允许用户在控制台中上下滚动。

(5) **脚本**: 一些 Shell 具有自定义的脚本语言。脚本语言的例子包括 Bash 或 DOS 批处理。

总之，Shell 是操作系统中的一个核心组件，它通过友好的用户界面简化了用户与系统的交互，同时提供了执行复杂任务的强大命令行工具。

### 1.3.4 英特尔 IA-32 架构（Intel IA-32 Architecture）

Intel IA-32 架构，也被称为 x86，是 Intel 开发的一种 32 位指令集架构。从 90 年代后期开始，IA-32 架构成为了广泛计算设备的基础，尤其是在个人计算机和服务器领域。它的发展重点之一是向后兼容性，使其能够运行几十年来开发的大量软件。这一特性使得 IA-32 架构在计算领域中发挥

了重要作用，成为了行业标准。

IA-32 是一种复杂指令集计算机 (CISC) 架构，允许在单条指令内执行复杂的操作。它包含多个通用寄存器，支持多种数据类型和寻址模式。此外，IA-32 架构还引入了保护模式 (Protected Mode)，这是一个操作模式，提供了硬件级别的安全功能，如特权级别、分段、虚拟内存和分页。这些特性使得 IA-32 架构不仅具有强大的性能，还为操作系统提供了增强的安全性和稳定性。

### 1.3.5 实模式 (Real Mode)

实模式 (Real Mode) 是一种处理器运行模式，最早由 Intel 8086 CPU 提出。它是最简单的 CPU 运行模式，也是所有 x86 处理器在启动时的默认模式。实模式主要特征如下：

(1) **内存寻址能力**：在实模式下，处理器只能直接寻址 1MB ( $2^{20}$  字节) 的内存。这是因为地址线只有 20 根，地址寄存器和段寄存器组合起来生成一个 20 位的物理地址。

(2) **无内存保护**：实模式不支持现代操作系统常用的多任务处理和内存保护机制。所有程序都对全内存有完全的访问权限，没有任何隔离机制，因此一个程序可以轻易地影响到系统中的其他程序。

(3) **兼容性**：实模式提供了与早期的 8086 和 8088 处理器的向后兼容性，允许旧的软件在新处理器上无修改地运行。

(4) **段寄存器**：使用段寄存器 (如 CS、DS、ES、SS) 来引用内存。这些寄存器存储一个 16 位的段地址，通过与偏移量相结合，形成一个完整的物理地址。例如，段地址放在段寄存器，偏移量放在其他如 IP 或 DI 等寄存器中，二者结合才能访问到具体的内存位置。

实模式的限制导致了后来保护模式 (Protected Mode) 的发展，保护模式支持更大的内存寻址空间 (最初是 16MB，后来扩展到更多)，并引入了内存保护和多任务等特性，是现代操作系统所依赖的运行模式。

### 1.3.6 保护模式 (Protected Mode)

保护模式 (Protected Mode) 是 x86 架构处理器支持的一种运行模式，首次出现在 Intel 80286 微处理器中。这种模式提供了比实模式更为高级的功能，包括更大的内存寻址能力、多任务支持以及严格的内存保护机制。保护模式的主要特点包括：

(1) **扩展的内存寻址**：在保护模式下，处理器可以寻址超过 1MB 的内存空间。例如，80286 可以寻址高达 16MB 的内存，而更高级的 80386 及以后的处理器可以寻址高达 4GB 的内存。

(2) **内存保护**：保护模式引入了内存保护机制，每个程序都在其自己的内存空间运行，互不干扰。操作系统可以控制程序对内存的访问权限，防止程序访问非授权的内存区域。

(3) **多任务支持**：保护模式支持硬件级的多任务处理。处理器可以在不同的程序之间快速切换，每个程序运行在独立的环境中，提高了系统的效率和稳定性。

(4) **分段和分页**：在保护模式下，内存依旧被分为不同的段，但每个段的属性 (如段的大小、访

问权限等) 可以通过段描述符在全局描述符表 (GDT) 或局部描述符表 (LDT) 中定义。此外，保护模式还支持分页机制，这允许操作系统将物理内存分成固定大小的页面，实现虚拟内存的管理，这样程序可以使用比物理内存更多的虚拟内存地址。

(5) **中断和异常处理**: 保护模式增强了对中断和异常的处理能力。每种中断和异常都可以通过中断描述符表 (IDT) 进行配置，使得系统能更有效地响应各种硬件和软件事件。

保护模式的引入标志着从单一的、不受保护的计算环境向现代多任务操作系统的重大转变。这种模式是现代操作系统如 Windows、Linux 等能够实现高效、稳定运行的基础。

### 1.3.7 Rust 语言

Rust 是一种系统级编程语言，近年来逐渐成为操作系统开发的重要工具。与大多数操作系统通常使用的 C 语言相比，Rust 在内存管理和安全性方面引入了全新的方法。

C 语言因其对硬件的低级别访问能力、直接内存操作的特性以及高效代码编写的能力而广泛用于操作系统开发。然而，这种低级别的控制权也带来了相应的风险。程序员必须手动管理内存，包括显式地分配和释放内存。这种灵活性虽然强大，但如果处理不当，就会引发严重的漏洞或问题，例如内存泄漏和悬空指针 (dangling pointers)。这些问题不仅难以调试，还可能被恶意攻击者利用，导致执行任意代码或泄露敏感用户数据。

为了避免这些问题，高级编程语言通常使用垃圾回收 (Garbage Collection) 技术，通过运行时环境自动管理内存。当程序运行时，垃圾回收器会定期暂停程序，扫描内存以寻找不再使用的变量并自动释放它们。然而，这种方法的缺点是会带来性能开销和长时间的暂停，因此这些语言通常不用于操作系统的开发。

Rust 编程语言采用了一种完全不同的内存管理方法，利用其独特的所有权 (Ownership) 机制在编译时确保内存操作的安全性，从而在运行时几乎不带来额外的开销。Rust 不仅保持了类似于 C 和 C++ 的低级别编程语言的性能，还提供了高级编程语言的内存安全性，结合了两者的优势而没有明显的缺点。

所有权机制的概念实际上非常直观，它遵循一组简单的规则来管理内存：

- (1) 每个值都有一个所有者
- (2) 同一时间只能有一个所有者
- (3) 当所有者超出作用域时，该值会被自动释放

这些规则使得 Rust 能够在保持与 C 和 C++ 类似的灵活性的同时，确保内存的安全性。Rust 的设计理念使其成为一个既能提供系统级语言性能，又能在编译时防止内存错误的强大工具，特别适合用于操作系统开发和其他需要高性能和安全性的应用程序。

### 1.4 文档结构

本文的文档结构如下：

- 
- (1) **引言**: 介绍 MinmusOS 的项目背景、项目简介, 以及与 MinmusOS 开发相关的一些基本概念, 提供读者对整个操作系统的设计理念和实现方法的初步了解。
  - (2) **开发环境**: 详细描述了项目的开发环境搭建, 包括操作系统、编程语言和所需软件的安装配置, 以确保开发环境的一致性和项目的可复现性。
  - (3) **系统设计**: 深入探讨操作系统的架构设计、系统设计原则, 以及各个模块如何协同工作, 从封装、模块性到安全性等方面详细解释设计决策。
  - (4) **引导程序实现**: 讲解操作系统启动的整个流程, 包括实模式到保护模式的切换, 详细介绍了如何通过引导程序加载和启动内核。
  - (5) **内核功能实现**: 详述内核的主要功能, 包括中断处理、驱动程序、多任务处理和系统调用等, 展示如何通过具体的代码实现这些复杂功能。
  - (6) **标准运行库实现**: 描述了操作系统中标准运行库的实现, 包括数学库、互斥库和打印库等, 解释了如何支持系统运行和用户程序的开发。
  - (7) **应用程序实现**: 介绍了如何构建和链接操作系统的用户空间应用程序, 包括应用程序的入口点、构建脚本和链接器脚本的配置。
  - (8) **系统演示**: 通过具体示例展示操作系统的功能和性能, 验证设计和实现的正确性, 并提供系统运行时的界面和操作演示。
  - (9) **总结与展望**: 总结项目的成果和遇到的挑战, 提出未来工作的方向和对操作系统发展的展望, 包括技术进步和功能扩展的可能性。

## 2 开发环境

在本节(章节 2)中,笔者将详细介绍 MinmusOS 项目的开发环境,包括使用的操作系统(Windows 11)、开发工具(Windows Subsystem for Linux、Ubuntu 22.04 LTS、QEMU 模拟器、JetBrains RustRover、Rust 语言),并深入讲解开发环境的搭建过程、项目配置及自动化构建的设置。最后,笔者还将介绍如何在配置好的环境中运行 MinmusOS 项目,以确保开发者能够顺利进行开发和调试。

### 2.1 开发环境概述

本项目的开发环境如下:

#### (1) 开发环境

- ① Windows 11 家庭中文版 23H2
- ② Windows Subsystem for Linux (WSL) 2.2.4.0
- ③ Ubuntu 22.04.3 LTS
- ④ QEMU 模拟机 (qemu-system-i386) 6.2.0

#### (2) 开发软件

- ① JetBrains RustRover 2024.1.7

#### (3) 开发语言

- ① Rust 语言 (Rustup: 1.27.1, Rustc: 1.82.0-nightly, Cargo: 1.82.0-nightly)

#### 2.1.1 Windows 11

使用 Windows 系统来开发操作系统内核,尽管可能不如 Linux 环境那样常见,但仍然有其独特的优势和便利性。以下是使用 Windows 系统开发操作系统内核的几个主要优势:

(1) **熟悉的开发环境**: 对于习惯使用 Windows 的开发者来说,继续在这个环境下工作可以减少学习新工具和操作系统的时间,使他们能够更专注于开发工作本身。Windows 的用户界面、文件系统管理、任务管理等都是开发者熟悉的。

(2) **强大的开发工具**: Windows 支持多种强大的开发环境,如 Visual Studio Code。这些 IDE 提供了先进的代码编辑、项目管理、版本控制和调试工具。

(3) **使用 WSL 提升开发效率**: 通过集成了 WSL (Windows Subsystem for Linux), Windows 环境可以无缝地运行 Linux 工具和软件,结合了 Linux 的命令行工具和 Windows 的图形界面优势。这允许开发者在不离开 Windows 的情况下使用 Linux 环境,例如使用 GCC、Make、GDB 等工具进行内核开发。

(4) **文档和社区资源**: Windows 拥有庞大的开发者社区和丰富的文档资源,对于解决开发中的问题和学习新技术都极为有用。

总体而言,虽然 Linux 通常是内核开发的首选环境,但 Windows 提供的工具、支持和兼容性使其成为许多情况下一个可行且有效的选择,尤其是结合了 WSL 后,Windows 在操作系统内核开发

方面的适用性大大提高。

### 2.1.2 Windows Subsystem for Linux (WSL)

使用 Windows Subsystem for Linux (WSL) 在开发操作系统内核时具有一系列优势，特别是对于习惯使用 Windows 环境的开发者来说。这些优势包括：

(1) **集成 Windows 和 Linux 环境**: WSL 允许开发者在 Windows 系统上运行 Linux 环境，无需重启进入另一个操作系统或使用虚拟机。这意味着可以利用 Windows 的图形界面和生产力工具（如 VSCode），同时执行 Linux 命令行工具和应用。

(2) **简化开发流程**: 对于需要同时访问 Windows 和 Linux 工具的开发任务，WSL 提供了极大的便利。开发者可以在相同的文件系统中访问项目文件，使用 Windows 编辑器编辑代码，然后在 Linux 环境中编译和测试，无需文件转移或复制。

(3) **资源占用更少**: 与传统的虚拟机相比，WSL 提供了更轻量级的解决方案。它直接在 Windows 内核上运行，减少了资源占用，启动和运行速度更快，对系统性能的影响也更小。

(4) **方便的环境管理**: WSL 允许开发者安装多个 Linux 发行版，可以在不同的项目或任务之间切换不同的环境。例如，可以在一个发行版上进行开发工作，而在另一个发行版上进行测试。

(5) **直接访问硬件和系统调用**: WSL 使用真实的 Linux 内核，这使得它在处理系统调用和操作硬件时表现得更接近传统 Linux 系统。这对于需要进行底层系统开发的项目尤其重要，因为它允许开发者在接近生产环境的条件下测试和开发。

(6) **持续集成和交叉编译支持**: 使用 WSL，开发者可以在同一机器上进行交叉编译，为不同的平台构建应用程序，包括 Linux、Windows 和其他操作系统。这种能力对于开发涉及多平台支持的内核或应用程序非常有用。

(7) **社区和官方支持**: Microsoft 对 WSL 的持续更新和支持确保了其与现代硬件和软件技术的兼容性。此外，广泛的开发者社区也提供了大量的教程、工具和第三方应用支持，这对于解决开发中的问题非常有帮助。

WSL 是为那些想要在 Windows 系统上利用 Linux 开发工具的开发者提供了一种非常实用的解决方案，它结合了两个系统的优点，提高了开发效率和灵活性。

### 2.1.3 Ubuntu 22.04 LTS

使用 Ubuntu 作为开发环境，尤其是在开发操作系统内核这类底层项目时，有几个显著的优势：

(1) **稳定性和长期支持**: Ubuntu 22.04 LTS (Long Term Support, 长期支持) 版本提供了长达五年的安全更新和维护。这意味着开发者可以在一个稳定且长期受支持的平台上工作，无需担心频繁更换操作系统或缺乏安全更新。

(2) **与生产环境一致**: 在 Ubuntu 上开发可以确保软件在生产环境中运行时表现得更加可靠和高效，因为开发和生产环境可以保持高度一致。

(3) 广泛的社区支持和资源: Ubuntu 拥有庞大的用户和开发者社区, 这意味着大量的文档、论坛和支持资源可供查阅。这对于解决开发中遇到的问题非常有帮助。

(4) 开源工具和库的兼容性: Ubuntu 提供了丰富的开源开发工具和库。对于操作系统开发而言, 这些工具 (如 GCC、Make、GDB) 都是不可或缺的, 而且通常在 Linux 系统上的兼容性和性能都非常好。

(5) 适合底层开发: Linux 系统提供了丰富的底层系统调用和接口, 这对于内核开发是非常重要的。开发者可以直接与硬件和底层系统资源交互, 更方便地实现和测试内核级功能。

(6) 环境一致性和隔离性: 使用容器和虚拟化技术 (如 Docker 和 QEMU), 可以在 Ubuntu 上轻松创建和管理隔离的开发环境。这对于测试不同的配置和开发环境至关重要。

总之, 选择 Ubuntu 作为开发环境, 可以为操作系统内核的开发提供一个稳定、高效、兼容性好的基础, 同时也利于将来在类似环境中部署和运行。

#### 2.1.4 QEMU 模拟机 (qemu-system-i386)

QEMU 是一个功能强大的开源机器模拟器和虚拟化解决方案, 对于操作系统内核的开发尤为重要的。以下是使用 QEMU 的一些主要优势:

(1) 多平台支持: QEMU 能够模拟多种处理器架构, 包括 x86、ARM、PowerPC、SPARC、和 MIPS 等。这意味着开发者可以在一个平台上开发和测试为其他平台设计的内核和应用程序, 非常适合交叉平台开发。

(2) 环境隔离: 使用 QEMU 进行开发可以确保测试环境与主机操作系统隔离。这种隔离可以防止潜在的软件错误影响到主机系统, 特别是在开发涉及底层硬件交互的系统软件时。

(3) 无需实际硬件: QEMU 允许开发者在没有物理目标硬件的情况下进行开发和测试。这不仅降低了成本, 还可以在硬件到达前开始开发工作, 加速开发周期。

(4) 调试支持: QEMU 与各种调试工具 (如 GDB) 集成, 可以进行详细的步进执行和调试。这对于操作系统内核开发尤为重要, 因为它允许开发者在内核运行时进行观察和修改。

(5) 快照和即时状态保存: QEMU 支持保存和恢复虚拟机的状态 (称为快照)。这使得开发者可以快速回滚到一个已知的良好状态, 并从那里重新开始测试, 极大地提高了测试的效率。

(6) 网络模拟: QEMU 还能模拟网络环境, 允许开发者测试内核的网络功能, 如协议栈和驱动程序, 而无需实际的网络硬件。

(7) 性能和资源利用: 虽然 QEMU 为功能丰富性提供了强大的支持, 但它在性能上的优化也非常有效。QEMU 的用户模式模拟可以运行应用程序和驱动程序, 而不必模拟整个操作系统, 从而减少资源消耗。

(8) 版本更新和社区支持: QEMU 持续更新和改进, 提供了最新的功能和改进, 确保开发者可以利用最新的技术进行开发。同时, QEMU 的广泛用户和开发者社区提供了丰富的文档、工具和支持。

总体来说，QEMU 提供了一个强大、灵活、成本效益高的平台，用于开发、测试和调试操作系统内核和其他系统级软件，特别是在多架构和虚拟化环境中。这使得它成为操作系统开发者的重要工具。

### 2.1.5 JetBrains RustRover

JetBrains 的 RustRover 是一款专门为 Rust 语言设计的集成开发环境（IDE），其提供了许多功能来支持 Rust 开发，特别是在开发操作系统内核这类复杂项目时。在使用 RustRover 开发操作系统内核时，具有如下优势：

(1) **代码编辑与分析**: RustRover 提供了强大的代码编辑器，支持代码高亮、自动格式化、智能补全等功能，能够极大地提高代码编写的效率。它具备深度的代码分析功能，可以帮助开发者快速定位到潜在的代码问题，比如生命周期错误、类型不匹配等常见于 Rust 开发中的问题。

(2) **高级调试工具**: RustRover 集成了强大的调试器，支持条件断点、表达式求值、变量观察等功能，这对于操作系统内核的调试至关重要。支持对内存布局、堆栈跟踪等低级特性进行检查，这对于底层系统编程尤其有用。

(3) **跨平台支持和远程开发**: RustRover 支持跨平台开发，可以在 Windows、Linux 或 macOS 上进行 Rust 开发。它还支持远程开发功能，允许开发者在远程服务器上直接编写代码、编译和调试，这对于操作系统开发尤其重要，因为操作系统常常需要在特定硬件或仿真环境（如 QEMU）中运行和测试。

(4) **版本控制集成**: RustRover 提供了与 Git 等版本控制系统的无缝集成，这使得管理大型项目的源代码变得更为方便。

(5) **项目和构建管理**: 该 IDE 提供了对 Cargo 的完整支持，包括依赖管理、构建配置和测试，并可以直接从 IDE 中运行构建脚本和测试，极大地简化了构建过程。

(6) **社区和插件**: JetBrains 拥有活跃的开发者社区，可以轻松找到大量有用的插件和资源，以扩展 IDE 的功能并适应特定开发需求。

总的来说，RustRover 为 Rust 操作系统内核开发提供了一套完整的工具和功能，使开发更加高效和专业。其强大的代码分析和调试功能，以及便捷的远程开发支持，是开发高性能和低级系统软件的理想选择。

### 2.1.6 Rust 语言

Rust 是一门现代的系统编程语言，专为提供高性能与内存安全而设计。Rust 具有如下特点：

(1) **语言设计和安全性特点**: Rust 的目标是允许开发者构建高效且可靠的系统级软件，同时通过语言层面的约束，消除传统语言（如 C 和 C++）中常见的内存错误。Rust 的独特之处在于其所有权模型，该模型通过精确控制资源（如内存）的所有权、借用规则和生命周期管理，确保在编译时就消除数据竞争和内存泄漏等问题。

(2) **内存管理机制**: Rust 通过其所有权和借用系统实现了编译期内存安全。每个值在 Rust 中有一个称为其“所有者”的变量，且同一时间内只能有一个所有者。当所有者超出作用域时，值和其占用的内存会被自动清理。此外，Rust 通过借用规则（可变借用和不可变借用），允许程序在保证安全性的同时访问数据，而不产生运行时开销。这些特性使 Rust 能够在不使用垃圾收集的情况下，有效管理内存，避免传统编程语言中常见的内存安全问题。

(3) **类型系统和数据抽象**: Rust 拥有一个表达能力强大的类型系统和类型推断功能，支持泛型、特征（traits）和生命周期（lifetimes）。这些特性允许 Rust 程序表达高层次的抽象而不牺牲性能。例如，特征可以用来定义共享的行为，泛型则允许在不同类型之间重用代码，而生命周期则确保引用的有效性，防止悬垂引用等问题。这样的类型系统增强了代码的安全性和维护性，使得复杂的系统设计更加简洁和健壮。

(4) **并发编程的支持**: Rust 的设计从一开始就考虑到并发，其所有权和借用机制自然地避免了数据竞争。Rust 提供了多种并发编程模型，包括使用消息传递来交换数据、共享状态的线程安全智能指针等。这些机制都是在编译时进行安全检查的，确保并发操作的安全性，大大简化了并发系统的开发和维护。

(5) **错误处理和异常管理**: Rust 采用 `Result<T, E>` 和 `Option<T>` 类型来显式处理可能的错误和不存在的值，这与 C 语言中常见的隐式错误代码或异常处理不同。这种方法鼓励开发者前置处理所有潜在的错误情况，从而增加代码的可靠性。此外，Rust 也支持通过 `panic!` 宏处理不可恢复的错误，当系统遇到严重错误时立即终止执行，这类似于其他语言的异常抛出机制。

通过这些现代的语言特性，Rust 提供了一个更安全、更可控的环境，适合开发需要高度可靠性和性能的操作系统内核。这使 Rust 成为一个对于系统级编程尤为有利的选择。

对于 C/C++ 语言在操作系统开发中的传统地位、C/C++ 语言在操作系统开发中的劣势以及 Rust 语言在操作系统开发中的优势在小节 3.3 中有详细阐述。

## 2.2 开发环境搭建

### 2.2.1 配置 Windows Subsystem for Linux (WSL)

WSL 是一个在 Windows 操作系统上运行 Linux 环境的兼容层，允许用户直接在 Windows 中安装和使用 Linux 环境。

配置 Windows Subsystem for Linux (WSL) 的步骤如下：

- (1) 在 Visual Studio Code 官网下载并安装 Visual Studio Code，如图 2.1。
- (2) 在 Visual Studio Code 中向本地安装扩展：WSL，以允许用户在 Windows 中使用 Linux 环境，如图 2.2。
- (3) 在“控制面板”>“程序”>“程序和功能”>“启用或关闭 Windows 功能”中，确定“适用于 Linux 的 Windows 子系统”已被选中。
- (4) 在“Windows 任务管理器”>“性能”>“CPU”选项卡中，确定“虚拟化”已启用。

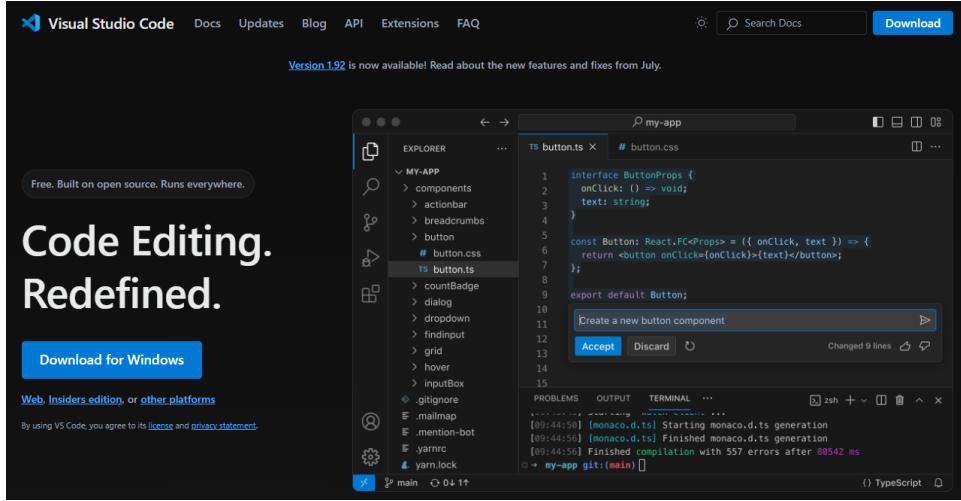


图 2.1 Visual Studio Code 官网



WSL v0.88.2  
Microsoft [microsoft.com](#) | 28,831,944 | ★★★★★ (78)  
Open any folder in the Windows Subsystem for Linux (WSL) and take advantage of Visual Studio Code's full feature set.  
 禁用  隔离  自动更新

图 2.2 安装 WSL 扩展

(5) 启动 Windows PowerShell，执行命令 `wsl --version`，以查看 WSL 版本，命令执行结果如图 2.3，以验证安装成功。

```
PS C:\Users\lenovo> wsl --version
WSL 版本: 2.2.4.0
内核版本: 5.15.153.1-2
WSLg 版本: 1.0.61
MSRDC 版本: 1.2.5326
Direct3D 版本: 1.611.1-81528511
DXCore 版本: 10.0.26091.1-240325-1447.ge-release
Windows 版本: 10.0.22631.4037
```

图 2.3 WSL 版本信息

## 2.2.2 配置 Ubuntu 22.04 LTS

Ubuntu 22.04 LTS 可以为操作系统内核的开发提供一个稳定、高效、兼容性好的基础，同时也利于将来在类似环境中部署和运行。

下面是配置 Ubuntu 22.04 LTS 的步骤：

(1) 启动 Windows PowerShell，执行命令 `wsl --install Ubuntu-22.04`，以安装 Ubuntu 22.04 LTS 版本到 WSL，命令执行结果如图 2.4。

(2) 在 Ubuntu 中执行命令 `lsb_release -a`，以查看 Linux 发行版的详细信息，命令执行结果如图 2.5，以验证安装成功。

```

PS C:\Users\lenovo> wsl --install Ubuntu-22.04
Ubuntu 22.04 LTS 已安装。
正在启动 Ubuntu 22.04 LTS...
Installing, this may take a few minutes...
wsl: 检测到 localhost 代理配置, 但未镜像到 WSL。NAT 模式下的 WSL 不支持 localhost 代理。
Please create a default UNIX user account. The username does not need to match your Windows username.
For more information visit: https://aka.ms/wslusers
Enter new UNIX username: minmuslin
wsl: 检测到 localhost 代理配置, 但未镜像到 WSL。NAT 模式下的 WSL 不支持 localhost 代理。
New password:
Retype new password:
passwd: password updated successfully
wsl: 检测到 localhost 代理配置, 但未镜像到 WSL。NAT 模式下的 WSL 不支持 localhost 代理。
wsl: 检测到 localhost 代理配置, 但未镜像到 WSL。NAT 模式下的 WSL 不支持 localhost 代理。
Installation successful!
wsl: 检测到 localhost 代理配置, 但未镜像到 WSL。NAT 模式下的 WSL 不支持 localhost 代理。
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

Welcome to Ubuntu 22.04.3 LTS (GNU/Linux 5.15.153.1-microsoft-standard-WSL2 x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

This message is shown once a day. To disable it please create the
/home/minmuslin/.hushlogin file.
minmuslin@MinmusLin:~$
```

图 2.4 配置 Ubuntu 22.04 LTS

```

minmuslin@MinmusLin:~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 22.04.3 LTS
Release:        22.04
Codename:       jammy
```

图 2.5 Ubuntu 版本信息

### 2.2.3 配置 Rust 语言开发环境

Rust 是一种注重安全、并发和性能的系统编程语言，旨在帮助开发者构建可靠和高效的软件。

RustRover 是 JetBrains 开发的针对于 Rust 编程语言的集成开发环境 (IDE)，对 Rust 语言开发提供了良好的支持。

配置 Rust 语言开发环境的步骤如下：

- (1) 在 JetBrains 官网下载并安装 JetBrains RustRover，如图 2.6。
- (2) 启动 RustRover，在“文件”>“远程开发”>“连接到 WSL”中，选择 WSL 实例为“Ubuntu-22.04”，点击“下一页”。
- (3) 在“选择 IDE 和项目”中，选择 IDE 版本为“RustRover 2024.1.7 (241.18034.106) | 下载最新”，选择项目目录为“/home/minmuslin/MinmusOS”，如图 2.7，点击“下载 IDE 并连接”。
- (4) 在“文件”>“设置”>“插件（主机）”中，安装插件“Chinese (Simplified) Language Pack / 中文语言版”和“Makefile Language”，如图 2.8。

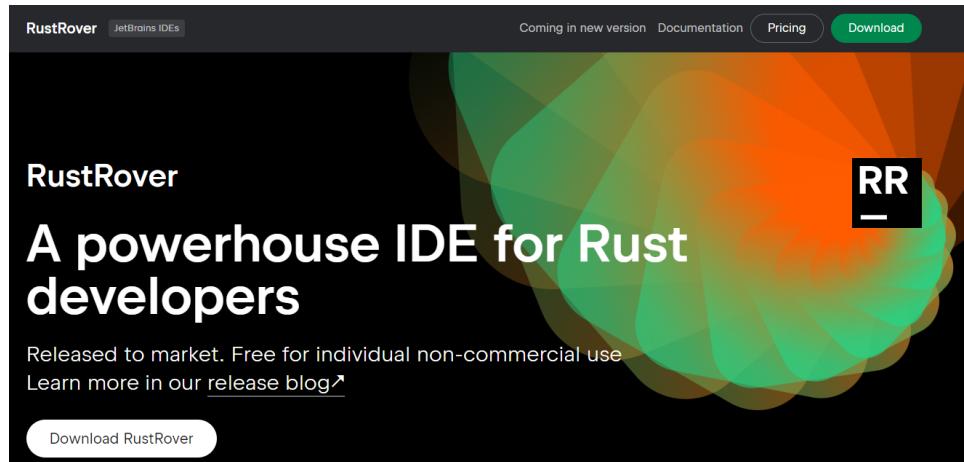


图 2.6 JetBrains 官网



图 2.7 配置远程开发

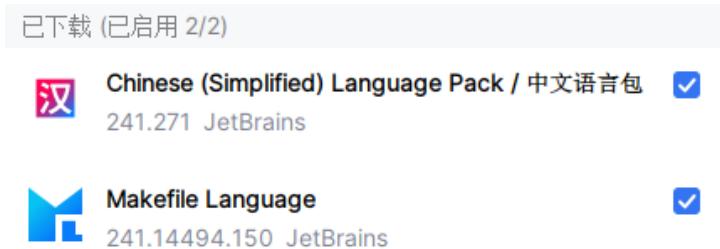


图 2.8 安装 RustRover 插件

(5) 新建终端，执行代码 2.9，以配置 Rust 工具链，代码执行结果如图 2.10，以验证安装成功。每行代码的作用如下：

- ① `curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh`: 这行代码通过 curl 命令从指定的 URL 下载 Rust 安装脚本，并通过管道(|)将其传递给 sh (Shell) 来执行。选项`--proto '=https'` 确保使用 HTTPS 协议，`--tlsv1.2` 指定使用 TLSv1.2 版本进行安全连接，`-sSf` 是几个选项的组合，`-s` 表示静默模式，`-S` 表示错误时显示错误，`-f` 表示失败时不显示 HTTP 错误。
- ② `source $HOME/.cargo/env`: 执行这行代码会加载 Rust 的环境配置文件，该文件通常由 rustup 安装程序在安装 Rust 时创建于`$HOME/.cargo` 目录下。`source` 命令用于执行该文件中的命令，以便在当前会话中设置环境变量，使得 Rust 工具链（如`rustc`、`cargo`）可以直接在命令行中使用。
- ③ `rustup update`: 这条命令用于更新 Rust 工具链到最新版本。`rustup` 是 Rust 的版本管理和安装工具，可以用来管理不同的 Rust 版本和相关工具。
- ④ `rustup default nightly`: 设置 Rust 的默认工作版本为“nightly”版本。Rust 有几个发布频道：`stable`、`beta` 和 `nightly`。`nightly` 频道提供最新的功能，但可能不够稳定。这行命令将 `nightly` 频道设置为默认的 Rust 版本。
- ⑤ `rustup --version`: 显示当前安装的 `rustup` 工具的版本信息。这对于确认 `rustup` 是否成功安装及其版本非常有用。
- ⑥ `rustup --version`: 显示 Rust 编译器 (`rustc`) 的版本信息。这用于确认 Rust 编译器已经安装并可用，同时显示当前编译器的版本。
- ⑦ `cargo --version`: 显示 Rust 的包管理器和构建工具 (`cargo`) 的版本信息。`Cargo` 用于 Rust 项目的依赖管理和构建，这行命令确认 `Cargo` 的安装状态及其版本。

```

1 curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
2 source $HOME/.cargo/env
3 rustup update
4 rustup default nightly
5 rustup --version
6 rustc --version
7 cargo --version

```

代码 2.9 配置 Rust 工具链

## 2.2.4 配置编译环境与 QEMU 模拟机 (qemu-system-i386)

在 Ubuntu 系统配置编译环境的过程中，需要安装一系列的基本工具和库，以支持软件的编译和构建。执行命令 `sudo apt update && sudo apt install build-essential mtools dosfstools fdisk qemu-system-x86` 以配置编译环境与 QEMU 模拟机 (qemu-system-i386)。

```
minmuslin@MinmusLin:~/MinmusOS$ rustup --version
rustup 1.27.1 (54dd3d00f 2024-04-24)
  info: This is the version for the rustup toolchain manager, not the rustc compiler.
  info: The currently active `rustc` version is `rustc 1.82.0-nightly (506052d49 2024-08-16)`
minmuslin@MinmusLin:~/MinmusOS$ rustc --version
rustc 1.82.0-nightly (506052d49 2024-08-16)
minmuslin@MinmusLin:~/MinmusOS$ cargo --version
cargo 1.82.0-nightly (2f738d617 2024-08-13)
```

图 2.10 Rust 版本信息

每个安装的包的作用如下：

(1) **build-essential**: 这是一个包含编译软件所需的基本编译工具（如 gcc 编译器、make 工具等）的元包。

(2) **mtools**: 一套用来访问 MS-DOS 磁盘和文件系统的工具，这在处理与 DOS 或 Windows 系统相关的磁盘映像时非常有用。

(3) **dosfstools**: 包含用于创建和验证 MS-DOS FAT 文件系统在 Unix 和类 Unix 系统上的磁盘的工具，如 mkfs.vfat 工具。

(4) **fdisk**: 一个磁盘分区表操作工具，用于创建和修改磁盘分区。

(5) **qemu-system-x86**: qemu-system-x86 是 QEMU (Quick Emulator) 项目的一部分，它是一个功能强大的开源模拟器和虚拟化工具。特别地，qemu-system-x86 是用于模拟 x86 和 x86\_64 (也称为 AMD64) 架构计算机的程序。这使得开发者和测试者可以在不同的操作系统和硬件配置上模拟和运行软件，而无需物理硬件。

## 2.3 MinmusOS 项目配置

### 2.3.1 项目工作空间和编译选项配置

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

代码 2.11 .cargo/config.toml 配置文件

配置文件.cargo/config.toml (代码 2.11) 的作用如下：

(1) **build-std**: 这个选项用于指定需要构建的标准库部分。在嵌入式开发或操作系统开发中，通常没有完整的标准库支持，所以需要指定要构建哪些核心库。

① **core**: 提供基础的 Rust 功能，适用于没有操作系统支持的环境。

② **compiler\_builtins**: 提供编译器相关的内建函数，这在没有标准库的情况下非常重要。

③ **alloc**: 提供内存分配支持，但不依赖全局堆，适用于特定的内存管理环境。

(2) **build-std-features**: 指定构建这些标准库时要启用的特性。**compiler-builtins-mem** 是

为了让 `compiler_builtins` 包含内存操作的实现，这在操作系统内核开发中可能会用到。

```

1 [workspace]
2 members = ["boot", "bootloader", "kernel", "apps/*"]
3 resolver = "2"
4
5 [workspace.package]
6 version = "0.4.0"
7 authors = ["Jishen Lin <minmuslin@outlook.com>"]
8 edition = "2021"
9
10 [profile.dev]
11 panic = "abort"
12 opt-level = 1
13
14 [profile.release]
15 panic = "abort"
16 opt-level = 1
17
18 [profile.dev.package.boot]
19 opt-level = "s"
20 codegen-units = 1
21 debug = false
22 overflow-checks = false
23
24 [profile.release.package.boot]
25 opt-level = "s"
26 codegen-units = 1
27 debug = false
28 overflow-checks = false

```

代码 2.12 Cargo.toml 配置文件

配置文件 Cargo.toml（代码 2.12）的作用如下：

- (1) `[workspace]`: 定义了工作空间，包含多个子项目。
- (2) `resolver = "2"`: 使用 Cargo 2 的依赖解析器，它提供了更严格的依赖解析机制，尤其是在处理不同版本的依赖时更加可靠。
- (3) `[workspace.package]`: 定义了整个工作空间的元数据，如版本号、作者信息和 Rust 的本本。
- (4) `[profile.dev]` 和 `[profile.release]`: 配置了开发和发布模式下的编译选项：
  - ① `panic = "abort"`: 设置 panic 策略为中止，这在嵌入式或操作系统开发中非常常见，因为通常没有复杂的错误恢复机制。
  - ② `opt-level`: 优化级别，设为 1 以提高编译速度。
  - ③ `codegen-units`: 减少代码生成单元数量，有助于提升优化效果。
  - ④ `overflow-checks`: 禁用溢出检查以提高性能，特别是在关键路径的代码中。

### 2.3.2 特定硬件平台的目标配置

裝

```
1  {
2      "arch": "x86",
3      "cpu": "i386",
4      "data-layout":
5          "e-m:e-p:32:32-p270:32:32-p271:32:32-p272:64:64-i128:128-f64:32:64-f80:32-n8:16:32-S128",
6      "dynamic-linking": false,
7      "executables": true,
8      "linker-flavor": "ld.lld",
9      "linker": "rust-lld",
10     "llvm-target": "i386-unknown-none-code16",
11     "max-atomic-width": 64,
12     "position-independent-executables": false,
13     "disable-redzone": true,
14     "target-c-int-width": "32",
15     "target-pointer-width": "32",
16     "target-endian": "little",
17     "panic-strategy": "abort",
18     "os": "none",
19     "vendor": "unknown",
20     "relocation-model": "static"
}
```

订

```
1  {
2      "arch": "x86",
3      "cpu": "i386",
4      "data-layout":
5          "e-m:e-p:32:32-p270:32:32-p271:32:32-p272:64:64-i128:128-f64:32:64-f80:32-n8:16:32-S128",
6      "dynamic-linking": false,
7      "executables": true,
8      "linker-flavor": "ld.lld",
9      "linker": "rust-lld",
10     "llvm-target": "i386-unknown-none",
11     "max-atomic-width": 64,
12     "position-independent-executables": false,
13     "disable-redzone": true,
14     "target-c-int-width": "32",
15     "target-pointer-width": "32",
16     "target-endian": "little",
17     "panic-strategy": "abort",
18     "os": "none",
19     "vendor": "unknown",
20     "relocation-model": "static",
21     "features": "+soft-float,-sse,-mmx"
}
```

代码 2.13 x86\_16-minmus.json 配置文件

代码 2.14 x86\_32-minmus.json 配置文件

这些 JSON 文件定义了自定义的目标平台，分别对应 16 位和 32 位的 x86 架构。

配置文件 x86\_16-minmus.json (代码 2.13) 和 x86\_32-minmus.json (代码 2.14) 的作用如下：

(1) `arch` 和 `cpu`: 定义目标架构和处理器类型。在这里都设置为 x86 和 i386，分别表示 x86 架构和 Intel 386 处理器。

(2) `data-layout`: 定义数据在内存中的布局规则，包括指针大小、对齐方式等。这对于内存管理和操作系统内核开发至关重要。

(3) `dynamic-linking` 和 `executables`: 表示是否支持动态链接和可执行文件。不使用动态链接，这在内核或嵌入式系统中很常见。

(4) `linker-flavor` 和 `linker`: 指定链接器的类型和名称，`ld.lld` 是 LLVM 的链接器 `rust-lld`，专门为生成无操作系统的裸机代码而设计。

(5) `llvm-target`: 指定 LLVM 的目标三元组，决定生成哪种目标代码。

(6) `max-atomic-width`: 最大支持的原子操作宽度，对于 i386 处理器设置为 64 位。

(7) `position-independent-executables`: 设置为 `false`，表示生成的二进制不是位置无关代码，这在操作系统开发中通常是关闭的。

(8) `disable-redzone`: 关闭 red zone (堆栈底部的一个特殊区域)，因为在操作系统开发中通常不使用这种优化。

(9) `target-c-int-width` 和 `target-pointer-width`: 定义 C 语言中的 int 和指针的宽度，分别为 32 位。

(10) `target-endian`: 定义目标平台的字节序，设置为 `little` 表示小端序。

(11) `panic-strategy`: 设定 panic 策略为 `abort`，这与 `.cargo/config.toml` 文件中的设置相一致。

(12) `relocation-model`: 设定为 `static`，表示使用静态链接模式，这在操作系统内核开发中非常普遍。

本项目使用了两个不同的 JSON 文件定义了不同的目标配置，分别用于 16 位和 32 位的 x86 代码编译。以下是需要两个 JSON 文件的原因：

## (1) 不同的目标模式 (16 位和 32 位)

① **x86\_16-minmus.json**: 这个文件配置了 16 位的目标架构 (i386)，通常用于生成低级别的启动代码 (如引导加载程序和启动代码)。在操作系统开发中，启动代码通常在实模式 (Real Mode) 下运行，这时 CPU 处于 16 位模式，因此需要专门的 16 位编译目标。

② **x86\_32-minmus.json**: 这个文件配置了 32 位的目标架构 (i386)，用于生成操作系统内核和其他高级别应用程序的代码。内核通常运行在保护模式 (Protected Mode) 下，此时 CPU 处于 32 位模式，因此需要 32 位编译目标。

## (2) 分离启动代码与内核代码

① **启动代码**: 引导加载程序 (bootloader) 和启动代码 (boot) 通常需要用 16 位模式编译，这是因为当计算机启动时，处理器处于实模式 (16 位)。引导加载程序负责加载内核并

将 CPU 从 16 位模式切换到 32 位模式，因此它必须与 CPU 的初始模式相兼容。

② **内核代码**: 内核代码通常运行在保护模式下，在这个模式下，CPU 以 32 位模式运行。

内核和高级别应用程序的编译需要使用 32 位目标配置。

### (3) 编译器和链接器的配置差异

① **指令集和数据布局**: 16 位和 32 位模式下，指令集的使用、数据布局和指针宽度等方面都存在显著差异。这些差异在操作系统开发中至关重要，因此必须使用不同的目标配置来确保生成的代码能够正确运行。

② **链接器行为**: 16 位和 32 位模式下的链接器行为也有所不同，尤其是在处理可执行文件格式、地址空间和段寄存器的管理时。因此，不同的 JSON 文件能够为这两种模式提供正确的编译和链接选项。

(4) **适应硬件平台的要求**: 在操作系统的启动过程中，需要直接访问硬件并执行一些低级别的操作，比如设置段寄存器、配置中断向量表等。这些操作通常是在 16 位模式下完成的，因此需要专门的 16 位编译配置。进入 32 位模式后，内核需要管理内存分页、任务切换和其他复杂操作，这些都依赖于 32 位的编译目标。

(5) **提高代码可维护性**: 使用两个不同的 JSON 文件可以将代码编译过程模块化。引导加载程序和内核可以独立编译和调试，彼此之间的依赖关系清晰明了，这有助于简化开发过程并提高代码的可维护性。

两个 JSON 文件分别为 16 位和 32 位的目标架构提供了不同的编译配置，确保生成的代码能够在各自的模式下正确执行。这种配置使得在操作系统开发过程中，可以同时管理启动过程和内核运行，从而确保系统的稳定性和正确性。

#### 2.3.3 Rust 工具链配置

`rust-toolchain.toml` 是一个用于配置 Rust 项目所需工具链的文件（代码 2.15）。它可以帮助为特定的 Rust 项目指定编译器版本及相关组件，从而确保项目在不同的开发环境中具有一致性。

```
1 [toolchain]
2 channel = "nightly"
3 components = ["rustfmt", "rust-src"]
```

代码 2.15 `rust-toolchain.toml` 配置文件

(1) **channel**: 指定项目使用的 Rust 工具链版本。

(2) **components = ["rustfmt", "rust-src"]**: 指定要安装的额外组件。

① **rustfmt**: Rust 的代码格式化工具，用于自动格式化 Rust 代码，使其符合社区标准和项目的风格指南。

② **rust-src**: Rust 标准库的源码。

rust-toolchain.toml 的作用：

- (1) **工具链版本固定**: 通过指定 channel, 这个文件确保了所有开发者和构建环境使用相同版本的 Rust 编译器，避免了因为编译器版本差异导致的潜在兼容性问题。
- (2) **自动安装指定组件**: 在项目所在目录中添加这个文件后，当你在该目录运行 Rust 工具（如 cargo build）时，Rust 工具链管理器 rustup 会自动确保指定的工具链和组件已安装。如果它们还没有安装，rustup 会自动下载和配置它们。
- (3) **项目一致性保障**: 无论项目是在本地开发环境、CI/CD 环境，还是在其他开发者的机器上，这个文件都确保了每次构建使用的工具链和组件完全一致，减少了因为环境差异导致的错误。

#### 2.3.4 项目自动化构建过程（Makefile）配置

Makefile 是一种用于自动化构建过程的文件，它定义了一组规则，用于编译和链接程序，以及管理项目中的各种文件和任务。在软件开发中，特别是像操作系统开发这种复杂的项目，Makefile 的作用至关重要。以下是 Makefile 的主要作用：

- (1) **自动化构建过程**: Makefile 可以自动执行编译、链接和生成可执行文件的所有步骤。开发者只需运行 make 命令，Makefile 会自动根据定义的规则来编译源代码、链接目标文件，并生成最终的可执行文件。这大大简化了编译过程，减少了手动编译的复杂性。
- (2) **管理依赖关系**: Makefile 可以追踪源代码文件之间的依赖关系。当某个源文件被修改时，Makefile 只会重新编译受影响的文件，而不会重新编译整个项目。这种增量编译机制可以节省大量的时间，特别是在处理大型项目时非常高效。
- (3) **定义可重复的构建步骤**: 在 Makefile 中，开发者可以定义各种任务（称为目标），如编译代码、生成文档、运行测试、清理编译产物等。通过这些定义，构建过程变得可重复且易于管理，确保不同开发者在不同环境中能一致地构建项目。
- (4) **多平台支持**: Makefile 可以通过条件判断和变量定义来支持多种编译环境和平台。例如，在一个跨平台项目中，可以在 Makefile 中定义特定平台的编译选项，确保在不同操作系统或硬件架构上正确构建项目。
- (5) **简化项目管理**: 对于复杂项目，Makefile 可以帮助开发者组织和管理多个模块或子项目。例如，可以定义不同的构建目标来编译各个模块，然后将它们链接在一起生成最终的可执行文件。此外，Makefile 还可以管理第三方库、生成中间文件、打包发布版本等。
- (6) **集成构建工具**: Makefile 可以与其他构建工具和脚本语言集成。例如，可以调用外部的编译器、链接器、测试框架、文档生成工具，甚至其他 Makefile，以实现更加复杂的构建流程。
- (7) **灵活性和可扩展性**: Makefile 提供了灵活的语法和功能，如变量定义、条件语句、循环等，使得开发者可以根据项目的具体需求，自定义各种构建过程。这种灵活性使得 Makefile 能适应从简单到复杂的各种项目需求。

Makefile 构建文件首先定义了一些常用的变量，如 MCOPY 和 OBJCOPY，用于指定执行相关

```
1 MCOPY := mcopy
2 OBJCOPY := objcopy
3 DISK_LAYOUT := "label: dos\n\
4           label-id: 0xffffb00b5\n\
5           device: disk.img\n\
6           unit: sectors\n\
7           sector-size: 512\n\
8           disk.img1: start=2048, size=2048, type=0\n\
9           disk.img2: start=4096, size=32768, type=0\n\
10          disk.img3: start=36864, size=94208, type=6"
11 APPS := $(notdir $(patsubst %,%,$(shell find apps -mindepth 1 -maxdepth 1 -type d)))
12 FILES := $(notdir $(wildcard files/*))
```

代码 2.16 Makefile 构建文件 (变量定义)

操作的命令。DISK\_LAYOUT 定义了磁盘布局，用于创建磁盘映像文件。APPS 变量使用 shell 命令找到 apps 目录下的所有子目录，并将它们存储为应用程序列表，FILES 变量则包含了 files 目录中的所有文件。这些变量简化了后续命令的编写和管理（代码 2.16）。

```
1 .PHONY: clean
2 clean:
3     @echo "[INFO] Cleaning up..."
4     @cargo clean
5     @rm -rf build
6     @echo "[INFO] Cleanup complete."
```

代码 2.17 Makefile 构建文件 (clean 命令)

clean 命令是一个伪目标 (PHONY)，用于清理构建生成的文件。这个命令通过 cargo clean 清除 Rust 构建生成的中间文件，并删除 build 目录，确保整个项目处于干净状态，为下一次构建做好准备（代码 2.17）。

```
1 .PHONY: build
2 build:
3     @echo "[INFO] Building boot, bootloader, and kernel..."
4     @cargo build --target=x86_16-minmus.json --package=boot
5     @cargo build --target=x86_16-minmus.json --package=bootloader
6     @cargo build --target=x86_32-minmus.json --package=kernel
7     @echo "[INFO] Building applications..."
8     $(foreach app,$(APPS),cargo build --target=x86_32-minmus.json --package=$(app);)
```

代码 2.18 Makefile 构建文件 (build 命令)

build 命令用于编译项目的各个模块。它首先编译 boot 和 bootloader，使用 16 位的目标配置文件 x86\_16-minmus.json，然后编译内核和所有应用程序，使用 32 位目标配置文件 x86\_32-minmus.json。这个命令确保所有模块都按正确的配置进行编译，为后续步骤做好准备（代码 2.18）。

objcopy 命令负责将编译生成的 ELF 格式二进制文件转换为纯二进制格式。这一步对于生成磁

```

1 .PHONY: objcopy
2 objcopy:
3     @echo "[INFO] Copying ELF binaries to binary format..."
4     @mkdir -p build/apps
5     @$(OBJCOPY) -I elf32-i386 -O binary target/x86_16-minmus/debug/boot build/boot.bin
6     @$(OBJCOPY) -I elf32-i386 -O binary target/x86_16-minmus/debug/bootloader
7         ↳ build/bootloader.bin
8     @$(OBJCOPY) -I elf32-i386 -O binary target/x86_32-minmus/debug/kernel build/kernel.bin
9     $(foreach app,$(APPS),$(OBJCOPY) -I elf32-i386 -O binary target/x86_32-minmus/debug/$(app)
10        ↳ build/apps/$(app).bin;)

```

代码 2.19 Makefile 构建文件 (objcopy 命令)

盘映像文件是必要的，因为磁盘引导程序和内核通常需要以特定的二进制格式存储和加载。该命令创建 build/apps 目录并将所有模块和应用程序转换为二进制文件，放置在适当的位置（代码 2.19）。

```

1 .PHONY: image
2 image:
3     @echo "[INFO] Creating disk image..."
4     @dd if=/dev/zero of=build/disk.img bs=67108864 count=1
5     @echo "[INFO] Applying disk layout..."
6     @echo $(DISK_LAYOUT) | /sbin/sfdisk build/disk.img
7     @echo "[INFO] Copying boot sector..."
8     @dd if=build/boot.bin of=build/disk.img conv=notrunc
9     @echo "[INFO] Preparing partition..."
10    @dd if=build/disk.img of=build/partition.img bs=512 skip=36864
11    @echo "[INFO] Formatting partition with FAT16..."
12    @mkfs.fat -F 16 build/partition.img
13    @echo "[INFO] Copying files to partition..."
14    $(foreach file,$(FILES),$(MCOPY) -i build/partition.img files/$(file) :: $(file););
15    $(foreach app,$(APPS),$(MCOPY) -i build/partition.img build/apps/$(app).bin :: $(app););
16    @echo "[INFO] Finalizing disk image..."
17    @dd if=build/partition.img of=build/disk.img bs=512 seek=36864 conv=notrunc
18    @rm -rf build/partition.img
19    @echo "[INFO] Copying bootloader and kernel to disk image..."
20    @dd if=build/bootloader.bin of=build/disk.img bs=512 seek=2048 conv=notrunc
21    @dd if=build/kernel.bin of=build/disk.img bs=512 seek=4096 conv=notrunc

```

代码 2.20 Makefile 构建文件 (image 命令)

image 命令创建磁盘映像文件并将所有必要的二进制文件写入其中。首先，它使用 dd 命令创建一个空的磁盘映像文件，并通过 sfdisk 工具应用磁盘布局。接着，将引导扇区复制到磁盘映像中，并格式化指定的分区为 FAT16 文件系统，最后，将应用程序和其他文件复制到磁盘映像的分区中。这一步生成了完整的磁盘映像，包含启动引导程序、内核和所有应用程序（代码 2.20）。

all 命令将 build、objcopy 和 image 目标组合在一起，形成一个完整的构建流程。执行 make all 命令将依次执行所有相关步骤，最终生成可运行的磁盘映像文件。这使得开发者可以通过一个简单的命令完成整个构建过程，确保所有步骤都正确执行（代码 2.21）。

run 命令在生成磁盘映像文件后，使用 QEMU 模拟器运行这个映像文件，启动开发中的操作

```
1 .PHONY: all
2 all: build objcopy image
3   @echo "[INFO] Image creation completed."
```

代码 2.21 Makefile 构建文件 (all 命令)

```
1 .PHONY: run
2 run: all
3   @echo "[INFO] Running system simulation with QEMU..."
4   @echo "[INFO] Press CTRL+C to terminate the simulation at any time."
5   @qemu-system-i386 -drive file=build/disk.img,index=0,media=disk,format=raw,if=ide
```

代码 2.22 Makefile 构建文件 (run 命令)

系统。QEMU 通过加载构建好的磁盘映像，模拟一个完整的 x86 计算机系统，使开发者可以在不依赖实际硬件的情况下测试和调试操作系统。这为操作系统的开发和调试提供了极大的便利性（代码 2.22）。

## 2.4 运行 MinmusOS 项目

通过 Makefile，可以灵活管理 MinmusOS 项目的构建、清理和运行过程。

(1) 首次运行或完整构建并启动：在终端中运行 `make` 或 `make run`。这将会：

- ① 编译项目的所有模块（boot、bootloader、kernel、应用程序）
- ② 将编译生成的 ELF 文件转换为纯二进制文件
- ③ 生成包含操作系统的磁盘映像文件
- ④ 使用 QEMU 模拟器启动 MinmusOS

(2) 只构建磁盘映像，不启动：运行 `make all`，编译项目的所有模块，生成包含操作系统的磁盘映像文件。

(3) 只进行编译，不生成磁盘映像：运行 `make build`，仅编译所有模块，但不会生成磁盘映像或启动操作系统。

(4) 清理项目：运行 `make clean` 清理项目中的所有构建文件，适合在重新构建或遇到问题时使用。

(5) 单独生成磁盘映像：如果已经编译了项目但需要重新生成磁盘映像，运行 `make image` 即可。

### 3 系统设计

在本节（章节 3）中，笔者将详细探讨 MinmusOS 项目的系统设计原则、系统架构设计、系统技术选型，以及项目结构设计。系统设计原则是确保操作系统可靠性、效率和可维护性的基础，包括封装、高内聚低耦合、面向对象设计、异常处理等多个方面。系统架构设计则涵盖了操作系统的 main 组成部分，如引导程序、内核、标准运行库和应用程序，每个部分的功能和设计细节将在后文中详细阐述。系统技术选型将解释为何选择特定的编程语言和硬件架构对于 MinmusOS 的开发至关重要，特别是选择 Rust 语言而不是传统的 C/C++，以及选择基于 IA-32 架构的硬件平台的原因。这些技术选型不仅反映了对现有技术和资源的最大利用，也展示了项目在安全性和性能优化方面的考量。项目结构设计部分将介绍 MinmusOS 的组织结构，包括代码的模块化布局和各组件之间的交互方式，以确保高效的开发和维护流程。

#### 3.1 系统设计原则

MinmusOS 的系统设计遵循了多个重要的软件工程原则，以确保其健壮性、可维护性和效率。

##### 3.1.1 封装（Encapsulation）

在 MinmusOS 中，封装是通过模块化设计实现的，每个模块负责特定的功能。例如，内核中的驱动程序模块、内存管理模块、文件系统模块等，每个模块都将其内部数据和实现细节隐藏起来，只通过定义良好的接口与外界交互。这种封装确保了模块之间的依赖性最小化，同时也便于每个模块独立更新和维护。

##### 3.1.2 高内聚低耦合（High Cohesion and Low Coupling）

MinmusOS 严格执行高内聚低耦合的设计原则。每个模块（如内存管理、进程调度等）都专注于一个具体的任务，确保模块内部各个部分紧密相关，共同完成单一功能，即高内聚。模块之间的交互通过简洁的接口进行，减少了依赖性。例如，内核与用户应用通过系统调用接口通信，系统调用接口提供了一个抽象层，使得用户程序不需要关心具体的硬件细节，即低耦合。

##### 3.1.3 面向对象（Object-Oriented）

尽管 MinmusOS 主要用 Rust 实现，它仍然采用了面向对象的设计原则。各种硬件驱动程序可能继承自一个通用的驱动程序接口，实现特定的接口函数，这支持了代码的重用和模块间的互操作性。

##### 3.1.4 异常处理（Exception Handling）

MinmusOS 通过全面的异常处理机制来增强系统的稳定性和安全性。内核设计包括对硬件和软件异常的捕获和处理，如中断、页错误和其他 CPU 异常。通过定义异常处理程序，系统能够优雅地

响应未预料的事件，保证系统的连续运行或安全地失败。

### 3.1.5 维护性 (Maintainability)

系统的设计允许容易地添加新功能或修改现有功能而不影响其他部分。通过使用版本控制系统 (Git)，MinmusOS 的维护和更新变得更加高效。代码的组织和文档化也支持快速理解和问题定位。

### 3.1.6 模块性 (Modularity)

MinmusOS 的模块性设计允许系统被划分为多个独立的部分，每个部分承担特定的职责。这种设计不仅使得各个模块可以独立开发、测试和维护，而且通过定义清晰的接口，减少了模块间的依赖关系，提高了整个系统的灵活性和可维护性。

### 3.1.7 扩展性 (Scalability)

MinmusOS 在设计时考虑到了扩展性，确保系统可以随着需求的增长而适应。系统支持动态加载和卸载功能模块，允许用户根据需要调整系统配置和资源利用，从而在用户和功能数量增加时仍能保持良好的性能和响应速度。

### 3.1.8 安全性 (Security)

安全性是 MinmusOS 设计的核心之一，系统采用多种机制确保运行的安全。利用 Rust 语言的内存安全特性，系统有效防止了内存泄漏和越界错误。此外，通过实施严格的访问控制和使用最小权限原则，MinmusOS 保护系统免受未授权访问和潜在的恶意攻击。

## 3.2 系统架构设计

MinmusOS 主要由四个模块组成：引导程序 (Bootloader)、内核 (Kernel)、标准运行库 (Standard Runtime Library) 和应用程序 (Applications)：

(1) **引导程序 (Bootloader)**：MinmusOS 的引导程序负责系统启动的初始阶段。它首先在实模式下运行，设置必要的环境后转入保护模式。

① **主引导记录**：存储在硬盘的第一个扇区，负责加载引导程序的其余部分。MBR 还包含了分区表，指明硬盘的分区信息。

② **BIOS 中断**：利用 BIOS 提供的中断调用来实现基本的输入输出系统功能，如读取磁盘数据和显示信息。

③ **磁盘读取器**：一个专门的模块，用于从磁盘读取操作系统的内核数据到内存。

④ **实模式**：CPU 的一种操作模式，允许引导程序访问系统的基础硬件资源，如内存和 I/O 端口。

⑤ **保护模式**：引导程序将系统从实模式切换到保护模式以支持更高级的功能，如虚拟内

存和多任务处理。

⑥ **全局描述符表**: 在保护模式下使用, 用于定义不同的内存段, 如代码段和数据段。

⑦ **打印机**: 在系统启动过程中用于输出调试信息到屏幕或其他输出设备。

(2) **内核 (Kernel)**: MinmusOS 的内核是系统的核心, 处理所有基本的系统任务, 如进程管理、内存管理和设备驱动。

① **中断**: 处理硬件和软件中断, 使内核能响应外部事件, 如硬件信号或软件异常。

② **驱动程序**: 内核包含各种设备驱动程序, 使其能够控制和管理硬件设备。

③ **多任务处理**: 实现进程调度和管理, 允许多个程序并发运行, 优化 CPU 使用。

④ **系统调用**: 提供一个接口供应用程序请求内核服务, 如文件操作和进程管理。

⑤ **VGA 文本模式**: 利用 VGA 硬件实现文本输出, 用于显示系统信息和用户界面。

⑥ **内存管理**: 包括虚拟内存管理和物理内存分配, 确保有效利用内存资源。

⑦ **文件系统**: 实现文件的存储、检索和管理, 支持数据的持久化。

⑧ **定时器**: 提供计时功能, 支持操作系统和应用程序的时间管理需求。

⑨ **命令行解释器**: 允许用户通过命令行界面与系统交互, 执行各种命令。

(3) **标准运行库 (Standard Runtime Library)**: 标准运行库提供一组基础库和工具, 支持内核和应用程序的运行。

① **math 库**: 支持各种数学运算, 如加减乘除和三角函数。

② **mutex 库**: 实现互斥锁, 支持多线程或多进程间的同步。

③ **print 库**: 提供基本的文本输出功能, 用于在终端或其他设备上显示信息。

④ **rand 库**: 包含多种随机数生成方法, 为系统和应用程序提供随机性支持。

⑤ **sort 库**: 实现各种排序算法, 如快速排序和归并排序, 供程序处理数据时使用。

⑥ **string 库**: 提供字符串处理功能, 如字符串拼接、分割和搜索。

(4) **应用程序 (Applications)**: MinmusOS 支持用户编写和运行应用程序。系统提供了一套应用程序编程接口 (API), 以及用于编译和链接应用程序的脚本。应用程序可以直接使用内核提供的系统调用, 以及标准运行库中的功能。

图 3.1 展示了 MinmusOS 的系统架构。

此架构设计旨在提供一个模块化和可扩展的操作系统, 每个部分都有明确的职责, 相互协作以提供高效稳定的系统运行。

### 3.3 系统技术选型

#### 3.3.1 编程语言

本系统选择 Rust 作为编程语言, 而不选择 C/C++ 编程语言。

##### A. C/C++ 语言在操作系统开发中的传统地位

C 语言自 1972 年诞生以来, 由于其接近硬件的编程能力和高效的性能表现, 迅速成为操作系

装  
订  
线

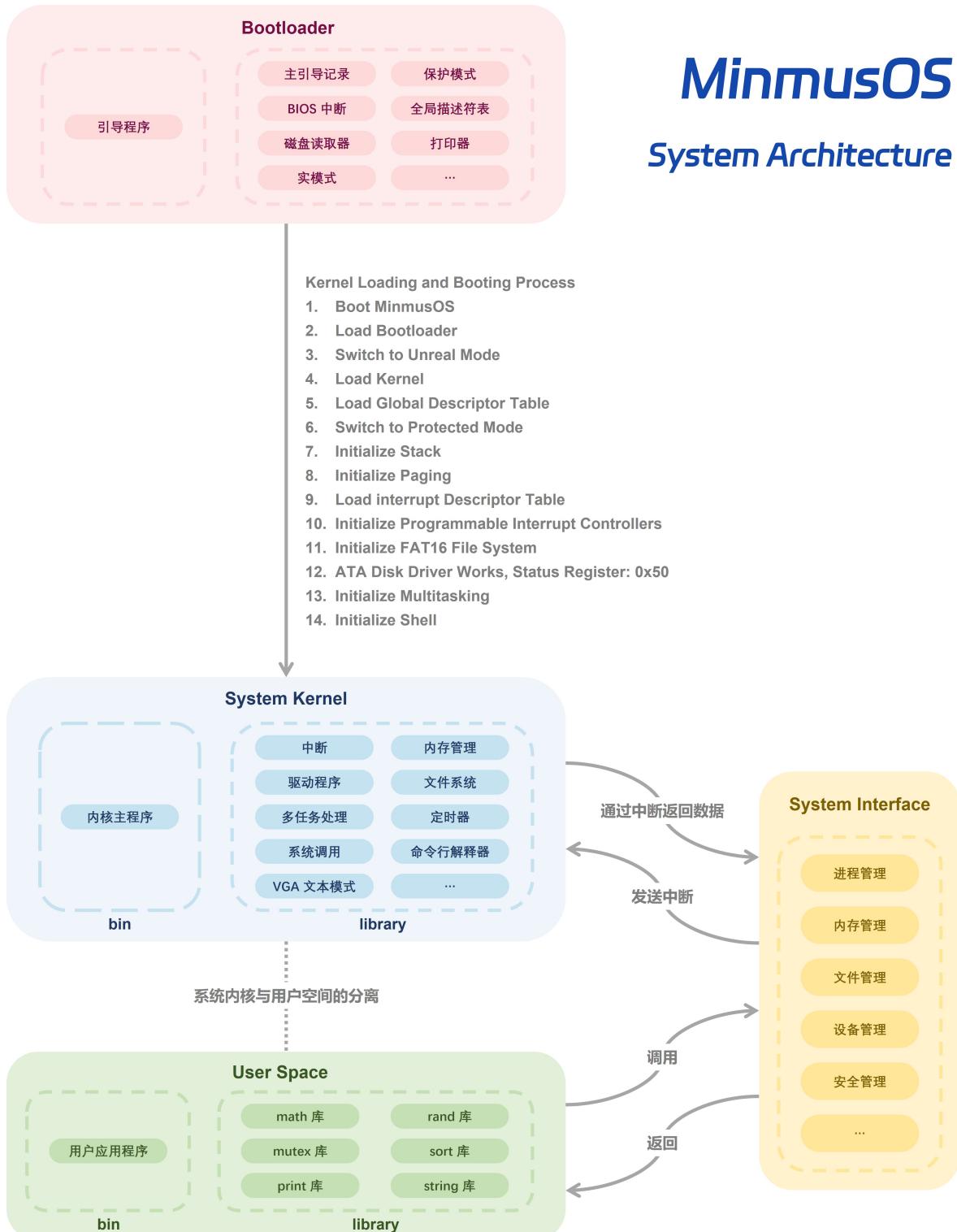


图 3.1 MinmusOS 系统架构

统开发的首选语言。最著名的例子是 Unix 操作系统，它几乎完全用 C 语言编写，此后引发了包括 Linux、Windows 和 macOS 在内的多个主流操作系统的开发倾向。这种历史悠久的使用背景为 C 语言在操作系统领域内奠定了坚实的基础，使其成为系统级编程的标准工具。

C 语言的主要技术优势在于其提供了极大的硬件操作能力和控制精度。开发者可以直接管理内存地址和硬件资源，这对于实现低级别的操作系统功能至关重要。此外，C 语言拥有强大的编译器优化能力，能够生成高效的机器代码，最大限度地提高运行时性能。

C 语言还受益于广泛的工具链支持，包括各种编译器（如 GCC 和 Clang）、调试工具（如 GDB）以及大量的库和开发资源。这些工具和资源的成熟与丰富，为操作系统开发提供了必要的技术支持和社区帮助。

C++ 作为 C 语言的后继者，最早于 1983 年由 Bjarne Stroustrup 开发，它在 C 语言的基础上增加了面向对象编程、泛型编程和其他高级特性。C++ 保留了 C 语言的硬件操作能力和性能，同时引入了类、继承和多态等面向对象的概念，使得开发者可以更好地组织和管理复杂的系统。

在操作系统开发中，C++ 提供了更高的抽象能力，使代码的可读性和可维护性得以提高。例如，Windows 内核在某些模块中就使用了 C++ 语言来实现，以便利用其面向对象的特性来组织复杂的系统组件和服务。同时，C++ 提供了强大的标准库（STL），这为开发过程中常见的数据结构和算法提供了开箱即用的实现，进一步提高了开发效率。

### B. C 语言在操作系统开发中的劣势

尽管 C 语言在操作系统开发中具有显著的优势，但它也存在一些不可忽视的劣势。其中最为人所诟病的是内存安全问题，包括内存泄漏、野指针和缓冲区溢出等，这些问题常常导致系统的不稳定甚至安全漏洞。同时，C 语言在并发和并行处理方面的支持较为原始，缺乏现代编程语言中常见的高级并发控制机制，这在多核处理器日益普及的今天显得尤为突出。

#### a. 内存管理问题

在 C 语言中，内存管理完全依赖于程序员的手动控制，包括内存的分配与释放。这种管理方式虽然提供了最大程度的控制，但同时也引入了诸多潜在问题，如内存泄漏、野指针、内存碎片以及空指针访问等。这些问题不仅增加了编程的复杂性，还可能导致程序运行时出现严重的稳定性和安全性问题。例如，忘记释放内存会造成内存泄漏，而错误地访问已释放的内存（悬垂指针）则可能导致不可预测的程序行为或系统崩溃。

#### b. 类型安全和数据完整性问题

C 语言的类型系统允许隐式类型转换和直接的指针运算，这虽然在某些情况下提供了便利，但也容易导致类型安全问题。例如，一个整数可以无警告地转换为较小的数据类型，造成数据溢出；指针类型之间也可以轻易地转换，增加了非法内存访问的风险。这种类型系统的松散性能够导致数据损坏、程序错误和安全漏洞。

#### c. 封装性和模块化的不足

C 语言作为一种面向过程的编程语言，缺乏现代面向对象语言所提供的类、继承和多态等封装

和抽象机制。虽然可以通过结构体和函数指针等手段模拟一些面向对象的功能，但这种做法通常较为笨拙且效率不高。缺乏高级的封装机制，使得在大型项目中实现代码的模块化和复用变得更加困难，从而影响了代码的可维护性和可扩展性。

#### d. 使用外部库的挑战

尽管 C 语言拥有大量的第三方库，但缺乏一个统一的、官方支持的包管理系统，使得管理和维护这些库变得复杂。程序员需要手动管理库的依赖关系、编译配置和命名冲突，这不仅增加了使用外部资源的门槛，也可能引入版本不一致等问题。

#### e. 异常处理的缺失

C 语言没有内置的异常处理机制。错误通常通过返回码来标示，或者使用全局变量来记录错误状态。这种方式要求程序员显式检查和处理每一个可能出错的函数调用，易造成错误处理代码分散且难以维护。缺乏结构化的异常处理机制，使得在面对复杂的错误情况时，程序的健壮性和可读性大大降低。

#### f. 总结

这些劣势表明，尽管 C 语言在操作系统开发中有其独特的优势和不可替代的地位，但在面对现代软件开发的复杂性和安全要求时，它的一些固有缺陷也逐渐显现出来。这促使了开发者探索如 Rust 这样的现代语言，以期提供更高的安全保障和更好的开发效率。

### C. C++ 语言在操作系统开发中的劣势

尽管 C++ 语言在操作系统开发中引入了许多强大的特性，如面向对象编程、泛型编程以及标准模板库（STL），但它在这一领域也存在一些显著的劣势，这些劣势在操作系统内核开发中尤其突出。

#### a. 运行时开销与性能问题

**异常处理开销：** C++ 的异常处理机制（如 try-catch 块）虽然为开发者提供了更好的错误处理方式，但在操作系统内核级别的开发中，这些机制会引入额外的运行时开销。操作系统内核通常要求极低的延迟和高效的性能，而 C++ 的异常处理机制可能在无意中增加函数调用的成本，降低代码执行效率。

**动态多态性开销：** C++ 中的虚函数表（vtable）用于实现动态多态性，这虽然为开发带来了灵活性，但也引入了额外的内存占用和运行时开销。对于操作系统内核，尤其是实时系统，这种开销是不可忽视的，可能会影响系统的整体性能。

#### b. 语言复杂性与错误易发性

**复杂的语言特性：** C++ 引入了比 C 语言更多的复杂特性，如模板元编程、多重继承、运算符重载等。这些特性虽然强大，但也增加了代码的复杂性，开发者容易在使用过程中引入难以调试的错误。例如，模板编程中可能产生的复杂错误信息、隐式类型转换带来的意外行为，都可能导致难以追踪的错误。

**资源管理复杂性：** C++ 提供了多种方式进行资源管理，如 RAII（Resource Acquisition Is Initial-

ization) 和智能指针，但这些特性在操作系统内核开发中引入了不必要的复杂性和潜在的错误。例如，智能指针的使用虽然在普通应用程序中能有效管理资源，但在内核环境中可能引发非预期的性能问题或资源死锁。

#### c. 内存控制的复杂性

**自动化与控制的冲突：**C++ 虽然通过智能指针和标准库等机制简化了内存管理，但在操作系统开发中，这种自动化机制可能与对硬件的精确控制需求产生冲突。操作系统内核通常需要对内存进行精细管理，而 C++ 的某些自动化特性（如垃圾回收、自动对象销毁）可能与此相悖，导致开发者对内存使用的控制权减少，甚至引发未预见的问题。

**全局构造与析构的不可控性：**C++ 中全局对象的构造与析构顺序在跨模块的操作系统内核中是不可控的，这会导致初始化顺序问题，特别是在依赖于严格启动顺序的系统中。这些问题在 C 语言中通常通过手动控制初始化顺序来避免，而 C++ 的全局对象特性可能导致不易调试的启动问题。

#### d. 工具链和调试复杂性

**调试的复杂性：**C++ 的复杂性也体现在调试过程中。由于 C++ 支持的特性如模板元编程、内联函数、运算符重载等，编译器生成的代码通常较为复杂，难以与原始代码对应。这使得在内核开发中，调试变得更加困难，开发者难以通过传统调试工具如 GDB 快速定位问题。

**工具链支持的局限性：**虽然 C++ 拥有广泛的编译器支持，但对于操作系统内核开发来说，某些 C++ 特性（如异常处理、RTTI）可能无法在不支持这些特性的编译器或特定平台上顺利使用。这限制了 C++ 在某些嵌入式或低级系统中的适用性。

#### e. 总结

综上所述，尽管 C++ 语言在一定程度上提高了编程的抽象能力和代码复用性，但在操作系统内核开发中，其复杂的特性和额外的运行时开销却可能带来不可忽视的挑战。因此，在开发需要高性能、精确控制和可靠性的操作系统时，C++ 并不总是最佳选择，开发者往往更倾向于选择控制性更强且开销更低的编程语言，如 Rust。

### D. Rust 语言在操作系统开发中的优势

Rust 是一门现代的系统编程语言，专为提供高性能与内存安全而设计。

#### a. 内存安全与防错设计

Rust 的内存安全机制是其在操作系统内核开发中最显著的优势之一。传统的 C 语言由于缺乏内置的内存管理工具，容易引发内存泄漏、悬垂指针、缓冲区溢出等问题，这些问题不仅难以调试，还可能导致系统崩溃或安全漏洞。而 Rust 通过所有权（ownership）、借用（borrowing）和生命周期（lifetimes）系统，从根本上消除了这些内存错误。Rust 编译器在编译时会严格检查内存使用情况，确保所有资源的生命周期被正确管理，从而避免了因内存错误导致的程序崩溃。

此外，Rust 提供了 `unsafe` 代码块的机制，让开发者在极少数情况下可以进行不受限的内存操作，如直接操作裸指针。尽管如此，Rust 依然会要求开发者在使用 `unsafe` 时显式声明，并且只能在明确的代码块中使用，这在确保高效内存操作的同时，也保持了总体代码的安全性。

## b. 类型系统的强化

Rust 拥有比 C 语言更丰富、更严格的类型系统。C 语言中，类型转换可以隐式发生，这可能导致数据丢失或意外的行为。而在 Rust 中，类型转换必须是显式的，编译器会拒绝未经许可的类型操作，从而避免类型相关的错误。

Rust 的类型系统还支持泛型编程，通过特征（traits）实现高效的代码复用和多态性。这种系统允许开发者在不牺牲性能的情况下，使用高级抽象来设计系统。例如，Rust 的 enum 和模式匹配机制让错误处理和状态管理变得更加直观和安全。这种类型安全的设计尤其适用于操作系统开发中的复杂数据结构和抽象管理。

## c. 外部库管理与依赖解决方案

Rust 通过其官方包管理工具 Cargo 及其生态系统 crates.io 提供了强大的外部库管理能力。Cargo 可以自动管理依赖关系，解决版本冲突，并简化库的编译和链接过程。这对于操作系统内核开发非常有利，因为内核开发往往需要依赖多个底层库，而这些库之间的依赖关系复杂且容易出错。

此外，Rust 还支持“无标准库”（no\_std）开发模式，这一特性允许开发者在不使用标准库的情况下进行系统编程。对于操作系统内核这样的低级系统软件，no\_std 模式提供了更高的灵活性，让开发者可以直接访问底层硬件，而无需依赖高级抽象层。这也使得 Rust 成为开发内核和嵌入式系统的理想选择。

## d. 封装性与代码组织

Rust 提供了先进的代码封装和模块化支持，使得复杂系统的开发和维护更加简洁和高效。Rust 允许通过 struct、enum 和 trait 来定义数据类型，并使用 impl 块为这些类型添加方法，从而将数据和行为有机地结合在一起。这种封装性提高了代码的可读性和维护性，并支持高度模块化的设计。

Rust 的模块系统允许开发者将代码组织成清晰的层次结构，并通过 mod 和 pub 关键字控制模块的可见性。Rust 还通过 use 关键字来引入外部模块或库，这有效避免了命名冲突并提高了代码的复用性。对于操作系统内核这样的大型项目，清晰的代码组织和强大的封装能力是确保项目成功的关键。

## e. 异常处理与错误回溯机制

Rust 提供了一种安全且明确的错误处理方式，区别于 C 语言的错误码返回机制。Rust 使用 Result<T, E> 类型来处理可恢复的错误，并通过编译器强制开发者显式处理这些错误，这种方法避免了错误处理的遗漏。对于不可恢复的错误，Rust 提供了 panic! 宏，该宏在遇到致命错误时会立即终止程序运行，并提供详细的错误信息和堆栈回溯。这种机制确保了操作系统内核在遇到严重问题时能够安全地停止，而不会导致系统进入未知或不安全的状态。

此外，Rust 的模式匹配特性使得错误处理更为直观和简洁，开发者可以轻松地根据不同的错误类型执行不同的处理逻辑。通过这种方式，Rust 在保证系统可靠性的同时，减少了错误处理代码的复杂性。

## f. 总结

Rust 通过其独特的语言设计和强大的编译时检查机制，提供了在操作系统内核开发中远超 C 语言的安全性和效率。Rust 的内存管理、类型系统、并发编程支持、外部库管理以及错误处理机制，使其成为现代操作系统开发的理想选择。在面对日益复杂和严苛的系统需求时，Rust 能够帮助开发者编写出更安全、更可靠的操作系统内核。

### 3.3.2 硬件架构

本项目选择 Intel IA-32（即 x86 架构）作为项目的硬件架构有以下几个主要原因：

#### A. 广泛的支持与成熟的生态系统

**长期应用与成熟性：**IA-32 是一款拥有超过三十年历史的经典架构，被广泛应用于个人计算机、服务器以及嵌入式系统中。它的成熟性确保了开发工具链、操作系统支持和硬件兼容性已非常完善，这使得在该架构上开发和调试系统变得更加便捷。

**丰富的资源：**由于其广泛的应用，IA-32 拥有大量的技术文档、教程和开源项目，开发者可以方便地获得所需的资料和支持。这一广泛的社区资源使得在 IA-32 上开发新的操作系统或系统级软件更加高效。

#### B. 硬件和工具链的广泛兼容性

**硬件可用性：**IA-32 架构在市场上拥有众多现成的处理器、主板和外围设备，从低端到高端，选择丰富，且成本相对低廉。开发者可以方便地获取所需硬件，并且在该平台上进行原型开发和测试。

**工具链支持：**针对 IA-32 的开发工具链（如 GCC、Clang、NASM 等）非常成熟，支持全面且经过充分测试。无论是编写汇编代码还是高级语言（如 C 或 C++），开发者都可以利用现有的工具进行高效开发。此外，IA-32 还支持多种虚拟化技术（如 QEMU、VirtualBox 和 VMware），方便进行操作系统的调试和测试。

#### C. 丰富的指令集与灵活性

**复杂指令集（CISC）优势：**IA-32 是一种复杂指令集计算（CISC）架构，提供了丰富的指令集和操作模式。这些指令能够直接进行复杂的内存操作、输入输出管理和中断处理，这对于操作系统开发非常有利。例如，IA-32 的保护模式、分页机制和虚拟内存管理功能，使得开发者能够实现高级的内存保护和任务管理功能。

**向后兼容性：**IA-32 保持了对旧版本指令集的兼容性，这意味着在这个平台上开发的系统可以在更广泛的硬件上运行，甚至可以兼容一些较为古老的设备。这种兼容性不仅扩大了应用范围，也减少了在开发和维护过程中可能遇到的兼容性问题。

#### D. 良好的调试与仿真支持

**丰富的调试工具：**针对 IA-32 架构的调试工具如 GDB、Bochs 和 QEMU 等，都提供了全面的调试支持。这些工具允许开发者对系统进行逐步执行、设置断点、监控寄存器和内存状态，从而高效定位和解决问题。

**仿真与虚拟化：**IA-32 支持多种仿真和虚拟化平台，这些平台可以模拟不同的硬件配置，允许

开发者在虚拟环境中快速验证操作系统的功能和性能。在虚拟环境中进行测试，不仅可以减少对物理硬件的依赖，还能够模拟各种不同的运行场景，提升系统的可靠性。

### 3.4 项目结构设计

MinmusOS 按照如下目录结构来组织代码：

```
1 MinmusOS          // MinmusOS 源代码
2 |--.cargo          // 项目配置
3 | `--config.toml  // 项目工作空间和编译选项配置文件
4 |--apps            // 应用程序
5 | `--hanoi         // 汉诺塔解决方案应用程序
6 |   |--src          // 汉诺塔解决方案应用程序源代码
7 |   | `--main.rs    // 汉诺塔解决方案应用程序入口
8 |   |--build.rs    // 汉诺塔解决方案应用程序构建脚本
9 |   |--Cargo.toml  // 汉诺塔解决方案应用程序 Rust 配置文件
10 |   `--linker.ld   // 汉诺塔解决方案应用程序链接器脚本
11 |--boot            // 第一阶段引导：实模式
12 | |--src           // 第一阶段引导源代码
13 |   | |--boot.asm  // 启动模块
14 |   | |--disk.rs   // 磁盘读取器
15 |   | `--main.rs   // 第一阶段引导主启动程序
16 |   |--build.rs    // 第一阶段引导构建脚本
17 |   |--Cargo.toml  // 第一阶段引导 Rust 配置文件
18 |   `--linker.ld   // 第一阶段引导链接器脚本
19 |--bootloader       // 第二阶段引导：保护模式与内核加载
20 | |--src           // 第二阶段引导源代码
21 |   | |--disk.rs   // 磁盘读取器
22 |   | |--gdt.rs    // 全局描述符表
23 |   | |--main.rs   // 第二阶段引导主启动程序
24 |   | `--print.rs  // 打印器
25 |   |--build.rs    // 第二阶段引导构建脚本
26 |   |--Cargo.toml  // 第二阶段引导 Rust 配置文件
27 |   `--linker.ld   // 第二阶段引导链接器脚本
28 |--files            // 示例文本文件
29 | |--cargo          // Cargo 包管理器介绍
30 | |--license         // MinmusOS 项目 MIT 许可证文件
31 | |--minmusos        // MinmusOS 欢迎文件与联系方式
32 | |--rust             // Rust 语言介绍
33 | `--thanks           // 致谢
34 |--kernel           // 内核
35 | |--src             // 内核源代码
36 |   | |--drivers      // 驱动程序
37 |   |   | |--disk.rs   // 高级技术附件磁盘驱动程序
38 |   |   | |--keyboard.rs // 键盘驱动程序
39 |   |   | |--mod.rs    // 驱动程序模块文件
40 |   |   | `--pic.rs    // 可编程中断控制器驱动程序
41 |   |   |--filesystem  // 文件系统
42 |   |   | |--fat.rs    // FAT16 文件系统
43 |   |   | `--mod.rs    // 文件系统模块文件
44 |   |   |--interrupts  // 中断
45 |   |   | |--exceptions.rs // CPU 异常
46 |   |   | |--idt.rs    // 中断描述符表
47 |   |   | |--mod.rs    // 中断模块文件
48 |   |   | `--timer.rs   // 上下文切换
```

```

49 |   |--memory           // 内存管理
50 |   |   |--allocator.rs // 内存分配器
51 |   |   |--mod.rs       // 内存管理模块文件
52 |   |   `--paging.rs    // 分页管理器
53 |   |   |--multitasking // 多任务处理
54 |   |   |   |--mod.rs    // 多任务处理模块文件
55 |   |   |   `--task.rs   // CPU 调度器与任务管理器
56 |   |   |--shell         // 命令行解释器
57 |   |   |   |--cal.rs    // cal 命令
58 |   |   |   |--color.rs   // color 命令
59 |   |   |   |--echo.rs    // echo 命令
60 |   |   |   |--kill.rs    // kill 命令
61 |   |   |   |--mod.rs    // 命令行解释器模块文件
62 |   |   |   `--shell.rs   // 命令行解释器
63 |   |   |--syscalls        // 系统调用
64 |   |   |   |--handler.rs // 系统调用处理器
65 |   |   |   |--mod.rs     // 系统调用模块文件
66 |   |   |   `--print.rs   // VGA 文本模式
67 |   |   |   |--timer       // 定时器
68 |   |   |   |   |--mod.rs  // 定时器模块文件
69 |   |   |   |   `--time.rs  // 定时器
70 |   |   |   `--main.rs    // 内核主启动程序
71 |   |   |--build.rs      // 内核构建脚本
72 |   |   |--Cargo.toml    // 内核 Rust 配置文件
73 |   |   `--linker.ld      // 内核链接器脚本
74 |--lib
75 |   |--src               // 标准运行库源代码
76 |   |   |--lib.rs          // lib 库
77 |   |   |--math.rs         // math 库
78 |   |   |--mutex.rs        // mutex 库
79 |   |   |--print.rs        // print 库
80 |   |   |--rand.rs         // rand 库
81 |   |   |--sort.rs         // sort 库
82 |   |   `--string.rs       // string 库
83 |   `--Cargo.toml        // 标准运行库 Rust 配置文件
84 |--Cargo.lock            // Rust 项目依赖配置文件
85 |--Cargo.toml            // Rust 配置文件
86 |--Makefile              // 项目自动化构建过程 (Makefile) 配置文件
87 |--rust-toolchain.toml  // Rust 工具链配置文件
88 |--x86_16-minmus.json   // x86_16 硬件平台的目标配置文件
89 `--x86_32-minmus.json   // x86_32 硬件平台的目标配置文件

```

装订线

## 4 引导程序实现

在本节（章节 4）中，笔者将详细讨论引导程序的实现，包括其定义、功能和操作系统引导过程的关键组成部分。引导加载器是启动操作系统的核心组件，它负责加载操作系统内核到内存并启动执行。整个引导过程在 Rust 语言的支持下，增强了系统的安全性和可靠性，通过精心设计的内存操作和错误管理，确保了引导过程的稳定性和效率。

### 4.1 引导过程概述

引导加载器（Bootloader）是一种专门用于启动操作系统的程序。其主要职责是将操作系统的内核加载到内存中并执行。在此之前，引导加载器需要完成一系列准备工作，包括将 CPU 从 16 位实模式切换到 32 位保护模式，并对内存段进行配置。市场上存在多种成熟的引导加载器，例如 GRUB，这类引导加载器能够启动 Linux、Windows 等复杂的操作系统。

但本项目采用了自制的引导加载器，全部使用 Rust 语言编写。本项目设计的引导加载器分为两个阶段。第一阶段的引导加载器由于必须适配磁盘的第一个扇区，因此在内存容量上极为受限。它的唯一目的是为了加载第二阶段的引导加载器。第一阶段的引导加载器功能简单，仅包含启动计算机和从磁盘读取第二阶段引导加载器所需的基本代码。一旦第一阶段的任务完成，控制权就会转交给第二阶段的引导加载器，后者负责更为复杂的任务，包括初始化额外的硬件设备、设置内存管理系统、将操作系统内核加载到内存中，并最终将控制权移交给内核。

这种分阶段的启动方式大大简化了各个阶段的职责，使每个部分都能集中处理其核心功能，避免过度复杂化。此外，采用 Rust 语言编写引导加载器增强了系统的安全性和可靠性。Rust 语言内置的内存安全特性可以有效减少常见的内存错误，为整个引导过程提供了坚实的保障。

### 4.2 第一阶段引导：实模式

#### 4.2.1 主引导记录（Master Boot Record, MBR）

主引导记录（MBR）是磁盘的第一个扇区（通常是扇区 0），也被称为启动扇区，因为它包含了启动操作系统所需的引导程序（通常是引导加载程序的第一阶段），大小为 512 字节。MBR 的结构（表 4.1）包括以下几个主要部分：

- (1) **MBR 引导代码（Boot Code, 0x000-0x1B7）**：包含用于启动计算机的基本输入输出系统（BIOS）的引导代码。在计算机启动时，BIOS 会加载并执行这段代码，这是启动序列的一部分。
- (2) **磁盘唯一标识符（Disk Signature, 0x1B8-0x1BB）**：用于存储磁盘的唯一标识符，这有助于操作系统识别和区分不同的磁盘。
- (3) **保留区域（Reserved Area, 0x1BC-0x1BD）**：通常未使用，保留供将来使用。
- (4) **分区表条目（Partition Entries, 0x1BE-0x1FD）**：存储关于硬盘上不同分区的信息，如起始地址、分区类型、分区大小等。每个分区表条目可以定义一个主分区或扩展分区。

(5) 标志性签名 (**Signature, 0x1FE-0x1FF**)：最后两个字节固定为 0x55AA，这是用来识别有效的引导扇区的标志。

表 4.1 MBR 结构

偏移量	大小(字节)	描述
0x000	440	MBR 引导代码
0x1B8	4	磁盘唯一标识符
0x1BC	2	保留区域
0x1BE	16	分区表条目
0x1CE	16	分区表条目
0x1DE	16	分区表条目
0x1EE	16	分区表条目
0x1FE	2	标志性签名 0x55AA

当从磁盘引导时，BIOS 自动将该磁盘的第一个扇区加载到内存地址 0x7C00，并跳转到该地址执行 MBR 引导程序。

在 MinmusOS 中，MBR 引导程序是引导加载器的第一阶段。这是一个极其受限的环境，因为必须只有 440 字节的程序能够将引导加载器的第二阶段加载到内存中并执行它。因此，这部分引导程序通常使用汇编语言编写，以实现尽可能的优化。然而，Rust 编译器也能生成优化的二进制文件，使其适用于此目的。因此，MinmusOS 的第一阶段引导程序主要用 Rust 编写，除了一小部分用汇编编写的程序，负责：

- (1) 禁用硬件中断
- (2) 将数据段寄存器置零
- (3) 设置堆栈
- (4) 调用 Rust 主函数

这样的设计充分利用了 Rust 的性能优势，同时保留汇编语言处理底层硬件操作的能力。

#### 4.2.2 BIOS 中断 (BIOS Interrupts)

在引导加载器的第一阶段，由于内存环境非常受限，它无法实现自己的磁盘驱动程序。因此，引导加载器使用 BIOS 中断来访问硬件，执行各种任务，如在屏幕上打印信息或从磁盘读取数据。

BIOS 中断调用是由 BIOS 提供的函数，用于简化和抽象硬件访问。这些中断调用只能在 16 位实模式下工作，因此它们不适合作为硬件驱动程序，而仅用于在启动过程中帮助引导加载器工作。当 CPU 进入 32 位保护模式时，BIOS 中断就无法工作，因此内核必须实现自定义硬件驱动程序。

MinmusOS 的引导加载器使用 BIOS 中断 0x13 从磁盘读取数据。这个中断要求设置一个磁盘地址包 (Disk Address Packet, DAP) 结构，用于指定要读取的扇区数、逻辑块地址 (LBA)，以及将数据写入内存的位置。数据结构定义如代码 4.1 所示：

```
1 #[repr(C, packed)]
2 struct DiskAddressPacket {
3     size: u8,
4     zero: u8,
5     sectors: u16,
6     offset: u16,
7     segment: u16,
8     lba: u64,
9 }
```

代码 4.1 DiskAddressPacket 数据结构定义

(1) **size**: 表示这个结构体的大小 (以字节为单位)。用来向 BIOS 提供这个结构体大小的信息，确保 BIOS 正确解释剩余的字段。

(2) **zero**: 这个字段通常设置为 0，用于填充或确保结构体的对齐。

(3) **sectors**: 指定要读取的扇区数量。因为 BIOS 中断 0x13 是以扇区为单位进行数据传输的，这个字段告诉 BIOS 一次操作需要读取多少扇区。

(4) **offset**: 数据应该被加载到的内存段内的偏移地址。这告诉 BIOS 从磁盘读取数据后，数据应该存储在哪个具体的内存位置。

(5) **segment**: 这是内存段的地址，与 offset 一起决定数据最终存放的物理内存位置。在实模式下，物理地址计算公式是  $segment \times 16 + offset$ 。

(6) **lba**: 逻辑块寻址 (Logical Block Addressing, LBA) 的起始地址。这是一个 64 位的值，用来指定从哪个扇区开始读取数据。LBA 模式允许以线性方式访问硬盘上的扇区，而不需要考虑物理磁盘的几何结构 (如柱面、磁头、扇区)。

这个结构的打包 (packed) 属性是必须的，因为它确保编译器不会在成员之间插入填充，从而满足 BIOS 对于这种结构数据严格的内存布局要求。这是在低级系统编程中常见的做法，用以确保与硬件之间的接口按预期工作。

在线发出中断之前，引导加载器需要设置一些 CPU 寄存器：

(1) **DS:SI**: 设置为 DAP (磁盘地址包) 的地址。DS 是段寄存器，SI 是偏移量寄存器，二者组合提供了数据结构的完整物理地址。

(2) **AH**: 设置为 0x42。在 INT 0x13 的多个功能中，AH=0x42 对应于扩展读盘操作，它支持使用逻辑块地址 (LBA) 而非传统的柱面-磁头-扇区 (CHS) 寻址方式。

(3) **DL**: 设置为驱动器号 (对于主驱动器是 0x80)。DL 寄存器指定了要访问的磁盘驱动器号。

然后发出 INT 0x13 中断将调用 BIOS 函数，从磁盘读取数据。如果在读取过程中发生错误，进位标志 (carry flag)<sup>3</sup> 将会被设置。MinmusOS 的引导加载器使用 JC 指令<sup>4</sup> 来检查这个标志，并通

<sup>3</sup>进位标志 (Carry Flag): 这是 CPU 状态寄存器中的一位，用来指示上一个算术或逻辑操作是否产生了进位或借位。在使用 INT 0x13 时，如果操作成功，进位标志将被清除 (设置为 0)；如果操作失败，进位标志将被设置 (设置为 1)。

<sup>4</sup>JC (Jump if Carry) 指令：这是一个条件跳转指令，只有在进位标志为 1 时才执行跳转。在引导加载器中，如果 INT 0x13 调用失败，进位标志会被设置，JC 指令将跳转到错误处理代码，通常会显示错误信息或停止进一步执行。

知用户错误发生。

另一个 BIOS 中断是 INT 0x10，INT 0x10 是一个 BIOS 提供的视频服务中断。这个中断提供了多种与显示相关功能，如设置显示模式、更改字符颜色、移动光标以及打印字符到屏幕等，它在引导加载器中被用来打印字符串到屏幕上。然而，这个中断只在引导加载器中使用，因为内核实现了一个更复杂的打印功能，该功能直接写入到视频内存。

#### 4.2.3 启动模块（Boot Module）

这个模块包括一个 Assembly 文件和其 Rust 接口的集成，负责禁用硬件中断，将数据段寄存器置零，设置堆栈，并且跳转到 Rust 主程序。

```

1 .section .boot, "awx"
2 .global _start
3 .code16
4
5 _start:
6     cli
7
8     xor ax, ax
9     mov ds, ax
10    mov es, ax
11    mov ss, ax
12    mov fs, ax
13    mov gs, ax
14
15    cld
16    mov sp, 0x7c00
17
18    call main
19
20 spin:
21     hlt
22     jmp spin

```

代码 4.2 boot/src/boot.asm

这段汇编代码（代码 4.2）是用于启动操作系统的 Master Boot Record（MBR）的引导程序的一部分，位于硬盘的最开始的扇区。详细解释如下：

##### (1) 节和全局设置

- ① `.section .boot, "awx"`: 定义一个名为 .boot 的段，属性为 “awx”，表示该段是可分配的、可写的以及可执行的。
- ② `.global _start`: 定义 \_start 标签为全局，使得链接器可以找到它作为程序的入口点。
- ③ `.code16`: 指定代码为 16 位模式，适用于 BIOS 在实模式下工作。

##### (2) 禁用外部中断

- ① `cli`: Clear Interrupt Flag，清除中断标志，禁用硬件中断，确保在初始化过程中不会被

外部事件打断。

### (3) 设置数据段寄存器为零

- ① `xor ax, ax`: 将 AX 寄存器清零。AX 寄存器是 x86 架构处理器中的一个通用寄存器，它是一个 16 位的寄存器，可以用于存储数据、执行算术和逻辑操作等。
- ② `mov ds, ax` 等: 将所有数据段寄存器 (DS、ES、SS、FS、GS) 设置为 0，初始化段寄存器，为后续程序的运行提供干净的段环境。

### (4) 设置栈指针

- ① `cld`: Clear Direction Flag，清除方向标志，确保字符串操作在内存中从低地址向高地址移动。
- ② `mov sp, 0x7c00`: 设置栈指针 SP 到 0x7c00，这是 BIOS 加载 MBR 程序的起始地址。由于栈是向下增长的，所以初始化栈指针到程序加载的起始地址是为了确保栈空间不会与程序空间冲突。

### (5) 调用 Rust 主函数

- ① `call main`: 调用标签为 main 的函数，这是用 Rust 编写的主函数，用于执行更复杂的任务。

### (6) 防止程序执行溢出

- ① `hlt`: Halt 指令用于停止 CPU 的执行直到下一个外部中断被触发。
- ② `jmp spin`: 无限循环，确保如果 main 函数返回，CPU 不会执行任何未定义的操作或跑到程序代码以外的地方去。

在完成了启动模块的基础设置和功能调用之后，系统的硬件环境和内存状态被适当配置和初始化，为主启动程序的加载和执行提供了必要的条件。此时，CPU 仍处于 16 位实模式，限制了对现代硬件特性的全面控制和访问。接下来，控制权将被传递到主启动程序，这一程序将负责进一步的系统启动过程，包括引导更复杂的操作系统核心组件。

#### 4.2.4 磁盘读取器模块 (Disk Reader Module)

这个模块负责从磁盘读取数据。它定义了 DiskReader 结构和相关方法，使用 BIOS 中断 0x13 来进行实际的磁盘操作。该模块使用了线性块地址 (LBA) 方式而不是传统的柱面-磁头-扇区 (CHS) 方式，这是现代磁盘访问的常用方法。这里使用的是一种称为磁盘地址包 (Disk Address Packet, DAP) 的数据结构 (见代码 4.1)，以支持这种访问方式。

DiskReader 结构体 (代码 4.3) 用于封装磁盘读取操作的状态，包括：

- (1) `lba`: 起始线性块地址
- (2) `target`: 数据应该加载到的内存地址

A. 方法 `new(lba: u64, target: u16) -> Self`

这个 `new` 方法是 DiskReader 结构体的构造函数，用于创建 DiskReader 实例。它接受两个参数：

```

1 pub struct DiskReader {
2     lba: u64,
3     target: u16,
4 }
```

代码 4.3 DiskReader 数据结构定义

lba 和 target。lba 参数是一个 u64 类型的值，代表线性块地址，用来指定从哪个扇区开始读取数据。这允许构造函数支持大容量存储设备上的磁盘操作。target 参数是一个 u16 类型的值，表示数据加载到内存中的目标偏移地址。

通过这个构造函数，用户可以方便地创建一个配置好的 DiskReader 实例，随后可用于执行磁盘读取操作。这个设计简洁而有效，确保了 DiskReader 在创建时就被正确地配置。

#### B. 方法 read\_sector()

read\_sector 方法（代码 4.4）的主要功能是读取从特定线性块地址（LBA）开始的一个扇区的数据，并将数据存储到指定的内存偏移地址。该方法使用 DiskAddressPacket 结构体来配置读取操作的细节，并通过 INT 0x13 BIOS 中断进行磁盘访问。

```

1 pub fn read_sector(&self) {
2     let dap = DiskAddressPacket {
3         size: size_of::<DiskAddressPacket>() as u8,
4         zero: 0,
5         sectors: 1,
6         offset: self.target,
7         segment: 0x0000,
8         lba: self.lba,
9     };
10
11     let dap_address = &dap as *const DiskAddressPacket;
12
13     unsafe {
14         core::arch::asm!(
15             "mov {1:x}, si",
16             "mov si, {0:x}",
17             "int 0x13",
18             "jc fail",
19             "mov si, {1:x}",
20             "in(reg) dap_address as u16",
21             "out(reg) _",
22             "in(\"ax\") 0x4200u16",
23             "in(\"dx\") 0x0080u16",
24         );
25     }
26 }
```

代码 4.4 read\_sector() 方法

在这个方法中，首先创建了一个 DiskAddressPacket 的实例 dap，具体字段配置如下：

(1) size：结构体的大小，使用 size\_of::<DiskAddressPacket>() 动态获取，保证与实际定

义匹配。

- (2) `zero`: 固定为 0, 作为填充字段。
- (3) `sectors`: 设置为 1, 表示此次操作只读取一个扇区。
- (4) `offset` 和 `segment`: 指定数据加载到内存中的位置。`offset` 为 `self.target`, 是调用时指定的内存地址偏移; `segment` 设置为 0x0000, 通常是在实模式下使用的段基址。
- (5) `lba`: 从该线性块地址读取数据, 对应于 `DiskReader` 实例中存储的 `lba` 字段。

在 `DiskAddressPacket` 配置完成后, 代码获取这个结构体的地址 `dap_address`, 然后通过内联汇编进行以下操作:

#### (1) 保存和设置寄存器

- ① `mov 1:x, si`: 保存原始 `si` 寄存器的值, 以便恢复。
- ② `mov si, 0:x`: 将 `DiskAddressPacket` 的地址放入 `si` 寄存器, 因为 INT 0x13 需要通过 `si` 传递参数。

#### (2) 执行 BIOS 中断

- ① `int 0x13`: 执行磁盘读操作。`ax` 寄存器设置为 0x4200, 表示执行读操作; `dx` 设置为 0x0080, 表示第一个硬盘。

#### (3) 错误检查

- ① `jc fail`: 如果操作失败 (进位标志被设置), 则跳转到错误处理标签 `fail` (需要在代码中定义该标签的行为)。

#### (4) 恢复寄存器

- ① `mov si, 1:x`: 恢复 `si` 寄存器的原始值。

这种方式直接使用 BIOS 提供的功能来访问硬件, 允许在没有操作系统支持的环境下 (如引导加载器) 进行低层次的磁盘操作。通过这个方法, `DiskReader` 可以读取磁盘上特定位置的数据, 对于操作系统的启动和运行至关重要。

### C. 方法 `read_sectors(sectors: u16)`

该 `read_sectors` 方法 (代码 4.5) 是 `DiskReader` 结构体的成员方法, 其功能是读取多个连续的扇区数据。方法接受一个参数 `sectors`, 表示需要读取的扇区数量。该方法通过循环调用 `read_sector` 方法来逐一读取指定数量的扇区, 并适当更新目标内存地址和 LBA 地址。

以下是对该方法的解释:

- (1) **初始化计数器**: 创建一个变量 `sectors_left` 用于追踪还剩多少扇区需要读取。
- (2) **循环读取扇区**: 使用一个循环来连续读取扇区, 直到 `sectors_left` 减到 0, 这意味着所有指定的扇区都已经读取完成。
- (3) **读取单个扇区**: 调用 `read_sector` 方法读取一个扇区的数据。
- (4) **更新内存地址和 LBA 地址**: 每读取一个扇区后, 将目标内存地址向前移动一个扇区的大小 (512 字节), 这样下一个扇区的数据就不会覆盖前一个扇区的数据; 将 LBA 地址递增 1, 以便下一

```

1 pub fn read_sectors(&mut self, sectors: u16) {
2     let mut sectors_left = sectors;
3     while sectors_left > 0 {
4         self.read_sector();
5         self.target += SECTOR_SIZE;
6         self.lba += 1;
7         sectors_left -= 1;
8     }
9 }
```

代码 4.5 `read_sectors(sectors: u16)` 方法

次读取操作定位到下一个扇区。

(5) **递减剩余扇区计数**: 每次循环结束后, 将剩余扇区数减 1。

这个方法通过更新内存偏移地址和 LBA 地址, 使得可以连续地读取多个扇区到指定的内存区域中。

#### 4.2.5 主启动程序 (Main Boot Program)

这个模块 (代码 4.6) 是系统的主入口, 使用 Rust 编写。它负责加载并执行 bootloader, 并处理初始化中的错误。

##### A. 全局和外部声明

`#![no_std]`: 指示编译器不使用标准库, 这是操作系统和裸机程序的常见需求, 因为标准库依赖于操作系统的功能。

`#![no_main]`: 表示没有常规的 `main` 函数入口, 这是裸机或操作系统开发中常见的设置。

`BOOTLOADER_LBA`: 定义引导加载器所在的逻辑块地址 (LBA)。

`BOOTLOADER_SIZE`: 定义引导加载器大小, 单位为扇区。

`global_asm!`: 插入全局汇编代码, 这里包括从 “`boot.asm`” 文件中读取的汇编代码, 通常用于设置初步的启动环境。

##### B. 主入口函数 `main`

`main` 函数是 MinmusOS 引导程序的核心入口点, 它负责初始化系统并引导操作系统。在裸机或操作系统开发中, 这个函数代替了常规应用程序中的标准 `main` 函数。`#[no_mangle]` 属性用于防止编译器修改函数名的装饰或混淆, 即确保函数名在编译后保持不变。以下是对这个函数中每一步操作的详细解释:

(1) **设置视频模式**: `mov ah, 0x00` 和 `mov al, 0x03` 这两条指令准备寄存器 `ah` 和 `al` 以设置 BIOS 视频中断 (int 0x10) 的参数。`ah = 0x00` 代表设置视频模式的功能, `al = 0x03` 指定具体的视频模式, 这里是标准的文本模式 (80 × 25 字符)。

(2) **打印启动信息**: 调用 `print` 函数来在屏幕上显示启动信息, 帮助用户了解当前引导进程的状态。

装订线

```
1  #![no_std]
2  #![no_main]
3
4  ...
5
6  const BOOTLOADER_LBA: u64 = 2048;
7  const BOOTLOADER_SIZE: u16 = 64;
8
9  global_asm!(include_str!("boot.asm"));
10
11 extern "C" { static _bootloader_start: u16; }
12
13 #[no_mangle]
14 pub extern "C" fn main() -> ! {
15     unsafe { asm!("mov ah, 0x00", "mov al, 0x03", "int 0x10"); }
16     print("[INFO] Booting MinmusOS...\r\n\0");
17     print("[INFO] Loading Bootloader...\r\n\0");
18     let bootloader_start: *const u16 = unsafe { &_bootloader_start };
19     let target = bootloader_start as u16;
20     let mut disk = DiskReader::new(BOOTLOADER_LBA, target);
21     disk.read_sectors(BOOTLOADER_SIZE);
22     unsafe { asm!("jmp {0:x}", in(reg) bootloader_start as u16); }
23     loop {}
24 }
25
26 fn print(message: &str) {
27     unsafe {
28         asm!(
29             "mov si, {0:x}", // 将 message 的地址加载到 SI 寄存器
30             "2:",           // 标签 2, 这是一个循环的开始点
31             "lodsb",        // 从 SI 指向的地址加载一个字节到 AL 寄存器, 并将 SI 自增
32             "or al, al",   // 将 AL 寄存器的内容与其自身进行 OR 操作, 主要用来设置零标志 (ZF)
33             "jz 3f",        // 如果结果为零 (即 AL 为零, 表明字符串结束), 则跳转到标签 3
34             "mov ah, 0x0e", // 将 0x0E 加载到 AH 寄存器, 设置 BIOS 的 teletype 输出
35             "mov bh, 0",    // 将显示页面设置为 0, 通常用于多页面管理
36             "int 0x10",     // 调用 BIOS 视频中断, 使用 AH = 0x0E 和 AL 的值在屏幕上打印字符
37             "jmp 2b",       // 无条件跳回到标签 2, 继续循环读取下一个字符
38             "3:",           // 标签 3, 循环结束的地方
39             in(reg) message.as_ptr(),
40         );
41     }
42 }
43
44 #[no_mangle]
45 pub extern "C" fn fail() -> ! {
46     print("[ERROR] Failed to Load Bootloader!\r\n\0");
47     loop {}
48 }
49
50 #[panic_handler]
51 fn panic(_info: &PanicInfo) -> ! { loop {} }
```

代码 4.6 boot/src/main.rs

(3) **获取引导加载器起始地址**: 通过外部链接的静态变量 `_bootloader_start` (由链接器脚本定义) 获取引导加载器的起始内存地址。

(4) **初始化磁盘读取器**: 将引导加载器的起始地址转换为 `u16` 类型, 以便作为磁盘读取的目标内存地址。然后创建一个 `DiskReader` 实例, 用于从指定的 LBA 地址开始读取数据到内存。

(5) **读取引导加载器到内存**: 调用 `DiskReader` 的 `read_sectors` 方法从磁盘读取 `BOOT-LOADER_SIZE` 个扇区的数据到内存中 `target` 指定的位置。

(6) **跳转到引导加载器执行**: 使用内联汇编指令 `jmp` 直接跳转到引导加载器的起始地址, 开始执行引导加载器的代码。

(7) **无限循环**: 在正常情况下, `jmp` 指令后的代码不应该被执行, 因为控制权已经转移给引导加载器。无限循环确保了在任何意外情况下, CPU 不会执行未定义的内存区域。

### C. 打印函数 `print`

在 MinmusOS 中的 `print` 函数负责将文本消息输出到屏幕, 它通过 BIOS 中断 `int 0x10` 实现。这个函数使用内联汇编来直接与硬件交互。

### D. 错误处理函数 `fail`

错误处理函数 `fail` 在 MinmusOS 的启动过程中加载引导加载器失败时被调用, 用于显示错误信息并将系统置于无限循环状态, 从而防止执行进一步的可能错误操作。

### E. 紧急停止处理函数 `panic`

紧急停止处理 (`panic handler`) 是一种特殊的函数, 用于处理运行时遇到的不可恢复的错误。它在程序遇到严重错误时被调用, 如内存访问错误、预期之外的执行分支等。该函数的主要目的是防止程序继续执行可能危险或未定义的操作, 它通过进入一个无限循环, 使系统停留在已知的状态, 便于调试和维护。在实际部署中, 这种机制对于保持系统的稳定性和安全性至关重要, 尤其是在裸机环境或操作系统的底层实现中。

## 4.2.6 链接器脚本 (Linker Script)

链接器脚本 (代码 4.7) 用于定义和控制操作系统引导程序的内存布局。脚本总体可以分为以下几个关键部分来概括其作用:

(1) **入口点配置**: 脚本指定了程序的入口点 `_start`, 确保程序加载后从正确的位置开始执行。

(2) **栈配置**: 脚本设置了栈的起始和结束位置, 确保程序在执行时具有正确的栈空间进行操作。

(3) **段定义**: 脚本详细定义了多个段, 包括引导代码段 `.boot`, 程序的主要执行代码段 `.text`, 只读数据段 `.rodata`, 以及包含初始化的全局变量和静态变量的数据段 `.data`。这些段的配置关键支持了程序数据和代码的正确加载和执行。

(4) **磁盘和分区表配置**: 脚本设置了磁盘标识符和两个分区表, 详细定义了启动分区和主分区的属性, 这对于系统启动和磁盘管理至关重要。

(5) **魔数和引导加载器位置**: 脚本的最后部分定义了引导扇区的魔数 `0xAA55`, 保证引导扇区

的有效性，同时标记了引导加载器的起始位置，确保引导过程可以正确地定位和加载引导加载器。

整个链接器脚本确保了引导程序的内存布局和磁盘布局的准确性，对于系统的引导和加载过程至关重要。

在 Rust 项目中，`build.rs` 文件充当构建脚本的角色，主要用于在项目编译前执行自定义的构建任务。此文件对于那些需要细粒度控制编译过程的项目尤为重要，如操作系统开发或需要特定编译器设置的应用程序。

脚本使用 `env!("CARGO_MANIFEST_DIR")` 来获取环境变量，该变量指向包含 `Cargo.toml` 文件的目录，即项目的根目录。通过打印特定格式的字符串动态地向 Cargo 传递编译参数。代码 4.8 配置了编译器在构建二进制文件时使用自定义的链接器脚本。这通过 `println!` 输出 `cargo:rustc-link-arg-bins==script={}` 实现，其中包含了链接器脚本的路径。通过计算出链接器脚本 `linker.ld` 的完整路径，并将其传递给编译器作为参数，`build.rs` 确保了链接过程按照预定义的内存布局和符号解析规则进行。这对于需要精确控制输出二进制格式的低级应用程序（如操作系统内核）是必需的。

## 4.3 第二阶段引导：保护模式与内核加载

### 4.3.1 保护模式（Protected Mode）

在保护模式引入之前，实模式（real mode）是 x86 CPU 上唯一的工作模式。因此，像 Intel 8088 这样的老旧 CPU 缺乏内存保护功能。在实模式下，内存使用 16 位地址进行访问，这意味着最大可寻址内存为 1MB。当时，这个限制并不成问题，因为所有系统的内存都不超过 640KB。

然而，如今这已成为一个巨大的限制，导致了 32 位 CPU 的引入以及一种称为保护模式（protected mode）的新操作模式。保护模式允许使用 32 位地址访问内存，将可寻址内存的上限提高到了 4GB。该模式还引入了许多实模式完全缺乏的新安全功能，例如新的分段类型、特权级别和分页机制。

出于兼容性原因，所有 x86 CPU 在启动时都从 16 位实模式开始，因此引导程序的早期阶段使用的是实模式。切换到保护模式包括三个步骤：

- (1) 设置全局描述符表
- (2) 设置 CR0 寄存器中的保护使能位（Protection Enable bit）<sup>5</sup>
- (3) 长跳转到保护模式代码段<sup>6</sup>

### 4.3.2 全局描述符表（Global Descriptor Table）

全局描述符表（Global Descriptor Table，简称 GDT）是 x86 架构中特有的一种数据结构。在保护模式下，GDT 用于定义内存段（Segments），并允许 CPU 根据这些段描述符对内存访问进行硬件级别的保护。

<sup>5</sup>保护使能位是 x86 处理器中的一个特定位，位于控制寄存器 CR0 中。具体来说，它是 CR0 寄存器的第 0 位，通常称为 PE 位（Protection Enable）。当该位被设置为 1 时，处理器从实模式进入保护模式，开始使用 32 位地址空间，并启用相关的内存保护机制和特权级别。

<sup>6</sup>长跳转是在 x86 架构中从实模式切换到保护模式时必须执行的一种跳转操作，用于重新加载代码段寄存器（CS）以确保代码的执行正确。在保护模式下，CPU 使用段选择子和段描述符来确定代码段的基址地址和限制等属性。简单来说，长跳转不仅改变程序的执行地址，还会更新代码段寄存器（CS）中的段选择子，以适应新的内存模型。

```

1  /* 配置程序入口点 */
2  ENTRY(_start)
3
4  /* 配置段的顺序和位置 */
5  SECTIONS {
6      /* 配置栈的开始和结束位置 */
7      . = 0X500;
8      _stack_start = .;
9      . = 0X7C00;
10     _stack_end = .;
11
12     /* 定义引导代码段, 包含所有 .boot 和 .boot.* 段的内容 */
13     .boot : { ... }
14
15     /* 定义文本段, 包含程序的主要执行代码 */
16     .text : { ... }
17
18     /* 定义只读数据段, 包含字符串常量等不可修改的数据 */
19     .rodata : { ... }
20
21     /* 定义数据段, 包含初始化的全局变量和静态变量 */
22     .data : { ... }
23
24     /* 调整内存布局中当前段的起始地址 */
25     . = 0X7C00 + 0X1B8;
26
27     /* 定义磁盘唯一标识符 */
28     .diskid : { ... }
29
30     /* 保留区域, 设置为 0 */
31     .reserved : { ... }
32
33     /* 第一个分区表: 用于存储引导加载器 */
34     .first_table : { ... }
35
36     /* 第二个分区表: 用于主分区 */
37     .second_table : { ... }
38
39     /* 定义设置魔数位置, 确保位于扇区的最后两个字节 */
40     . = 0X7C00 + 0X1FE;
41
42     /* 定义引导扇区有效的结束标记, 必须是 0XAA55 */
43     .magic_number : { ... }
44
45     /* 定义引导加载器的起始位置标记 */
46     _bootloader_start = .;
47 }
```

代码 4.7 boot/linker.ld

```

1 fn main() {
2     let local_path = std::path::Path::new(env!("CARGO_MANIFEST_DIR"));
3     println!("cargo:rustc-link-arg-bins==script={}", local_path.join("linker.ld").display());
4 }
```

代码 4.8 boot/build.rs

GDT 的主要作用是为 CPU 提供一个内存管理机制，通过它可以划分和管理内存段，从而实现更复杂的内存保护和访问控制。每个内存段的属性（如基地址、段的大小、权限等）都由 GDT 中的条目（Segment Descriptor）来描述，段描述符结构如图 4.9 所示。

GDT 中的每个条目称为段描述符（Segment Descriptor），它们的长度都是 8 字节。段描述符中包含的信息决定了该段的起始地址、段长、访问权限和其他特性。CPU 通过这些信息来管理和控制程序对内存的访问。在 GDT 中，第一个条目通常是空的（全为 0），称为空描述符（Null Descriptor）。这是为了防止 CPU 意外使用无效的段选择子，并作为一种安全措施。GDT 中可以定义多种类型的段，包括代码段、数据段和系统段，每种段都有不同的访问权限和使用规则。

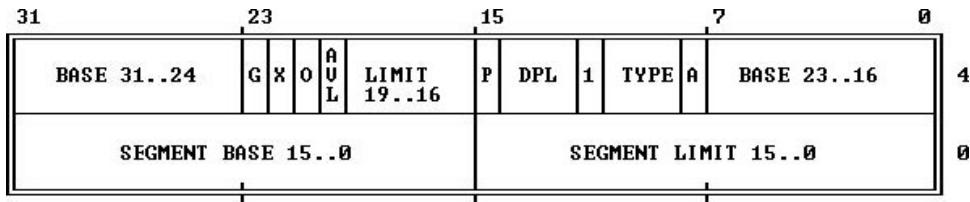


图 4.9 段描述符结构

通过设置 GDT，系统可以在保护模式下实现内存的分段管理和保护，使得不同的程序和任务可以在各自的内存段中独立运行，避免相互干扰，并且增强系统的安全性和稳定性。

### 4.3.3 磁盘读取器模块 (Disk Reader Module)

磁盘读取器模块 (Disk Reader Module) 定义了磁盘操作，使用 BIOS 中断 (INT 0x13) 来实现从磁盘读取数据。它包括一个 Disk 结构体和 DiskAddressPacket 结构体，用于管理磁盘 I/O 操作和地址映射。该模块允许从指定的逻辑块地址 (LBA) 读取多个扇区数据到内存中指定位置。DAP 的数据结构定义如代码 4.1 所示，在此不再赘述。

该模块定义了 Disk 结构体，其中包含用于指定从哪个磁盘块 (LBA) 开始读取以及数据应加载到内存的哪个位置 (buffer) 的字段。通过调用 init 方法，可以对这些字段进行配置，为读取操作做好准备。

read\_sector 方法通过配置 DiskAddressPacket (DAP)，然后使用 BIOS 中断 INT 0x13 来从磁盘读取一个扇区的数据。这种方法允许直接与硬件交互，无需操作系统层的支持。该方法的实现与代码 4.4 相同，在此不再赘述。

read\_sectors 方法 (代码 4.10) 允许从磁盘连续读取多个扇区。它通过在循环中反复调用 read\_sector 方法，并逐步将数据加载到目标内存位置，从而实现连续读取。以下是这个函数的实现逻辑：

(1) **初始化**: 方法初始化两个关键变量: sectors\_left 用于跟踪还需读取的扇区数，而 current\_target 标记了数据应当被复制到的内存起始地址。循环将持续进行，直到所有扇区都被读取完毕。

(2) **扇区读取**: 每次循环迭代，首先通过 read\_sector 方法从磁盘读取单个扇区到缓冲区。接下来，通过内部循环，将缓冲区中的数据逐字节复制到目标内存地址。这一过程涉及到寄存器操作和

内联汇编，确保数据正确地从源头移动到目标位置。

- ① **读取字节的汇编代码：**这条汇编指令负责从由 byte\_address 指向的内存位置读取单个字节的数据。byte\_address 是一个寄存器，它包含了要读取数据的具体内存地址。mov 0, [1:e] 的指令格式表示将位于内存地址 [1:e] (由 byte\_address 指定) 的数据移动到 byte 寄存器中。这里的 byte 作为输出寄存器，用于存储读取的数据，以便后续操作可以使用这个数据。
- ② **写入字节的汇编代码：**这条汇编指令用于将先前读取并存储在 byte 寄存器中的字节数据写入到由 current\_target 寄存器指定的内存地址中。mov [0:e], 1 的指令格式将 byte 寄存器中的数据移动到 current\_target 指定的内存位置。在这个过程中，current\_target 寄存器作为输入，提供目标地址，而 byte 寄存器则提供要写入的数据。这种方式确保了数据能够被正确地从源位置传输到目标位置，实现数据在内存中的精确控制和布局。

(3) **状态更新：**在每次完成单个扇区的读取和复制后，更新磁盘读取位置 self.lba 和剩余扇区计数 sectors\_left。

```

1 pub fn read_sectors(&mut self, sectors: u16, target: u32) {
2     let mut sectors_left = sectors;
3     let mut current_target = target;
4     while sectors_left > 0 {
5         self.read_sector();
6         let mut byte_address = self.buffer;
7         for _byte_index in 0..SECTOR_SIZE {
8             unsafe {
9                 let mut byte: u8;
10                core::arch::asm!("mov {0}, [{1:e}]", out(reg_byte) byte, in(reg)
11                                ← byte_address);
12                core::arch::asm!("mov [{0:e}], {1}", in(reg) current_target, in(reg_byte)
13                                ← byte);
14            }
15            current_target += 1;
16            byte_address += 1;
17        }
18        self.lba += 1;
19        sectors_left -= 1;
20    }
}

```

代码 4.10 read\_sectors(sectors: u16, target: u32) 方法

#### 4.3.4 全局描述符表模块 (GDT Module)

全局描述符表模块 (GDT Module) 包含设置全局描述符表 (Global Descriptor Table, GDT) 的功能，这是进入保护模式所必需的。描述符表告诉 CPU 如何处理内存的不同段。全局描述符表 (GDT) 是一个数据结构，它告诉 CPU 各个内存段的基地址、大小、权限和其他属性。在 MinmusOS 中使用的是平坦内存模型，这意味着整个 4GB 的内存空间被视为一个连续的内存区块，而不是被分割

成多个独立的段。

尽管 MinmusOS 使用平坦内存模型，其 GDT 实际上包含三个条目：

(1) **第一个条目（空描述符）**：在任何 GDT 中，第一个条目总是被设置为零。这是一个空描述符，不用于任何内存段的访问，但其存在是为了符合 Intel 处理器的要求，该处理器规定 GDT 的第一个条目必须是无效的。

(2) **第二个条目（代码段描述符）**：这个条目定义了一个代码段，其基址设为 0x0，限制设为 0xFFFF，并且粒度位被设置为 1。粒度位为 1 表示段限制以 4KB 为单位计量，而 0xFFFF 的限制在这种设置下允许访问全部 4GB 的内存空间（因为  $0xFFFF * 4KB = 4GB$ ）。代码段的设置允许 CPU 执行存储在其中的代码。

(3) **第三个条目（数据段描述符）**：与代码段类似，数据段也设定了基址为 0x0 和相同的限制，意味着数据段也覆盖整个 4GB 的地址空间。数据段用于定义普通内存操作的访问权限，如读写数据等。

使用平坦内存模型的主要优点是简化了内存管理。在此模型下，所有程序都使用相同的内存地址映射，这消除了段间切换的开销并简化了程序编写。只要有适当的权限所有的内存都可以被任何执行代码访问，这在多任务操作系统中特别有用，因为它允许简单有效地进行任务切换。

代码 4.11 通过定义和设置 GDT 条目来配置操作系统的内存段属性，涵盖了内存段的存在状态、类型、权限、大小以及其他关键属性，确保操作系统在保护模式下能够有效地管理内存访问。

在 MinmusOS 中，全局描述符表(GDT)的管理涉及三个核心的数据结构：`GdtEntry`(代码 4.12)、`GdtDescriptor`(代码 4.13) 和 `GlobalDescriptorTable`(代码 4.14)。这些结构体是操作系统与硬件之间沟通如何处理内存段的桥梁。下面详细介绍这些数据结构的作用与实现。

#### A. `GdtEntry` 数据结构

`GdtEntry` 结构体表示全局描述符表中的一个条目。在 GDT 中，每个条目（或称为段描述符）定义了一个内存段的基址、限制、访问权限等属性。此结构体的字段 `entry` 是一个 64 位的整数，包含了段的所有描述信息，按照 x86 架构的段描述符格式编码。使用 `#[repr(C, packed)]` 确保结构体没有任何内存对齐填充，与硬件要求的内存布局严格一致。`#[derive(Copy, Clone, Debug)]` 在 Rust 中自动为类型实现 `Copy`、`Clone` 和 `Debug` trait<sup>7</sup>，以支持类型的无缝复制、克隆和调试格式化输出。

#### B. `GdtDescriptor` 数据结构

`GdtDescriptor` 结构体用于在使用 `lgdt` 指令加载 GDT 时，向处理器提供 GDT 的大小和内存地址。`size` 字段存储的是 GDT 的字节大小减去一（因为 GDT 的界限是从零开始计数的）。`offset` 是一个指向 `GlobalDescriptorTable` 的指针，表示 GDT 在内存中的位置。

#### C. `GlobalDescriptorTable` 数据结构

<sup>7</sup>在 Rust 语言中，trait 是一种定义和共享接口或行为的机制，类似于其他编程语言中的接口（如 Java 的接口）或抽象基类。它允许程序员规定具有哪些方法的类型可以做什么，而不必关心这些方法是如何实现的。这样做可以确保不同的数据类型可以共用一套功能，提高代码的复用性和灵活性。

```

1  pub static GDT: GlobalDescriptorTable = {
2      let limit: u64 = {                                // 定义段限制大小 (0xFFFF 表示所有 32 位内存)
3          let limit_low: u64 = 0xFFFF << 0;           // 低位部分, 直接设置为 0xFFFF
4          let limit_high: u64 = 0xF << 48;           // 高位部分, 向左移 48 位
5          limit_low | limit_high                   // 将低位和高位合并, 形成完整的段限制大小, 覆盖全部
6              ↳ 4GB 内存
7      };
8
9      let base: u64 = {                               // 定义基址
10         let base_low: u64 = 0x0000 << 16;        // 基址低位, 向左移 16 位
11         let base_high: u64 = 0x00 << 56;         // 基址高位, 向左移 56 位
12         base_low | base_high                    // 将低位和高位合并, 形成完整的基址, 从 0 开始
13     };
14
15     let access: u64 = {                           // 定义访问字节
16         let p: u64 = 0b1 << 47;                 // Present 位, 必须为 1, 表示内存段是存在的
17         let dpl: u64 = 0b00 << 46;             // 描述符特权级别, 0 是最高特权级别
18         let s: u64 = 0b1 << 44;                 // 描述符类型位, 标记为代码段或数据段
19         let e: u64 = 0b0 << 43;                // 可执行位, 标记为非执行, 用于数据段
20         let dc: u64 = 0b0 << 42;                // 方向位 / 一致位, 不适用于数据段
21         let rw: u64 = 0b1 << 41;                // 可读可写位, 数据段可写, 代码段可读
22         let a: u64 = 0b0 << 40;                // 访问位, 用于跟踪段是否被访问过
23         p | dpl | s | e | dc | rw | a       // 合并所有位, 设置访问权限和段类型
24     };
25
26     let flags: u64 = {                           // 定义标志位
27         let g: u64 = 0b1 << 55;                // 粒度标志, 设为 1 表示段限制以 4KB 为单位
28         let db: u64 = 0b1 << 54;              // 操作数大小, 设为 1 表示 32 位操作
29         let l: u64 = 0b0 << 53;              // 长模式标志, 对于非 64 位代码不设定
30         let r: u64 = 0b0 << 52;              // 保留位, 未使用
31         g | db | l | r                     // 合并标志位, 确定段的属性
32     };
33
34     let executable: u64 = 0b1 << 43;          // 定义可执行位, 专用于代码段, 使代码段为可执行
35
36     let zero = GdtEntry {                      // 定义第一个条目 (空描述符)
37         entry: 0
38     };
39
40     let code = GdtEntry {                      // 定义第二个条目 (代码段描述符)
41         entry: limit | base | access | flags | executable
42     };
43
44     let data = GdtEntry {                      // 定义第三个条目 (数据段描述符)
45         entry: limit | base | access | flags
46     };
47
48     GlobalDescriptorTable {                  // 定义全局描述符表
49         entries: [zero, code, data]
50     };
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
617
618
619
619
620
621
622
623
624
625
625
626
627
627
628
629
629
630
631
631
632
632
633
633
634
634
635
635
636
636
637
637
638
638
639
639
640
640
641
641
642
642
643
643
644
644
645
645
646
646
647
647
648
648
649
649
650
650
651
651
652
652
653
653
654
654
655
655
656
656
657
657
658
658
659
659
660
660
661
661
662
662
663
663
664
664
665
665
666
666
667
667
668
668
669
669
670
670
671
671
672
672
673
673
674
674
675
675
676
676
677
677
678
678
679
679
680
680
681
681
682
682
683
683
684
684
685
685
686
686
687
687
688
688
689
689
690
690
691
691
692
692
693
693
694
694
695
695
696
696
697
697
698
698
699
699
700
700
701
701
702
702
703
703
704
704
705
705
706
706
707
707
708
708
709
709
710
710
711
711
712
712
713
713
714
714
715
715
716
716
717
717
718
718
719
719
720
720
721
721
722
722
723
723
724
724
725
725
726
726
727
727
728
728
729
729
730
730
731
731
732
732
733
733
734
734
735
735
736
736
737
737
738
738
739
739
740
740
741
741
742
742
743
743
744
744
745
745
746
746
747
747
748
748
749
749
750
750
751
751
752
752
753
753
754
754
755
755
756
756
757
757
758
758
759
759
760
760
761
761
762
762
763
763
764
764
765
765
766
766
767
767
768
768
769
769
770
770
771
771
772
772
773
773
774
774
775
775
776
776
777
777
778
778
779
779
780
780
781
781
782
782
783
783
784
784
785
785
786
786
787
787
788
788
789
789
790
790
791
791
792
792
793
793
794
794
795
795
796
796
797
797
798
798
799
799
800
800
801
801
802
802
803
803
804
804
805
805
806
806
807
807
808
808
809
809
810
810
811
811
812
812
813
813
814
814
815
815
816
816
817
817
818
818
819
819
820
820
821
821
822
822
823
823
824
824
825
825
826
826
827
827
828
828
829
829
830
830
831
831
832
832
833
833
834
834
835
835
836
836
837
837
838
838
839
839
840
840
841
841
842
842
843
843
844
844
845
845
846
846
847
847
848
848
849
849
850
850
851
851
852
852
853
853
854
854
855
855
856
856
857
857
858
858
859
859
860
860
861
861
862
862
863
863
864
864
865
865
866
866
867
867
868
868
869
869
870
870
871
871
872
872
873
873
874
874
875
875
876
876
877
877
878
878
879
879
880
880
881
881
882
882
883
883
884
884
885
885
886
886
887
887
888
888
889
889
890
890
891
891
892
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
900
901
901
902
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
1351
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1360
1361
1361
1362
1362
1363
1363
1364
1364
1365
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1370
1371
1371
1372
1372
1373
1373
1374
1374
1375
1375
1376
1376
1377
1377
1378
1378
1379
1379
1380
1380
1381
1381
1382
1382
1383
1383
1384
1384
1385
1385
1386
1386
1387
1387
1388
1388
1389
1389
1390
1390
1391
1391
1392
1392
1393
1393
1394
1394
1395
1395
1396
1396
1397
1397
1398
1398
1399
1399
1400
1400
1401
1401
1402
1402
1403
1403
1404
1404
1405
1405
1406
1406
1407
1407
1408
1408
1409
1409
1410
1410
1411
1411
1412
1412
1413
1413
1414
1414
1415
1415
1416
1416
1417
1417
1418
1418
1419
1419
1420
1420
1421
1421
1422
1422
1423
1423
1424
1424
1425
1425
1426
1426
1427
1427
1428
1428
1429
1429
1430
1430
1431
1431
1432
1432
1433
1433
1434
1434
1435
1435
1436
1436
1437
1437
1438
1438
1439
1439
1440
1440
1441
1441
1442
1442
1443
1443
1444
1444
1445
1445
1446
1446
1447
1447
1448
1448
1449
1449
1450
1450
1451
1451
1452
1452
1453
1453
1454
1454
1455
1455
1456
1456
1457
1457
1458
1458
1459
1459
1460
1460
1461
1461
1462
1462
1463
1463
1464
1464
1465
1465
1466
1466
1467
1467
1468
1468
1469
1469
1470
1470
1471
1471
1472
1472
1473
1473
1474
14
```

```
1 #[derive(Copy, Clone, Debug)]
2 #[repr(C, packed)]
3 pub struct GdtEntry {
4     entry: u64,
5 }
```

代码 4.12 GdtEntry 数据结构定义

```
1 pub struct GdtDescriptor {
2     size: u16,
3     offset: *const GlobalDescriptorTable,
4 }
```

代码 4.13 GdtDescriptor 数据结构定义

GlobalDescriptorTable 结构体包含一个固定大小的 GdtEntry 数组，即实际的 GDT，存储着所有的段描述符。提供 load 方法，用于加载 GDT 到 CPU。此方法创建一个 GdtDescriptor 实例，设置 GDT 的大小和地址，然后通过 lgdt 指令告诉 CPU 新的 GDT 的位置。使用 #[repr(C, packed)] 确保结构体没有任何内存对齐填充，与硬件要求的内存布局严格一致。

```
1 #[repr(C, packed)]
2 pub struct GlobalDescriptorTable {
3     entries: [GdtEntry; GDT_ENTRIES],
4 }
5
6 impl GlobalDescriptorTable {
7     pub fn load(&self) {
8         let descriptor = GdtDescriptor {
9             size: (GDT_ENTRIES * size_of::<GdtEntry>() - 1) as u16,
10            offset: self,
11        };
12        unsafe {
13             core::arch::asm!("lgdt [{0:e}]", in(reg) &descriptor);
14         }
15     }
16 }
```

代码 4.14 GlobalDescriptorTable 数据结构定义

### 4.3.5 打印器模块 (Printer Module)

打印器模块 (Printer Module) 负责输出信息到屏幕。这个模块利用 BIOS 中断 (INT 0x10) 来在屏幕上打印字符和字符串。它实现了 fmt::Write trait，使得可以使用 Rust 的格式化宏如 print! 和 println! 等来输出格式化文本。

#### A. 结构体和全局变量

定义为 pub static mut PRINTER: Printer = Printer {}, 这是一个全局可变的 Printer 实

例。它声明为 `mut` 表示它是可变的。`Printer` 为一个空结构体，作为方法的容器，用于实现打印功能。

### B. 实现 `fmt::Write` trait

为 `Printer` 实现 `fmt::Write` trait 的 `write_str` 方法（如代码 4.15），允许使用 Rust 的格式化字符串功能。此方法接受一个字符串切片 `s` 并调用 `prints` 方法来逐字符打印。成功后返回 `fmt::Result` 类型的 `Ok(())`。

```

1  impl fmt::Write for Printer {
2      fn write_str(&mut self, s: &str) -> fmt::Result {
3          self.prints(s);
4          Ok(())
5      }
6  }
```

代码 4.15 `write_str` 方法

### C. 实现 `Printer`

`Printer` 提供了三种方法（代码 4.16），`printc` 方法用于在屏幕上打印一个字符，`prints` 方法用于在屏幕上打印一个字符串，`clear` 方法用于清除屏幕并将显示重置为文本模式。

(1) **`printc` 方法：**`printc` 方法用于在屏幕上打印一个字符。该方法使用内联汇编代码调用 INT 0x10 BIOS 中断来将字符显示在屏幕上。在这里，`al` 寄存器同样装载字符的 ASCII 值，`ah` 寄存器设置为 0x0e 以选择 BIOS 的“显示字符”的功能，`bx` 寄存器设置为 0 用于选择默认的显示页。这段代码使得 BIOS 根据 `al` 寄存器的值在屏幕上打印相应的字符。

(2) **`prints` 方法：**`prints` 方法接收一个字符串，并遍历其中的每个字符，使用 `printc` 方法逐个打印到屏幕。这是通过一个简单的循环实现的，循环遍历字符串的每个字符并将其传递给 `printc` 方法。这种方法简洁有效地处理了字符串的逐字符显示，适用于在屏幕上输出完整的文本行或消息。

(3) **`clear` 方法：**`clear` 方法用于清除屏幕并将显示重置为文本模式。这是通过调用 INT 0x10 BIOS 中断实现的，其中 `ax` 寄存器设置为 0x0003。这个特定的中断和 `ax` 值的组合告诉 BIOS 将视频模式重置到标准文本模式，并清空当前显示的内容。使用这种方法可以快速清除屏幕中的所有文本。

### D. 实现宏

MinmusOS 实现了三种宏：`clear!` 宏用于清除屏幕显示，`print!` 宏提供格式化文本输出到屏幕，而 `println!` 宏在 `print!` 的基础上额外添加了自动换行功能。宏实现工具函数见代码 4.17。

(1) **`clear!` 宏：**`clear!` 宏提供了一个简单而直接的方式来清除屏幕的内容。它的实现包括一条单独的语句，调用内部定义的 `_clear()` 函数。这个函数通过执行 `Printer` 类的 `clear` 方法来清空屏幕，后者使用 BIOS 中断 INT 0x10 来设置视频模式并清除显示。通过这个宏，开发者可以在任何地方简单地写 `clear!();` 来清除屏幕，而不需要直接调用底层清屏函数。

(2) **`print!` 宏：**`print!` 宏允许开发者以类似于 Rust 标准库中的 `print!` 功能的方式打印格式化文本。这个宏接收任意数量的参数，使用 `format_args!` 宏将这些参数转换成一个

```
1  impl Printer {
2      pub fn printc(&self, c: char) {
3          unsafe {
4              asm!(
5                  "int 0x10",
6                  "in(%al)" c as u8,
7                  "in(%ah)" 0x0eu8,
8                  "in(%bx)" 0u16,
9                  );
10         }
11     }
12
13     pub fn prints(&self, s: &str) {
14         for c in s.chars() {
15             self.printc(c);
16         }
17     }
18
19     #[allow(dead_code)]
20     pub fn clear(&self) {
21         unsafe {
22             asm!("int 0x10", in("ax") 0x0003u16);
23         }
24     }
25 }
```

代码 4.16 Printer 实现

fmt::Arguments 类型，这是一个预格式化的文本表示。之后，它调用 \_print 函数，该函数实际上通过 write\_fmt 方法将文本输出到屏幕。这使得打印操作不仅支持简单的字符串，还支持复杂的格式化操作。

(3) `println!` 宏: `println!` 宏扩展了 `print!` 宏的功能，添加了自动换行的特性。它在功能上类似于 `print!` 宏，但在每次调用结束时自动加上换行符。这个宏同样使用 `format_args!` 来处理输入的参数并格式化它们，然后调用 `_print` 来输出最终的字符串。`println!` 宏非常适合用于输出日志信息或其他需要按行处理的文本输出，简化了每次输出后手动添加换行符的需要。

```
1  pub fn _print(args: fmt::Arguments) {
2      use core::fmt::Write;
3      unsafe {
4          PRINTER.write_fmt(args).unwrap();
5      }
6  }
7
8  pub fn _clear() {
9      unsafe {
10         PRINTER.clear();
11     }
12 }
```

代码 4.17 宏实现工具函数

### 4.3.6 主启动程序 (Main Boot Program)

主启动程序是引导加载器的入口点。它负责初始化硬件设备、设置内存模式、加载内核到内存，并转交控制权到内核。此模块包含了转换到不真实模式和保护模式的逻辑，以及核心启动逻辑。

错误处理函数 `fail` 和紧急停止处理函数 `panic` 的实现和子小节 4.2.5 类似，在此不再赘述。

#### A. 常量定义

代码 4.18 定义了内核数据的存放位置和大小，以及内存中的目标位置。

```

1 const KERNEL_LBA: u64 = 4096;           // 内核的逻辑块地址 (LBA)
2 const KERNEL_SIZE: u16 = 2048;          // 内核大小，以扇区为单位
3 const KERNEL_BUFFER: u16 = 0xBE00;       // 用于复制内核的缓冲区的内存地址
4 const KERNEL_TARGET: u32 = 0x00100000;   // 内核被加载到内存中的目标地址

```

代码 4.18 常量定义

#### B. 切换到不真实模式 (Unreal Mode)

`unreal_mode` 函数（代码 4.19）主要是用来设置处理器从实模式（Real Mode）切换到所谓的“Unreal Mode”，代码实现逻辑如下：

```

1 fn unreal_mode() {
2     // 保存段寄存器值
3     let ds: u16;
4     let ss: u16;
5     unsafe {
6         asm!("mov {0:x}, ds", out(reg) ds);
7         asm!("mov {0:x}, ss", out(reg) ss);
8     }
9
10    // 加载全局描述符表 (GDT)
11    GDT.load();
12
13    unsafe {
14        // 读取并修改 CR0 寄存器
15        let mut cr0: u32;
16        asm!("mov {0:e}, cr0", out(reg) cr0);
17        let cr0_protected = cr0 | 1;
18        asm!("mov cr0, {0:e}", in(reg) cr0_protected);
19
20        // 设置新的段寄存器值
21        asm!("mov {0:x}, 0x10", "mov ds, {0:x}", "mov ss, {0:x}", out(reg)_);
22
23        // 恢复 CR0 和段寄存器
24        asm!("mov cr0, {0:e}", in(reg) cr0);
25        asm!("mov ds, {0:x}", in(reg) ds);
26        asm!("mov ss, {0:x}", in(reg) ss);
27    }
28}

```

代码 4.19 切换到不真实模式 (Unreal Mode) 实现

(1) **保存段寄存器值**: 通过内联汇编命令保存当前的数据段寄存器 (DS) 和栈段寄存器 (SS)。这是为了后面能够恢复这些寄存器的原始值，确保在修改后可以返回到原始状态。

(2) **加载全局描述符表 (GDT)**: 调用加载全局描述符表，GDT 中定义了不同的内存段，包括其大小、权限等信息。加载 GDT 是为了使得后续可以设置更大的段限制。

(3) **读取并修改 CR0 寄存器**: 首先读取 CR0 寄存器的当前值，然后设置最低位 (启用保护模式的标志位)。

(4) **设置新的段寄存器值**: 将段寄存器 DS 和 SS 设置为 GDT 中新的段选择子 (0x10)，这指向一个具有更大段限制的描述符。

(5) **恢复 CR0 和段寄存器**: 最后，将 CR0 寄存器恢复到原来的值，并将 DS 和 SS 段寄存器恢复到之前保存的值。这是为了确保系统的稳定性和正确回到原始状态。

Unreal Mode 的主要功能是允许实模式下的应用程序访问更大的内存空间。虽然 CPU 仍然在实模式下运行，但通过修改段寄存器的限制，可以访问高于 1MB 的内存。这在早期的 DOS 扩展中非常有用，允许 16 位程序突破 640KB 内存限制。在现代操作系统开发中，这种模式可以在启动过程中用于设置和初始化更复杂的内存管理机制。

### C. 切换到保护模式 (Protected Mode)

protected\_mode 函数 (代码 4.20) 实现了在一个操作系统的启动过程中从实模式切换到保护模式。保护模式是现代操作系统中使用的模式，它支持更高级的特性，如更大的内存寻址、虚拟内存和进程隔离等。代码实现逻辑如下：

#### (1) 设置 CR0 寄存器以启用保护模式

- ① `mov eax, cr0`: 将控制寄存器 CR0 的值移动到 EAX 寄存器中。CR0 寄存器中的位控制着处理器的操作模式和状态。
- ② `or al, 1`: 将 EAX 寄存器的最低字节 AL 与 1 进行逻辑或操作，这样设置了 CR0 寄存器的最低位 (PE 位, Protection Enable)，用以启动保护模式。
- ③ `mov cr0, eax`: 将修改后的 EAX 寄存器的值回写到 CR0 寄存器，从而实际启用保护模式。

(2) **推送内核入口地址**: 这一步将内核的目标入口地址推送到栈上。这个地址稍后将用于跳转到内核执行。

(3) **长跳转以刷新管线和真正切换到保护模式**: 执行一个长跳转，这里的 \$0x8 是新的代码段选择子，指向 GDT 中定义的保护模式代码段。\$2f 是标签，跳转到下一行代码，这种跳转是必需的，因为它使 CPU 的内部管线刷新，确认从实模式完全切换到保护模式。

#### (4) 设置段寄存器和跳转到内核

- ① `.code32`: 指示开始使用 32 位代码。
- ② `mov 0:e, 0x10`: 设置各个段寄存器指向 GDT 中的相应保护模式段。
- ③ `mov ds, 0:e, mov es, 0:e, mov ss, 0:e`: 分别将数据段、附加段和栈段寄存器设

置为保护模式下的值。

- ④ `pop 1:e`: 从栈中弹出之前保存的内核入口地址。
- ⑤ `call 1:e`: 跳转到内核的入口地址开始执行内核代码。

```

1 fn protected_mode() {
2     unsafe {
3         // 设置 CR0 寄存器以启用保护模式
4         asm!("mov eax, cr0", "or al, 1", "mov cr0, eax");
5
6         // 推送内核入口地址
7         asm!("push {0:e}", in(reg) KERNEL_TARGET);
8
9         // 长跳转以刷新管线和真正切换到保护模式
10        asm!("ljmp $0x8, $2:", "2:", options(att_syntax));
11
12        // 设置段寄存器和跳转到内核
13        asm!(
14            ".code32",
15            "mov {0:e}, 0x10",
16            "mov ds, {0:e}",
17            "mov es, {0:e}",
18            "mov ss, {0:e}",
19            "pop {1:e}",
20            "call {1:e}",
21            out(reg) _,
22            in(reg) KERNEL_TARGET,
23        );
24    }
25 }
```

代码 4.20 切换到保护模式（Protected Mode）实现

#### D. 主入口函数

代码 4.21 定义了一个操作系统引导加载器中的主入口函数 `_start`。这个函数是操作系统从实模式到保护模式转换过程中的关键部分，并且负责加载内核到内存中。下面是对该函数中各部分的解释：

- (1) `#[no_mangle]`: 这个属性防止 Rust 编译器改变这个函数的名称，这是确保启动代码能正确链接到这一特定入口点非常重要的一步。
- (2) `#[link_section = ".start"]`: 指示编译器将这个函数放在 ELF 文件<sup>8</sup>的 `.start`段中。这是 bootloader 约定用于存放入口点代码的段。
- (3) **切换到 Unreal Mode**: 调用 `unreal_mode` 函数，这个函数设置处理器以允许访问扩展的内存空间，虽然仍然在 16 位模式下运行。
- (4) **加载内核**: 在 `unsafe` 块中，调用 `DISK.init()` 和 `DISK.read_sectors()` 来初始化磁盘并

<sup>8</sup>ELF (Executable and Linkable Format) 文件格式是一种广泛使用的标准文件格式，用于定义可执行文件、对象代码、共享库和核心转储。这种格式主要在 Unix 和 Unix-like 系统（如 Linux、Solaris、IRIX、FreeBSD、NetBSD 等）中使用，但也可在其他操作系统上实现。ELF 文件的主要用途包括可执行文件、对象文件、共享库、核心转储文件等。ELF 格式提供了一种灵活且强大的方式来组织和使用不同类型的代码和数据。这种格式的统一性和广泛支持使其成为了现代操作系统在处理可执行文件和库时的首选格式。它也支持高级特性如动态链接和执行时重定位，这些特性对于现代操作系统的动态性和模块化非常重要。

```
1 #[no_mangle]
2 #[link_section = ".start"]
3 pub extern "C" fn _start() -> ! {
4     println!("[INFO] Switching to Unreal Mode...");
5     unreal_mode();
6     println!("[INFO] Loading Kernel...");
7     unsafe {
8         DISK.init(KERNEL_LBA, KERNEL_BUFFER);
9         DISK.read_sectors(KERNEL_SIZE, KERNEL_TARGET);
10    }
11    println!("[INFO] Loading Global Descriptor Table...");
12    GDT.load();
13    println!("[INFO] Switching to Protected Mode...");
14    protected_mode();
15    loop {}
16 }
```

代码 4.21 主入口函数

从磁盘读取内核到内存中的指定位置。这需要使用 `unsafe` 语句，因为它涉及直接的硬件操作和内存操作，这些操作可能导致未定义行为。

(5) 加载全局描述符表 (GDT)：调用 `GDT.load()` 函数来设置 CPU 的段描述符，这是进入保护模式前必需的步骤。

(6) 切换到 Protected Mode：调用 `protected_mode()` 函数，该函数负责将 CPU 从 Unreal Mode 完全切换到 32 位的保护模式。此模式支持更高级的内存管理和保护功能。

(7) 无限循环：`loop {}` 确保 `_start` 函数在执行完成后不会返回或执行任何其他未定义的操作。

#### 4.3.7 链接器脚本 (Linker Script)

这个链接器脚本（代码 4.22）专为操作系统的引导程序设计，目的是定义和控制引导程序在内存中的布局。链接器脚本在系统启动过程中起到至关重要的作用，确保引导程序正确地加载和执行。脚本总体可以分为以下几个关键部分来概括其作用：

(1) 程序入口点：指定 `_start` 函数为程序的入口点，这是系统开始执行引导程序的地方。

(2) 段的配置：设置引导程序的加载地址。`0x7C00` 是 BIOS 传统上将引导扇区加载到内存的位置，加上 `512` 字节（一个扇区的大小）是为了紧接引导扇区之后开始。

(3) 各种内存段的定义：`.start` 包含引导程序的实际入口代码，`.text` 包含程序的机器代码，`.bss` 用于存储程序中未初始化的数据，`.rodata` 包含只读数据，`.data` 包含已初始化的全局变量和静态变量，`.eh_frame` 和 `.eh_frame_hdr` 包含用于支持运行时错误处理的异常处理信息。

(4) 内存布局的结束配置：将位置指针设置为引导加载器起始位置加上 `32KB` 减 `2` 字节，确保引导加载器的大小正好为 `32KB`。在内存中设置一个结束标记（使用 `SHORT(0XDEAD)`），用于标识引导加载器的结束。

在 Rust 项目中，`build.rs` 文件充当构建脚本的角色，主要用于在项目编译前执行自定义的构建

```

1  /* 配置程序入口点 */
2 ENTRY(_start)
3
4 /* 配置段的顺序和位置 */
5 SECTIONS {
6     /* 配置引导加载器的开始地址为 0x7C00 加 512B (即一个扇区后), 这是 BIOS 启动扇区的传统位置 */
7     . = 0X7C00 + 512;
8     _bootloader_start = .;
9
10    /* 定义启动段, 包含启动代码的实际入口点 */
11    .start : {
12        *(.start)
13    }
14
15    /* 定义代码段, 包含程序的机器代码 */
16    .text : {
17        *(.text .text.*)
18    }
19
20    /* 定义 BSS 段, 包含程序中未初始化的数据 */
21    .bss : {
22        *(.bss .bss.*)
23    }
24
25    /* 定义只读数据段, 包含常量等不应被程序修改的数据 */
26    .rodata : {
27        *(.rodata .rodata.*)
28    }
29
30    /* 定义数据段, 包含已初始化的全局变量和静态变量 */
31    .data : {
32        *(.data .data.*)
33    }
34
35    /* 配置异常处理信息, 用于支持运行时错误处理 */
36    .eh_frame : {
37        *(.eh_frame .eh_frame.*)
38    }
39    .eh_frame_hdr : {
40        *(.eh_frame_hdr .eh_frame_hdr.*)
41    }
42
43    /* 将位置指针设置为引导加载器起始位置加上 32KB 减 2B, 确保引导加载器大小为 32KB */
44    . = _bootloader_start + 0X8000 - 2;
45
46    /* 在内存中设置一个结束标记, 用于标识引导加载器的结束 */
47    .end_marker :
48    {
49        SHORT(0XDEAD)
50    }
51}

```

代码 4.22 bootloader/linker.ld

任务。构建脚本的实现与子小节 4.2.6 相同，在此不再赘述。

通过精确地控制各个内存段的位置和大小，这个脚本确保引导程序能够被正确地加载到预期的内存地址，并且所有的代码和数据都按预期方式进行组织。通过配置 .eh\_frame 和 .eh\_frame\_hdr，增强了程序的健壮性，使得在出现运行时错误时能够进行有效的异常处理。

至此我们实现了从实模式到保护模式的切换，并完成了内核的加载（图 4.23）。

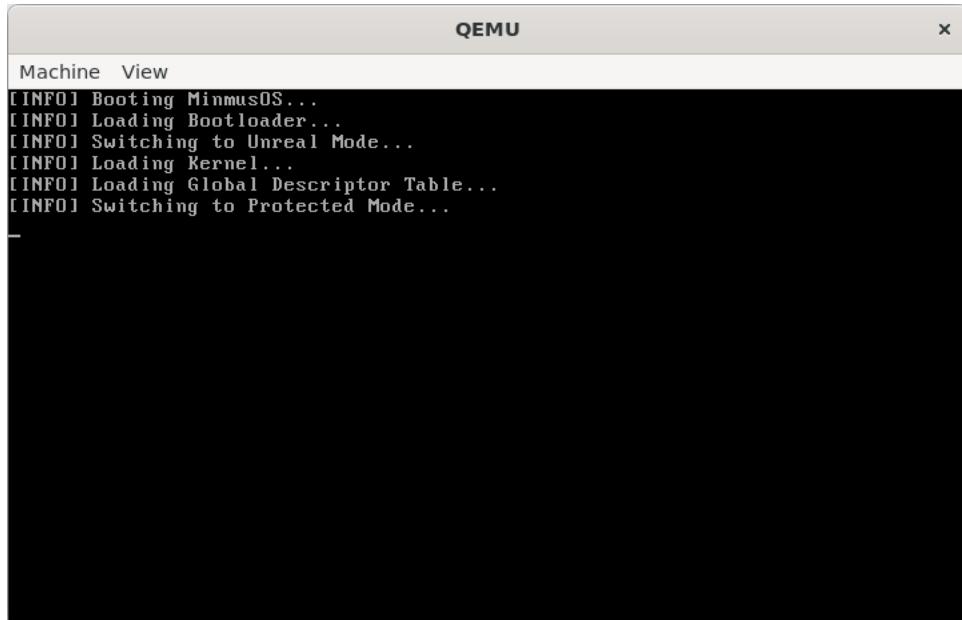


图 4.23 引导加载器演示

## 5 内核功能实现

内核是操作系统的核心组件，负责管理内存和设备，同时为软件应用程序提供使用这些资源的接口。根据复杂性和目标的不同，内核的开发可以遵循不同的模型。主要有两种模型：微内核<sup>9</sup> 和 宏内核<sup>10</sup>。微内核的目标是将大多数服务运行在用户空间，从而提高安全性和模块化。

MinmusOS 采用的是宏内核模式，因此其所有服务和设备驱动程序都在单一地址空间内运行，并处于特权模式下。这种模式提高了效率并减少了代码复杂性，但缺点是其中任何一个组件的错误都可能导致整个系统崩溃。

在本节（章节 5）中，笔者将详细探讨 MinmusOS 内核各项功能的实现，包括中断管理、驱动程序、多任务处理、系统调用、内存管理、文件系统、计时器以及命令行解释器等。

### 5.1 中断 (Interrupts)

中断是操作系统中关键的概念之一，主要用于处理外部设备的信号或内部事件。中断机制使得 CPU 可以响应并处理紧急事件，同时保持对其他任务的执行。中断主要分为三种类型：

(1) **异常 (Exceptions)**：同步中断，由 CPU 执行指令过程中的事件或错误引发，例如除以零错误、执行非法指令或访问无效内存地址。异常通常分为两类：可屏蔽异常和不可屏蔽异常。可屏蔽异常允许操作系统根据当前的需求和优先级来决定是否立即处理。

(2) **硬件中断 (Hardware Interrupts)**：异步中断，由外部硬件设备产生，例如键盘输入、鼠标移动或网络数据包到达。硬件中断使得设备可以通知 CPU 它们需要立即处理的事件，这对于实时数据处理非常重要。

(3) **软件中断 (Software Interrupts)**：由执行特定的 CPU 指令集（如 INT 指令）显式触发，常用于操作系统的系统调用，比如文件访问、网络通信或其他服务请求。软件中断还可以用来执行 BIOS（基本输入输出系统）例程，或者其他低级操作。

当中断发生时，CPU 会采取以下步骤来处理：

(1) **保存当前状态**：CPU 首先保存当前正在执行的任务的状态，以便中断处理完成后可以恢复。

(2) **识别中断类型**：CPU 通过查询中断向量表（一个存储了所有中断处理程序入口地址的表格）来识别中断的具体类型和来源。

(3) **执行中断服务例程 (ISR, Interrupt Service Routine)**：每种中断类型都有对应的服务程序。CPU 执行这些程序来处理中断。

(4) **恢复状态**：中断服务完成后，CPU 恢复之前中断的任务状态，继续之前的操作。

中断处理机制的设计对于操作系统的效率和响应时间至关重要。优秀的中断处理策略可以显著提高系统的整体性能和用户体验。

<sup>9</sup>微内核是一种操作系统内核设计，其核心理念是将内核功能最小化，只保留最基本的操作系统服务，如最低级的内存管理、进程调度和通信机制。其他更复杂的服务，如文件系统、网络协议、设备驱动等，则运行在用户空间。

<sup>10</sup>宏内核则是传统的内核设计方式，将大部分的系统服务和管理功能，如设备驱动、文件系统管理、网络服务等，都直接在内核空间内实现。

### 5.1.1 中斷描述符表 (Interrupt Descriptor Table)

中断描述符表 (IDT) 是 x86 架构中特有的一个关键数据结构，用于管理中断的处理。它的主要目的是提供一个中断服务例程 (ISR) 的地址映射，当中断发生时，CPU 能够知道跳转到哪个位置执行相应的中断处理程序。

`IdtEntry` 是中断描述符表 (IDT) 中的基本单元，它定义了每个中断处理函数的具体属性和位置。`IdtEntry` 结构体的定义如代码 5.1 所示。`#[derive(Copy, Clone, Debug)]` 使用 derive 宏为结构体自动实现 Copy、Clone 和 Debug 这三个 trait，`#[repr(C, packed)]` 用于指定结构体使用 C 语言风格的内存布局并取消字段对齐，以确保字段在内存中紧凑排列。

```

1 #[derive(Copy, Clone, Debug)]
2 #[repr(C, packed)]
3 pub struct IdtEntry {
4     offset_low: u16,           // 偏移地址的低 16 位
5     segment_selector: u16,     // 中断处理程序代码所在的段选择器
6     reserved: u8,             // 保留字段，设置为 0
7     flags: u8,                // 位标志
8     offset_high: u16,         // 偏移地址的高 16 位
9 }
```

代码 5.1 `IdtEntry` 数据结构

位标志，包含门类型 (Gate Type)、DPL (Descriptor Privilege Level) 和 P (Presence)：

(1) **门类型 (Gate Type)**：决定 IDT 条目是中断门还是陷阱门。例如，0xE (1110b) 代表 32 位中断门。

(2) **DPL (Descriptor Privilege Level)**：指定可以调用中断的最低特权级 (CPL)，通常为 0，表示只有内核模式可以使用。

(3) **P (Presence)**：此位必须设置为 1，表示此中断向量是有效的。

`IdtEntry` 结构体提供了 `set` 方法（代码 5.2）用于设置中断处理程序的地址，这个方法接受一个 32 位的整数 `offset`，代表中断处理程序的地址。然后将这个地址分解为两个 16 位的部分，分别存储到 `offset_low` 和 `offset_high`。

```

1 pub fn set(&mut self, offset: u32) {
2     self.offset_low = ((offset << 16) >> 16) as u16;
3     self.offset_high = (offset >> 16) as u16;
4 }
```

代码 5.2 `IdtEntry` 的 `set` 方法

`IdtDescriptor` 结构体（代码 5.3）定义了一个用于加载中断描述符表 (IDT) 的描述符。它用于指定 IDT 的大小和内存中的位置，从而允许处理器正确地找到并使用 IDT。`size` 表示 IDT 的大小，但实际上是 IDT 中最后一个有效条目的偏移量加一。因为中断向量表的条目索引从 0 开始，所以 `size`

字段的值等于 IDT 的字节长度减一。offset 是一个指向 InterruptDescriptorTable 的指针，表示 IDT 的起始物理地址。这个指针告诉处理器从哪里开始读取中断向量表的数据。

```

1 #[repr(C, packed)]
2 pub struct IdtDescriptor {
3     size: u16,
4     offset: *const InterruptDescriptorTable,
5 }
```

代码 5.3 IdtDescriptor 数据结构

InterruptDescriptorTable 结构体（代码 5.4）实现了一个关键的系统级数据结构，即中断描述符表（IDT）。这个表是操作系统中管理中断向量的核心结构，用于定义每个中断或异常的处理程序地址。

```

1 #[repr(C, packed)]
2 pub struct InterruptDescriptorTable {
3     entries: [IdtEntry; IDT_ENTRIES],
4 }
5
6 impl InterruptDescriptorTable {
7     pub fn init(&mut self) {
8         for i in 0..IDT_ENTRIES {
9             self.entries[i] = exceptions::generic_handler as u32;
10        }
11    }
12
13    pub fn add(&mut self, int: usize, handler: u32) {
14        self.entries[int].set(handler);
15    }
16
17    pub fn load(&self) {
18        let descriptor = IdtDescriptor {
19            size: (IDT_ENTRIES * size_of::<IdtEntry>() - 1) as u16,
20            offset: self,
21        };
22        unsafe {
23            asm!("lidt [{0:e}]", in(reg) &descriptor);
24        }
25    }
26
27    pub fn add_exceptions(&mut self) {
28        self.add(0x00, exceptions::division_error as u32);
29        ...
30        self.add(0x15, exceptions::control_protection_exception as u32);
31    }
32 }
```

代码 5.4 InterruptDescriptorTable 数据结构

中断描述符表（IDT）的结构体定义与方法实现：

(1) **结构体定义**: entries 包含了 IDT\_ENTRIES (256) 数量的 IdtEntry 结构体实例。每个 IdtEntry

对应一个中断向量，存储中断处理程序的具体位置和相关属性。

(2) **init** 方法：初始化 IDT 的每个条目，将所有中断向量的处理程序设置为一个通用的处理函数 `exceptions::generic_handler`。

(3) **add** 方法：设置特定中断向量的处理程序。它接受中断向量的索引和处理程序的地址（函数指针），然后调用 `IdtEntry` 的 `set` 方法来更新相应的条目。

(4) **load** 方法：将 IDT 加载到 CPU 的 IDT 寄存器。它创建一个 `IdtDescriptor` 结构体，该结构体包含了 IDT 的大小和地址，然后使用 `lidt` 汇编指令加载这个描述符。

(5) **add\_exceptions** 方法：为常见的 CPU 异常配置特定的处理程序，如除零错误、无效操作码、双重故障、通用保护和页面错误等，中断向量表见表 5.1。

`IDT_ENTRY` 是 `IdtEntry` 类型的静态变量，代码 5.5 定义了一个默认的 IDT 条目模板。这个模板被用来初始化 IDT 数组中的每个条目。

```
1 pub static IDT_ENTRY: IdtEntry = {
2     // 段选择器，确定中断处理程序的代码段
3     let segment_selector: u16 = {
4         let rpl = 0b00 << 0;           // 请求特权级 (Ring Privilege Level)，0 表示最高权限级别
5         let ti = 0b0 << 2;          // 表指示符 (Table Indicator)，0 表示使用全局描述符表 (GDT)
6         let index = 0b1 << 3;        // GDT 中的索引，这里是 0x8
7         rpl | ti | index          // 组合成完整的段选择器
8     };
9
10    // 保留字段，必须设为 0
11    let reserved: u8 = 0;
12
13    // 标志位字段，定义了 IDT 条目的属性
14    let flags: u8 = {
15        let gate_type = 0xe << 0; // 门类型，0xE 表示 32 位中断门，0xF 表示 32 位陷阱门
16        let zero = 0 << 3;       // 保留位，必须为 0
17        let dpl = 0 << 5;        // 描述符特权级，0 表示只有内核模式可以使用此中断
18        let p = 1 << 7;          // 存在位 (Presence bit)，1 表示此中断有效
19        gate_type | zero | dpl | p // 组合成完整的标志位
20    };
21
22    // 构造 IdtEntry 结构体实例
23    IdtEntry {
24        offset_low: 0,           // 中断处理程序的低 16 位地址，初始化设置为 0，后续指定具体值
25        segment_selector,        // 上面定义的段选择器
26        reserved,               // 保留位，始终为 0
27        flags,                  // 标志位，定义了 IDT 条目的特性
28        offset_high: 0,          // 中断处理程序的高 16 位地址，初始化设置为 0，后续指定具体值
29    }
30}
```

代码 5.5 IDT 条目模板

IDT 是 `InterruptDescriptorTable` 类型的静态变量，它实际上是一个包含 256 个 `IdtEntry` 条目的数组。这个数组构成了整个中断描述符表，用于响应各种中断请求。代码 5.6 使用 `IDT_ENTRY` 来初始化每个元素，这意味着每个条目初始时都复制了 `IDT_ENTRY` 的设置。特定的中断向量将通过调

用 `InterruptDescriptorTable` 的 `add` 方法来设置特定的处理函数。

```
1 pub static mut IDT: InterruptDescriptorTable = InterruptDescriptorTable {
2     entries: [IDT_ENTRY; IDT_ENTRIES]
3 }
```

代码 5.6 中断描述符表

IDT 定义了操作系统如何响应硬件和软件生成的中断。每个条目对应一个中断向量，系统在收到中断时会查询 IDT 以确定如何处理该中断。通过提供一个统一的方式来处理所有类型的中断，操作系统能够更加有效地管理资源和响应外部事件，从而增强系统的稳定性和响应性。

### 5.1.2 中断服务例程 (Interrupt Service Routines)

中断服务例程 (ISR) 是一种特定的程序，用于处理来自硬件或软件的中断信号。当系统发生中断时，CPU 会自动查找中断描述符表 (IDT) 中相应的服务例程，并跳转到该例程执行相关操作。与普通的 C 函数不同，ISR 由 CPU 直接调用，并使用特定的 `iret` 指令返回，这种指令用于恢复中断前的状态，确保系统能安全地继续执行其他任务。

在编写 ISR 时，需要特别注意 CPU 在调用中断服务例程之前会自动将一些寄存器的值压栈，这些寄存器包括 `EFLAGS`、`EIP` 和 `CS`。`EFLAGS` 寄存器包含了系统的状态标志，`EIP` 是程序计数器，指示下一条执行指令的地址，`CS` 是代码段寄存器，指示当前执行代码的段基址。

大部分操作系统使用专门的汇编语言函数来实现中断处理程序，以确保处理过程的高效和准确。MinmusOS 采用 Rust 的内联汇编技术来实现 ISR。这种方法允许在中断服务例程中直接调用其他 Rust 函数，然后通过 `iretd` 指令安全返回，这种技术的使用使得中断处理程序既可以保持高效，也能利用 Rust 语言的安全特性。

```
1 #[naked]
2 pub extern "C" fn generic_handler() {
3     unsafe {
4         asm!(
5             "push 0xFF",           // 向堆栈推送一个错误代码
6             "call exception_handler", // 调用 exception_handler 函数，用于显示错误信息并处理中断
7             "add esp, 4",          // 调整堆栈指针，以清除之前压入的错误代码
8             "iretd",               // 使用中断返回指令，恢复之前由 CPU 自动保存的状态
9             options(noreturn),    // 表明汇编代码块执行后不会返回到原始的 Rust 函数中
10        );
11    }
12 }
```

代码 5.7 通用中断处理器

代码 5.7 定义了一个用于处理多种中断的通用中断服务例程 (ISR)。使用 `#[naked]` 属性表示

这个函数没有由编译器生成的 prologue<sup>11</sup> 和 epilogue<sup>12</sup>，使得函数可以完全控制所有的堆栈操作。这种处理方式对于中断处理程序来说是必要的，因为它们需要精确控制寄存器和堆栈的状态，但也增加了复杂性和出错的风险。

```

1  #[no_mangle]
2  pub extern "C" fn exception_handler(int: u32, eip: u32, cs: u32, eflags: u32) {
3      unsafe {
4          PRINTER.set_colors(COLOR_WHITE, COLOR_BLUE);
5          PRINTER.clear();
6      }
7      lib::println!();
8      lib::println!(" ===== MinmusOS v1.0 =====");
9      lib::println!(" ↳ Lin");
10     lib::println!();
11     lib::println!(" SERIOUS KERNEL ERROR!");
12     lib::println!(" MinmusOS has encountered a fatal error and the system has been halted.");
13     lib::println!();
14     lib::println!();
15     lib::println!();
16     lib::println!();
17     lib::println!(" EXCEPTION DESCRIPTION:");
18     lib::println!();
19     match int {
20         0x00 => {
21             lib::println!(" DIVISION ERROR!");
22         }
23         ...
24         _ => {
25             lib::println!(" EXCEPTION!");
26         }
27     }
28     lib::println!();
29     lib::println!();
30     lib::println!(" TECHNICAL INFORMATION:");
31     lib::println!();
32     lib::println!("     ERROR_CODE      : {}", int);
33     lib::println!("     INSTRUCTION_PTR : 0x{:X}", eip);
34     lib::println!("     CODE_SEGMENT    : 0x{:X}", cs);
35     lib::println!("     EXTENDED_FLAGS : 0b{:b}", eflags);
36     lib::println!();
37     lib::println!();
38     lib::print!(" Please restart your computer. :) ");
39     loop {}
40 }
```

代码 5.8 异常处理器

在调试过程中，在代码中通过内联汇编使用 int 命令可以手动触发中断。代码 5.8 为 MinmusOS 提供了友好的用户交互（图 5.9）。蓝屏界面采用了经典的蓝底白字风格，类似于传统的 Windows 蓝屏死机（BSOD），并显示了详细的错误信息，包括错误类型和技术信息（如错误代码、指令指针、

<sup>11</sup> 函数的 prologue 是函数实际执行之前的准备阶段。在这个阶段，编译器或者程序会保存旧的基指针、设置新的基指针和为局部变量分配空间。

<sup>12</sup> 函数的 epilogue 是函数执行完成后的清理阶段。在这个阶段，编译器或者程序会恢复堆栈指针、恢复基指针和返回到调用者。

代码段、扩展标志等)，有助于开发人员定位问题，便于调试和修复。

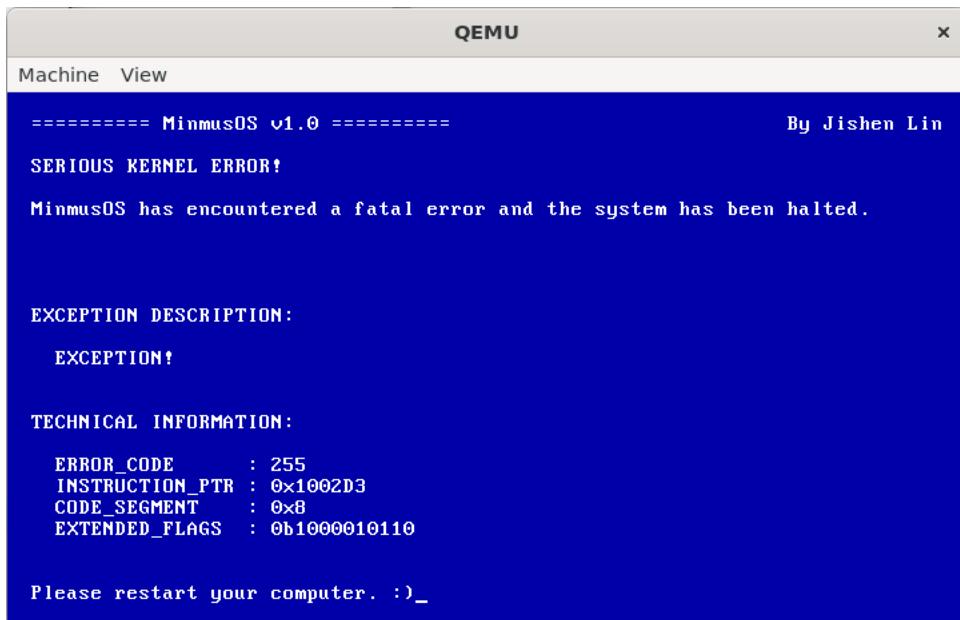


图 5.9 异常处理器演示

### 5.1.3 CPU 异常 (CPU Exceptions)

异常是 CPU 在执行当前指令过程中遇到错误时自身触发的一种中断。在 x86 架构上，大约有 20 种不同的 CPU 异常类型，见表 5.1。表 5.1 参考自 Intel® 64 和 IA-32 架构软件开发者手册。

表 5.1 中断向量表

向量	助记符	描述	类型	错误代码	来源
0	#DE	除零错误	故障	否	DIV 和 IDIV 指令
1	#DB	调试异常	故障/陷阱	否	指令、数据和 I/O 断点；单步执行等
2		NMI 中断	中断	否	不可屏蔽的外部中断
3	#BP	断点	陷阱	否	INT3 指令
4	#OF	溢出	陷阱	否	INTO 指令
5	#BR	BOUND 范围超出	故障	否	BOUND 指令
6	#UD	无效操作码 (未定义操作码)	故障	否	UD 指令或保留操作码
7	#NM	设备不可用 (无数学协处理器)	故障	否	浮点数或 WAIT/FWAIT 指令
8	#DF	双重故障	中止	是 (零)	任何能生成异常、NMI 或 INTR 的指令
9		协处理器段溢出 (保留)	故障	否	浮点数指令
10	#TS	无效的 TSS <sup>13</sup>	故障	是	任务切换或 TSS 访问

续下页

<sup>13</sup>TSS (Task State Segment) 是 Intel x86 架构中用于任务切换和状态维护的数据结构。它存储了处理器在执行任务时需要的所有状态信息，使得操作系统能够管理多任务环境中的任务切换。TSS 的使用主要体现在硬件级别的任务切换和保护模式下的任务状态管理。

续表 5.1

向量	助记符	描述	类型	错误代码	来源
11	#NP	段不存在	故障	是	加载段寄存器或访问系统段
12	#SS	栈段故障	故障	是	栈操作和 SS 寄存器加载
13	#GP	通用保护	故障	是	任何内存引用和其他保护检查
14	#PF	页面错误	故障	是	任何内存引用
15		(Intel 保留)		否	
16	#MF	x87 FPU 浮点错误 (数学故障)	故障	否	x87 FPU 浮点或 WAIT/FWAIT 指令
17	#AC	对齐检查	故障	是 (零)	内存中的任何数据引用
18	#MC	机器检查	中止	否	错误代码 (如果有) 和来源取决于模型
19	#XM	SIMD 浮点异常	故障	否	SSE/SSE2/SSE3 浮点指令
20	#VE	虚拟化异常	故障	否	EPT 违规
21	#CP	控制保护异常	故障	是	RET/IRET/RSTORSSP/SETSSBSY 指令
22-31		(Intel 保留)			
32-255		用户定义 (非保留) 中断	中断		外部中断或 INT n 指令

#### 5.1.4 可编程中断控制器 (Programmable Interrupt Controller)

可编程中断控制器 (Programmable Interrupt Controller, 简称 PIC)，尤指 8259 PIC，是早期计算机主板上用于管理硬件中断的一个小型芯片。在现代计算机中，这一芯片通常已被整合进 CPU 的设计中。PIC 的核心作用是接收来自硬件设备的中断信号，并在 CPU 准备接收时，将这些信号有效地分派给 CPU，从而响应外部设备的事件或请求。

PIC 有 28 个引脚，其中 8 个引脚是中断输入线，用于连接那些可以触发中断的硬件组件。这些中断线路将外部设备如键盘、鼠标或其他 I/O 设备的中断请求直接传输到 PIC。由于单个 PIC 只能处理有限的 8 个中断，现代计算机系统为了扩展中断处理能力，通常会使用两个 PIC 芯片，配置为一主一从模式。这种配置允许系统处理更多的中断线路，总共可达 15 个中断（一个用于连接主从 PIC）。虽然现代计算机系统普遍采用了更为先进的高级可编程中断控制器 (APIC)，以支持更大数量的中断线路和更复杂的中断管理功能，PIC 依旧因其向后兼容性而被支持在新的系统中，确保老旧硬件和软件的正常运行。

CPU 在设计时已经为处理器内部的异常（如除零错误、访问违规等）预留了前 32 个中断向量。因此，来自外部硬件的中断必须被映射到 32 号以上的中断向量，以避免与这些处理器异常冲突。

重新映射中断是通过向 PIC 的命令和数据端口发送特定的命令来完成的。对于主 PIC，其命令端口位于 0x20，数据端口位于 0x21；从 PIC 的命令端口位于 0xA0，数据端口位于 0xA1。初始化 PIC 时，会发送一个初始化命令，设定中断向量的偏移量，指定哪个是主 PIC，哪个是从 PIC，并设置运行模式。

可编程中断控制器 (Programmable Interrupt Controller) 驱动程序的实现见子小节 5.2.1。

## 5.2 驱动程序 (Drivers)

设备驱动程序是一种软件组件，其目的是使特定的硬件设备与内核进行接口交互。默认情况下，操作系统并不知道如何与硬件通信，因此需要驱动程序将操作系统的请求转换成硬件能够理解和执行的命令。

### 5.2.1 可编程中断控制器驱动程序 (Programmable Interrupt Controller Driver)

kernel/src/drivers/pic.rs 定义了一个用于管理可编程中断控制器 (Programmable Interrupt Controller, PIC) 的驱动程序，特别针对老式的 8259 PIC 设计。这些 PICs 主要用于中断管理，使得硬件中断能被适当地映射并处理。

相关常量定义如下：

- (1) `MASTER_PIC_COMMAND_PORT`: 主 PIC 接收命令的端口地址是 0x20
- (2) `MASTER_PIC_DATA_PORT`: 主 PIC 数据读写的端口地址是 0x21
- (3) `SLAVE_PIC_COMMAND_PORT`: 从 PIC 接收命令的端口地址是 0xA0
- (4) `SLAVE_PIC_DATA_PORT`: 从 PIC 数据读写的端口地址是 0xA1
- (5) `COMMAND_INIT`: 用于初始化 PIC 的命令字节，其值为 0x11
- (6) `COMMAND_EOF`: 用于告诉 PIC 中断处理完成的命令字节，其值为 0x20
- (7) `MODE`: 设置 PIC 的工作模式为 8086 模式，其值为 0x01
- (8) `OFFSET`: 定义中断向量表中断向量的起始偏移量，这里设为 32，这是因为 0-31 号中断已被 CPU 的异常占用

(9) `IRQ_COUNT`: 每个 PIC 能够处理的中断数量，这里设为 8，表示每个 PIC 可以管理 8 个中断  
代码 5.10 定义了一个名为 `PICS` 的静态变量，它是 `Pics` 类型的实例，代表系统中的主从可编程中断控制器 (PIC)。该实例包括两个 `Pic` 结构体，分别配置了主和从 PIC 的中断偏移量、命令端口和数据端口。此静态变量在整个程序运行期间全局可访问，用于初始化和管理硬件中断，确保操作系统能够正确响应外部设备的中断请求。

```

1 pub static PICS: Pics = Pics {
2     master: Pic {
3         offset: OFFSET,
4         command_port: MASTER_PIC_COMMAND_PORT,
5         data_port: MASTER_PIC_DATA_PORT,
6     },
7     slave: Pic {
8         offset: OFFSET + IRQ_COUNT,
9         command_port: SLAVE_PIC_COMMAND_PORT,
10        data_port: SLAVE_PIC_DATA_PORT,
11    },
12 }

```

代码 5.10 主从可编程中断控制器 (PIC) 实例

代码 5.11 定义了一个可编程中断控制器 (PIC) 的基本属性和操作方法。这个结构体和其方法主要用于对硬件中断进行低级控制，允许操作系统正确地响应外部中断请求。

```
1 struct Pic {
2     offset: u8,           // 中断向量的起始偏移量，这个偏移量是中断向量表中该 PIC 控制的中断开始位置
3     command_port: u8,    // PIC 的命令端口地址
4     data_port: u8,        // PIC 的数据端口地址
5 }
6
7 impl Pic {
8     // 从 PIC 的数据端口读取一个字节的数据
9     pub fn read_data(&self) -> u8 {
10         let data: u8;
11         unsafe {
12             asm!("in al, dx", out("al") data, in("dx") self.data_port as u16);
13         }
14         data
15     }
16
17     // 将数据写入到 PIC 的数据端口
18     pub fn write_data(&self, data: u8) {
19         unsafe {
20             asm!("out dx, al", in("dx") self.data_port as u16, in("al") data);
21         }
22     }
23
24     // 向 PIC 的命令端口发送指定的命令
25     pub fn send_command(&self, command: u8) {
26         unsafe {
27             asm!("out dx, al", in("dx") self.command_port as u16, in("al") command);
28         }
29     }
30
31     // 通知 PIC 一个中断处理已经结束
32     pub fn end_interrupt(&self) {
33         unsafe {
34             asm!("out dx, al", in("dx") self.command_port as u16, in("al") COMMAND_EOF);
35         }
36     }
37
38     // 检查给定的中断号是否由当前的 PIC 处理，判断依据是中断号是否位于该 PIC 的控制范围内
39     pub fn handles_interrupt(&self, interrupt: u8) -> bool {
40         self.offset <= interrupt && interrupt < self.offset + IRQ_COUNT
41     }
42 }
```

代码 5.11 Pic 数据结构

代码 5.12 定义了一个包含两个可编程中断控制器 (PIC)，即主 PIC 和从 PIC 的组合。该结构体及其方法的设计与实现主要用于系统中断管理，允许系统初始化和管理来自硬件的中断信号。

在与硬件通信时，尤其是在初始化或配置硬件如可编程中断控制器 (PIC) 时，必须确保在发送连续的配置命令之间有足够的间隔，让硬件有机会响应每个命令。wait 函数（代码 5.13）通过向一个通常未使用的端口 (0x80) 写入一个值（通常是 0），来创造一个小的延时，这样的操作需

```

1  pub struct Pics {
2      master: Pic, // 主 PIC
3      slave: Pic, // 从 PIC
4  }
5
6  impl Pics {
7      // 初始化
8      pub fn init(&self) {
9          // 读取当前的中断屏蔽状态
10         let mask1 = self.master.read_data();
11         let mask2 = self.slave.read_data();
12         // 向每个 PIC 发送初始化命令
13         self.master.send_command(COMMAND_INIT);
14         wait();
15         self.slave.send_command(COMMAND_INIT);
16         wait();
17         // 设置偏移量
18         self.master.write_data(self.master.offset);
19         wait();
20         self.slave.write_data(self.slave.offset);
21         wait();
22         // 向主 PIC 发送一个数值 4, 表示从 PIC 通过主 PIC 的 IRQ2 连接, 配置主 PIC 知道如何与从
23         // PIC 通信
24         self.master.write_data(4);
25         wait();
26         // 向从 PIC 发送一个数值 2, 指定它是通过主 PIC 的 IRQ2 接入的, 为从 PIC 配置连接到主 PIC
27         // 的方式
28         self.slave.write_data(2);
29         wait();
30         // 配置为 8086 模式
31         self.master.write_data(MODE);
32         wait();
33         self.slave.write_data(MODE);
34         wait();
35         // 最终恢复初始屏蔽状态
36         self.master.write_data(mask1);
37         self.slave.write_data(mask2);
38     }
39
40     // 检查指定的中断号是否由当前 Pics 实例的主或从 PIC 处理。此方法为中断服务例程提供决策支持, 决
41     // 定是否需要响应某个特定中断
42     pub fn handles_interrupt(&self, interrupt: u8) -> bool {
43         self.master.handles_interrupt(interrupt) || self.slave.handles_interrupt(interrupt)
44     }
45
46     // 通知 PIC 中断处理已经结束。如果从 PIC 处理了中断, 则先通知从 PIC, 然后无论如何都通知主 PIC
47     pub fn end_interrupt(&self, interrupt: u8) {
48         if self.handles_interrupt(interrupt) {
49             if self.slave.handles_interrupt(interrupt) {
50                 self.slave.end_interrupt();
51             }
52             self.master.end_interrupt();
53         }
54     }
55 }

```

代码 5.12 Pics 数据结构

要硬件访问时间，从而实现延迟的目的。

```
1 pub fn wait() {
2     unsafe {
3         asm!("out dx, al", in("dx") 0x80u16, in("al") 0u8);
4     }
5 }
```

代码 5.13 wait 函数

设计与实现考量：

- (1) **安全与底层访问**：由于直接与硬件端口交互，多数操作都在 `unsafe` 块中执行，这符合 Rust 的安全原则，仅在必要时放宽规则。
- (2) **中断管理**：`Pics` 结构体及其方法设计考虑了中断的优先级和处理流程，确保系统对中断的响应既快速又正确。
- (3) **灵活性和扩展性**：通过独立控制每个 `PIC`，系统可以灵活地配置中断处理，适应不同的硬件和系统需求。

### 5.2.2 键盘驱动程序 (Keyboard Driver)

键盘驱动程序是计算机操作系统中的一个重要组件，负责处理和解释来自物理键盘的输入信号。无论是传统的 `PS/2` 键盘还是现代的 `USB` 键盘，驱动程序都扮演着至关重要的角色。以下是键盘驱动程序的几个主要功能和特点：

- (1) **信号解析**：键盘驱动程序首先需要从键盘控制器读取扫描码，这些扫描码是键盘在按键操作时生成的。这些扫描码包含关于哪个键被按下或释放的信息。
- (2) **中断处理**：当按键操作发生时，键盘会向计算机发送一个中断信号，这个信号会触发操作系统中断处理程序。键盘驱动程序必须能够响应这些中断，并从键盘控制器中读取相应的扫描码。
- (3) **数据转换**：驱动程序读取到扫描码后，需要将其解析并转换成具体的键盘事件，如按键按下或释放。这个过程涉及到将扫描码映射到特定的键盘布局和语言设置。
- (4) **事件传递**：解析完成后，驱动程序会生成一个或多个高级的输入事件，这些事件将被操作系统接收并传递给相应的应用程序处理。
- (5) **兼容性和模拟**：对于使用 `PS/2` 接口的旧键盘，现代主板通常通过模拟方式支持，将这些设备表现为 `PS/2` 设备，以便支持旧软件。这种模拟也需要键盘驱动程序处理特定的兼容性问题。

MinmusOS 的 `PS/2` 键盘驱动程序负责处理来自 `PS/2` 键盘的输入，将扫描码 (`scancode`) 转换为 `ASCII` 字符，并相应地更新系统状态。该驱动程序利用中断请求 (`IRQ1`, 即键盘中断) 来响应键盘事件，确保即使在多任务环境下也能及时处理键盘输入。

`Keyboard` 结构体是键盘状态的主要表示形式，包含了各种键盘修饰键的状态（如 `Shift`、`Ctrl`、`Alt` 和 `Caps Lock`）。这些状态用布尔值表示，分别对应左右 `Shift` 键、左右 `Ctrl` 键、左右 `Alt` 键及 `Caps`

Lock 键的开关状态。该结构体通过代码 5.14 定义。

```

1 pub struct Keyboard {
2     left_shift: bool,
3     right_shift: bool,
4     left_ctrl: bool,
5     right_ctrl: bool,
6     left_alt: bool,
7     right_alt: bool,
8     caps_lock: bool,
9 }
```

代码 5.14 Keyboard 数据结构

KEYBOARD 是 Keyboard 结构体的一个静态实例，系统中的任何部分都可以访问此实例来查询当前的键盘状态。该静态实例通过代码 5.15 定义。

```

1 pub static mut KEYBOARD: Keyboard = Keyboard {
2     left_shift: false,
3     right_shift: false,
4     left_ctrl: false,
5     right_ctrl: false,
6     left_alt: false,
7     right_alt: false,
8     caps_lock: false,
9 };
```

代码 5.15 KEYBOARD 静态实例

KeyMap 结构体和 KEYMAP 静态数组是键盘驱动中用于映射扫描码到字符的核心部分。这些定义使得驱动程序能够将从键盘硬件接收到的扫描码转换为相应的字符输出，包括考虑到是否按下了 Shift 键或 Caps Lock 键的情况。KeyMap 结构体定义了键盘上每个键的扫描码及其对应的常规字符和 Shift 状态下的字符，该结构体通过代码 5.16 定义。

```

1 #[derive(Copy, Clone, Debug)]
2 struct KeyMap {
3     scancode: u8,
4     normal: char,
5     shifted: char,
6 }
```

代码 5.16 KeyMap 数据结构

代码 5.17 定义了一个静态数组，包含了所有主要键盘按键的 KeyMap 条目。这个数组作为键盘驱动的查找表，用于在接收到键盘扫描码时快速确定对应的字符输出。

keyboard 函数（代码 5.18）在 MinmusOS 中充当键盘中断的处理入口点，专门用于在发生键盘相关硬件中断时被调用。它是一个裸函数（naked function），这意味着它不会自动产生入口和出口代

```
1 static KEYMAP: &[KeyMap] = &[
2     KeyMap { scancode: 0x02, normal: '1', shifted: '!' },
3     KeyMap { scancode: 0x03, normal: '2', shifted: '@' },
4     ...
5     KeyMap { scancode: 0x10, normal: 'q', shifted: 'Q' },
6     KeyMap { scancode: 0x11, normal: 'w', shifted: 'W' },
7     ...
8     KeyMap { scancode: 0x1A, normal: '[' , shifted: '{' },
9     KeyMap { scancode: 0x1B, normal: ']' , shifted: '}' },
10    ...
11];
```

代码 5.17 KEYMAP 静态数组

码，如保存寄存器或建立栈帧，从而为编写极具控制性的低级硬件交互代码提供了可能。该函数首先通过一系列 push 指令保存关键状态或数据，然后调用 keyboard\_handler 来处理具体的键盘输入，如字符解析和状态更新。在处理完毕后，通过调整栈指针（esp）来清理栈，并使用 iretd 指令完成从中断返回，这样 CPU 可以恢复到中断发生前的状态并继续执行其他任务。这种方法确保了键盘输入的及时和准确处理，对于操作系统的响应性和稳定性至关重要。

```
1 #[naked]
2 pub extern "C" fn keyboard() {
3     unsafe {
4         asm!(
5             "push 0x6D6E6276",
6             "push 0x63787A6C",
7             "push 0x6B6A6867",
8             "push 0x66647361",
9             "push 0x706F6975",
10            "push 0x79747265",
11            "push 0x77713039",
12            "push 0x38373635",
13            "push 0x34333231",
14            "call keyboard_handler",
15            "add esp, 36",
16            "iretd",
17            options(noreturn),
18        );
19    }
20}
```

代码 5.18 keyboard 函数

cancode\_to\_char 函数（代码 5.19）是 MinmusOS 键盘驱动程序中用于将键盘扫描码转换为对应字符的关键函数。它首先检查是否有任何 Shift 键被按下，并查询 Caps Lock 的状态，这些都是通过访问全局 KEYBOARD 结构体得到的。此函数遍历 KEYMAP 数组，寻找与给定扫描码匹配的条目。一旦找到匹配，函数根据是否按下 Shift 键选择对应的字符（正常或 Shift 状态下的字符）。如果所选字符是字母，并且考虑到 Caps Lock 状态与 Shift 键的组合，函数可能会将字符转换为大写或小写。

如果没有找到匹配的扫描码，函数返回空字符串，表示无有效输入。

```

1 fn scancode_to_char(scancode: u8) -> char {
2     let shift_pressed: bool = unsafe { KEYBOARD.left_shift || KEYBOARD.right_shift };
3     let caps_lock: bool = unsafe { KEYBOARD.caps_lock };
4     for key in KEYMAP.iter() {
5         if key.scancode == scancode {
6             let mut character = if shift_pressed {
7                 key.shifted
8             } else {
9                 key.normal
10            };
11            if character.is_ascii_alphabetic() {
12                if caps_lock && !shift_pressed || !caps_lock && shift_pressed {
13                    character = character.to_ascii_uppercase();
14                } else {
15                    character = character.to_ascii_lowercase();
16                }
17            }
18            return character;
19        }
20    }
21    '\0'
22 }
```

代码 5.19 cancode\_to\_char 函数

keyboard\_handler 函数（代码 5.20）是 MinmusOS 键盘驱动中负责处理键盘中断的核心函数。这个函数直接从键盘硬件读取扫描码，并根据这些扫描码更新系统内的键盘状态或者执行对应的动作。以下是函数实现逻辑的详细介绍：

(1) **读取扫描码**：使用 `asm!` 宏直接从键盘数据端口（0x60 端口）读取一个字节的扫描码到 `scancode` 变量中。这是一个底层的硬件交互操作，直接与键盘输入硬件通信。

(2) **处理特殊前缀**：检查读取的扫描码是否为 0xE0，这是多字节扫描码的开始，常见于特殊功能键（如右 Ctrl、右 Alt 等）。如果是，再次从端口读取扫描码，并将 `E0_PREFIX` 标志设置为 `true`，以指示后续的扫描码处理应按照特殊键处理。

(3) **结束中断**：调用 `PICS.end_interrupt(KEYBOARD_INT)` 来通知可编程中断控制器（PIC），键盘中断已经被处理完毕，可以继续接收其他中断。

(4) **处理键状态**：使用两个不同的 `match` 语句来根据是否有 `E0_PREFIX` 标志分别处理扫描码。

(5) **字符输出**：使用 `scancode_to_char` 函数将扫描码转换为相应的字符。如果结果字符不是空字符，则添加到 `shell` 中，实现字符的输出。

这个键盘驱动支持显示数字 0-9、字母 a-z、A-Z、空格以及 32 个常用的 ASCII 特殊字符，并能够处理左右 Shift、左右 Ctrl、左右 Alt 以及 Caps Lock 键的功能。例如，按下和释放 Shift 键会切换对应的布尔状态（如 `left_shift` 或 `right_shift`），从而控制字符的大小写转换和特殊符号的输入。Ctrl 和 Alt 键的状态同样通过布尔变量管理，支持执行快捷操作和特殊功能。Caps Lock 键则通过切换其

装  
订  
线

```
1 #[allow(improper_ctypes_definitions)]
2 #[no_mangle]
3 pub extern "C" fn keyboard_handler() {
4     static mut EO_PREFIX: bool = false;
5     let mut scancode: u8;
6     unsafe {
7         asm!("in al, dx", out("al") scancode, in("dx") 0x60u16);
8         if scancode == 0xE0 {
9             asm!("in al, dx", out("al") scancode, in("dx") 0x60u16);
10            EO_PREFIX = true;
11        }
12        // lib::print!("[0x{:X}]", scancode);
13        PICS.end_interrupt(KEYBOARD_INT);
14        if EO_PREFIX {
15            match scancode {
16                0x1D => KEYBOARD.right_ctrl = true,
17                0x9D => KEYBOARD.right_ctrl = false,
18                0x38 => KEYBOARD.right_alt = true,
19                0xB8 => KEYBOARD.right_alt = false,
20                _ => {}
21            }
22            EO_PREFIX = false;
23        } else {
24            match scancode {
25                0x2A => KEYBOARD.left_shift = true,
26                0xAA => KEYBOARD.left_shift = false,
27                0x36 => KEYBOARD.right_shift = true,
28                0xB6 => KEYBOARD.right_shift = false,
29                0x1D => KEYBOARD.left_ctrl = true,
30                0x9D => KEYBOARD.left_ctrl = false,
31                0x38 => KEYBOARD.left_alt = true,
32                0xB8 => KEYBOARD.left_alt = false,
33                0x3A => {
34                    KEYBOARD.caps_lock = !KEYBOARD.caps_lock;
35                    return;
36                }
37                0x0E => {
38                    SHELL.backspace();
39                    return;
40                }
41                0x1C => {
42                    SHELL.enter();
43                    return;
44                }
45                _ => {}
46            }
47        }
48        let key: char = scancode_to_char(scancode);
49        if key != '\0' {
50            SHELL.add(key);
51        }
52    }
53}
```

代码 5.20 keyboard\_handler 函数

布尔状态来控制键盘输入的大小写模式，提供灵活的文本输入方式。键盘驱动演示如图 5.21。

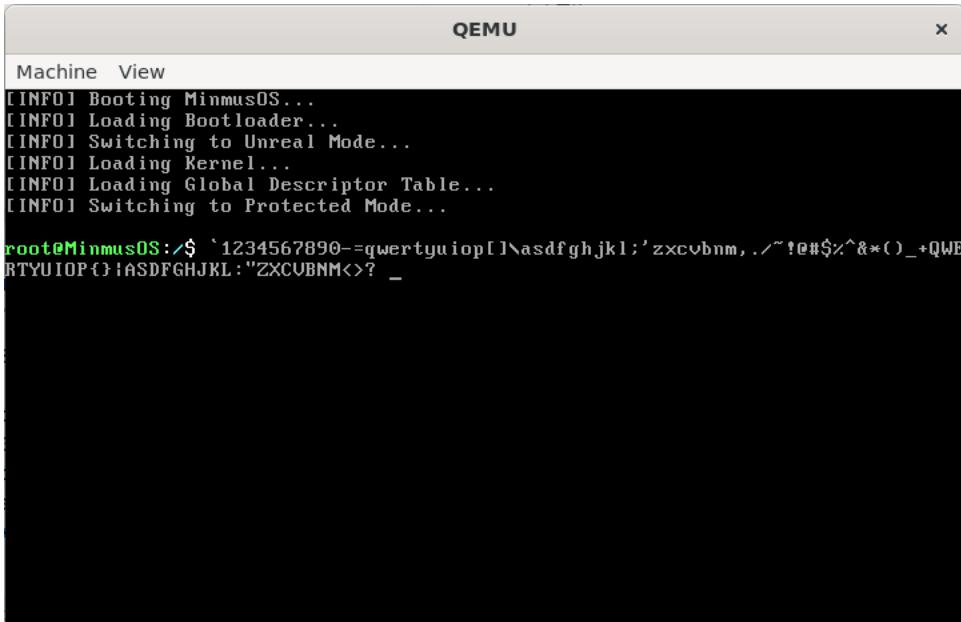


图 5.21 键盘驱动程序演示

### 5.2.3 高级技术附件磁盘驱动程序（Advanced Technology Attachment Disk Driver）

ATA（高级技术附件）磁盘驱动器是一种用于连接存储设备的标准接口，支持硬盘、软盘和光盘驱动器。虽然现代系统中更常见的是使用 SATA 接口，但 ATA 接口因其简单性在一些系统中仍被使用。

ATA 磁盘驱动程序主要用于控制和管理磁盘读写操作。其核心功能之一是“读取”操作，该操作涉及以下几个步骤：

(1) **检查驱动器状态**: 在进行读取之前，驱动程序会检查 ATA 驱动器是否启用并确保其处于非忙碌状态。

(2) **设置寄存器**: 驱动程序设置 ATA 控制器的寄存器，输入必要的数据，如逻辑块地址 (LBA) 和要读取的扇区数量。

(3) **发送读取命令**: 驱动程序向控制器发送读取命令，启动数据传输过程。

(4) **数据传输**: 在读取命令发出后，控制器从磁盘读取数据到其内部缓冲区。驱动程序随后通过一个循环，从缓冲区将数据传输到指定的内存位置。由于磁盘的每个扇区大小通常为 512 字节，而缓冲区可能更小，例如 4 字节，因此需要多次迭代来完整地传输一个扇区的数据。

(5) **循环读取**: 在每次迭代中，驱动程序都需要检查控制器是否已准备好下一次数据传输，确保不会在数据未完全就绪时进行读取。

这种 PIO（程序输入/输出）模式虽然对 CPU 负担较重，但由于其实现简单，因此在一些基础或旧的系统中仍然被采用。通过使用这种方法，ATA 磁盘驱动程序能够有效地管理数据在存储设备

与计算机内存之间的传输。

`kernel/src/drivers/disk.rs` 定义了一个高级技术附件磁盘驱动程序 (Advanced Technology Attachment Disk Driver)，这个磁盘驱动程序在实现上非常直接，主要通过直接与硬件寄存器交互来控制硬盘操作，而不依赖于操作系统提供的抽象层。

相关常量定义如下：

(1) `DATA_REGISTER`: 数据寄存器端口，用于读写数据。当进行数据传输时，数据会通过这个端口传入或传出。

(2) `SECTOR_COUNT_REGISTER`: 扇区计数寄存器，用于指定在一个操作中要传输的扇区数量。

(3) `LBA_LOW_REGISTER`: LBA (逻辑块地址) 的低位寄存器，用于指定 LBA 地址的低 8 位。

(4) `LBA_MID_REGISTER`: LBA (逻辑块地址) 的中位寄存器，用于指定 LBA 地址的中 8 位。

(5) `LBA_HIGH_REGISTER`: LBA (逻辑块地址) 的高位寄存器，用于指定 LBA 地址的高 8 位。

(6) `DRIVE_REGISTER`: 驱动器寄存器，用于选择操作的硬盘和该硬盘的哪个分区。

(7) `STATUS_COMMAND_REGISTER`: 状态和命令寄存器。作为命令寄存器时用于发送命令给 ATA 控制器；作为状态寄存器时用于读取当前的设备状态。

(8) `READ_COMMAND`: 读取命令的代码，当要从硬盘读取数据时，通过状态命令寄存器发送这个命令。

(9) `STATUS_BSY`: 状态寄存器中的忙碌位标志，当硬盘忙碌时该位为 1。

(10) `STATUS_RDY`: 状态寄存器中的就绪位标志，当硬盘准备好进行数据传输时该位为 1。

Disk 结构体定义了一个简单的 ATA 硬盘驱动程序，并包含多个方法用于硬盘的操作和状态检查。字段 `enabled` 类型为 `bool`，标识硬盘是否被启用。

## A. `is_busy` 方法

`is_busy` 方法的主要功能是检查 ATA 硬盘的状态，确保在硬盘忙于处理其他命令时不会发起新的操作。这个方法通过查询硬盘的状态命令寄存器来判断是否有正在执行的任务，从而防止命令冲突和数据损坏。

代码 5.22 使用 Rust 的内联汇编直接从硬盘的状态命令寄存器 (`STATUS_COMMAND_REGISTER`) 读取状态，特别关注忙碌位 (`BSY`)。方法内部通过检查这个忙碌位是否设置 (值为 1)，来确定硬盘是否处于忙碌状态，并返回相应的布尔值。这种直接与硬件寄存器交互的方式使得该方法既高效又精确。

## B. `is_ready` 方法

`is_ready` 方法的功能是检查 ATA 硬盘是否已经准备好接受新的命令。这个方法通过读取硬盘的状态命令寄存器来判断就绪位 (`RDY`)，如果就绪位被设置，表示硬盘处于空闲状态，可以安全地发送新的读写命令。

代码 5.23 利用 Rust 的内联汇编语句从硬盘的状态命令寄存器 (`STATUS_COMMAND_REGISTER`) 直接读取状态值。方法通过检查就绪位 (`RDY`)，这个位在状态寄存器中的特定位置标识硬盘

```

1 pub fn is_busy(&self) -> bool {
2     let status: u8;
3     unsafe {
4         asm!("in al, dx", out("al") status, in("dx") STATUS_COMMAND_REGISTER);
5     }
6     (status & STATUS_BSY) != 0
7 }
```

代码 5.22 is\_busy 方法

是否准备好进行操作。通过与就绪位掩码 (STATUS\_RDY) 进行按位与操作来确定硬盘是否就绪，如果结果为真，则返回 true，表示硬盘已经准备好接受新的操作。这种直接操作硬件寄存器的方法确保了检查的及时性和准确性。

```

1 pub fn is_ready(&self) -> bool {
2     let status: u8;
3     unsafe {
4         asm!("in al, dx", out("al") status, in("dx") STATUS_COMMAND_REGISTER);
5     }
6     (status & STATUS_RDY) != 0
7 }
```

代码 5.23 is\_ready 方法

### C. check 方法

check 方法的功能是验证 ATA 硬盘的操作状态，并更新硬盘的启用状态。此方法通过读取状态寄存器，检查硬盘是否正常响应命令，从而确定硬盘是否工作正常。如果状态寄存器的值表明硬盘处于异常状态（如完全不响应或返回错误状态），则硬盘将被标记为禁用。

代码 5.24 使用 Rust 的内联汇编来直接从硬盘的状态命令寄存器读取当前状态值。通过判断这个状态值是否为 0 或 0xFF（通常表示硬件无响应或通信故障），来决定硬盘是否启用。如果状态值显示硬盘正常（即不是 0 或 0xFF），方法将 enabled 设置为 true 并记录信息日志；反之，设置为 false 并记录错误日志。这种方法可以及时捕捉到硬盘的异常状态，帮助进行故障诊断和维护。

### D. reset 方法

reset 方法的主要功能是重置 ATA 硬盘，将其恢复到初始状态。这个操作对于解决硬盘冻结或响应异常的情况非常有用，通过发送特定的重置命令来清除任何挂起的操作或错误状态，确保硬盘能够重新开始接受命令。

代码 5.25 通过 Rust 的内联汇编语句直接向硬盘的控制寄存器（位于端口 0x3F6）发送重置命令。具体地，它首先发送一个重置启动的命令 (0b00000110)，随后发送一个重置结束的命令 (0b00000010)。这种直接与硬件寄存器交互的方式确保了命令的即时执行和硬盘状态的即时更新，从而有效地重置硬盘。

### E. read 方法

```
1 pub fn check(&mut self) {
2     let status: u8;
3     unsafe {
4         asm!("in al, dx", out("al") status, in("dx") STATUS_COMMAND_REGISTER);
5     }
6     if status != 0 && status != 0xFF {
7         self.enabled = true;
8         lib::println!("[INFO] ATA Disk Driver is Working. Status Register: 0x{:X}", status);
9     } else {
10        self.enabled = false;
11        lib::println!("[ERROR] ATA Disk Driver is not Working! Status Register: 0x{:X}",
12                     status);
13    }
14 }
```

代码 5.24 check 方法

```
1 pub fn reset(&self) {
2     unsafe {
3         asm!("out dx, al", in("dx") 0x3F6, in("al") 0b00000110u8);
4         asm!("out dx, al", in("dx") 0x3F6, in("al") 0b00000010u8);
5     }
6 }
```

代码 5.25 reset 方法

read 方法（代码 5.26）的功能是从 ATA 硬盘上指定的逻辑块地址（LBA）开始读取一定数量的扇区数据到指定的内存地址。此方法在硬盘已启用且处于空闲状态时执行，通过设置必要的控制寄存器和发送读取命令，逐扇区地从硬盘读取数据并存储到提供的目标内存位置。该方法是硬盘驱动中实现数据读取的核心功能，确保了数据能够安全且有效地从硬盘传输到系统内存中。方法的实现逻辑如下：

（1）**检查硬盘是否启用**：方法首先检查 Disk 结构体的 enabled 字段。如果硬盘未启用，则输出错误信息并提前返回，不执行任何读取操作。

（2）**等待硬盘不忙**：在开始设置命令之前，方法循环调用 is\_busy 方法来确保硬盘不处于忙碌状态。这是必要的步骤，因为如果硬盘正忙于处理其他命令，尝试读取可能会导致命令失败或数据损坏。

（3）**设置控制寄存器**：使用内联汇编，方法设置了相关的控制寄存器来准备硬盘进行数据读取。

① **扇区计数寄存器**：设置要读取的扇区数。

② **LBA 寄存器**：设置逻辑块地址，分为低、中、高位，确保完整的 LBA 地址被正确设置。

③ **驱动器寄存器**：设置目标驱动器和 LBA 的最高位。

④ **命令寄存器**：发送读取命令到硬盘。

（4）**读取数据**：初始化一个循环，根据指定的扇区数进行迭代。每个扇区通常是 512 字节。对于每个扇区，方法执行另一个循环 128 次（因为每次从数据寄存器读取 4 字节， $512/4 = 128$ ）。

- ① 内部循环先检查硬盘是否忙碌，然后检查是否就绪。
- ② 使用内联汇编从数据寄存器读取 4 字节数据到本地变量 `buffer`。
- ③ 使用 `core::ptr::write_unaligned` 将读取的数据写入目标内存地址，并将目标指针移动 4 字节以准备下次写入。

(5) **重置硬盘**: 在所有扇区读取完成后，调用 `reset` 方法将硬盘重置到初始状态，确保硬盘不会因为留下未完成的命令而出现问题。

```

1 pub fn read<T>(&self, target: *mut T, lba: u64, sectors: u16) {
2     if !self.enabled {
3         lib::println!("[ERROR] Failed to Read Disk!");
4         return;
5     }
6
7     while self.is_busy() {}
8
9     unsafe {
10        asm!("out dx, al", in("dx") 0x3F6, in("al") 0b00000010u8);
11        asm!("out dx, al", in("dx") SECTOR_COUNT_REGISTER, in("al") sectors as u8);
12        asm!("out dx, al", in("dx") LBA_LOW_REGISTER, in("al") lba as u8);
13        asm!("out dx, al", in("dx") LBA_MID_REGISTER, in("al") (lba >> 8) as u8);
14        asm!("out dx, al", in("dx") LBA_HIGH_REGISTER, in("al") (lba >> 16) as u8);
15        asm!("out dx, al", in("dx") DRIVE_REGISTER, in("al") (0xE0 | ((lba >> 24) & 0xF)) as
16            ← u8);
17        asm!("out dx, al", in("dx") STATUS_COMMAND_REGISTER, in("al") READ_COMMAND);
18    }
19
20    let mut sectors_left = sectors;
21    let mut target_pointer = target;
22
23    while sectors_left > 0 {
24        for _i in 0..128 {
25            while self.is_busy() {}
26            while !self.is_ready() {}
27            let buffer: u32;
28            unsafe {
29                asm!("in eax, dx", out("eax") buffer, in("dx") DATA_REGISTER);
30                core::ptr::write_unaligned(target_pointer as *mut u32, buffer);
31                target_pointer = target_pointer.byte_add(4);
32            }
33            sectors_left -= 1;
34        }
35
36        self.reset();
37    }
}

```

代码 5.26 read 方法

整个方法的实现逻辑既考虑了硬件操作的精确性，也确保了通过适当的检查和等待来避免硬件冲突。这种低级的硬盘操作需要精确控制硬件寄存器，同时确保数据的完整性和正确性。

图 5.27 展示了 ATA 磁盘驱动程序正在工作，且状态寄存器的值为 0x50。状态寄存器的值是一

个 8 位的数，在硬盘接口中有特定的含义，每个位代表硬盘的一个不同的状态。对于状态寄存器值 0x50，将其转换为二进制表达为 0101 0000，分析如下：

- (1) **位 7 (BSY)**: 0 - 硬盘当前不忙
- (2) **位 6 (DRDY)**: 1 - 硬盘已准备好接受命令
- (3) **位 5 (DF)**: 0 - 无设备故障
- (4) **位 4 (DSC)**: 1 - 设备搜索操作已完成
- (5) **位 3 (DRQ)**: 0 - 当前没有数据请求
- (6) **位 2 (CORR)**: 0 - 无已纠正的数据
- (7) **位 1 (IDX)**: 0 - 在检测到状态的时刻，磁盘的旋转周期没有重新开始
- (8) **位 0 (ERR)**: 0 - 无错误



图 5.27 ATA 磁盘驱动程序演示

### 5.3 多任务处理 (Multitasking)

多任务处理是现代操作系统中一项基本功能，允许单个处理器同时运行多个任务。在单核系统中，虽然物理上不可能真正同时执行多个任务，但通过合理的时间分配，操作系统能够让用户感受到多个任务似乎是在同时运行。这种技术称为“并发”(Concurrency)，它与真正的“并行”(Parallelism)不同，后者指的是在多核或多处理器系统中同时执行多个任务。

MinmusOS 是一个支持抢占式多任务的操作系统。与协作式多任务系统不同，协作式系统中的任务需要主动放弃 CPU 使用权，抢占式系统则不需要任务主动放弃处理器。在抢占式系统中，操作系统通过定时器中断来强制从当前任务切换到另一个任务。每当定时器发送中断信号时，中断处理程序会调用调度程序，由调度程序决定下一个要执行的任务。这种方式确保了操作系统能高效地

管理 CPU 资源，响应性好，可以处理大量任务，同时也增强了系统对于紧急处理需求的能力。

### 5.3.1 上下文切换 (Context Switching)

在多任务操作系统中，上下文切换是一个至关重要的过程，它使得操作系统能够在多个任务之间高效切换，保证每个任务都能得到适当的处理器时间。上下文切换主要涉及保存当前任务的 CPU 状态，并在恢复执行另一个任务时加载其之前的状态。这样，即使任务被暂停，稍后恢复时也能如同从未停止过一样继续执行。

上下文的两个主要组成部分是任务栈 (Task Stack) 和 CPU 状态 (CPU State)：

(1) **任务栈**：每个任务都有自己的栈，这使得任务可以在需要时由调度器恢复。栈中保存着执行任务所需的所有信息，包括局部变量、函数调用的返回地址等。

(2) **CPU 状态**：包括各种寄存器的值，如程序计数器、堆栈指针、状态寄存器等，这些寄存器的值决定了任务的执行状态。

在 MinmusOS 中，定时器中断 (IRQ9) 负责在每个定时器滴答时触发调度程序。调度程序的一个关键指令是 `mov esp, eax`，这条指令将堆栈指针 (ESP) 设置为从定时器处理程序返回的值。该处理程序返回下一个要执行任务的堆栈指针。由于这个栈的底部包含了旧的 CPU 状态，所以弹出 (popping) CPU 寄存器就会恢复到该任务被暂停前的 CPU 状态。这种机制确保了操作系统能够在多个任务之间平滑且高效地切换，从而实现高度并发的执行，增强了系统的响应性和多任务处理能力。

代码 5.28 使用了裸函数 (`# [naked]`)，这意味着该函数不会由编译器生成入口和退出代码，完全依赖于程序员手写的汇编代码来控制。这是必须的，因为上下文切换需要精确地控制栈和寄存器的状态。

代码 5.29 是上下文切换逻辑的核心。在 `timer_handler` 函数中，上下文切换的实现涉及调用任务调度器以选择下一个执行的任务并获取其栈指针，计算该任务在内存中的目标地址，然后更新页表以映射到新的任务地址。这样，当中断处理完成后，CPU 可以从新任务的保存状态继续执行，实现任务间的无缝切换。

这种结合汇编和高级语言的方式允许操作系统精确控制硬件，实现高效的多任务处理和上下文切换。通过这样的实现，即使在单核处理器上，操作系统也能够提供平滑的多任务环境。

### 5.3.2 CPU 调度器 (CPU Scheduler)

MinmusOS 使用了轮转调度 (round robin) 算法来管理任务的调度，该算法通过定时器中断处理程序触发。每当发生一个定时器滴答，调度算法就会从任务列表中选择下一个任务，并返回一个指向该任务 CPU 状态的指针。

这个返回的指针非常关键，因为它指向了被选择任务的 CPU 状态。在上下文切换时，上下文切换器会将栈指针设置为这个指针，并从中弹出 (pop) 寄存器的值，以此恢复任务的状态。这样，

```
1 #[naked]
2 pub extern "C" fn timer() {
3     unsafe {
4         core::arch::asm!(
5             "cli",                      // 禁用中断，确保上下文切换不被打断
6             "push ebp",                // 保存所有寄存器，以保持当前任务的状态
7             "push edi",
8             "push esi",
9             "push edx",
10            "push ecx",
11            "push ebx",
12            "push eax",
13            "push esp",                // 将当前栈指针 (esp) 压栈，作为参数传递给 C 函数
14            "call timer_handler",      // 调用 Rust 部分的处理函数
15            "mov esp, eax",           // 将 C 函数返回的新栈指针设为当前栈指针
16            "pop eax",                // 恢复所有寄存器
17            "pop ebx",
18            "pop ecx",
19            "pop edx",
20            "pop esi",
21            "pop edi",
22            "pop ebp",
23            "sti",                     // 重新开启中断
24            "iretd",                   // 从中断返回
25            options(noreturn),
26        );
27    }
28 }
```

代码 5.28 timer 函数

```
1 #[no_mangle]
2 pub extern "C" fn timer_handler(esp: u32) -> u32 {
3     unsafe {
4         // 调用任务调度器，并传递当前的栈指针，返回新任务的栈指针
5         let new_esp: u32 = TASK_MANAGER.schedule(esp as *mut CPUState) as u32;
6         // 获取当前任务的槽位
7         let slot = TASK_MANAGER.get_current_slot();
8         // 计算当前任务的内存目标地址
9         let target = APP_TARGET + (slot as u32 * APP_SIZE);
10        // 更新页表，将任务的代码映射到正确内存位置，确保 CPU 在执行新任务时访问到正确的代码和数据
11        TABLES[8].set(target);
12        // 使页表更改生效
13        PAGING.set_table(8, &TABLES[8]);
14        // 发送中断结束信号到 PIC，告知中断处理完成
15        PICS.end_interrupt(TIMER_INT);
16        // 返回新的栈指针
17        new_esp
18    }
19 }
```

代码 5.29 timer\_handler 函数

当任务再次运行时，就可以从上次停止的地方继续执行，实现任务间的平滑切换。

轮转调度 (Round Robin, RR) 算法是一种非常常见的任务调度方法，尤其适用于时间分片的操作系统中。在 RR 算法中，每个任务被赋予一个固定时间的时间片或量子 (quantum)，所有任务按照一定的顺序排列，形成一个循环队列。

轮转调度算法的主要特点和步骤包括：

(1) **时间片**：系统为每个任务分配一个固定长度的时间片，这是任务可以连续运行的最大时间。时间片的长度通常由操作系统确定，需要平衡响应时间和系统效率。

(2) **任务切换**：当一个任务的时间片用完时，如果它还没有完成，则会被操作系统挂起。调度器将 CPU 控制权转移给队列中的下一个任务。

(3) **循环队列**：所有等待运行的任务形成一个循环队列。调度器按照队列的顺序为每个任务分配 CPU 时间，一旦到达队列末尾，调度器就从队列头开始新一轮的任务调度。

(4) **公平性**：RR 算法的一个显著优点是它的公平性。每个任务都有相同长度的时间运行，这意味着没有单个任务能长时间独占 CPU，从而保证了所有任务都能获得足够的处理时间。

(5) **响应时间**：由于任务经常被切换，RR 算法能提供较好的响应时间，适合需要频繁交互的应用。但是，频繁的任务切换也可能导致较高的开销，因为每次切换都需要进行上下文保存和恢复。

轮转调度算法适用于多任务环境，尤其是处理许多短任务的系统。通过合理设置时间片长度，可以使系统达到良好的性能平衡。

```

1 pub fn schedule(&mut self, cpu_state: *mut CPUState) -> *mut CPUState {
2     if self.task_count <= 0 {
3         return cpu_state;
4     }
5     if self.current_task >= 0 {
6         self.tasks[self.current_task as usize].cpu_state_ptr = cpu_state as u32;
7     }
8     self.current_task = self.get_next_task();
9     self.tasks[self.current_task as usize].cpu_state_ptr as *mut CPUState
10 }
11
12 pub fn get_next_task(&self) -> i8 {
13     let mut i = self.current_task + 1;
14     while i < MAX_TASKS {
15         if self.tasks[i as usize].running {
16             return i;
17         }
18         i = (i + 1) % MAX_TASKS;
19     }
20     -1
21 }
```

代码 5.30 轮转调度 (Round Robin, RR) 算法

代码 5.30 通过两个主要函数 `schedule` 和 `get_next_task` 来实现轮转调度算法。`schedule` 函数首先检查是否有任务在运行，如果有，它会保存当前任务的 CPU 状态。然后，它调用 `get_next_task` 函数来选择下一个任务。

数，从任务列表中按顺序选择下一个可运行的任务，并返回该任务的 CPU 状态指针，供系统加载和执行。

`get_next_task` 函数实现了任务的轮转选择逻辑。它从当前任务的下一个任务开始查找，直到找到一个正在运行的任务。如果到达任务列表末尾，它会循环回到列表的开头，继续查找。如果找不到任何运行的任务，则返回 -1。

这种实现方式确保了所有任务都能依次得到执行机会，公平地分配 CPU 时间，符合轮转调度算法的特点。

### 5.3.3 任务管理器（Task Manager）

任务管理器（Task Manager）是操作系统中负责管理任务的组件，主要职责包括插入、移除和调度任务。任务管理器由多个函数组成，用于操作任务列表，并包含一个数据结构，存储任务数组、任务数量以及当前正在运行任务的 ID。

#### A. CPUSState 数据结构

`CPUSState` 是一个表示 CPU 状态的结构体（代码 5.31），使用 `#[repr(C, packed)]` 属性确保其在内存中的布局与 C 语言的结构体一致且紧密排列。这个结构体保存了寄存器的状态，其中包括手动压入栈的寄存器（如 `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, `ebp`）和 CPU 自动压入的寄存器（如 `eip`, `cs`, `eflags`, `esp`, `ss`）。这些寄存器的值通常在上下文切换或异常处理中被保存和恢复，以确保线程或异常处理返回时能够继续正确的执行。

```
1 #[repr(C, packed)]
2 pub struct CPUSState {
3     eax: u32,
4     ebx: u32,
5     ecx: u32,
6     edx: u32,
7     esi: u32,
8     edi: u32,
9     ebp: u32,
10    eip: u32,
11    cs: u32,
12    eflags: u32,
13    esp: u32,
14    ss: u32,
15 }
```

代码 5.31 `CPUSState` 数据结构

#### B. Task 数据结构

`Task` 结构体（代码 5.32）表示操作系统中的一个任务。它包含了任务运行所需的关键信息，包括一个固定大小的栈 `stack`，一个指向 CPU 状态保存位置的指针 `cpu_state_ptr`，以及一个布尔值 `running`，用于表示任务是否正在运行。`stack` 用于存储任务的局部变量和函数调用信息，而 `cpu_state_ptr` 指向

CPUState 结构体，以保存任务的寄存器状态，方便在任务切换时进行恢复。

```

1 #[derive(Copy, Clone, Debug)]
2 pub struct Task {
3     pub stack: [u8; STACK_SIZE],
4     pub cpu_state_ptr: u32,
5     pub running: bool,
6 }
7
8 impl Task {
9     pub fn init(&mut self, entry_point: u32) {
10         self.running = true;
11         let mut state = &self.stack as *const u8;
12         unsafe {
13             state = state.byte_add(STACK_SIZE);
14             state = state.byte_sub(size_of::<CPUState>());
15         }
16         self.cpu_state_ptr = state as u32;
17         let cpu_state = self.cpu_state_ptr as *mut CPUState;
18         unsafe {
19             (*cpu_state).eax = 0;
20             (*cpu_state).ebx = 0;
21             (*cpu_state).ecx = 0;
22             (*cpu_state).edx = 0;
23             (*cpu_state).esi = 0;
24             (*cpu_state).edi = 0;
25             (*cpu_state).ebp = 0;
26             (*cpu_state).eip = entry_point;
27             (*cpu_state).cs = 0x8;
28             (*cpu_state).eflags = 0x202;
29         }
30     }
31 }
```

代码 5.32 Task 数据结构

Task 结构体实现了一个 init 方法，用于初始化任务。初始化过程中，将 running 标记为 true，表示任务开始运行。然后计算 CPUState 的位置，cpu\_state\_ptr 指向栈顶减去 CPUState 大小的位置，确保 CPUState 有足够的空间存储在栈中。接着，初始化 CPUState 结构体中的各个寄存器值：

(1) **通用寄存器 (eax, ebx, ecx, edx, esi, edi, ebp)**：设置为 0，以保证任务开始时寄存器处于已知状态。

(2) **程序计数器 (eip)**：设置为传入的 entry\_point，表示任务从这个地址开始执行。entry\_point 是任务的入口地址，通常指向任务的第一条指令。

(3) **代码段寄存器 (cs)**：设置为 0x8，这是一个常见的代码段选择子，通常在操作系统中表示内核代码段。

(4) **标志寄存器 (eflags)**：设置为 0x202。0x202 表示启用了中断 (IF 位为 1)，并且设置了固定的保留位。这样可以确保任务在开始执行时，CPU 的标志状态正确。

NULL\_TASK 是一个静态变量（代码 5.33），用作任务的默认空白模板，表示未初始化或未运

行的任务。它的 stack 被初始化为全零，cpu\_state\_ptr 为 0，running 标志设置为 false。NULL\_TASK 在创建新任务时作为基础，并通过调用 init 方法进行具体的初始化，避免每次创建任务时都手动构建任务结构体。

```
1 static NULL_TASK: Task = Task {
2     stack: [0; STACK_SIZE],
3     cpu_state_ptr: 0u32,
4     running: false,
5 }
```

代码 5.33 NULL\_TASK 静态变量

### C. TaskManager 数据结构

TaskManager 是一个管理操作系统中所有任务的结构体（代码 5.34）。它包含以下几个重要字段：

- (1) **tasks**: 一个固定大小的数组，用于存储系统中的所有任务。数组的大小由常量 MAX\_TASKS 决定，每个元素都是一个 Task 结构体实例。
- (2) **task\_count**: 记录当前运行的任务数量。
- (3) **current\_task**: 表示当前正在执行的任务的索引。使用 -1 表示没有当前任务或系统处于空闲状态。

TaskManager 的方法实现如下：

- (1) **init**: 添加一个空闲任务 (idle)，以确保即使没有其他任务运行，系统也有一个默认任务执行。idle 函数是一个空闲任务，在一个无限循环中执行 hlt 指令，以使 CPU 进入低功耗状态，等待外部中断恢复。
- (2) **add\_task**: 向任务管理器中添加一个新任务。它会先找到一个空闲的任务槽（未使用的 Task 实例），然后初始化该任务，并将 task\_count 计数器加 1。
- (3) **remove\_task**: 删除指定 ID 的任务。该方法会重置任务槽为 NULL\_TASK，并减少任务数量。
- (4) **remove\_current\_task**: 删除当前正在运行的任务。它调用 remove\_task 方法，并传入当前任务的 ID。
- (5) **schedule** 和 **get\_next\_task**: 通过轮转调度算法进行任务调度。保存当前任务的 CPU 状态，然后选择下一个任务并返回其 CPU 状态指针，具体实现见子节 5.3.2，在此不再赘述。
- (6) **get\_free\_slot**: 找到第一个空闲任务槽的索引。如果没有找到空闲槽，返回 -1。
- (7) **get\_current\_slot**: 返回当前正在运行的任务的索引。
- (8) **list\_tasks**: 列出所有正在运行的任务的 PID。

TASK\_MANAGER 静态变量（代码 5.35）是整个操作系统中唯一的 TaskManager 实例，用于全局管理所有的任务。它的作用包括：

- (1) **任务管理的核心控制**: 跟踪和调度系统中的所有任务。它持有任务列表 (tasks)，管理当前

```

1  pub struct TaskManager {
2      pub(crate) tasks: [Task; MAX_TASKS as usize],
3      task_count: i8,
4      current_task: i8,
5  }
6
7  impl TaskManager {
8      pub fn init(&mut self) {
9          self.add_task(idle as u32);
10     }
11
12     pub fn add_task(&mut self, entry_point: u32) {
13         self.tasks[self.get_free_slot() as usize].init(entry_point);
14         self.task_count += 1;
15     }
16
17     pub fn remove_task(&mut self, id: usize) {
18         if id != 0 {
19             self.tasks[id] = NULL_TASK;
20             self.task_count -= 1;
21         }
22     }
23
24     pub fn remove_current_task(&mut self) {
25         self.remove_task(self.current_task as usize);
26     }
27
28     pub fn schedule(&mut self, cpu_state: *mut CPUState) -> *mut CPUState { ... }
29
30     pub fn get_next_task(&self) -> i8 { ... }
31
32     pub fn get_free_slot(&self) -> i8 {
33         let mut slot: i8 = -1;
34         for i in 0..MAX_TASKS {
35             if self.tasks[i as usize].running == false {
36                 slot = i;
37                 return slot;
38             }
39         }
40         slot
41     }
42
43     pub fn get_current_slot(&self) -> i8 {
44         self.current_task
45     }
46
47     pub fn list_tasks(&self) {
48         lib::println!("Running tasks:");
49         for i in 0..MAX_TASKS {
50             if self.tasks[i as usize].running {
51                 lib::println!("PID: {}", i);
52             }
53         }
54     }
55 }
```

代码 5.34 TaskManager 数据结构

```
1 pub static mut TASK_MANAGER: TaskManager = TaskManager {  
2     tasks: [NULL_TASK; MAX_TASKS as usize],  
3     task_count: 0,  
4     current_task: -1,  
5 }
```

代码 5.35 TASK\_MANAGER 静态变量

任务的状态 (current\_task)，并记录系统中总共存在的任务数量 (task\_count)。

(2) **提供全局访问**: 系统中各个部分可以方便地访问这个任务管理器，这对于任务调度、任务创建、删除、切换等操作非常重要。

(3) **任务初始化和调度**: 在系统启动时初始化默认任务 (例如 idle 任务)，并在系统运行过程中负责调度各个任务。这使得操作系统能够在多任务环境下正确地进行任务切换，确保系统的稳定运行。

(4) **持久存在的状态管理**: 作为静态变量，TASK\_MANAGER 的生命周期贯穿整个系统的运行期，它的状态在整个操作系统运行过程中是持久的。这意味着任务的状态不会在函数调用结束时丢失，可以在任意时间点被访问和修改。

## 5.4 系统调用 (System Calls)

### 5.4.1 系统调用概述

在 MinmusOS 中，kernel/src/syscalls/handler.rs 文件（代码 5.36）负责处理系统调用。系统调用是用户空间中的程序请求内核服务的一种方式。通过这个接口，内核可以建立一个安全模型，对某些进程可以执行的操作类型和范围进行限制，它是内核功能对用户空间的一种暴露。

在 MinmusOS 中，系统调用通过 `syscall()` 函数使用软件中断（中断向量 0x80）来触发。这种机制类似于 Linux 在 x86 架构上使用的方法。

目前，MinmusOS 只支持两个系统调用：

- (1) **系统调用 0**: 打印由 EBX 寄存器指向的地址开始的字符串，长度由 ECX 寄存器指定。
- (2) **系统调用 1**: 从任务列表中移除当前运行的任务。

### 5.4.2 系统调用实现

以下是处理系统调用的关键步骤：

(1) **中断触发和保存寄存器**: 使用 `asm!` 宏定义的 `syscall()` 函数负责保存寄存器状态并调用 `syscall_handler()` 函数。这里 eax, ebx 和 ecx 寄存器被推入堆栈，并在调用完 `syscall_handler` 之后恢复，最后执行 `iretd` 从中断返回。在 `syscall()` 函数中，eax, ebx 和 ecx 寄存器的值被依次推入栈中，每个寄存器都是 32 位，即 4 字节。所以，总共推入了 12 字节。函数执行结束后，需要将这些值从栈上移除，以恢复调用前的栈状态。这就是 `add esp, 12` 的作用：它通过增加 esp 的值

来“弹出”这些值，即将栈指针向上移动 12 字节，回到调用 `syscall_handler` 函数之前的位置。

(2) 处理系统调用：`syscall_handler()` 函数接收来自 `eax`, `ebx` 和 `ecx` 寄存器的值作为参数，这些寄存器在 `syscall()` 中被传递。`eax` 用于指定系统调用的类型，`ebx` 和 `ecx` 用于传递额外的参数或数据。

(3) 系统调用的分发：通过检查 `eax` 寄存器的值来确定要执行的具体系统调用。

- ① 系统调用 0：打印字符串。通过 `ebx` 寄存器传递的地址读取字符串，`ecx` 指定字符串的长度。使用 Rust 的 `slice::from_raw_parts()` 和 `str::from_utf8()` 将字节转换为字符串。
- ② 系统调用 1：移除当前活动任务，通过调用任务管理器的 `remove_current_task()` 方法来实现。

(4) 结束中断处理：调用 `PICS.end_interrupt()` 方法，标志着对此次中断的处理已经完成，可以继续执行其他操作。

```

1 #[naked]
2 pub extern "C" fn syscall() {
3     unsafe {
4         asm!(
5             "push eax",
6             "push ebx",
7             "push ecx",
8             "call syscall_handler",
9             "add esp, 12",
10            "iretd",
11            options(noreturn),
12        );
13    }
14 }
15
16 #[no_mangle]
17 pub extern "C" fn syscall_handler(ecx: u32, ebx: u32, eax: u32) {
18     unsafe {
19         match eax {
20             0 => {
21                 let s = {
22                     let slice = slice::from_raw_parts(ebx as *const u8, ecx as usize);
23                     str::from_utf8(slice)
24                 };
25                 print::PRINTER.prints(s.unwrap());
26             }
27             1 => {
28                 TASK_MANAGER.remove_current_task();
29             }
30             _ => {}
31         }
32         PICS.end_interrupt(SYSCALL_INT);
33     }
34 }
```

代码 5.36 kernel/src/syscalls/handler.rs

通过这种方式，MinmusOS 能够提供一种有效且安全的机制，允许用户程序执行核心级的操作，如 IO 操作和任务管理。

## 5.5 VGA 文本模式 (VGA Text Mode)

VGA 文本模式是 VGA 标准的一种模式，专门用于显示标准的文本字符而非图形。这种模式通常用于计算机启动过程中和一些基本的用户界面，因为它可以提供快速且低消耗的文本显示功能。

在 VGA 文本模式中，屏幕通常被分成一个个字符单元，每个字符单元可以显示一个字符。常见的配置是 80 列和 25 行，即屏幕上可以同时显示 2000 个字符。每个字符单元由一个字符码和一个属性码组成，字符码用来确定显示哪个字符，属性码定义了字符的前景色和背景色以及是否闪烁等属性。

VGA 文本模式支持 ASCII 字符集的扩展，包括英文字符和各种符号，以及一些特殊的图形符号，这使得它能够用来绘制简单的图形和边框。由于其简洁和高效的特性，VGA 文本模式在早期的个人计算机和一些嵌入式系统中得到了广泛使用。

### 5.5.1 标准色彩定义

在 VGA 文本模式中，标准色彩定义为前景色和背景色的组合，从而在文本显示中实现不同的视觉效果。如代码 5.37 所示，我们定义了一系列的颜色常量，包括基本的黑色、蓝色、绿色、红色等，以及它们的“亮”版本，如亮黑色、亮蓝色等。这些颜色常量使用一个无符号 8 位整数 (u8) 表示，其中每个颜色对应一个唯一的数值，例如黑色为 0，蓝色为 1 等。

这些颜色可以在 VGA 文本模式下用于设置字符的前景色和背景色。在设置字符属性时，前景色和背景色的值可以通过位操作组合在一起，其中背景色占高四位，前景色占低四位。这种色彩配置方法允许快速和简单地调整文本的视觉样式，是文本模式下高效显示的关键。

```
1 pub const COLOR_BLACK: u8 = 0x0;
2 pub const COLOR_BLUE: u8 = 0x1;
3 pub const COLOR_GREEN: u8 = 0x2;
4 pub const COLOR_CYAN: u8 = 0x3;
5 pub const COLOR_RED: u8 = 0x4;
6 pub const COLOR_MAGENTA: u8 = 0x5;
7 pub const COLOR_YELLOW: u8 = 0x6;
8 pub const COLOR_WHITE: u8 = 0x7;
9 pub const COLOR_LIGHT_BLACK: u8 = 0x8;
10 pub const COLOR_LIGHT_BLUE: u8 = 0x9;
11 pub const COLOR_LIGHT_GREEN: u8 = 0xA;
12 pub const COLOR_LIGHT_CYAN: u8 = 0xB;
13 pub const COLOR_LIGHT_RED: u8 = 0xC;
14 pub const COLOR_LIGHT_MAGENTA: u8 = 0xD;
15 pub const COLOR_LIGHT_YELLOW: u8 = 0xE;
16 pub const COLOR_LIGHT_WHITE: u8 = 0xF;
```

代码 5.37 标准色彩定义

### 5.5.2 printc 方法

printc 方法（代码 5.38）是 VGA 文本模式中用于在屏幕上打印单个字符的函数。该方法接受一个字符参数，并根据当前光标位置将字符显示在屏幕上。在 VGA 文本模式中，屏幕被视为一个由多个字符单元组成的数组，每个单元可以显示一个字符和相应的颜色属性。

该方法首先检查传入的字符是否为换行符。如果是，它将调用 new\_line 方法来移动到下一行。如果不是换行符，方法将计算当前字符应该放置的内存位置。这个位置由 VGA\_START (VGA 内存的起始地址) 加上基于当前行 (self.y) 和列 (self.x) 的偏移量计算得出。每个字符单元占用两个字节：第一个字节用于存储字符的 ASCII 码，第二个字节用于存储该字符的颜色属性（由前景色和背景色组成）。

方法中还包含了对屏幕边界的检查。如果字符位置达到屏幕的最右侧或最下方，它会通过调用 scroll 方法来滚动屏幕内容，为新字符腾出空间。每打印一个字符后，方法会通过调用 set\_cursor\_position 更新光标位置，确保下一个字符能够被正确打印。

```

1 pub fn printc(&mut self, c: char) {
2     if c == '\n' {
3         self.new_line();
4         return;
5     }
6     let target: *mut u8 = (VGA_START + ((self.y * VGA_WIDTH + self.x) * 2) as u32) as *mut u8;
7     unsafe {
8         if self.y >= VGA_HEIGHT - 1 && self.x >= VGA_WIDTH - 1 {
9             *target = c as u8;
10            *target.byte_add(1) = self.bg_color << 4 | self_fg_color;
11            self.scroll();
12            self.x = 0;
13        } else {
14            *target = c as u8;
15            *target.byte_add(1) = self.bg_color << 4 | self_fg_color;
16            self.x += 1;
17            if self.x >= VGA_WIDTH {
18                self.x = 0;
19                self.y += 1;
20            }
21        }
22    }
23    self.set_cursor_position();
24 }
```

代码 5.38 printc 方法

### 5.5.3 prints 方法

prints 方法（代码 5.39）用于在 VGA 文本模式中打印一个字符串。此方法接受一个字符串参数，并逐字符调用 printc 方法将字符串的每个字符输出到屏幕上。

首先，此方法通过调用 get\_cursor\_position 获取当前光标的位置，确保字符串从正确的位置开

始打印。光标位置以 (u16, u16) 格式返回，表示当前光标的行 (y) 和列 (x)。获取到这个位置后，方法将此位置赋值给内部的 x 和 y 属性，从而准备字符打印的起始点。

接着，方法遍历字符串中的每一个字符，并逐个调用 printc 方法进行打印。由于 printc 方法已经内置了处理换行和屏幕滚动的逻辑，prints 方法可以无缝地继续打印字符串，即使内容跨越多行。

在字符串的所有字符都被打印后，prints 方法会再次调用 set\_cursor\_position 以更新屏幕的光标位置，确保后续的输出能够从正确的位置开始。通过这种方式，prints 方法提供了一种高效且方便的方式来在 VGA 文本模式下输出完整的字符串。

```
1 pub fn prints(&mut self, s: &str) {
2     let cursor: (u16, u16) = self.get_cursor_position();
3     self.x = cursor.0;
4     self.y = cursor.1;
5     for c in s.chars() {
6         self.printc(c);
7     }
8     self.set_cursor_position();
9 }
```

代码 5.39 prints 方法

#### 5.5.4 delete 方法

delete 方法（代码 5.40）在 VGA 文本模式中的实现允许用户擦除屏幕上最后一个显示的字符。此方法通过检查当前光标位置并适当调整，以定位需要擦除的字符。如果光标不在行首，它将向左移动一个位置；如果在行首但非屏幕顶端，则移动到上一行的末尾。接着，方法在计算出的地址位置写入空格字符，实际上是用空格覆盖了原字符，同时保持字符的颜色属性不变。这个过程不仅包括了字符的视觉擦除，也正确地更新了光标的位置，确保了继续输入或进一步删除的准确性。

```
1 pub fn delete(&mut self) {
2     if self.x > 0 {
3         self.x -= 1;
4     } else if self.y > 0 {
5         self.y -= 1;
6         self.x = VGA_WIDTH - 1;
7     } else {
8         return;
9     }
10    let target: *mut u8 = (VGA_START + ((self.y * VGA_WIDTH + self.x) * 2) as u32) as *mut u8;
11    unsafe {
12        *target = b' ' as u8;
13        *target.byte_add(1) = self.bg_color << 4 | self_fg_color;
14    }
15    self.set_cursor_position();
16 }
```

代码 5.40 delete 方法

### 5.5.5 get\_cursor\_position 方法

`get_cursor_position` 方法（代码 5.41）在 VGA 文本模式中用于获取当前光标的位置。此方法是理解和操作光标在屏幕上的位置的关键，它返回一个包含两个元素的元组 (`u16, u16`)，分别代表光标的列 (x 坐标) 和行 (y 坐标)。

该方法的实现涉及到对硬件端口的直接操作，以查询视频硬件当前的光标位置。具体步骤如下：

- (1) **初始化变量**：方法开始时，初始化一个用于存储光标位置索引的变量 `index`。
- (2) **读取光标低字节**：通过向 VGA 控制寄存器（端口 0x3D4）写入命令 0x0F，请求返回光标位置的低字节。随后从数据寄存器（端口 0x3D5）读取这个字节并存入 `index`。
- (3) **读取光标高字节**：再次向控制寄存器写入命令 0x0E，请求返回光标位置的高字节。读取该字节后，将其左移 8 位并与 `index` 合并，从而得到完整的光标位置索引。
- (4) **计算坐标**：使用 `index` 值计算光标的行和列坐标。光标的列坐标是 `index % VGA_WIDTH` (`index` 与屏幕宽度的余数)，行坐标是 `index / VGA_WIDTH` (`index` 除以屏幕宽度的商)。

通过这种方法，`get_cursor_position` 函数能够准确地从底层硬件中检索光标的当前位置，为屏幕上的文本操作提供必要的坐标信息。

```

1 pub fn get_cursor_position(&self) -> (u16, u16) {
2     let mut index: u16 = 0;
3     unsafe {
4         asm!("out dx, al", in("dx") 0x3D416>, in("al") 0x0Fu8);
5         let mut a: u8;
6         asm!("in al, dx", out("al") a, in("dx") 0x3D5);
7         index |= a as u16;
8         asm!("out dx, al", in("dx") 0x3D416>, in("al") 0x0Eu8);
9         let b: u8;
10        asm!("in al, dx", out("al") b, in("dx") 0x3D5);
11        index |= (b as u16) << 8;
12    }
13    (index % VGA_WIDTH, index / VGA_WIDTH)
14 }
```

代码 5.41 `get_cursor_position` 方法

### 5.5.6 set\_cursor\_position 方法

`set_cursor_position` 方法（代码 5.42）在 VGA 文本模式中用于设置光标的位置。此方法是调整屏幕上光标位置的关键函数，确保文本输入和编辑能够在正确的屏幕位置进行。它通过直接与硬件通信，将光标定位到由内部状态 (`self.x` 和 `self.y`) 指定的新位置。

该方法的执行过程涉及以下步骤：

- (1) **计算光标索引**：首先，根据光标的行 (`self.y`) 和列 (`self.x`) 坐标，计算光标在 VGA 文本缓冲区中的线性位置索引。这是通过表达式 `self.y * VGA_WIDTH + self.x` 完成的，其中 `VGA_WIDTH` 是屏幕的宽度（通常为 80 个字符）。

(2) **设置光标低字节**: 使用汇编指令向 VGA 控制寄存器 (端口 0x3D4) 写入光标位置低字节的命令 0x0F, 然后通过数据寄存器 (端口 0x3D5) 设置光标位置的低字节, 即 `index & 0xFF` (索引的低 8 位)。

(3) **设置光标高字节**: 继续使用汇编指令向 VGA 控制寄存器写入光标位置高字节的命令 0x0E, 再通过数据寄存器设置光标位置的高字节, 即 `((index >> 8) & 0xFF)` (索引的高 8 位)。

通过这种方法, `set_cursor_position` 函数能够精确地将光标移动到所需的屏幕位置, 这对于实现屏幕文本的正确渲染和用户交互的流畅性是至关重要的。

```
1 pub fn set_cursor_position(&self) {
2     let index: u16 = self.y * VGA_WIDTH + self.x;
3     unsafe {
4         asm!("out dx, al", in("dx") 0x3D4u16, in("al") 0x0Fu8);
5         asm!("out dx, al", in("dx") 0x3D5u16, in("al") (index & 0xFF) as u8);
6         asm!("out dx, al", in("dx") 0x3D4u16, in("al") 0x0Eu8);
7         asm!("out dx, al", in("dx") 0x3D5u16, in("al") ((index >> 8) & 0xFF) as u8);
8     }
9 }
```

代码 5.42 `set_cursor_position` 方法

### 5.5.7 scroll 方法

`scroll` 方法 (代码 5.43) 在 VGA 文本模式中用于滚动屏幕内容, 使得当文本到达屏幕底部时可以继续显示新的文本。这个方法是文本模式下管理屏幕输出的重要组成部分, 尤其是在处理长文本或连续输出时非常有用。

该方法的基本逻辑如下:

(1) **内存复制**: 方法首先遍历屏幕上的每一行。对于每一行, 它计算当前行在 VGA 内存中的起始地址, 并将其上一行的内容复制到当前行的位置。这种操作实质上是将屏幕上所有行的内容向上移动一行。

(2) **处理最后一行**: 由于最顶部的一行被移出屏幕, 最底部的新行需要被清空或用新的内容填充。在此实现中, 新的行通常会被清空或设置为默认的背景和前景色, 以等待新的文本输入。

(3) **内存操作安全性**: 所有内存操作都是在 `unsafe` 块中执行的, 因为它们涉及直接的指针操作。这是必要的, 因为 VGA 文本模式直接与硬件相关的内存交互, 需要确保操作的正确性和内存安全。

通过这种方式, `scroll` 方法有效地管理了屏幕空间, 确保即使在连续输出的情况下也能保持屏幕内容的连续性和可读性。

### 5.5.8 `set_colors` 方法和 `reset_colors` 方法

`set_colors` 方法 (代码 5.44) 用于设置 VGA 文本模式下字符的前景色和背景色。此方法接受两个参数: `fg_color` 和 `bg_color`, 分别代表前景色和背景色的颜色代码。调用此方法后, 所有后续打印

```

1 pub fn scroll(&mut self) {
2     for i in 0..VGA_HEIGHT {
3         for j in (VGA_WIDTH * i)..((VGA_WIDTH * i) + VGA_WIDTH) {
4             let new: *mut u8 = (VGA_START + (j * 2)) as *mut u8;
5             let old: *const u8 = (VGA_START + ((j + VGA_WIDTH) * 2)) as *const u8;
6             unsafe {
7                 *new = *old;
8                 *new.byte_add(1) = *old.byte_add(1);
9             }
10        }
11    }
12 }
```

代码 5.43 scroll 方法

到屏幕的字符都将使用这些颜色设置，直到再次更改颜色或重置颜色。这为用户提供了自定义屏幕文本外观的灵活性，非常适合需要强调或区分不同文本部分的场景。`reset_colors` 方法（代码 5.44）是 `set_colors` 方法的一个特例，它将前景色和背景色重置为默认值。

```

1 pub fn set_colors(&mut self, fg_color: u8, bg_color: u8) {
2     self.fg_color = fg_color;
3     self.bg_color = bg_color;
4 }
5
6 pub fn reset_colors(&mut self) {
7     self.set_colors(COLOR_WHITE, COLOR_BLACK)
8 }
```

代码 5.44 set\_colors 方法和 reset\_colors 方法

### 5.5.9 new\_line 方法

`new_line` 方法（代码 5.45）用于在 VGA 文本模式下移动光标到下一行的开始位置。如果光标已经在屏幕的最后一行，这个方法会触发 `scroll` 方法，将屏幕内容向上滚动一行，从而为新的一行腾出空间。这个方法简化了文本输出的处理，特别是在处理多行文本时，保证了文本的连续和整洁展示。

### 5.5.10 clear 方法

`clear` 方法（代码 5.46）用于清除 VGA 文本模式下的整个屏幕，设置所有字符单元为空格，并应用当前的颜色设置。此操作将光标重置到屏幕的左上角，为新的文本输出提供一个干净的环境。这个方法通常在程序开始运行时或者需要完全清除屏幕内容时使用，确保了屏幕内容的清晰和整洁。

图 5.47 展示了 VGA 文本模式。VGA 文本模式提供了一种低资源消耗且执行高效的方式，用于在早期计算机系统中进行基本文本显示和简单用户界面的构建。

```
1 pub fn new_line(&mut self) {
2     if self.y == VGA_HEIGHT - 1 {
3         self.scroll();
4     } else {
5         self.y += 1;
6     }
7     self.x = 0;
8     self.set_cursor_position();
9 }
```

代码 5.45 new\_line 方法

```
1 pub fn clear(&mut self) {
2     self.x = 0;
3     self.y = 0;
4     for i in 0..(VGA_WIDTH * VGA_HEIGHT) {
5         let target: *mut u8 = (VGA_START + (i * 2)) as *mut u8;
6         unsafe {
7             *target = b' ' as u8;
8             *target.byte_add(1) = self.bg_color << 4 | self_fg_color;
9         }
10    }
11    self.set_cursor_position();
12 }
```

代码 5.46 clear 方法

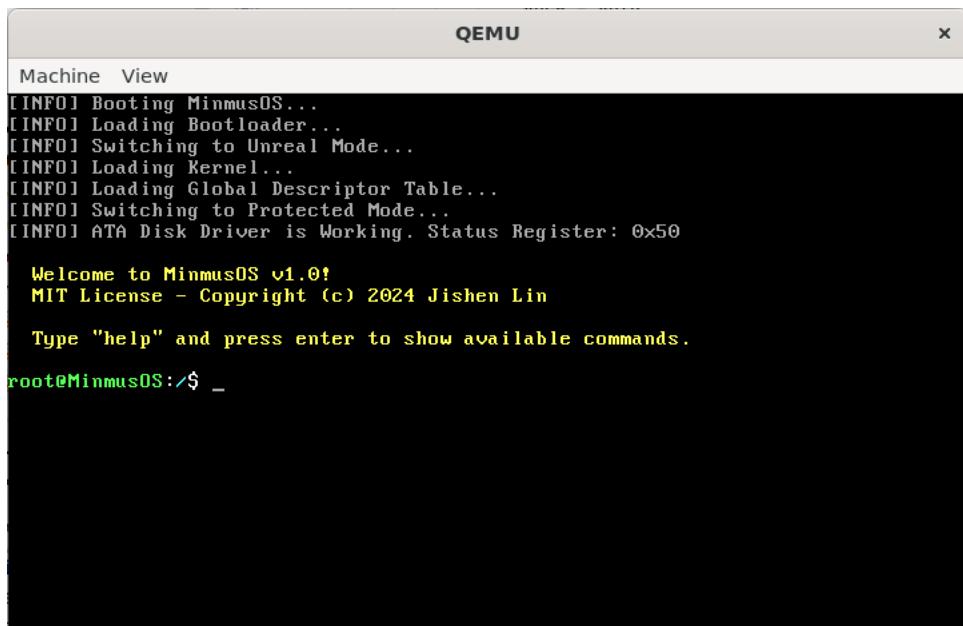


图 5.47 VGA 文本模式演示

## 5.6 内存管理 (Memory Management)

在 MinmusOS 中，内存管理由两个核心组件处理：内存分配器 (Memory Allocator) 和分页管理器 (Paging Manager)。kernel/src/memory/allocator.rs 定义了一个名为 Allocator 的内存分配器，它实现了 Rust 标准库的 GlobalAlloc 特质，提供了基本的内存分配和释放接口。kernel/src/memory/paging.rs 包含分页管理逻辑，定义了 PageDirectory 和 PageTable 结构，用于处理虚拟地址到物理地址的映射。这包括设置页表、启用分页机制，并提供标识映射的功能，即将一系列预定义的虚拟地址直接映射到相同的物理地址。这样的内存管理设计旨在支持操作系统的基础内存操作需求，为更复杂的内存管理特性和优化提供基础。

### 5.6.1 内存分配器 (Memory Allocator)

MinmusOS 的内存分配器设计相对简单，使用了单链表结构管理内存块。这种设计虽然易于实现和理解，但在复杂的场景下，可能会因为频繁的链表操作而导致性能瓶颈。因此，在未来版本中，可能需要考虑引入更高级的内存管理机制，如伙伴分配器或堆碎片整理策略，以提升内存分配的效率和鲁棒性。

#### A. Block 数据结构

Block 结构体（代码 5.48）表示内存分配器中的一个内存块，包含两个字段：size 表示内存块的大小，next 是一个 Option<NonNull<Block>> 类型的字段，指向下一个内存块。自动派生的 Copy 和 Clone 特质使得 Block 结构体可以按值进行复制，Debug 特质允许对其进行调试输出。

```
1 #[derive(Copy, Clone, Debug)]
2 struct Block {
3     size: usize,
4     next: Option<NonNull<Block>>,
5 }
```

代码 5.48 Block 数据结构

#### B. Allocator 数据结构

Allocator 结构体（代码 5.49）包含一个 AtomicPtr<Block> 类型的头指针 head，它指向一个内存块链表的起始位置。链表中的每个节点表示一个可用的内存块，其大小由 size 字段表示，next 字段指向下一个内存块。通过使用原子指针，Allocator 能够在多线程环境中安全地更新链表。

```
1 pub struct Allocator {
2     head: AtomicPtr<Block>,
3 }
```

代码 5.49 Allocator 数据结构

init 方法（代码 5.50）用于初始化内存分配器的堆空间。该方法接受堆的起始地址和大小，并

将堆视为一个大的空闲内存块。这个初始内存块被存储在 `head` 指针中，准备好供后续的分配操作使用。

```
1  impl Allocator {
2      pub const fn new() -> Self {
3          Allocator {
4              head: AtomicPtr::new(ptr::null_mut())
5          }
6      }
7
8      #[allow(dead_code)]
9      pub unsafe fn init(&self, heap_start: usize, heap_size: usize) {
10         let block = Block {
11             size: heap_size,
12             next: None,
13         };
14         let block_ptr: *mut Block = heap_start as *mut Block;
15         ptr::write(block_ptr, block);
16         self.head.store(block_ptr, Ordering::SeqCst);
17     }
18 }
```

代码 5.50 Allocator 数据结构实现

### 5.6.2 全局分配器接口 (GlobalAlloc Trait)

代码 5.51 实现了 GlobalAlloc 特质中的 `alloc` 方法。该方法接受一个 `Layout` 参数，表示所需内存的大小和对齐要求。分配器会遍历内存块链表，找到第一个满足要求的内存块并将其分配出去。如果找到合适的块，它将更新链表头指针，指向下一个空闲块。

```
1  unsafe fn alloc(&self, layout: Layout) -> *mut u8 {
2      let mut current: *mut Block = self.head.load(Ordering::SeqCst);
3      let alloc_size: usize = layout.size().max(layout.align());
4      while !current.is_null() {
5          let current_block: &mut Block = &mut *current;
6          if current_block.size >= alloc_size {
7              self.head.store(current_block.next.map_or(ptr::null_mut(), |b| b.as_ptr()), 
8                             Ordering::SeqCst);
8              return current as *mut u8;
9          }
10         current = current_block.next.map_or(ptr::null_mut(), |b| b.as_ptr());
11     }
12     ptr::null_mut()
13 }
```

代码 5.51 alloc 内存分配接口

代码 5.52 实现了 GlobalAlloc 特质中的 `dealloc` 方法，用于释放已分配的内存。`dealloc` 方法会将释放的内存块重新插入到链表的头部，从而使其能够被未来的内存分配请求重用。

这个内存分配器的实现为操作系统的基础内存管理提供了核心支持，允许操作系统在不同的场

```

1 unsafe fn deallocate(&self, ptr: *mut u8, layout: Layout) {
2     let mut new_block = Block {
3         size: layout.size(),
4         next: None,
5     };
6     new_block.next = NonNull::new(self.head.load(Ordering::SeqCst));
7     let new_block_ptr: *mut Block = ptr as *mut Block;
8     ptr.write(new_block_ptr, new_block);
9     self.head.store(new_block_ptr, Ordering::SeqCst);
10 }

```

代码 5.52 deallocate 内存释放接口

景中动态分配和释放内存资源，从而更好地支持应用程序的执行和资源管理。

### 5.6.3 分页管理器 (Paging Manager)

分页管理器 (Paging Manager) 负责操作系统中的分页内存管理机制，通过维护分页目录 (Page Directory) 和页表 (Page Table) 来管理虚拟内存与物理内存的映射关系。分页管理器的核心任务包括创建和初始化分页结构，设置页表项，并启用分页模式，使系统能够在启用分页后正确映射和管理内存。

#### A. 全局分页变量 (Global Paging Variables)

代码 5.53 定义了操作系统分页管理中使用的全局分页变量，包括分页目录 (Page Directory) 和页表 (Page Table)。这些变量在整个系统中是全局共享的，确保分页机制能够在各个模块中一致地运作。为了适应分页内存管理的需求，这些数据结构被静态定义，并且通过特定的内存对齐方式进行存储，以确保其高效且正确的访问。

全局变量 PAGING 定义了一个全局的分页目录，用于存储虚拟内存与物理内存的映射关系。分页目录包含 1024 个条目，每个条目对应一个页表指针。全局变量 TABLES 是一个包含 16 个页表的数组，这些页表用于进一步细化内存映射，每个页表也包含 1024 个条目，指向物理内存中的实际页。初始状态下，这些页表被定义为 NULL\_TABLE，即一个默认的空页表。

NULL\_TABLE 作为一个全局常量，被初始化为全零的页表，用于未被分配或尚未使用的页表项。这个设计确保了系统在初始状态下的安全性与稳定性，防止未初始化的页表带来的潜在问题。通过这些全局变量，操作系统能够高效地管理内存映射关系，为不同进程提供隔离和保护。

```

1 pub static mut PAGING: PageDirectory = PageDirectory { entries: [0x00000002; 1024] };
2 pub static mut TABLES: [PageTable; 16] = [NULL_TABLE; 16];
3 pub static NULL_TABLE: PageTable = PageTable { entries: [0; 1024] };

```

代码 5.53 全局分页变量

#### B. 分页目录 (Page Directory)

PageDirectory 数据结构 (代码 5.54) 使用了一个长度为 1024 的 entries 数组，每个数组项为 32

位无符号整数类型 (u32)，用来存储页表的物理地址和控制标志。数组中的每个条目对应着 4MB 的虚拟内存范围，其内容是页表的起始地址加上分页控制位。这种设计允许操作系统通过访问 `entries` 数组的不同索引来管理不同的内存段。

`PageDirectory` 提供了几个关键的方法来操作分页目录：

(1) **set\_table** 方法：该方法用于将一个页表的地址设置到分页目录中的指定索引位置。具体实现上，它接受一个页表指针，将其转换为对应的物理地址，并将该地址与必要的控制位（例如存在位和读写位）进行按位或运算，最终存储在 `entries` 数组的指定位置。这确保了该页表在分页系统中被正确识别和使用。

(2) **enable** 方法：该方法用于启用分页模式。它首先将分页目录的地址加载到控制寄存器 CR3 中，然后通过修改 CR0 寄存器的相应标志位来启用分页功能。这个操作是分页机制中至关重要的一步，允许操作系统从此时开始使用分页来管理内存。

(3) **identity** 方法：该方法用于进行内存的身份映射 (identity mapping)，将特定的物理地址范围直接映射到相同的虚拟地址范围。在实现上，它遍历 TABLES 数组的前八个页表，将每个页表初始化为对应的物理地址区间，并将这些页表加载到分页目录的前八个条目中。这种映射方式通常用于内核空间的地址映射，确保内核在启用分页后能够继续正常访问内存。

### C. 页表 (Page Table)

`PageTable` 数据结构由一个长度为 1024 的 `entries` 数组构成，每个数组项为 32 位无符号整数类型 (u32)，用于存储指向具体物理内存页的地址和相关的控制位。该数据结构被定义为带有 4096 字节对齐的结构体，确保页表在内存中的对齐要求与分页机制保持一致。这种对齐方式是分页系统中至关重要的一部分，有助于提高内存访问的效率和准确性。

在 `PageTable` 结构体 (代码 5.55) 中，每个 `entries` 数组元素对应一个 4KB 的内存页，因此整个 `PageTable` 能够管理 4MB 的虚拟内存范围。每个数组项存储的内容包括物理页的起始地址以及一些控制位，这些控制位用于指定页面是否存在、可读写、用户权限等。

`PageTable` 提供了 `set` 方法。`set` 方法用于初始化页表中的所有条目，将它们映射到一个从指定起始地址开始的物理内存范围。方法接受一个 `from` 参数，表示物理地址的起始点。方法通过循环遍历 `entries` 数组，将每个条目设置为一个物理页的地址和相关的控制位。具体地，物理地址的计算是通过将数组索引乘以页面大小 (4KB) 并加上 `from` 参数来实现的。控制位则通过按位或运算加入到地址中，通常包括页面的存在位和读写权限位。这种方式允许操作系统将连续的一段虚拟地址映射到连续的物理地址，提供了对内存的有效管理。

#### 5.6.4 分页目录 (Page Directory) 与页表 (Page Table)

图 5.56 展示了分页目录与页表的结构与联系。

分页目录是分页机制的第一层结构，用于管理整个虚拟内存空间的映射。一个分页目录包含 1024 个条目，每个条目指向一个页表。分页目录本身的每个条目管理着一个大的虚拟地址块，通常

```

1     #[repr(align(4096))]
2 pub struct PageDirectory {
3     pub entries: [u32; 1024],
4 }
5
6 impl PageDirectory {
7     pub fn set_table(&mut self, index: usize, table: &PageTable) {
8         self.entries[index] = (table as *const PageTable) as u32 | 0b011;
9     }
10
11    pub fn enable(&self) {
12        unsafe {
13            let address: u32 = (self as *const PageDirectory) as u32;
14            core::arch::asm!(
15                "mov cr3, eax",
16                "mov eax, cr0",
17                "or eax, 0x80000001",
18                "mov cr0, eax",
19                "in(\"eax\") address",
20                );
21        }
22    }
23
24    pub fn identity(&mut self) {
25        unsafe {
26            for i in 0..8 {
27                TABLES[i].set((0x00400000 * i) as u32);
28                PAGING.set_table(i, &TABLES[i]);
29            }
30        }
31    }
32 }

```

代码 5.54 PageDirectory 数据结构

```

1 #[derive(Copy, Clone, Debug)]
2 #[repr(align(4096))]
3 pub struct PageTable {
4     pub entries: [u32; 1024],
5 }
6
7 impl PageTable {
8     pub fn set(&mut self, from: u32) {
9         for i in 0..1024 {
10             self.entries[i] = (((i * 0x1000) + from as usize) | 0b011) as u32;
11         }
12     }
13 }

```

代码 5.55 PageTable 数据结构

是 4MB 的虚拟内存。因此，分页目录的主要作用是将虚拟地址空间划分成更小的段，并通过页表进一步细化这些段的地址映射。

页表是分页机制的第二层结构，用于细化虚拟地址到物理地址的映射。每个页表也包含 1024 个条目，每个条目对应一个 4KB 的物理内存页。因此，页表的作用是将从分页目录获得的虚拟地址范围进一步分解，精确映射到物理内存的具体页面。

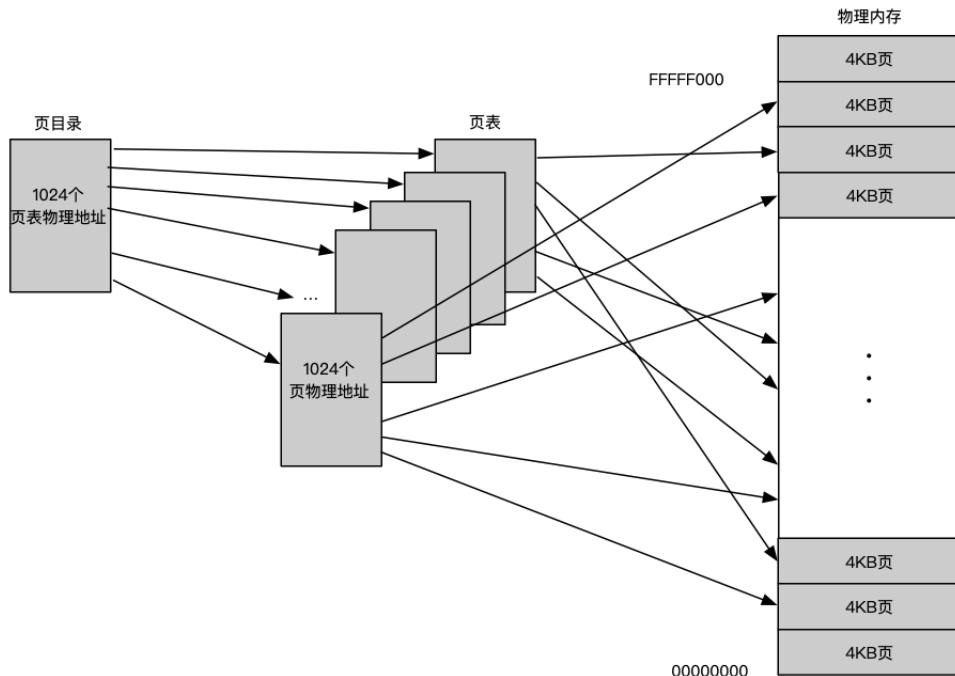


图 5.56 分页目录与页表

分页目录的主要功能是通过其条目指向不同的页表，从而管理大块虚拟地址空间。它起到了一个索引的作用，将虚拟地址的高位部分映射到对应的页表。页表的功能是将分页目录指向的虚拟地址块进一步细化，映射到具体的物理内存页。它完成了从虚拟地址到物理地址的最终转换。

分页管理器通过组合分页目录和页表，提供了一种灵活的内存管理机制，使得操作系统能够有效地管理和保护内存资源，支持多任务和内存隔离的实现。

## 5.7 文件系统 (File System)

### 5.7.1 FAT16 文件系统

FAT16 (File Allocation Table 16-bit, 16 位文件分配表) 是微软最早开发的文件系统之一，最早在 MS-DOS 操作系统中引入。FAT16 的设计目的是为了在早期的磁盘驱动器上高效地管理文件和目录。以下是 FAT16 文件系统的详细介绍：

(1) **文件分配表 (FAT)**：FAT 是文件系统的核心部分，用于记录每个文件在磁盘上的存储位置。FAT 表本质上是一个链表，每个文件由多个簇 (Cluster) 组成，而每个簇对应于 FAT 表中的一个条

目。FAT 表记录了每个簇的下一个簇的位置，文件的最后一个簇标记为 0xFFFF，表示文件结束。

① **16 位条目**: FAT16 使用 16 位条目来表示每个簇的索引，因此它最多可以管理  $2^{16} = 65536$  个簇。

② **簇链**: 文件在磁盘上的簇可能不是连续存储的，FAT 表将这些分散的簇链在一起，形成文件的完整数据。

(2) **簇 (Cluster)**: 簇是 FAT 文件系统中磁盘的基本分配单位。簇由多个扇区（通常为 512 字节）组成，文件在磁盘上的存储是以簇为单位进行分配的。簇的大小决定了文件系统的最大容量和性能。

① **簇大小**: 通常簇大小为 4KB、8KB 或更大，具体取决于磁盘的大小。较大的簇有助于减少 FAT 表的大小，但也可能导致磁盘空间浪费（特别是存储小文件时）。

(3) **文件和目录**: 在 FAT16 文件系统中，文件和目录的信息存储在根目录和子目录中。每个文件和目录的元数据保存在一个 32 字节的目录项中。

① **文件名**: FAT16 使用 8.3 格式（8 个字符的文件名加上 3 个字符的扩展名）来表示文件名。

② **目录结构**: 目录也是一种文件，它包含子目录和文件的列表。根目录有固定的大小，不能动态扩展。

③ **元数据**: 每个目录项包含文件或目录的名称、文件大小、起始簇号、创建时间、修改时间、属性（如只读、隐藏、系统文件等）等信息。

(4) **扇区 (Sector) 和磁盘布局**: 磁盘在 FAT16 文件系统中按照一定的布局分为多个区域，每个区域有特定的功能：

① **引导扇区 (Boot Sector)**: 磁盘的第一个扇区，包含引导程序和文件系统的基本信息，如每簇的扇区数、FAT 表的起始位置、根目录的起始位置等。

② **FAT 表区域**: 存放 FAT 表的数据，通常有两份 FAT 表用于冗余备份。

③ **根目录区域**: 存放根目录的目录项，FAT16 的根目录区大小固定，通常位于 FAT 表之后。

④ **数据区**: 存放实际文件和目录数据，由簇构成。

(5) **文件操作**: 在 FAT16 中，文件的操作包括创建、删除、读取和写入文件等。这些操作都是通过更新 FAT 表和目录项来实现的。

① **创建文件**: 在根目录或子目录中创建一个新的目录项，为文件分配簇，并在 FAT 表中记录簇链。

② **删除文件**: 标记目录项为删除，并释放 FAT 表中对应的簇。

③ **读取文件**: 根据目录项中的起始簇号，遍历 FAT 表获取文件的所有簇，并读取这些簇的数据。

④ **写入文件**: 找到文件的空闲簇，更新 FAT 表和目录项，将数据写入相应的簇。

(6) **限制和特点**: 由于使用 16 位的簇索引，FAT16 的最大卷大小为 2GB（如果使用 32KB 的

簇)，且文件名必须遵循 8.3 格式，这对文件命名有一定的限制。此外 FAT16 没有内建的文件权限管理功能，不支持复杂的安全策略。

### 5.7.2 FAT16 全局常量定义

代码 5.57 在 FAT16 文件系统实现中分别定义了以下内容：

- (1) `ENTRY_COUNT`: 定义了根目录中最多可以容纳的目录项数量为 512，用于限制根目录的文件和子目录数量。
- (2) `FAT_START`: 指定了文件分配表 (FAT) 在磁盘上的起始位置 (逻辑块地址 LBA)，其值为 36864，用于定位 FAT 表在磁盘上的位置。
- (3) `FAT_SIZE`: 定义了 FAT 表的大小为 256，表示 FAT 表中可以存储的簇链条目数量，用于跟踪文件在磁盘上的簇链。

```
1 const ENTRY_COUNT: usize = 512;
2 const FAT_START: u16 = 36864;
3 const FAT_SIZE: usize = 256;
```

代码 5.57 FAT16 全局常量定义

### 5.7.3 Header 数据结构

Header 结构体 (代码 5.58) 表示 FAT16 文件系统的引导扇区信息，用于存储文件系统的元数据，包括引导加载器代码、文件系统参数 (如扇区大小、簇大小、FAT 表数量等)、卷标信息和保留区域。这个结构体在加载 FAT16 文件系统时至关重要，因为它提供了文件系统布局的基本信息，帮助操作系统正确解析和操作磁盘上的数据。

`NULL_HEADER` 是一个用于初始化 Header 结构体的静态变量，其作用是提供一个默认的、空的 FAT16 文件系统头部信息。当创建或初始化一个 Header 变量时，如果尚未从磁盘加载实际的文件系统信息，可以使用 `NULL_HEADER` 作为占位符。它确保在没有加载有效数据之前，Header 结构体中的字段都被安全地设置为默认值 (通常是零或空值)，防止意外访问未初始化的数据。

### 5.7.4 Entry 数据结构

Entry 结构体 (代码 5.59) 表示 FAT16 文件系统中的目录项，用于存储文件或目录的元数据。每个目录项包含文件的名称、属性、创建和修改时间、文件的起始簇号以及文件大小等信息。这个结构体是 FAT16 文件系统管理文件和目录的基础单元，它通过这些元数据来描述文件在磁盘上的位置和相关属性，从而使操作系统能够正确地访问和管理文件。

`NULL_ENTRY` 是一个用于初始化 Entry 结构体的静态变量，其作用是提供一个默认的、空的目录项。当创建或初始化一个 Entry 变量时，如果尚未加载实际的文件或目录信息，可以使用

```

1 #[derive(Copy, Clone, Debug)]
2 #[repr(C, packed)]
3 pub struct Header {
4     boot_jump_instructions: [u8; 3], // 引导跳转指令, 用于启动时跳转到引导程序
5     oem_identifier: [u8; 8], // OEM 标识符, 标识文件系统创建的操作系统或工具
6     bytes_per_sector: u16, // 每扇区字节数, 通常为 512 字节
7     sectors_per_cluster: u8, // 每簇包含的扇区数, 决定了簇的大小
8     reserved_sectors: u16, // 保留扇区数, 通常包括引导扇区和其他保留数据
9     fat_count: u8, // FAT 表的数量, 通常为 2, 用于冗余备份
10    dir_entries_count: u16, // 根目录的目录项数量, 表示根目录最多可以容纳的文件和目录
11    total_sectors: u16, // 总扇区数, 如果为 0, 则使用 large_sector_count 字段
12    media_descriptor_type: u8, // 媒体描述符类型, 标识存储设备的类型
13    sectors_per_fat: u16, // 每个 FAT 表占用的扇区数
14    sectors_per_track: u16, // 每个磁道的扇区数, 用于 CHS (柱面-磁头-扇区) 寻址模式
15    heads: u16, // 磁头数, 用于 CHS 寻址模式
16    hidden_sectors: u32, // 隐藏扇区数, 表示从磁盘起始到分区起始之间的扇区数
17    large_sector_count: u32, // 总扇区数 (如果 total_sectors 为 0), 用于支持大于 32MB
18    drive_number: u8, // 驱动器号, 通常为 0x80 表示硬盘
19    reserved: u8, // 保留字段, 通常为 0, 未使用
20    signature: u8, // 扩展引导签名, 用于验证引导扇区是否有效
21    volume_id: u32, // 卷序列号, 用于唯一标识卷
22    volume_label: [u8; 11], // 卷标, 用于显示卷的名称
23    system_id: [u8; 8], // 文件系统类型标识符, 如 "FAT16"
24    zero: [u8; 460], // 填充字段, 通常为 0, 用于对齐和保留空间
25 }

```

代码 5.58 Header 数据结构

```

1 #[derive(Copy, Clone, Debug)]
2 #[repr(C, packed)]
3 pub struct Entry {
4     pub name: [u8; 11], // 文件名, 采用 8.3 格式 (8 个字符的文件名 + 3 个字符的扩展名)
5     attributes: u8, // 文件属性, 如只读、隐藏、系统文件等
6     reserved: u8, // 保留字段, 未使用
7     created_time_tenths: u8, // 文件创建时间的十分之一秒, 用于提高时间精度
8     created_time: u16, // 文件创建时间 (小时、分钟、秒)
9     created_date: u16, // 文件创建日期 (年、月、日)
10    accessed_date: u16, // 最近访问日期
11    first_cluster_high: u16, // 文件的起始簇号高 16 位 (仅用于 FAT32, FAT16 中未使用)
12    modified_time: u16, // 最近修改时间 (小时、分钟、秒)
13    modified_date: u16, // 最近修改日期 (年、月、日)
14    first_cluster_low: u16, // 文件的起始簇号低 16 位, 指向文件内容在数据区的起始位置
15    size: u32, // 文件大小, 以字节为单位
16 }

```

代码 5.59 Entry 数据结构

NULL\_ENTRY 作为占位符。它的存在确保在遍历或操作目录项时，如果遇到无效或空的目录项，能够安全地返回 NULL\_ENTRY，从而避免处理无效数据，保持程序的稳定性和安全性。

### 5.7.5 FatDriver 数据结构

FatDriver 结构体（代码 5.60）是 FAT16 文件系统驱动程序的核心数据结构，设计用于管理和操作文件系统。它包含四个关键部分：header 存储 FAT16 文件系统的引导扇区信息，entries 存储根目录中的所有目录项，table 存储文件分配表（FAT），buffer 用于临时存储从磁盘读取的文件数据。通过这种设计，FatDriver 能够全面管理 FAT16 文件系统的基本操作，包括文件和目录的读取、写入以及簇链的管理。

```
1 #[derive(Copy, Clone, Debug)]
2 pub struct FatDriver {
3     pub header: Header,
4     pub entries: [Entry; ENTRY_COUNT],
5     pub table: [u16; FAT_SIZE],
6     pub buffer: [u8; 2048],
7 }
```

代码 5.60 FatDriver 数据结构

FAT 是一个全局静态变量（代码 5.61），使用 Mutex 保护 FatDriver 实例，以确保线程安全。它在程序启动时被初始化，将 header、entries 和 table 等结构体字段设置为默认的空值或零值，防止未初始化的访问。在实际操作过程中，FAT 作为整个 FAT16 文件系统的入口点，负责管理和调度文件系统的基本操作。

```
1 pub static mut FAT: Mutex<FatDriver> = Mutex::new(FatDriver {
2     header: NULL_HEADER,
3     entries: [NULL_ENTRY; ENTRY_COUNT],
4     table: [0; FAT_SIZE],
5     buffer: [0; 2048],
6 });
```

代码 5.61 FAT 全局静态变量

### 5.7.6 FAT16 文件操作

#### A. 加载文件系统头部信息

load\_header 方法（代码 5.62）用于从磁盘的指定位置（由 FAT\_START 和 reserved\_sectors 决定）加载 FAT16 文件系统的头部信息（Header 结构体）到 FatDriver 的 header 字段中。这个操作是文件系统初始化的第一步，获取文件系统的基本参数如每扇区字节数、每簇扇区数等，为后续文件操作提供必要的元数据。

```

1 pub fn load_header(&mut self) {
2     let target: *mut Header = &mut self.header as *mut Header;
3     let lba: u64 = FAT_START as u64;
4     let sectors: u16 = 1;
5     unsafe { DISK.read(target, lba, sectors); }
6 }
```

代码 5.62 load\_header 方法

### B. 加载根目录的目录项

load\_entries 方法（代码 5.63）从磁盘中加载根目录的所有目录项到 entries 数组中。通过计算目录项的起始逻辑块地址（LBA）和所需的扇区数，该方法将所有目录项数据从磁盘读取到内存中，以便后续对文件和目录的操作。根目录的目录项包含文件和子目录的元数据。

```

1 pub fn load_entries(&mut self) {
2     let target: *mut Entry = &mut self.entries as *mut Entry;
3     let entry_size: u16 = size_of::<Entry>() as u16;
4     let lba: u64 = FAT_START as u64 + (self.header.reserved_sectors +
5         self.header.sectors_per_fat * self.header.fat_count as u16) as u64;
6     let size: u16 = entry_size * self.header.dir_entries_count;
7     let sectors: u16 = size / self.header.bytes_per_sector;
8     unsafe { DISK.read(target, lba, sectors); }
9 }
```

代码 5.63 load\_entries 方法

### C. 列出根目录中的文件和目录

list\_entries 方法（代码 5.64）遍历 entries 数组，并列出其中所有有效的目录项（文件或目录），打印出每个文件的名称和大小。这一方法用于展示根目录中的内容，帮助用户或系统查看文件系统中存在哪些文件和目录。

```

1 pub fn list_entries(&self) {
2     lib::println!("Filename      Size");
3     for i in 0..ENTRY_COUNT {
4         if self.entries[i].name[0] != 0 {
5             for c in self.entries[i].name { lib::print!("{}", c as char); }
6             let size = self.entries[i].size;
7             lib::println!("  {} bytes", size);
8         }
9     }
10 }
```

代码 5.64 list\_entries 方法

### D. 加载文件分配表（FAT 表）

load\_table 方法（代码 5.65）从磁盘加载文件分配表（FAT）到 table 数组中。FAT 表记录了文件在磁盘上的簇链信息，表示文件的各个部分（簇）在磁盘中的物理位置。这个方法是文件读取和写

入操作的前提，通过 FAT 表可以找到文件的所有数据块。

```
1 pub fn load_table(&mut self) {
2     let target: *mut u16 = &mut self.table as *mut u16;
3     let lba: u64 = FAT_START as u64 + self.header.reserved_sectors as u64;
4     let sectors: u16 = 1;
5     unsafe { DISK.read(target, lba, sectors); }
6 }
```

代码 5.65 load\_table 方法

#### E. 将文件内容读取到缓冲区

read\_file\_to\_buffer 方法（代码 5.66）用于将指定文件的内容从磁盘读取到 buffer 缓冲区中。它通过计算文件数据的起始 LBA，并使用 FAT 表找到文件的所有簇链，读取文件的内容到内存中的缓冲区中。这个方法适用于读取小文件或进行临时处理的场景。

```
1 pub fn read_file_to_buffer(&self, entry: &Entry) {
2     let target: *mut u8 = self.buffer.as_ptr() as *mut u8;
3     let data_lba: u64 = FAT_START as u64 + (self.header.reserved_sectors +
4         self.header.sectors_per_fat * self.header.fat_count as u16 + 32) as u64;
5     let lba: u64 = data_lba + ((entry.first_cluster_low - 2) * self.header.sectors_per_cluster
6         as u16) as u64;
7     let sectors: u16 = self.header.sectors_per_cluster as u16;
8     unsafe { DISK.read(target, lba, sectors); }
```

代码 5.66 read\_file\_to\_buffer 方法

#### F. 将文件内容读取到指定目标内存

read\_file\_to\_target 方法（代码 5.67）用于将文件的内容从磁盘读取到指定的目标内存位置。该方法通过遍历 FAT 表，逐簇读取文件内容，并将其写入目标内存地址，直到文件的最后一个簇。这种操作适用于需要将文件内容直接加载到指定内存区域的场景，如加载可执行文件。

#### G. 在目录中搜索文件

search\_file 方法（代码 5.68）在 entries 数组中搜索指定名称的文件或目录，并返回对应的目录项（Entry 结构体）。它逐个比较目录项中的文件名与目标名称，忽略大小写，并返回找到的文件或目录的元数据。如果未找到，返回 NULL\_ENTRY 作为占位符。这个方法用于查找文件系统中的具体文件或目录。

#### 5.7.7 示例文本文件

MinmusOS 提供了五个示例文本文件，加载到了根目录中：

- (1) **cargo**: Cargo 包管理器介绍。（图 5.69）
- (2) **license**: MinmusOS 项目 MIT 许可证文件。（图 5.70）

(3) **minmusos**: MinmusOS 欢迎文件与联系方式。(图 5.71)

(4) **rust**: Rust 语言介绍。(图 5.72)

(5) **thanks**: 致谢。(图 5.73)

可以在命令行解释器 (Shell) 中通过 cat 命令查看这些文件。

```

1 pub fn read_file_to_target(&self, entry: &Entry, target: *mut u32) {
2     let mut next_cluster: u16 = entry.first_cluster_low;
3     let mut current_target: *mut u32 = target;
4     loop {
5         let data_lba: u64 = FAT_START as u64 + (self.header.reserved_sectors +
6             self.header.sectors_per_fat * self.header.fat_count as u16 + 32) as u64;
7         let lba: u64 = data_lba + ((next_cluster - 2) * self.header.sectors_per_cluster as
8             u16) as u64;
9         let sectors: u16 = self.header.sectors_per_cluster as u16;
10        unsafe { DISK.read(current_target, lba, sectors); }
11        next_cluster = self.table[next_cluster as usize];
12        unsafe {
13            let cluster_size: u16 = self.header.sectors_per_cluster as u16 *
14                self.header.bytes_per_sector;
15            current_target = current_target.byte_add(cluster_size as usize);
16        }
17    }
18 }
```

代码 5.67 read\_file\_to\_target 方法

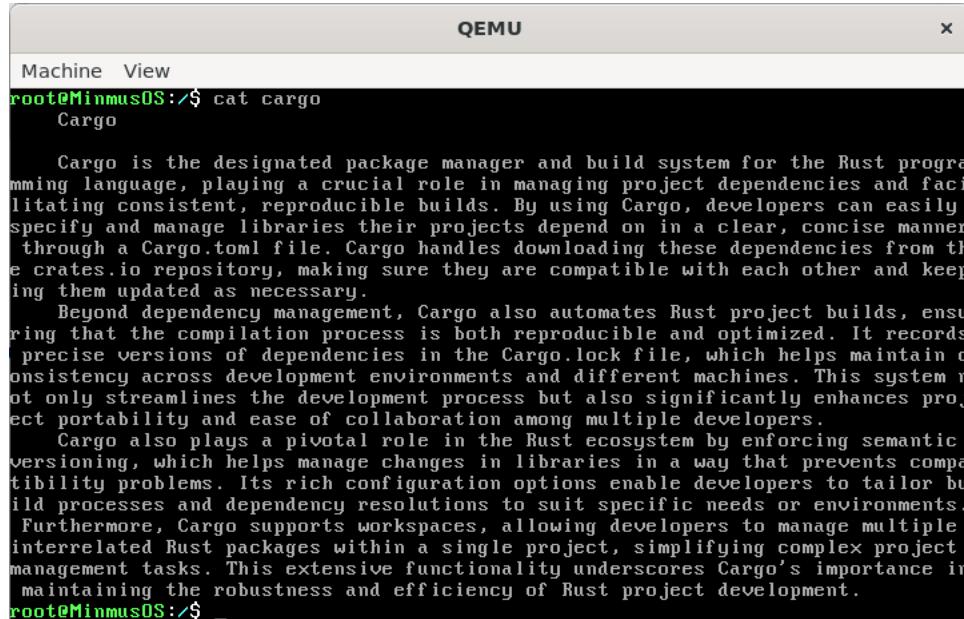
```

1 pub fn search_file(&self, name: &[char]) -> &Entry {
2     for entry in self.entries.iter() {
3         let mut found: bool = true;
4         let mut i: usize = 0;
5         for n in name {
6             let mut c: char = n.clone();
7             if c.is_ascii_lowercase() { c = c.to_ascii_uppercase(); }
8             if (c != entry.name[i] as char) && (name[i] != '\0') { found = false; }
9             i += 1;
10        }
11        if found {
12            return entry;
13        }
14    }
15    &NULL_ENTRY
16 }
```

代码 5.68 search\_file 方法

## 5.8 定时器 (Timer)

在操作系统内核中，定时器 (Timer) 是一种核心组件，用于测量和控制时间的流逝，支持任务调度、超时管理、和性能监测等关键功能。它可以基于硬件也可以基于软件实现，确保操作系统



```
QEMU
Machine View
root@MinmusOS:~$ cat cargo
Cargo

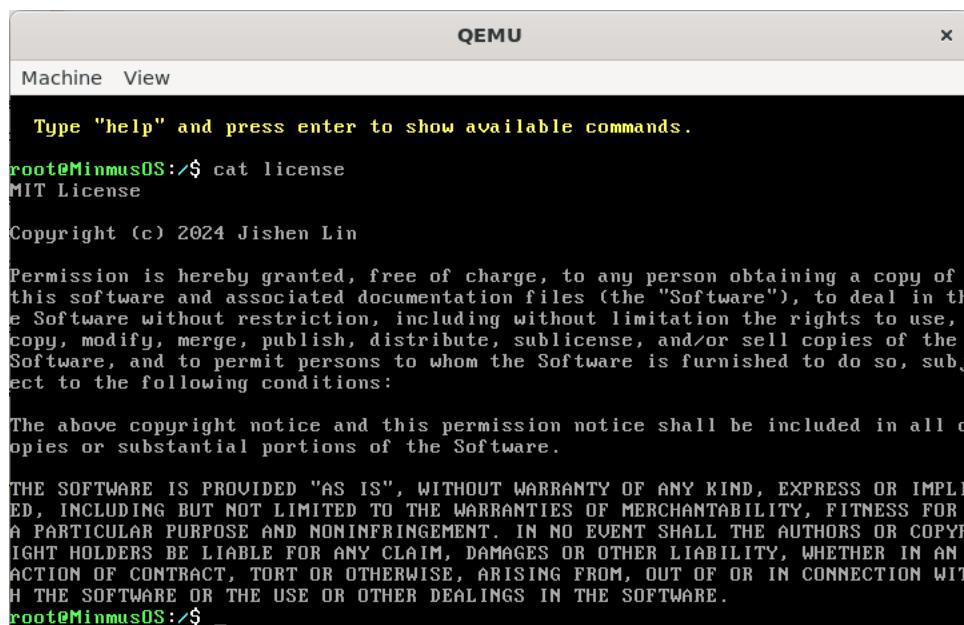
Cargo is the designated package manager and build system for the Rust programming language, playing a crucial role in managing project dependencies and facilitating consistent, reproducible builds. By using Cargo, developers can easily specify and manage libraries their projects depend on in a clear, concise manner through a Cargo.toml file. Cargo handles downloading these dependencies from crates.io repository, making sure they are compatible with each other and keeping them updated as necessary.

Beyond dependency management, Cargo also automates Rust project builds, ensuring that the compilation process is both reproducible and optimized. It records precise versions of dependencies in the Cargo.lock file, which helps maintain consistency across development environments and different machines. This system not only streamlines the development process but also significantly enhances project portability and ease of collaboration among multiple developers.

Cargo also plays a pivotal role in the Rust ecosystem by enforcing semantic versioning, which helps manage changes in libraries in a way that prevents compatibility problems. Its rich configuration options enable developers to tailor build processes and dependency resolutions to suit specific needs or environments. Furthermore, Cargo supports workspaces, allowing developers to manage multiple interrelated Rust packages within a single project, simplifying complex project management tasks. This extensive functionality underscores Cargo's importance in maintaining the robustness and efficiency of Rust project development.

root@MinmusOS:~$ _
```

图 5.69 文本文件 cargo 演示



```
QEMU
Machine View
Type "help" and press enter to show available commands.

root@MinmusOS:~$ cat license
MIT License

Copyright (c) 2024 Jishen Lin

Permission is hereby granted, free of charge, to any person obtaining a copy of
this software and associated documentation files (the "Software"), to deal in the
Software without restriction, including without limitation the rights to use,
copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the
Software, and to permit persons to whom the Software is furnished to do so, subject
to the following conditions:

The above copyright notice and this permission notice shall be included in all c
opies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYR
IGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN
ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH
THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

root@MinmusOS:~$ _
```

图 5.70 文本文件 license 演示

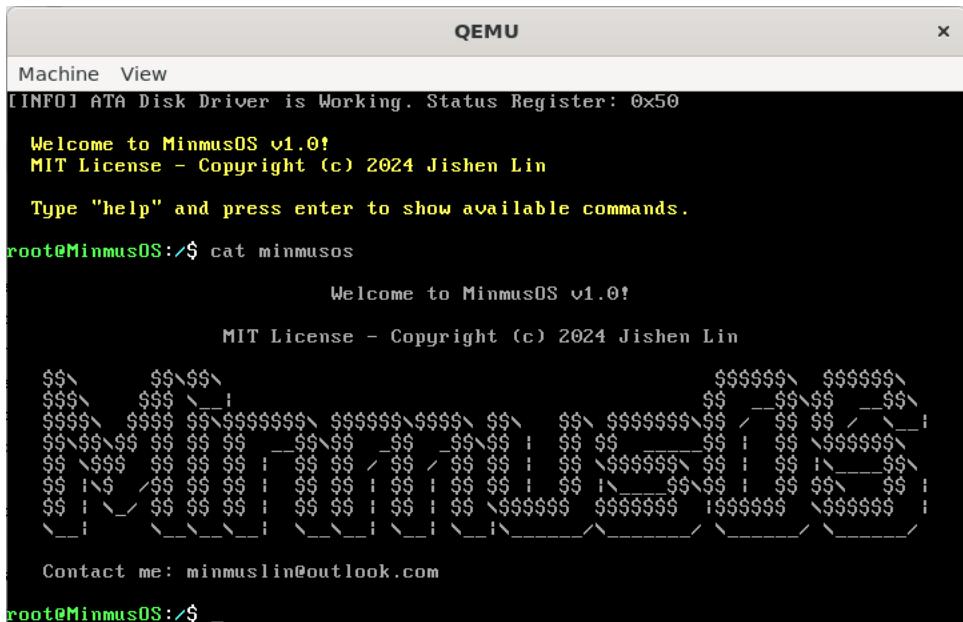


图 5.71 文本文件 minmusos 演示

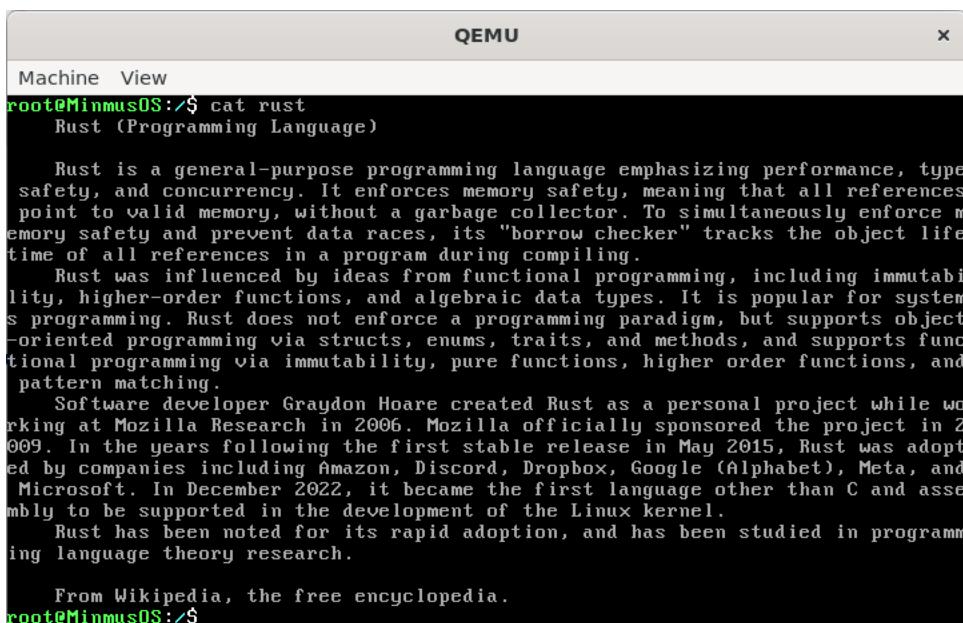
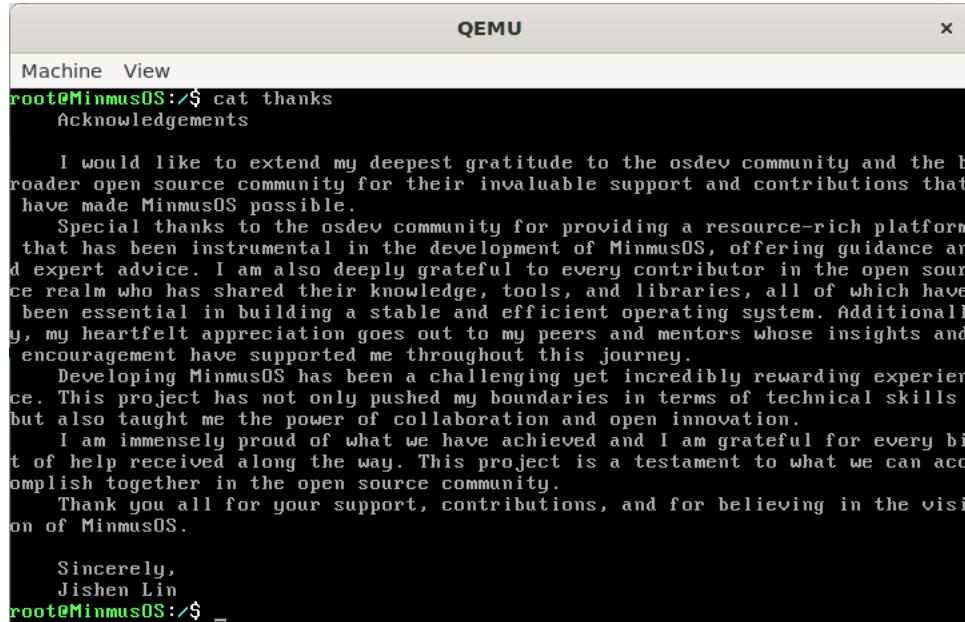


图 5.72 文本文件 rust 演示



The screenshot shows a terminal window titled "QEMU" with the command "Machine View". The terminal displays the contents of a file named "thanks". The file contains several paragraphs of text expressing gratitude to the osdev community and contributors for their support and contributions that made MinmusOS possible. It also expresses appreciation to the osdev community for providing a resource-rich platform and expert advice. The text concludes with a signature from Jishen Lin and a closing message of thanks for support and contributions.

```
root@MinmusOS:~$ cat thanks
Acknowledgements

I would like to extend my deepest gratitude to the osdev community and the broader open source community for their invaluable support and contributions that have made MinmusOS possible.

Special thanks to the osdev community for providing a resource-rich platform that has been instrumental in the development of MinmusOS, offering guidance and expert advice. I am also deeply grateful to every contributor in the open source realm who has shared their knowledge, tools, and libraries, all of which have been essential in building a stable and efficient operating system. Additionally, my heartfelt appreciation goes out to my peers and mentors whose insights and encouragement have supported me throughout this journey.

Developing MinmusOS has been a challenging yet incredibly rewarding experience. This project has not only pushed my boundaries in terms of technical skills but also taught me the power of collaboration and open innovation.

I am immensely proud of what we have achieved and I am grateful for every bit of help received along the way. This project is a testament to what we can accomplish together in the open source community.

Thank you all for your support, contributions, and for believing in the vision of MinmusOS.

Sincerely,
Jishen Lin
root@MinmusOS:~$ _
```

图 5.73 文本文件 thanks 演示

能有效地管理和分配处理器时间，从而维持系统的高效运行和响应性。定时器的精确操作对于实现多任务处理、实时系统的需求和系统稳定性至关重要，使其成为任何现代操作系统不可或缺的组成部分。

### 5.8.1 日期时间 (Datetime)

Time 数据结构（代码 5.74）用于封装日期和时间信息，包括年、月、日、小时、分钟和秒。此结构体还包含用于记录自 Unix 纪元以来的秒数的时间戳（timestamp）和 CPU 时钟周期数（ticks），这使得它可以为操作系统提供精确的时间跟踪和性能监测功能。Time 结构体的设计允许系统在执行任务调度、时间管理和资源监控等操作时，能够高效且准确地处理时间相关数据。这种数据结构的实现，是操作系统能够提供稳定和可靠服务的基础之一。

```
1 #[derive(Copy, Clone, Debug)]
2 pub struct Time {
3     year: u16,
4     month: u8,
5     day: u8,
6     hour: u8,
7     minute: u8,
8     second: u8,
9     timestamp: u64,
10    ticks: u64,
11 }
```

代码 5.74 Time 数据结构

代码 5.75 定义的 `get_current_datetime()` 函数是用于从硬件的实时时钟 (RTC)<sup>14</sup> 中直接获取当前的日期和时间。该函数返回一个元组，包含年、月、日、小时、分钟和秒，格式为 (`u16, u8, u8, u8, u8, u8`)。这种方法通常用于操作系统级别的时间管理，如设置系统时钟或执行与时间相关的操作。

函数利用 x86 I/O 指令 `out` 和 `in` 来读取 RTC 的时间数据。例如，通过设置 RTC 寄存器地址并读取对应的数据，从而分别获取秒、分钟、小时、日、月、年的值。每个时间部分的数据都是通过一系列的内联汇编命令独立获取的。

读取的时间数据通常以二进制编码十进制 (BCD) 格式返回，其中每个字节的两个半字节分别表示十位和个位。函数 `bcd_to_binary(bcd: u8) -> u8` 负责将这些 BCD 编码的数据转换为普通的二进制整数值，转换方式是将低半字节的值加上高半字节值乘以 10。

在处理完所有的时间数据后，由于读取的年份是从 2000 年开始的偏移量，函数会将这个偏移量加上 2000 来得到完整的年份。最后，函数将这些处理过的数据组装成一个元组返回。

由于直接进行硬件访问和使用内联汇编，这些操作被包裹在 `unsafe` 代码块中，以标识潜在的风险。

### 5.8.2 UNIX 时间戳 (UNIX Timestamp)

UNIX 时间戳 (UNIX Timestamp)，也称为 POSIX 时间或 Epoch 时间，是一种广泛使用的时间表示方法，其定义为自 1970 年 1 月 1 日 00:00:00 协调世界时 (UTC) 起至当前时刻的总秒数，不计入闰秒。UNIX 时间戳为整型数字，便于计算机系统进行日期和时间的计算与比较。它是操作系统、数据库和各类应用程序中处理时间的一种基准格式，允许简便地进行时间的加减和转换。由于其简单和一致性，UNIX 时间戳在全球范围内被广泛应用于多种编程环境和网络协议中。

代码 5.76 实现了一个函数 `get_unix_timestamp`，该函数用于将给定的日期时间元组转换为自 1970 年 1 月 1 日 00:00:00 UTC 以来的总秒数，即 UNIX 时间戳。

### 5.8.3 CPU 时钟周期计数 (CPU Ticks)

CPU 时钟周期计数 (CPU Ticks) 是指在中央处理器 (CPU) 中进行操作的基本时间单位，每个时钟周期代表 CPU 完成其基本操作的能力的一个周期。这些时钟周期通过使用特定的硬件指令，如 x86 架构的 `rdtsc` (Read Time-Stamp Counter)，直接从 CPU 的时间戳计数器中读取。CPU 时钟周期数提供了一种测量代码执行时间和系统性能的直接方法，常用于性能优化、系统监测、和精确的时间测量。由于每个时钟周期的持续时间取决于 CPU 的时钟频率，所以它可以作为评估不同 CPU 和不同环境下程序性能的重要指标。

`get_cpu_ticks()` 函数 (代码 5.77) 用于获取 CPU 的时钟周期计数，即 CPU Ticks，这是衡量处理器执行时间的一个基本单位。函数使用 x86 架构的 `rdtsc` 指令来读取 CPU 的时间戳计数器。在执

<sup>14</sup>RTC (Real-Time Clock) 即实时时钟，是计算机系统中的一种硬件设备，用于维持当前的日期和时间。即使在计算机断电或重启的情况下，它也能够继续运行，这是因为 RTC 通常由一个小电池供电，这使得它能在没有外部电源时持续工作。

装  
订  
线

```
1 fn get_current_datetime() -> (u16, u8, u8, u8, u8, u8) {
2     let mut second: u8;
3     let mut minute: u8;
4     let mut hour: u8;
5     let mut day: u8;
6     let mut month: u8;
7     let mut year: u8;
8     unsafe {
9         asm!(
10            "out 0x70, al",
11            "in al, 0x71",
12            "in(al) 0x00u8",
13            lateout("al") second,
14        );
15        asm!(
16            "out 0x70, al",
17            "in al, 0x71",
18            "in(al) 0x02u8",
19            lateout("al") minute,
20        );
21        asm!(
22            "out 0x70, al",
23            "in al, 0x71",
24            "in(al) 0x04u8",
25            lateout("al") hour,
26        );
27        asm!(
28            "out 0x70, al",
29            "in al, 0x71",
30            "in(al) 0x07u8",
31            lateout("al") day,
32        );
33        asm!(
34            "out 0x70, al",
35            "in al, 0x71",
36            "in(al) 0x08u8",
37            lateout("al") month,
38        );
39        asm!(
40            "out 0x70, al",
41            "in al, 0x71",
42            "in(al) 0x09u8",
43            lateout("al") year,
44        );
45    }
46    second = Self::bcd_to_binary(second);
47    minute = Self::bcd_to_binary(minute);
48    hour = Self::bcd_to_binary(hour);
49    day = Self::bcd_to_binary(day);
50    month = Self::bcd_to_binary(month);
51    year = Self::bcd_to_binary(year);
52    (2000 + year as u16, month, day, hour, minute, second)
53 }
```

代码 5.75 get\_current\_datetime 方法

```

1 fn get_unix_timestamp(datetime: (u16, u8, u8, u8, u8, u8)) -> u64 {
2     let (year, month, day, hour, minute, second) = datetime;
3     let mut days_since_epoch: u64 = 0;
4     for y in 1970..year {
5         days_since_epoch += if Self::is_leap_year(y) { 366 } else { 365 };
6     }
7     let month_days: [i32; 13] = if Self::is_leap_year(year) {
8         [0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
9     } else {
10        [0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
11    };
12    for m in 1..month {
13        days_since_epoch += month_days[m as usize] as u64;
14    }
15    days_since_epoch += day as u64 - 1;
16    let hours_since_epoch: u64 = days_since_epoch * 24 + hour as u64;
17    let minutes_since_epoch: u64 = hours_since_epoch * 60 + minute as u64;
18    let seconds_since_epoch: u64 = minutes_since_epoch * 60 + second as u64;
19    seconds_since_epoch
20 }

```

代码 5.76 get\_unix\_timestamp 方法

行 rdtsc 指令时，时钟周期的计数被分为高位和低位存储。eax 寄存器存储低 32 位，而 edx 寄存器存储高 32 位。通过位操作将高位和低位组合成一个 64 位的整数。具体操作是将高位 (high) 左移 32 位，然后与低位 (low) 进行位或操作 (|)，这样就能得到一个完整的 64 位的时钟周期计数。

由于 rdtsc 指令获取的是从 CPU 启动到当前的总时钟周期数，可以用来精确测量代码执行的时间，这是性能分析和优化中常用的一种方法。

```

1 fn get_cpu_ticks() -> u64 {
2     let mut low: u32;
3     let mut high: u32;
4     unsafe {
5         asm!(
6             "rdtsc",
7             out("eax") low,
8             out("edx") high,
9             options(nostack, nomem),
10            );
11     }
12     ((high as u64) << 32) | (low as u64)
13 }

```

代码 5.77 get\_cpu\_ticks 方法

## 5.9 命令行解释器（Shell）

命令行解释器（Shell）是一个用户界面，用于向操作系统提供用户指令。用户可以通过键入命令来与系统交互，这些命令然后由 Shell 解释并传递给操作系统的内核执行。

MinmusOS 提供了 20 条可以执行的命令（代码 5.78），见表 5.2。通过 Help 命令可以展示可用命令（图 5.79）。

表 5.2 可用命令

命令	描述
cal	显示当前月份的日历
cat <filename>	显示文件的内容
clear	清除终端屏幕
color	显示 VGA 文本模式颜色
date	显示当前日期和时间
echo <text>	输出文本
exit	退出当前会话
help	显示可用命令
hostname	显示主机名
kill <pid>	终止指定进程
ls	列出根目录条目
ps	列出运行中的任务
pwd	显示当前目录
reboot	重启系统
run <appname>	运行应用程序
shutdown	关闭系统
ticks	显示当前 CPU 时钟周期计数
timestamp	显示当前 UNIX 时间戳
uname	显示系统信息
whoami	显示当前用户

### 5.9.1 Shell 数据结构

Shell 结构（代码 5.82）定义了操作系统的命令行界面，包含三个成员：`buffer`（一个 256 字符的数组，用于存储用户输入的命令行文本）、`arg`（一个 11 字符的数组，专门用于解析命令参数）和 `cursor`（一个 `usize` 类型，记录当前用户输入的位置）。这个结构体提供了基础的文本输入存储和命令解析功能，使得操作系统能够接收、处理和响应用户的命令操作。

Shell 的实现需要定义三个常量（代码 5.80），用于操作系统中应用程序的加载和执行管理。它们的作用如下：

(1) `APP_TARGET`: 表示应用程序在内存中的目标加载地址。操作系统在加载应用程序时，会将其放置在这个内存地址开始的位置。

(2) `APP_SIZE`: 表示每个应用程序的大小上限 (64KB)。这个大小用于划分内存区域，以确保

```

1 const HELP: &'static str = "Available commands:
2 cal           - Shows current month's calendar
3 cat <filename> - Shows content of a file
4 clear         - Clears terminal screen
5 color          - Shows VGA text mode color
6 date          - Shows current datetime
7 echo <text>   - Outputs text
8 exit          - Exits current session
9 help           - Shows available commands
10 hostname      - Shows hostname
11 kill <pid>    - Terminates specified process
12 ls            - Lists root directory entries
13 ps            - Lists running tasks
14 pwd           - Shows current directory
15 reboot        - Reboot system
16 run <appname> - Runs an application
17 shutdown      - Shutdowns system
18 ticks          - Shows current CPU ticks
19 timestamp     - Shows current timestamp
20 uname          - Shows system information
21 whoami        - Shows current user";

```

代码 5.78 Help 命令提示

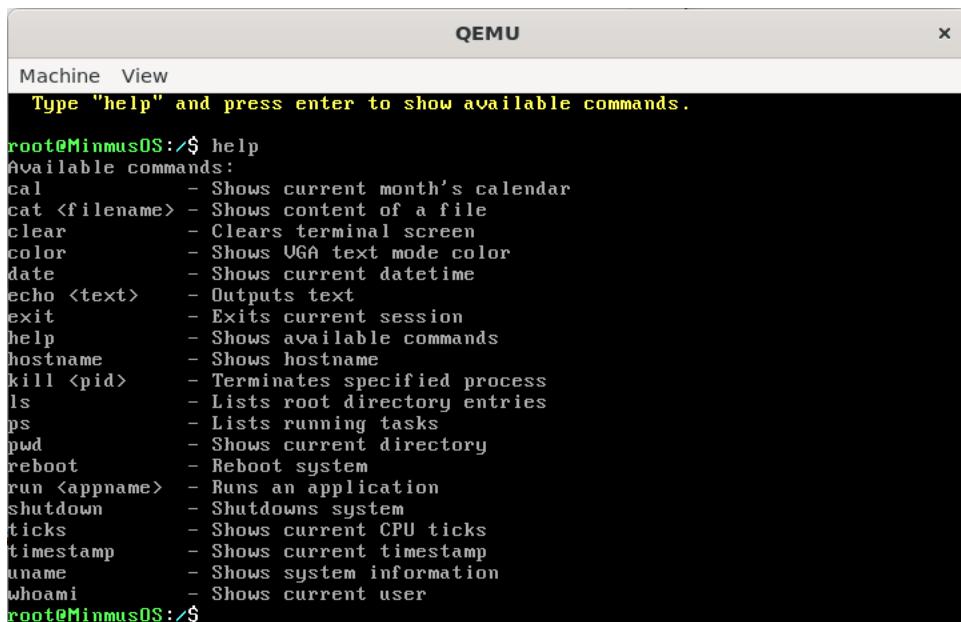


图 5.79 Help 命令演示

```

1 const APP_TARGET: u32 = 0x00A00000;
2 const APP_SIZE: u32 = 0x00010000;
3 const APP_SIGNATURE: u32 = 0xB16B00B5;

```

代码 5.80 Shell 常量定义

每个应用程序在独立的内存空间中运行，避免相互干扰。

(3) APP\_SIGNATURE: 这是一个标识符，用于验证加载的文件是否为有效的可执行文件。操作系统在加载应用程序后，会检查这个签名，以确认该文件确实是一个可以执行的应用程序。

SHELL 是一个可变的静态变量（代码 5.81），用于全局管理操作系统的命令行界面（Shell）的状态。SHELL 使用了 mut 关键字，表示该静态变量是可变的。这意味着操作系统可以在运行时动态地更新 SHELL 的状态，例如响应用户输入或执行命令时。SHELL 作为一个全局的 Shell 实例，使得操作系统中的各个模块可以通过访问这个变量来控制和管理命令行界面。它在整个系统中保持单一实例，以便统一管理 Shell 的行为和状态。

```
1 pub static mut SHELL: Shell = Shell {  
2     buffer: [0 as char; 256],  
3     arg: [0 as char; 11],  
4     cursor: 0,  
5 };
```

代码 5.81 SHELL 静态变量

Shell 实现了如下方法：

(1) **init** 方法：init 方法用于初始化 Shell 的状态。它将 buffer 清空为全零，并将 cursor 重置为 0，然后通过 PRINTER 设置命令提示符的颜色和格式。具体来说，它设置提示符为“root@MinmusOS:\$”，使用不同的颜色（如绿色、白色、青色等）分别标识用户名、主机名、目录和提示符符号。这个方法确保每次命令输入后，Shell 界面都会重新初始化并显示提示符，等待用户输入新的命令。

(2) **add** 方法：add 方法将用户输入的字符 c 添加到 buffer 的当前位置（由 cursor 指定），然后将 cursor 向前移动一位。它同时使用 lib::print! 输出该字符，显示在屏幕上。这个方法用于逐个收集用户输入的命令字符，并在输入时即时显示。

(3) **backspace** 方法：backspace 方法用于处理用户按下退格键的情况。它会将 cursor 位置向后移动一位，并将该位置的字符从 buffer 中删除（即设置为 0 as char），同时调用 PRINTER.delete() 从屏幕上移除最后一个字符。这个方法确保用户可以删除输入的字符。

(4) **enter** 方法：enter 方法在用户按下回车键时调用。它首先调用 PRINTER.new\_line() 在屏幕上换行，然后调用 interpret() 方法解析和执行当前 buffer 中的命令，最后再次调用 init() 方法重置 Shell 状态并显示新的提示符，准备接收下一条命令。

(5) **interpret** 方法：interpret 方法负责解析并执行 buffer 中的命令。它通过检查 buffer 是否匹配特定的命令来决定执行相应的操作（如 cal, cat, ls, run 等）。如果命令匹配，调用对应的函数执行命令逻辑；如果命令未匹配，显示“命令未找到”的错误信息。这个方法是 Shell 的核心，处理和响应用户输入的命令。

(6) **cat** 方法：cat 方法用于显示文件内容。它从 buffer 中提取文件名参数，并调用 FAT 文件系统驱动搜索文件。如果文件存在，读取文件内容并显示在屏幕上；如果文件不存在，显示“文件未

找到”的错误信息。这个方法通过操作文件系统，使得用户可以查看文本文件的内容。关于 `cat` 命令的实现详见子小节 5.9.3。

(7) **run 方法**: `run` 方法用于运行指定的应用程序。它从 `buffer` 中提取应用程序名称，并调用 FAT 文件系统驱动查找应用程序文件。如果文件存在且签名正确，将其加载到内存的特定位置，并将其作为任务添加到任务管理器中执行。如果文件签名不正确，显示“该文件不是有效的可执行文件”的错误信息；如果文件不存在，显示“应用程序未找到”的错误信息。这个方法实现了操作系统中应用程序的加载与执行。关于 `run` 命令的实现详见子小节 5.9.16。

(8) **is\_command 方法**: `is_command` 方法用于判断 `buffer` 中的内容是否与指定的命令字符串匹配（大小写不敏感）。它逐字符比较 `buffer` 和 `command`，如果匹配且 `buffer` 中命令后的字符是空格或终止符，则返回 `true`；否则返回 `false`。这个方法用于在 `interpret` 方法中识别用户输入的命令，并决定执行相应的操作。

### 5.9.2 cal 命令

`kernel/src/shell/cal.rs` 实现了 `cal` 命令。`cal` 命令的实现分为以下几个步骤：

(1) **获取当前日期信息**: 在 `calendar` 函数的开头，通过调用 `Time::init()` 获取当前时间的实例，然后分别提取当前的年、月和日。这个时间信息是接下来生成日历的基础。

(2) **确定月份标题**: 使用 `match` 表达式根据 `month` 值确定月份的名称，如“January”、“February”等。然后将月份名称与年份一起打印在日历的顶部，作为日历的标题。

(3) **计算月份的第一天**: 使用 Zeller's Congruence 算法（代码 5.83）计算出当月第一天是星期几。这一步得到一个从 0 到 6 的数值，表示星期一到星期日。这是日历对齐的关键，用于决定该月的日期从星期几开始显示。

(4) **确定当月的天数**: 通过 `match` 表达式，根据月份值确定当月的天数。如果是二月，还需判断该年是否为闰年，从而决定二月是 28 天还是 29 天。

(5) **生成和显示日历**: 首先，根据计算出的 `first_day_of_month` 在日历的第一行前打印适当数量的空格，然后逐日打印该月的日期。打印过程中，如果当前日期正好是当天，则会高亮显示（将日期文字设为黄色）。每当日期跨过一个星期时，换行继续打印，直到打印完当月所有日期。

(6) **最终输出**: 日历打印完成后，通过调用 `lib::println!()` 确保日历输出的最后一行与后续输出之间有一个空行隔开。

### 5.9.3 cat <filename> 命令

代码 5.84 实现了 `cat` 命令：

(1) **解析命令参数**: 当用户输入 `cat <filename>` 时，`buffer` 中会包含命令和文件名。`cat` 函数首先需要从 `buffer` 中提取文件名。它通过定位文件名的起始和结束位置，确保能够正确解析用户输入的文件名。如果文件名解析失败（例如没有提供文件名或文件名格式不正确），函数会显示用法提示

```
1 pub struct Shell {
2     buffer: [char; 256],
3     arg: [char; 11],
4     cursor: usize,
5 }
6
7 impl Shell {
8     pub fn init(&mut self) { ... }
9
10    pub fn add(&mut self, c: char) {
11        self.buffer[self.cursor] = c;
12        self.cursor += 1;
13        lib::print!("{}", c);
14    }
15
16    pub fn backspace(&mut self) {
17        if self.cursor > 0 {
18            unsafe {
19                PRINTER.delete();
20            }
21            self.cursor -= 1;
22            self.buffer[self.cursor] = 0 as char;
23        }
24    }
25
26    pub fn enter(&mut self) {
27        unsafe {
28            PRINTER.new_line();
29        }
30        self.interpret();
31        self.init();
32    }
33
34 #[allow(unused_unsafe)]
35 fn interpret(&mut self) { ... }
36
37 unsafe fn cat(&mut self, b: &[char]) { ... }
38
39 unsafe fn run(&mut self, b: &[char]) { ... }
40
41 fn is_command(&self, command: &str) -> bool {
42     let command_len: usize = command.len();
43     for (i, command_char) in command.chars().enumerate() {
44         let buffer_char: char = self.buffer[i].to_ascii_lowercase();
45         if buffer_char == '\0' || buffer_char != command_char.to_ascii_lowercase() {
46             return false;
47         }
48     }
49     self.buffer[command_len].to_ascii_lowercase() == ' ' || self.buffer[command_len] ==
50         '\0'
51 }
```

代码 5.82 Shell 数据结构

```

1 let first_day_of_month: usize = {
2     let (y, m): (i32, u8) = if month < 3 { (year as i32 - 1, month + 12) } else { (year as
3         ~ i32, month) };
4     let k: i32 = y % 100;
5     let j: i32 = y / 100;
6     let h: i32 = (1 + (13 * (m as i32 + 1)) / 5 + k + k / 4 + j / 4 + 5 * j) % 7;
7     ((h + 5) % 7) as usize
8 };

```

代码 5.83 Zeller's Congruence 算法

“Usage: cat <filename>” 并返回。

(2) **设置文件名参数**: 将提取的文件名字符逐个复制到 Shell 结构体的 arg 数组中，确保文件名被正确存储。超过 arg 数组长度的部分会被截断，同时在未使用的部分填充 0 以确保字符串正确结束。

(3) **查找文件**: cat 函数通过 FAT.acquire() 获取当前使用的 FAT 文件系统驱动，然后调用 textttfat.search\_file(&self.arg) 在文件系统中查找与 arg 中指定的文件名匹配的文件条目 Entry。如果没有找到对应的文件，会显示错误信息 “File not found!”，并结束命令执行。

(4) **读取和显示文件内容**: 如果找到文件，cat 函数会将文件内容读取到缓冲区并打印在屏幕上。通过调用 fat.read\_file\_to\_buffer(entry)，文件内容会被读取到文件系统的缓冲区 fat.buffer 中，然后逐个字符地打印出来，直到文件结束。

(5) **释放文件系统驱动**: 在操作结束后，cat 函数调用 FAT.free() 释放文件系统驱动，使其可以被其他操作使用，从而确保系统资源的合理管理。

(6) **结束处理**: 最后，cat 函数调用 lib::println!() 输出一个空行，以保持命令行界面的整洁，标志命令执行的结束。

#### 5.9.4 clear 命令

clear 命令的实现非常简单，当用户输入 clear 命令时，程序通过 is\_command("clear") 检查用户输入的命令是否为 clear。如果匹配，程序调用 PRINTER.clear() 方法，该方法负责清空终端屏幕内容，从而实现屏幕的清除效果。整个过程通过直接调用打印系统的清除功能，确保命令行界面恢复到一个干净的状态。

#### 5.9.5 color 命令

color 命令的实现通过遍历前景色 (fg) 和背景色 (bg) 的所有可能组合（共 16 种），并使用 PRINTER.set\_colors(fg, bg) 设置不同的颜色组合。每种组合都在屏幕上显示其对应的颜色代码（例如 “01” 表示前景色为 1，背景色为 0），以可视化方式展示所有可能的颜色配置。最后，调用 PRINTER.reset\_colors() 恢复默认颜色设置。这个命令主要用于展示终端支持的颜色选项和效果。

```
1 unsafe fn cat(&mut self, b: &[char]) {
2     ...
3     let fat: &FatDriver = FAT.acquire();
4     let entry: &Entry = fat.search_file(&self.arg);
5     if entry.name[0] != 0 {
6         fat.read_file_to_buffer(entry);
7         for c in fat.buffer {
8             if c != 0 {
9                 lib::print!("{} ", c as char);
10            }
11        }
12    } else {
13        PRINTER.set_colors(COLOR_LIGHT_RED, COLOR_BLACK);
14        lib::println!("File not found!");
15        PRINTER.reset_colors();
16    }
17    FAT.free();
18 }
```

代码 5.84 cat 方法

### 5.9.6 date 命令

date 命令的实现通过调用 `Time::init()` 获取当前系统时间，并提取年、月、日、小时、分钟和秒等时间信息。然后使用 `lib::println!` 以 `YYYY-MM-DD HH:MM:SS UTC` 的格式将时间打印到终端上，显示当前的系统日期和时间。整个过程通过简单的时间提取和格式化输出，实现了显示当前日期和时间的功能。

### 5.9.7 echo <text> 命令

echo 命令的实现（代码 5.85）用于在终端中输出用户输入的文本。函数首先跳过命令本身（`echo` 及其后的空格），然后逐字符遍历用户输入的文本字符。如果遇到空格字符且前一个字符也是空格，则跳过该空格；否则，将字符输出到终端。这样做的目的是避免输出连续的多个空格。在所有字符处理完成后，调用 `lib::println!()` 输出换行符，确保命令执行后的光标位置正确。通过这种方式，`echo` 命令可以将用户输入的文本整齐地显示在终端上。

### 5.9.8 exit 命令

exit 命令用于退出当前会话。实现时，通过使用内联汇编指令 `core::arch::asm!`，将特定值输出到 I/O 端口，以触发系统退出或关机。这是直接通过硬件端口操作实现退出功能。

### 5.9.9 help 命令

help 命令用于显示系统中可用的所有命令及其简要描述。实现时，直接通过 `lib::println!` 输出预先定义好的帮助文本 `HELP`，让用户查看命令列表和使用说明。

```

1 pub fn echo(b: &[char]) {
2     let mut last_was_space = true;
3     for &ch in b.iter().skip(5) {
4         if ch == '\0' {
5             break;
6         }
7         if ch == ' ' {
8             if !last_was_space {
9                 lib::print!(" ");
10                last_was_space = true;
11            }
12        } else {
13            lib::print!("{}", ch);
14            last_was_space = false;
15        }
16    }
17    lib::println!();
18 }

```

代码 5.85 echo 方法

### 5.9.10 hostname 命令

hostname 命令用于显示当前操作系统的主机名。实现时，通过 `lib::println!("MinmusOS")` 输出主机名为“MinmusOS”，让用户了解当前系统的标识。

### 5.9.11 kill <pid> 命令

代码 5.86 实现了 kill 命令：

(1) **解析命令参数**：当用户输入 `kill <pid>` 命令时，首先需要从输入的命令行中提取 PID 参数。函数从 `b` 中跳过命令本身（即 `kill` 及其后面的空格字符）来定位 PID 的起始位置 `start`。如果 PID 的起始位置不合法（即没有提供有效的 PID），函数会立即返回，并显示“`Usage: kill <pid>`”提示用户正确的使用方法。

(2) **验证和解析 PID**：一旦 PID 的起始位置确定，函数继续向后寻找 PID 的结束位置。通过再次遍历 `b`，寻找第一个空格或空字符来确定 PID 的结束位置 `end`。在提取出 PID 的字符序列后，函数将逐字符检查并将其转换为整数 `task_id`。在这个过程中，如果发现任何非数字字符，函数会判定 PID 输入无效，并立即返回，同时显示“`Please enter a valid PID (1-31)`”的错误信息。

(3) **验证 PID 范围**：在成功解析并转换 PID 为整数后，函数会进一步检查该 PID 是否在合法范围内，即是否在 1 到 `MAX_TASKS` 之间。如果 PID 超出了这个范围，函数会显示错误提示“`Please enter a valid PID (1-31)`”，并结束操作。这一检查确保了用户输入的 PID 是系统内一个有效的任务标识符。

(4) **移除任务**：`kill` 命令会调用 `TASK_MANAGER.remove_task(task_id)` 来终止系统中对应的任务，如果 PID 验证通过且在合法范围内。`TASK_MANAGER` 是一个全局的任务管理器，负责管理

所有运行中的任务。通过调用 `remove_task` 方法，系统会根据用户提供的 PID 查找到对应的任务，并将其从任务队列中移除。移除成功后，函数会使用 `lib::println!("Task (PID {}) has been removed.", task_id)` 输出确认信息，告知用户任务已成功终止。

(5) **输出与提示：**在整个命令执行过程中，`kill` 函数使用 `PRINTER` 进行文本输出。`PRINTER` 负责控制终端的显示，包括设置文本颜色和输出内容。为了区分正常操作和错误信息，`kill` 命令在输出错误信息时，会将文本颜色设置为 `COLOR_LIGHT_RED`，而在输出提示信息时，会将颜色设置为 `COLOR_LIGHT_MAGENTA`。在所有操作完成后，函数会调用 `PRINTER.reset_colors()` 将颜色恢复到默认状态。通过这种方式，`kill` 命令确保了用户在使用过程中可以清晰地接收到正确的反馈信息，增强了用户体验的友好性和操作的安全性。

```
1 pub fn kill(b: &[char]) {
2     ...
3     for i in start + 5..end {
4         if b[i].is_digit(10) { task_id = task_id * 10 + (b[i] as u8 - b'0') as usize; }
5         else { is_valid_id = false; break; }
6     }
7     if is_valid_id {
8         if task_id > 0 && task_id < MAX_TASKS as usize {
9             unsafe {
10                 if !TASK_MANAGER.tasks[task_id].running {
11                     PRINTER.set_colors(COLOR_LIGHT_RED, COLOR_BLACK);
12                     lib::println!("Task with PID {} not found!", task_id);
13                     PRINTER.reset_colors();
14                     return;
15                 }
16                 TASK_MANAGER.remove_task(task_id);
17                 lib::println!("Task (PID {}) has been removed.", task_id);
18             }
19         } else {
20             unsafe { PRINTER.set_colors(COLOR_LIGHT_RED, COLOR_BLACK); }
21             lib::println!("Please enter a valid PID (1-31).");
22             unsafe { PRINTER.reset_colors(); }
23         }
24     } else {
25         unsafe { PRINTER.set_colors(COLOR_LIGHT_RED, COLOR_BLACK); }
26         lib::println!("Please enter a valid PID (1-31).");
27         unsafe { PRINTER.reset_colors(); }
28     }
29 }
```

代码 5.86 kill 方法

### 5.9.12 ls 命令

`ls` 命令用于列出根目录中的所有文件和目录项。实现时，通过调用 `FAT.acquire()` 获取文件系统驱动，并使用 `list_entries()` 方法列出根目录中的所有条目。操作完成后，调用 `FAT.free()` 释放文件系统资源，确保系统资源的有效管理。

### 5.9.13 ps 命令

ps 命令用于显示当前系统中所有正在运行的任务列表。实现时，调用全局任务管理器 `TASK_MANAGER` 的 `list_tasks()` 方法，该方法会遍历系统中的所有任务并将其状态输出到终端，帮助用户查看系统中正在执行的任务。

### 5.9.14 pwd 命令

pwd 命令用于显示当前工作目录。在这个实现中，命令直接输出根目录 “/”，表明当前工作目录为系统的根目录。由于此 Shell 环境中没有复杂的目录结构，pwd 命令只返回根目录。

### 5.9.15 reboot 命令

reboot 命令用于重新启动系统。实现时，通过内联汇编 `core::arch::asm!` 发送特定命令到系统的 I/O 端口，触发硬件重启。这个直接的硬件操作确保了系统能够立即重新启动。

### 5.9.16 run <appname> 命令

代码 5.87 实现了 run 命令：

(1) **解析应用程序名称**: `run` 方法首先从用户输入的命令行中提取应用程序的名称。通过跳过 `run` 及其后的空格字符，定位应用程序名称的起始位置 `start`。如果没有提供有效的应用程序名称，或者名称格式不正确，方法会立即返回并显示用法提示 “`Usage: run <appname>`”。

(2) **设置应用程序名称参数**: 在成功提取到应用程序名称后，方法将其逐字符复制到 Shell 结构体的 `arg` 数组中，并确保名称字符串以 0 结束。如果应用程序名称超出 `arg` 数组的长度，则会被截断。

(3) **查找应用程序文件**: `run` 方法接下来会在文件系统中查找指定名称的应用程序文件。首先通过 `FAT.acquire()` 获取 FAT 文件系统驱动，然后调用 `fat.search_file(&self.arg)` 在文件系统中搜索与 `arg` 数组中存储的名称匹配的文件条目。如果找不到对应的文件，方法会显示 “`Application not found!`” 的错误信息，并结束操作。

(4) **加载并执行应用程序**: 如果找到应用程序文件，方法会尝试将其加载到内存中并执行。首先，方法通过 `TASK_MANAGER.get_free_slot()` 获取一个空闲的任务槽位，并计算应用程序的加载目标地址 `target`。然后，设置分页表 `TABLES[8]` 并调用 `PAGING.set_table`，将应用程序加载到指定的内存位置。如果应用程序文件包含有效的签名（与 `APP_SIGNATURE` 匹配），`run` 方法会通过 `TASK_MANAGER.add_task(target + 4)` 将其作为一个新任务添加到任务管理器中执行。如果签名不匹配，方法会显示 “`This file is not a valid executable!`” 的错误信息。

(5) **清理与释放资源**: 最后，无论运行应用程序是否成功，方法都会调用 `FAT.free()` 释放文件系统驱动，以确保资源的合理使用和系统的稳定性。

装  
订  
线

```
1 unsafe fn run(&mut self, b: &[char]) {
2     let start: usize = b.iter().skip(4).position(|&c| c != ' ' && c !=
3         '\0')unwrap_or(b.len());
4     if start + 4 >= b.len() {
5         unsafe {
6             PRINTER.set_colors(COLOR_LIGHT_MAGENTA, COLOR_BLACK);
7         }
8         lib::println!("Usage: run <appname>");
9         unsafe {
10             PRINTER.reset_colors();
11         }
12         return;
13     }
14     let end: usize = b.iter().skip(start + 4).position(|&c| c == ' ' || c ==
15         '\0').map_or(b.len(), |p| p + start + 4);
16     if end <= start + 4 {
17         unsafe {
18             PRINTER.set_colors(COLOR_LIGHT_MAGENTA, COLOR_BLACK);
19         }
20         lib::println!("Usage: run <appname>");
21         unsafe {
22             PRINTER.reset_colors();
23         }
24         return;
25     }
26     let mut j: usize = 0;
27     for i in start + 4..end {
28         self.arg[j] = b[i];
29         j += 1;
30     }
31     for i in j..self.arg.len() {
32         self.arg[i] = '\0';
33     }
34     let fat: &FatDriver = FAT.acquire();
35     let entry: &Entry = fat.search_file(&self.arg);
36     if entry.name[0] != 0 {
37         let target: u32 = APP_TARGET + (TASK_MANAGER.get_free_slot() as u32 * APP_SIZE);
38         TABLES[8].set(target);
39         PAGING.set_table(8, &TABLES[8]);
40         fat.read_file_to_target(&entry, target as *mut u32);
41         if *(target as *mut u32) == APP_SIGNATURE {
42             TASK_MANAGER.add_task(target + 4);
43         } else {
44             unsafe { PRINTER.set_colors(COLOR_LIGHT_RED, COLOR_BLACK); }
45             lib::println!("This file is not a valid executable!");
46             unsafe { PRINTER.reset_colors(); }
47         }
48     } else {
49         PRINTER.set_colors(COLOR_LIGHT_RED, COLOR_BLACK);
50         lib::println!("Application not found!");
51         PRINTER.reset_colors();
52     }
53     FAT.free();
54 }
```

代码 5.87 run 方法

### 5.9.17 shutdown 命令

`shutdown` 命令用于关闭系统。实现时，通过内联汇编 `core::arch::asm!`，将特定的值输出到 I/O 端口，触发系统的关机操作。这是一个直接的硬件控制命令，用于立即停止系统的运行。

### 5.9.18 ticks 命令

`ticks` 命令用于显示自系统启动以来的 CPU 时钟滴答数。实现时，通过调用 `Time::init()` 初始化时间模块，然后使用 `get_ticks()` 方法获取当前的滴答数，并通过 `lib::println!` 将其输出到终端，供用户查看。

### 5.9.19 timestamp 命令

`timestamp` 命令用于显示当前的时间戳。实现时，同样调用 `Time::init()` 初始化时间模块，并使用 `get_timestamp()` 方法获取当前的 Unix 时间戳，表示自 1970 年 1 月 1 日以来的秒数。然后将其通过 `lib::println!` 输出，方便用户查看当前时间戳。

### 5.9.20 uname 命令

`uname` 命令用于显示操作系统的名称和版本信息。实现时，直接通过 `lib::println!` 输出操作系统的名称和版本信息，例如“MinmusOS v1.0 IA-32 x86”，以告知用户当前运行的操作系统的基本信息。

### 5.9.21 whoami 命令

`whoami` 命令用于显示当前用户的身份。实现时，通过 `lib::println!("root")` 直接输出用户名“root”，表明当前操作系统的用户身份为超级用户。

## 5.10 主启动程序 (Main Boot Program)

### 5.10.1 编译器指令

在 Rust 开发中，编译器指令（pragmas）以 `#![...]` 的形式出现，它们为编译器提供了特定的指示。这里使用的指令（代码 5.88）具体意义如下：

(1) `#![no_std]`：这个属性指示 Rust 编译器不要链接到 Rust 的标准库（std），因为标准库依赖于操作系统的底层功能，例如线程和文件系统，这在裸机环境中通常是不可用的。使用 `#![no_std]`，需要依赖于 core 库，它是一个不依赖于操作系统的 Rust 库，提供了基本的语言构建块，如迭代器、基本类型和切片等。

(2) `#![no_main]`：在 Rust 程序中，默认情况下，`main` 函数是程序的入口点。在裸机或内核开发中，常常需要自定义入口点（如 `_start`），因为标准的 `main` 函数需要某些操作系统级别的初始化，

这在操作系统开发中是不适用的。`#![no_main]` 属性用于告知编译器，这个程序将不使用标准的 `main` 函数作为入口点。

(3) `#![feature(naked_functions)]`: “裸函数” (naked functions) 是一种特殊类型的函数，它们允许开发者完全控制函数的入口和出口序列。这意味着这些函数不会生成任何额外的汇编指令来处理变量保存、寄存器恢复等，使得开发者可以精确地控制函数的所有汇编操作。这在写操作系统或其他需要精确控制汇编的环境中非常有用。

这些属性和特性对于开发依赖于硬件操作和低级内存管理的应用程序，如操作系统内核，是非常关键的，它们使得 Rust 代码能够以更底层、更控制的方式运行。

```
1 #![no_std]
2 #![no_main]
3 #![feature(naked_functions)]
```

代码 5.88 编译器指令

### 5.10.2 常量定义

代码 5.89 中定义的这些常量具有关键的意义，主要用于内存布局的配置：

```
1 const KERNEL_START: u32 = 0x00100000;
2 const KERNEL_SIZE: u32 = 0x00100000;
3 const STACK_SIZE: u32 = 0x00100000;
4 const STACK_START: u32 = KERNEL_START + KERNEL_SIZE + STACK_SIZE;
```

代码 5.89 常量定义

(1) `KERNEL_START`: 这个常量定义了内核的起始物理地址。在这个示例中，`0x00100000` (十六进制) 通常是指内存中的 1MB 位置。这是一个常见的起点，因为低于 1MB 的内存可能会被系统的 BIOS 或其他固件使用。选择这个地址作为内核的起始位置有助于避免与这些系统组件冲突。

(2) `KERNEL_SIZE`: 这个常量定义了内核所占用的内存大小，也是 1MB。这意味着你的内核被设计为在物理内存的这个范围内运行。内核大小的设定需要足够容纳内核的所有代码和数据。

(3) `STACK_SIZE`: 栈大小同样被设置为 1MB。栈是用于存储函数局部变量、返回地址和函数调用时的其他数据的内存区域。确保有足够的栈空间对于防止栈溢出和支持多层函数调用非常重要。

(4) `STACK_START`: 这个常量计算的是栈的起始地址。它是通过在内核结束的地址 (`KERNEL_START + KERNEL_SIZE`) 之后再加上栈的大小 (`STACK_SIZE`) 来确定的。这确保了栈空间在内核之后、且不与其他内存区域重叠，从而在物理内存中保持良好的组织和分隔。

这些常量的设定对于内核的内存管理极其关键，它们确保了内核、栈和其他关键数据结构在物理内存中有合理的布局和足够的空间，从而提供了稳定运行的基础。

### 5.10.3 内核启动过程

在 MinmusOS 中，主启动程序（Main Boot Program）（代码 5.90）位于 `kernel/src/main.rs` 文件中，是整个操作系统内核的入口点。这个程序负责初始化系统的关键组件并进入操作系统的主循环。以下是主启动程序的主要步骤和它们的功能：

(1) **堆栈初始化**：在系统的最初阶段，需要设置堆栈指针（esp），确保在后续的程序中有足够的空间来存储局部变量和函数调用的返回地址。这一步是通过直接汇编指令设置 `esp` 寄存器完成的。

(2) **分页初始化**：操作系统启动时，需要配置内存分页，以支持虚拟内存的使用。这里首先进行身份映射（所有物理地址直接映射到相同的虚拟地址），随后启用分页机制。

(3) **加载中断描述符表（IDT）**：中断描述符表（IDT）是用于定义如何处理各种中断的关键数据结构。程序在此阶段初始化 IDT，并添加了处理器异常、定时器中断、系统调用和键盘中断的处理函数。

(4) **初始化可编程中断控制器（PIC）**：可编程中断控制器（PIC）负责管理硬件中断，这一步骤将 PIC 初始化，以便能够接收和处理外部设备的中断请求。

(5) **初始化 FAT16 文件系统**：如果磁盘被检测为启用状态，此时将加载 FAT16 文件系统。这包括加载文件系统的头部、表格和条目，确保系统可以读写文件。

(6) **初始化多任务处理**：为支持多任务处理，此阶段初始化任务管理器，设置任务调度和管理机制。

(7) **初始化 Shell**：最后，系统初始化一个简单的 Shell 环境，允许用户通过命令行与操作系统交互。初始化完毕后，开启中断处理（通过 `sti` 指令），让操作系统开始响应中断。

这些步骤共同构成了 MinmusOS 的核心启动逻辑，是系统能够顺利运行和管理硬件及应用程序的基础。通过这个主启动程序，MinmusOS 能够配置必要的环境并提供一个稳定的运行时环境。

### 5.10.4 PANIC 处理器

在许多编程语言和操作系统中，尤其是 Rust，`panic` 指的是程序运行中遇到无法处理的错误或无法恢复的状态时的行为。这通常是由于程序逻辑错误、资源耗尽或不符合预期的操作导致的。

`Panic` 通常意味着程序或系统必须停止执行当前任务。在 Rust 中，`panic` 可能导致线程崩溃或整个程序终止，除非有特定的捕获机制来处理这种情况。

MinmusOS 通过一个自定义的 `panic_handler` 函数（代码 5.91）来处理 `panic`。这个函数被标记为 `#[panic_handler]`，这是 Rust 操作系统开发中用来指定处理系统 `panic` 的函数。

图 5.92 是 MinmusOS 在遇到内核 `panic` 时的用户界面展示。当系统遭遇严重错误并停止运行时，界面以红色背景和白色文字清晰地显示了错误的详细信息。它首先标明了“`KERNEL PANIC!`”，紧接着解释系统已经因错误停止，并具体说明了错误的性质——这里是一个数组越界问题。技术细节部分指出了错误发生的具体文件和行号，帮助开发者快速定位问题。页面底部友好地提示用户重新启动计算机，这是用户唯一的交互方式，因为系统已无法继续操作。整个设计旨在在紧急情况下

```
1 #[no_mangle]
2 #[link_section = ".start"]
3 pub extern "C" fn _start() -> ! {
4     unsafe {
5         PRINTER.prints("[INFO] Initializing Stack...\n");
6         asm!("mov esp, {}", in(reg) STACK_START);
7
8         PRINTER.prints("[INFO] Initializing Paging...\n");
9         PAGING.identity();
10        PAGING.enable();
11        asm!("xchg bx, bx");
12
13        PRINTER.prints("[INFO] Loading Interrupt Descriptor Table...\n");
14        IDT.init();
15        IDT.add_exceptions();
16        IDT.add(
17            interrupts::timer::TIMER_INT as usize,
18            interrupts::timer::timer as u32,
19        );
20        IDT.add(
21            syscalls::handler::SYSCALL_INT as usize,
22            syscalls::handler::syscall as u32,
23        );
24        IDT.add(
25            drivers::keyboard::KEYBOARD_INT as usize,
26            drivers::keyboard::keyboard as u32,
27        );
28        IDT.load();
29
30        PRINTER.prints("[INFO] Initializing Programmable Interrupt Controllers...\n");
31        PICS.init();
32
33        PRINTER.prints("[INFO] Initializing FAT16 File System...\n");
34        DISK.check();
35        if DISK.enabled {
36            let fat = FAT.acquire_mut();
37            fat.load_header();
38            fat.load_table();
39            fat.load_entries();
40            FAT.free();
41        }
42
43        PRINTER.prints("[INFO] Initializing Multitasking...\n");
44        TASK_MANAGER.init();
45
46        PRINTER.prints("[INFO] Initializing Shell...\n");
47        print_info();
48        SHELL.init();
49        asm!("xchg bx, bx");
50        asm!("sti");
51
52        loop {}
53    }
54 }
```

代码 5.90 \_start 内核启动函数

```

1 #[panic_handler]
2 fn panic(info: &PanicInfo) -> ! {
3     unsafe { PRINTER.set_colors(COLOR_LIGHT_WHITE, COLOR_RED); PRINTER.clear(); }
4     lib::println!();
5     lib::println!(" ===== MinmusOS v1.0 =====");
6     lib::println!(" ↳ Lin");
7     lib::println!();
8     lib::println!(" KERNEL PANIC!");
9     lib::println!();
10    lib::println!(" MinmusOS has encountered a panic and the system has been halted.");
11    lib::println!();
12    lib::println!();
13    lib::println!();
14    lib::println!();
15    lib::println!();
16    lib::println!(" PANIC DESCRIPTION:");
17    lib::println!();
18    lib::println!(" {}", info.message());
19    lib::println!();
20    lib::println!();
21    lib::println!(" TECHNICAL INFORMATION:");
22    lib::println!();
23    lib::println!(" FILE : {}", info.location().unwrap().file());
24    lib::println!(" LINE : {}", info.location().unwrap().line());
25    lib::println!();
26    lib::println!();
27    lib::print!(" Please restart your computer. :) ");
28    loop {}
29 }

```

代码 5.91 PanicHandler 函数



图 5.92 PANIC 处理器演示

提供清晰、直接的信息，确保用户知晓发生了什么并鼓励采取适当的响应措施。

### 5.11 链接器脚本 (Linker Script)

链接器脚本（代码 5.93）是用来指导链接器如何组织 MinmusOS 内核的不同部分（即各个段）到最终的可执行文件中。链接器脚本定义了内存布局，确保内核的不同数据和代码段按照预期的地址和顺序放置。

链接器脚本还对操作系统的性能和功能安全起到了关键作用。正确配置的链接器脚本能够确保程序的不同部分被放置在适合执行的内存区域，减少运行时的错误，提高系统的效率和响应速度。通过精确控制如 `.text`, `.data`, `.bss`, `.rodata` 等段的放置，链接器脚本直接影响了程序的加载和执行。例如，将代码和只读数据保持在紧密的内存区域可以优化访问速度和缓存利用率。

在 Rust 项目中，`build.rs` 文件充当构建脚本的角色，主要用于在项目编译前执行自定义的构建任务。构建脚本的实现与子小节 4.2.6 相同，在此不再赘述。

通过精确地控制各个内存段的位置和大小，这个脚本确保内核能够被正确地加载到预期的内存地址，并且所有的代码和数据都按预期方式进行组织。通过配置 `.eh_frame` 和 `.eh_frame_hdr`，增强了程序的健壮性，使得在出现运行时错误时能够进行有效的异常处理。

至此我们完成了内核的加载与启动，如图 5.94。



图 5.94 内核启动演示

在本章中，我们详细探讨了 MinmusOS 内核的核心功能实现，从中断处理到驱动程序，再到多任务处理和系统调用，每一部分都是为了提高操作系统的性能和稳定性。通过深入了解中断描述符表、中断服务例程以及可编程中断控制器，我们得以构建一个响应迅速且可靠的中断管理系统，它是操作系统稳定运行的基石。

```

1  /* 配置程序入口点 */
2  ENTRY(_start)
3
4  /* 配置段的顺序和位置 */
5  SECTIONS {
6      /* 配置内核加载的起始地址为 1MB */
7      . = 0x00100000;
8
9      /* 定义内核开始的位置 */
10     _kernel_start = .;
11
12     /* 定义启动段, 包含启动代码的实际入口点 */
13     .start : {
14         *(.start)
15     }
16
17     /* 定义代码段, 包含程序的机器代码 */
18     .text : {
19         *(.text .text.*)
20     }
21
22     /* 定义 BSS 段, 包含程序中未初始化的数据 */
23     .bss : {
24         *(.bss .bss.*)
25     }
26
27     /* 定义只读数据段, 包含常量等不应被程序修改的数据 */
28     .rodata : {
29         *(.rodata .rodata.*)
30     }
31
32     /* 定义数据段, 包含已初始化的全局变量和静态变量 */
33     .data : {
34         *(.data .data.*)
35     }
36
37     /* 配置异常处理信息, 用于支持运行时错误处理 */
38     .eh_frame : {
39         *(.eh_frame .eh_frame.*)
40     }
41     .eh_frame_hdr : {
42         *(.eh_frame_hdr .eh_frame_hdr.*)
43     }
44
45     /* 配置地址对齐与预留空间 */
46     . = _kernel_start + 0x00100000 - 2;
47
48     /* 在内存中设置一个结束标记, 用于标识内核的结束 */
49     .end_marker :
50     {
51         SHORT(0xDEAD)
52     }
53 }
```

代码 5.93 kernel/linker.ld

我们还介绍了驱动程序的实现，包括键盘和硬盘驱动程序，这些是与硬件设备交互的关键组件。此外，多任务处理的讨论，尤其是上下文切换和 CPU 调度器的详细描述，展示了如何有效地在多个进程之间分配 CPU 时间，确保系统资源的高效使用。

系统调用作为用户程序与操作系统内核之间的桥梁，其稳健的实现对于操作系统的功能性和保护模式的安全性至关重要。VGA 文本模式的管理则展示了基础的用户界面交互方式，提供了一种简单而直观的方式来输出文本信息。

在内存管理方面，我们讨论了从内存分配到分页管理的策略，这些策略确保了内存的高效使用和程序的隔离性，防止了许多常见的安全问题。文件系统的实现讨论则涵盖了数据持久化和文件操作的基础知识，为存储和检索用户数据提供了方法。

最后，命令行解释器的介绍展示了如何通过简单的文本命令与操作系统交互，使用户能够执行各种任务，如文件管理、系统监控和配置。

通过这些章节的讨论，不仅加深了对操作系统设计和实现的理解，也为日后在 MinmusOS 项目中的进一步开发和优化奠定了坚实的基础。

## 6 标准运行库实现

在本节（章节 6）中，笔者将详细介绍 MinmusOS 的标准运行库，这些库为操作系统提供了核心的运行支持和基础功能实现。标准运行库是操作系统架构中不可或缺的一部分，它们提供了从基本的数据处理到复杂的系统交互所需的所有基础工具和功能。通过对这些库的实现，MinmusOS 能够提供稳定、高效的环境，以支持各种应用和系统级的操作。

笔者将依次探讨六个核心库的功能和实现方式，包括 **math** 库、**mutex** 库、**print** 库、**rand** 库、**sort** 库和 **string** 库。这些库覆盖了从数学计算到字符串处理、从生成随机数到数据排序等多个方面，是实现操作系统功能的基石。通过章节 6 开发者将能更深入地理解如何利用这些基础库来构建和优化自己的操作系统内核。

### 6.1 标准运行库概述

标准运行库（Standard Runtime Library），通常指的是提供编程语言核心运行支持的库，这些库包含了编程语言运行所必需的基本函数和数据结构。这些库的函数可以包括输入输出处理、内存管理、字符串操作、数学计算等基础功能。

标准库提供了编译和在操作系统上运行程序所需的所有实现。它实现了各种有用的算法和数据结构，但最重要的是，它提供了系统调用的封装，使程序员可以更轻松地与操作系统交互，而不必直接调用系统调用。

MinmusOS 提供了六个标准运行库的实现：

(1) **math 库**: 提供数学计算的函数和算法，包括基本的算术运算、复杂的数学函数（如三角函数、对数函数等）和数值分析方法。这个库可以用于执行各种数学运算，支持操作系统内核及其它模块进行数学相关的处理。

(2) **mutex 库**: 实现互斥锁（Mutex）的功能，用于控制对共享资源的访问。这是多线程或多进程环境中同步和防止数据竞争的关键工具。互斥锁可以帮助确保当一个线程或进程在使用一个共享资源时，其他线程或进程必须等待，直到资源被释放。

(3) **print 库**: 提供打印宏的实现，可以用于打印到控制台、日志文件或其他输出设备的功能，用于调试和追踪系统运行时的状态和错误。

(4) **rand 库**: 随机数生成是计算机科学中的一个基本需求，尤其在操作系统的设计中扮演着重要角色。MinmusOS 中的 rand 库不仅支持基本的随机数生成，还包括多种伪随机数生成算法，如线性同余生成器（LCG）、梅森旋转（Mersenne Twister）等。这些算法能够生成高质量的随机序列，用于多种系统级功能，包括但不限于内存分配的随机化、进程调度的公平性保证、以及增强安全性的密码学应用。

(5) **sort 库**: 实现各种排序算法的库。这个模块可以对数据进行排序，支持基本的排序算法如冒泡排序、快速排序、归并排序等。排序功能常用于数据处理和优化存储或检索。

(6) **string 库**: 提供字符串处理的函数和算法。这个库包括字符串的搜索、比较、拼接等常见

操作。字符串处理是大多数操作系统功能的基础，用于文件系统、用户界面和网络通信等方面。

## 6.2 math 库

MinmusOS 的 math 库提供了 64 个数学函数（表 6.1），覆盖了广泛的数学运算和计算需求。它们包括基本的算术运算（加、减、乘、除），比较函数（最大值和最小值），三角函数（正弦、余弦、正切等），反三角函数，以及双曲函数。此外，还有对数、指数、平方根和立方根等高级数学函数。库中也包括了一些特殊函数如伽马函数、贝塞尔函数和误差函数。向量运算如加法、减法、点积和叉积也被实现。此外，库中还提供了统计函数，用于计算数列的总和、乘积、平均值、中位数、众数、范围等统计指标。复数运算也得到了支持，包括复数的加法、减法、乘法和除法。整体而言，这个数学库为操作系统提供了一套全面而强大的数学计算工具。

表 6.1 math 库函数

函数名	输入	输出	功能
add	a: f64, b: f64	f64	计算两个浮点数的和
sub	a: f64, b: f64	f64	计算两个浮点数的差
mul	a: f64, b: f64	f64	计算两个浮点数的乘积
div	a: f64, b: f64	f64	计算两个浮点数的商
max	a: T, b: T	T	返回两个可比较值中的最大值
min	a: T, b: T	T	返回两个可比较值中的最小值
ceil	x: f64	f64	计算大于等于输入的最小整数
floor	x: f64	f64	计算小于等于输入的最大整数
round	x: f64	f64	四舍五入到最接近的整数
pow	base: f64, exp: i32	f64	计算基数的指数次幂
exp	x: f64	f64	计算 e 的 x 次方
sqrt	x: f64	f64	计算平方根
cbrt	x: f64	f64	计算立方根
abs	value: i64	i64	计算整数的绝对值
fabs	value: f64	f64	计算浮点数的绝对值
sin	x: f64	f64	计算正弦值（输入为角度）
cos	x: f64	f64	计算余弦值（输入为角度）
tan	x: f64	f64	计算正切值（输入为角度）
cot	x: f64	f64	计算余切值（输入为角度）
sec	x: f64	f64	计算正割值（输入为角度）
csc	x: f64	f64	计算余割值（输入为角度）
arcsin	x: f64	f64	计算反正弦值
arccos	x: f64	f64	计算反余弦值
arctan	x: f64	f64	计算反正切值

续下页

续表 6.1

函数名	输入	输出	功能
arccot	x: f64	f64	计算反余切值
arcsec	x: f64	f64	计算反正割值
arccsc	x: f64	f64	计算反余割值
sinh	x: f64	f64	计算双曲正弦值
cosh	x: f64	f64	计算双曲余弦值
tanh	x: f64	f64	计算双曲正切值
ln	x: f64	f64	计算自然对数
log2	x: f64	f64	计算以 2 为底的对数
log10	x: f64	f64	计算以 10 为底的对数
log	base: f64, x: f64	f64	计算以任意底的对数
is_prime	n: i64	bool	判断整数是否为质数
gcd	a: i64, b: i64	i64	计算两个整数的最大公约数
lcm	a: i64, b: i64	i64	计算两个整数的最小公倍数
sum	numbers: [f64]	f64	计算数组中数值的总和
difference	numbers: [f64]	f64	计算数组中数值的总差
product	numbers: [f64]	f64	计算数组中数值的总积
division	numbers: [f64]	f64	计算数组中数值的总商
max_value	numbers: [f64]	f64	返回数组中的最大值
min_value	numbers: [f64]	f64	返回数组中的最小值
mode	numbers: [f64]	f64	返回数组中的众数
median	numbers: mut [f64]	f64	返回数组中的中位数
range	numbers: [f64]	f64	返回数组中的范围
mean	numbers: [f64]	f64	返回数组中数值的算术平均值
geometric_mean	numbers: [f64]	f64	返回数组中数值的几何平均值
harmonic_mean	numbers: [f64]	f64	返回数组中数值的调和平均值
weighted_mean	numbers: [f64], weights: [f64]	f64	返回数组中数值的加权平均值
trimmed_mean	numbers: mut [f64], percent: f64	f64	返回数组中数值的截尾平均值
polynomial	x: f64, n: [f64]	f64	计算多项式的值
vector_add	v1: [f64], v2: [f64], result: mut [f64]	0	计算两个向量的加法
vector_sub	v1: [f64], v2: [f64], result: mut [f64]	0	计算两个向量的减法
dot_product	v1: [f64], v2: [f64]	f64	计算两个向量的点积
cross_product	v1: [f64], v2: [f64], result: mut [f64]	0	计算两个向量的叉积
bessel_j0	x: f64	f64	计算第一类零阶贝塞尔函数的值
gamma	x: f64	f64	计算伽玛函数的值
erf	x: f64	f64	计算误差函数的值
complex_add	a_real: f64, a_imag: f64, b_real: f64, b_imag: f64	(f64, f64)	计算两个复数的加法

续下页

续表 6.1

函数名	输入	输出	功能
complex_sub	a_real: f64, a_imag: f64, b_real: f64, b_imag: f64	(f64, f64)	计算两个复数的减法
complex_mul	a_real: f64, a_imag: f64, b_real: f64, b_imag: f64	(f64, f64)	计算两个复数的乘法
complex_div	a_real: f64, a_imag: f64, b_real: f64, b_imag: f64	(f64, f64)	计算两个复数的除法
complex_conjugate	a_real: f64, a_imag: f64	(f64, f64)	计算复数的共轭

## 6.3 mutex 库

### 6.3.1 互斥对象 (Mutual Exclusion Object)

Mutex，全称是“mutual exclusion object”，即“互斥对象”。它是一种用于多线程编程中的同步机制，用来确保同一时间只有一个线程能够访问某个共享资源，从而避免竞争条件（race conditions）导致的数据不一致问题。

Mutex 的工作原理类似于锁，线程在访问共享资源前需要先获取这个锁，获取锁成功后才能访问资源。在资源访问完成后，线程必须释放锁，允许其他线程访问该资源。若其他线程在此期间尝试访问该资源，由于锁已经被占用，这些线程将会被阻塞，直到锁被释放为止。

简单来说，Mutex 通过锁定机制确保了多线程环境下对共享资源的访问是安全的，有序的，避免了多个线程同时修改数据而导致的错误。

### 6.3.2 Mutex 数据结构

MinmusOS 实现了一个简单的 Mutex（互斥锁），用于保证多线程环境中对共享数据的独占访问。代码 6.1 定义了 Mutex 结构体。

```

1 pub struct Mutex<T> {
2     target: T,
3     free: AtomicBool,
4 }
```

代码 6.1 Mutex 数据结构定义

Mutex 结构体包含两个字段：target 和 free。target 是一个泛型字段，用于存储 Mutex 需要保护的数据，使得该互斥锁可以适用于多种数据类型。free 是一个 AtomicBool 类型的字段，用于标示锁的状态，确保在多线程环境中对共享数据的访问是安全的。

代码 6.2 实现了 Mutex 结构体的方法：

(1) **new 方法**：这个构造函数用于创建 Mutex 的新实例。它接受一个泛型参数 value，该参数是 Mutex 将要保护的数据。在初始化时，free 字段被设置为 true，表示锁是可用的。这样的设计使得锁的初始化直接与它将要保护的数据绑定，保证了数据保护的始终性。

(2) **acquire\_mut 方法**: 这个方法用于获取对受保护数据的可变引用。它首先检查 free 字段的状态，如果锁被占用（即 free 为 false），则进入等待循环直到锁变为可用。一旦 free 为 true，方法将其设置为 false 并返回受保护数据的可变引用。这种方式确保了在任何时刻，只有一个线程可以修改受保护的数据。

(3) **acquire 方法**: 与 acquire\_mut() 方法类似，这个方法提供对受保护数据的不可变引用。它同样会检查 free 字段，等待直到锁变为可用，然后将其设置为占用状态。这个方法是为了那些只需要读取而不需要修改数据的场景设计的，它保证了数据在读取期间不会被其他线程修改。

(4) **free 方法**: 这个方法用于释放 Mutex。当数据访问完成后，调用此方法将 free 字段设置回 true，表示锁现在是空闲的，其他线程可以尝试获取这个锁。这是保证多线程程序正确运行的关键环节，防止了死锁的发生并提高了资源的利用效率。

```

1  impl<T> Mutex<T> {
2      pub const fn new(value: T) -> Self {
3          Self {
4              target: value,
5              free: AtomicBool::new(true),
6          }
7      }
8
9      pub fn acquire_mut(&mut self) -> &mut T {
10         while !self.free.load(Ordering::SeqCst) {}
11         self.free.store(false, Ordering::SeqCst);
12         &mut self.target
13     }
14
15     pub fn acquire(&mut self) -> &T {
16         while !self.free.load(Ordering::SeqCst) {}
17         self.free.store(false, Ordering::SeqCst);
18         &self.target
19     }
20
21     pub fn free(&self) {
22         self.free.store(true, Ordering::SeqCst);
23     }
24 }
```

代码 6.2 Mutex 数据结构实现

### 6.3.3 Mutex 库作用

在 MinmusOS 内核中，Mutex 库扮演了一个至关重要的角色，主要通过提供基本的线程同步机制来保证内核的稳定性和效率。以下是 Mutex 在 MinmusOS 中的重要意义：

(1) **数据一致性和完整性**: 在多线程环境中，线程经常需要访问和修改共享资源。Mutex 库确保在任一时刻，只有一个线程可以访问这些资源，从而避免了数据的竞争条件和不一致性。这对于操作系统内核尤为重要，因为内核需要管理和维护核心的数据结构，如进程表、文件系统缓存等。

(2) **系统稳定性**: 通过有效的锁机制, Mutex 库帮助内核维持操作稳定性。防止了多个线程同时修改同一资源可能导致的系统崩溃或不可预期的行为。此外, 正确的锁策略还可以预防死锁的情况, 保持系统的高效运行。

(3) **性能优化**: Mutex 的实现通常尽量减少锁的持有时间, 以及优化等待锁的线程的调度。这有助于减少线程阻塞和上下文切换的开销, 从而提升系统整体的性能。

(4) **设计简洁性**: 提供一个简单而通用的 Mutex 实现使得内核代码更加清晰和易于维护。开发者可以重复使用这一同步原语, 而无需为每个需要同步的场景重新发明轮子。这降低了代码的复杂性, 同时提高了开发效率和代码的可重用性。

总体而言, Mutex 库为 MinmusOS 提供了一个关键的同步工具, 它是实现多线程安全访问共享资源的基石, 对保证操作系统的整体安全性、稳定性和性能至关重要。

## 6.4 print 库

print 库 (代码 6.3) 为 MinmusOS 操作系统内核提供了基础的输出功能, 使得开发者能够在底层系统级别上打印和管理输出信息。这个库通过实现 Printer 数据结构和相关的 print 及 println 宏, 简化了字符串的格式化和输出过程。通过将 Rust 的强大宏系统与操作系统的中断服务结合使用, print 库不仅支持基础的字符串输出, 还允许复杂的格式化操作, 极大地增强了开发和调试的效率。此外, 它的实现方式考虑了系统资源的直接管理和优化, 确保了输出操作的高效性和安全性, 使其成为系统级编程中不可或缺的工具。

### 6.4.1 Printer 数据结构

Printer 数据结构在这个实现中是一个简单的空结构体。它的主要目的是提供一个具体的实体来实现字符串的输出功能。在操作系统级别的编程中, 通常不会涉及到复杂的对象状态管理, 因此, 一个空结构体已足够用于封装所需的输出功能。

Printer 结构体实现了 `fmt::Write` trait, 这是 Rust 标准库提供的一个 trait, 用于定义一个可以输出字符串的类型。具体来说, `fmt::Write` trait 要求实现一个名为 `write_str` 的方法, 该方法接收一个字符串切片并返回一个 `fmt::Result`。

`write_str` 方法是 `fmt::Write` trait 的核心实现。在 Printer 结构体中, `write_str` 方法调用了内部的 `prints` 方法来实际输出字符串, 然后返回 `Ok(())` 表示成功。这样的设计允许 Printer 结构体通过 Rust 的格式化宏 (如 `write!` 或 `format!`) 进行输出。

`prints` 方法是 Printer 的自定义方法, 用于将字符串输出到底层系统。方法内部使用了不安全代码块, 因为它涉及到直接的内存地址访问和汇编指令调用。具体步骤包括:

#### (1) 寄存器推栈

- ① `push eax`: 将 `eax` 寄存器的当前值推入堆栈。`eax` 通常用于在系统调用中传递功能号或返回值。

```

1  use core::fmt;
2
3  pub static mut PRINTER: Printer = Printer {};
4
5  pub struct Printer {}
6
7  impl fmt::Write for Printer {
8      fn write_str(&mut self, s: &str) -> fmt::Result {
9          self.prints(s);
10         Ok(())
11     }
12 }
13
14 impl Printer {
15     pub fn prints(&self, s: &str) {
16         unsafe {
17             let ptr: *const u8 = s.as_ptr();
18             let len: usize = s.len();
19             core::arch::asm!(
20                 "push eax",
21                 "push ebx",
22                 "push ecx",
23                 "int 0x80",
24                 "pop ecx",
25                 "pop ebx",
26                 "pop eax",
27                 "in(\"eax\") 0",
28                 "in(\"ebx\") ptr as u32",
29                 "in(\"ecx\") len as u32",
30             );
31         }
32     }
33 }
34
35 pub fn _print(args: fmt::Arguments) {
36     use core::fmt::Write;
37     unsafe {
38         PRINTER.write_fmt(args).unwrap();
39     }
40 }
41
42 #[macro_export]
43 macro_rules! print {
44     ...
45 }
46
47 #[macro_export]
48 macro_rules! println {
49     ...
50 }

```

代码 6.3 lib/src/print.rs

② `push ebx`: 将 ebx 寄存器的当前值推入堆栈。在这段代码中，ebx 用于传递字符串的地址。

③ `push ecx`: 将 ecx 寄存器的当前值推入堆栈。ecx 用于传递字符串的长度。

#### (2) 中断调用

① `int 0x80`: 这条指令触发编号为 0x80 的中断，这通常是 UNIX 和 Linux 系统中用于系统调用的中断。通过此中断，操作系统可以执行用户请求的服务。

#### (3) 设置寄存器值

① `in("eax") 0`: 将 eax 寄存器设置为 0。

② `in("ebx") ptr as u32`: 将 ptr (字符串的地址) 转换为 u32 类型，并设置给 ebx 寄存器。

③ `in("ecx") len as u32`: 将 len (字符串的长度) 转换为 u32 类型，并设置给 ecx 寄存器。

#### (4) 寄存器出栈

① `pop ecx`: 从堆栈中弹出一个值到 ecx 寄存器，恢复其原始的上下文值。

② `pop ebx`: 从堆栈中弹出一个值到 ebx 寄存器，恢复其原始的上下文值。

③ `pop eax`: 从堆栈中弹出一个值到 eax 寄存器，恢复其原始的上下文值。

此汇编代码段利用了操作系统的系统调用接口来执行打印操作，通过堆栈操作确保了寄存器的原始值在调用结束后能够恢复，保证了调用的安全性和稳定性。这种方式允许操作系统内核代码直接与硬件交互，实现了在内核级别的基本输出功能。

### 6.4.2 print 宏

print 宏在 Rust 中是一种非常强大的工具，允许开发者通过简洁的语法进行复杂的代码扩展。在 MinmusOS 中，print 宏的实现展示了如何在底层系统级别使用 Rust 的宏系统来创建用户友好的打印接口。

\_print 函数是 print 宏背后的核心函数，其参数 args 是一个 `fmt::Arguments` 类型，这是一个由 `format_args!` 宏生成的结构体，它包含了所有需要格式化的参数。这个结构体使得参数能够在运行时被有效地处理和格式化。print 宏通过 `macro_rules!` 定义，这允许它接受不确定数量的参数，这些参数通过 `format_args!` 宏被转换成 `fmt::Arguments` 结构体。

这种实现使得 print 宏不仅在开发过程中为调试提供便利，还能在最终产品中用于日志记录和用户交互信息的输出。通过将复杂的格式化操作封装在简单的宏调用后，大大降低了开发者在系统级编程时的工作量。

### 6.4.3 println 宏

println 宏在 Rust 中是用于打印输出后自动换行的宏，它在 print 宏的基础上增加了换行功能。

`println` 宏通过 `macro_rules!` 被定义为两种形式，以处理不同的调用情况：

(1) **无参数调用**：当 `println` 宏没有任何参数时，它直接打印一个换行符。

(2) **带参数调用**：当提供参数时，宏首先使用 `print` 宏来处理和格式化这些参数，然后再次调用 `PRINTER.prints` 方法来添加一个换行符。这种方式使得宏在输出完指定的内容后能自动开始新的一行。

通过将 `print` 宏的功能扩展到自动添加换行符，`println` 宏极大地简化了日常编程中的打印任务。它允许开发者不必在每次打印后手动添加换行代码，提高了代码的可读性和编写效率。

`println` 宏尤其适用于输出日志信息、调试信息以及任何需要结构化显示和清晰分隔的输出场景。在系统开发和调试阶段，能够清晰地看到每条信息的起始和结束，极大地帮助了问题的定位和修复。

## 6.5 rand 库

MinmusOS 的 `lib/src/rand.rs` 文件为该操作系统提供了一系列的随机数生成工具，以支持系统和应用程序的需求。通过实现多种随机数生成算法，该库确保了足够的灵活性和应用广泛性，从简单的线性同余发生器到复杂的梅森旋转算法，都为不同的应用场景提供了优化的解决方案。这些算法支持从基本的伪随机序列生成到加密级的随机数生成，满足操作系统级的多样化需求。

以下是 `rand` 库中包含的八种随机数生成算法：

- (1) 线性同余生成器 (Linear Congruential Generator)
- (2) 异或移位算法 (Xorshift)
- (3) 中平方方法 (Middle Square Method)
- (4) 斐波那契线性同余生成器 (Fibonacci Linear Congruential Generator)
- (5) 梅森旋转算法 (Mersenne Twister)
- (6) 时间种子生成器 (Time Seed Generator)
- (7) 线性反馈移位寄存器 (Linear Feedback Shift Register)
- (8) 组合生成器 (Combined Generator)

### 6.5.1 线性同余生成器 (Linear Congruential Generator)

线性同余生成器 (Linear Congruential Generator, LCG) (代码 6.4) 是一种简单但广泛使用的随机数生成算法，其原理基于线性方程。这种生成器的核心在于一个递归公式，其中每个新的随机数都是基于前一个数计算得来。具体来说，该算法通过特定的增量、乘数和模数来更新状态，从而产生一个伪随机数序列。在 MinmusOS 中实现的线性同余生成器设定了固定的乘数 ( $a=1664525$ )、增量 ( $c=1013904223$ ) 和模数 ( $m=2^{32}$ )，这些值的选择关系到生成随机数序列的质量和周期长度。

在使用上，线性同余生成器首先通过一个初始种子值来设定起始状态，之后每次调用 `next` 方法都会根据上述公式计算出下一个随机数。这种方法的优势在于其实现简单，运算速度快，适用于需

要大量随机数的场合。然而，它的缺点也很明显，包括可能的周期性和预测性，尤其是当参数选择不当时。

尽管存在这些限制，线性同余生成器仍然是许多系统和应用中的首选，因为它提供了一种权衡性能和复杂度的有效方式。在 MinmusOS 中，这个算法被用来支持各种系统级和应用级功能，例如内存管理、进程调度等，其中对随机性的需求较为宽松。总的来说，线性同余生成器是一个在实践中表现良好的工具，尤其是在资源有限的环境中。

```
1 pub struct LCG {
2     state: u32,
3     a: u32,
4     c: u32,
5     m: u32,
6 }
7
8 impl LCG {
9     pub fn new(seed: u32) -> Self {
10         LCG {
11             state: seed,
12             a: 1664525,
13             c: 1013904223,
14             m: 2_u32.pow(32),
15         }
16     }
17
18     pub fn next(&mut self) -> u32 {
19         self.state = (self.a.wrapping_mul(self.state).wrapping_add(self.c)) % self.m;
20         self.state
21     }
22 }
```

代码 6.4 线性同余生成器

### 6.5.2 异或移位算法（Xorshift）

异或移位算法（Xorshift）（代码 6.5）是一种基于位运算的快速伪随机数生成器，由 George Marsaglia 提出。该算法的核心在于通过几次简单的异或（XOR）和位移（shift）操作来生成随机数，具有实现简单和运行高效的特点。在 MinmusOS 中，Xorshift32 结构体采用了 32 位无符号整数（u32）作为其状态，通过连续的位运算保证了生成随机数的速度和随机性。

具体到实现，算法首先从一个非零的种子开始，以保证随机数生成的连续性。然后，它执行一系列的异或和位移操作，这些操作包括将当前状态左移 13 位、右移 17 位，再左移 5 位，最后将这些结果再次异或。这些操作的选择不是随机的，而是基于实验和理论分析，以达到良好的随机分布和周期性。

由于其简单和高效的特性，Xorshift 算法在需要快速生成大量随机数的场景中非常有用，例如在模拟或游戏开发中。同时，这种算法也是研究随机数生成理论的一个良好案例，展示了简单操作

如何通过巧妙组合产生复杂的随机序列。

```

1 pub struct Xorshift32 {
2     state: u32,
3 }
4
5 impl Xorshift32 {
6     pub fn new(seed: u32) -> Self {
7         Xorshift32 {
8             state: seed,
9         }
10    }
11
12    pub fn next(&mut self) -> u32 {
13        let mut x: u32 = self.state;
14        x ^= x << 13;
15        x ^= x >> 17;
16        x ^= x << 5;
17        self.state = x;
18        x
19    }
20 }
```

代码 6.5 异或移位算法

### 6.5.3 中平方方法 (Middle Square Method)

中平方方法 (Middle Square Method) (代码 6.6) 是一种古老且直观的伪随机数生成算法，最初由冯·诺依曼在 1940 年代提出。该算法的基本思想是取一个数 (种子)，将其平方，然后再取中间的几位数作为下一个随机数，这个过程不断重复。在 MinmusOS 中，这个算法通过 MiddleSquare 结构体实现，使用 32 位无符号整数 (u32) 来存储当前状态。

具体实现中，MiddleSquare 首先接受一个初始种子值来设置起始状态。随机数的生成过程中，首先计算当前状态的平方，然后通过位移和位掩码操作提取平方结果的中间部分作为新的随机数。在此实现中，通过右移 8 位并应用 16 位掩码 (0xFFFF)，从而获取中间 16 位。这种方法的优点是实现简单，但其缺点也很明显，包括可能的短周期和模式预测性，特别是当种子或中间位的选择不恰当时。

### 6.5.4 斐波那契线性同余生成器 (Fibonacci Linear Congruential Generator)

斐波那契线性同余生成器 (Fibonacci Linear Congruential Generator, FLCG) (代码 6.7) 是一种结合了线性同余算法和斐波那契序列特性的伪随机数生成器。这种生成器利用了两种算法的优点，通过线性同余方法的数学基础与斐波那契数列的自然递归特性，创造出一个更为复杂且周期更长的随机序列生成过程。

在 MinmusOS 中的实现中，FibonacciLCG 由两个状态 (state1 和 state2) 和一个模数 (m) 组成，初始状态通过两个种子值设定。随机数生成过程首先计算这两个状态的和，并取模得到新的随机

```
1 pub struct MiddleSquare {
2     state: u32,
3 }
4
5 impl MiddleSquare {
6     pub fn new(seed: u32) -> Self {
7         MiddleSquare {
8             state: seed,
9         }
10    }
11
12    pub fn next(&mut self) -> u32 {
13        let squared: u32 = self.state.wrapping_mul(self.state);
14        let middle: u32 = (squared >> 8) & 0xFFFF;
15        self.state = middle;
16        middle
17    }
18 }
```

代码 6.6 中平方方法

数。然后，状态更新，其中第一个状态变为第二个，而第二个状态则更新为新计算出的值。这种更新机制保证了生成的随机数序列具有斐波那契数列的递归特性，同时线性同余方法确保了数值的均匀分布。

斐波那契线性同余生成器的特点是它能够生成更加复杂的随机序列，相比传统的线性同余生成器具有更长的周期和更好的统计特性，这使得它在需要较高随机性的应用中非常有用，如模拟和密码学领域。尽管如此，算法的效率和周期性还是受到种子选择和模数设置的影响，因此在实际应用中需要仔细选择这些参数以确保最优的随机性表现。

### 6.5.5 梅森旋转算法 (Mersenne Twister)

梅森旋转算法 (Mersenne Twister) (代码 6.8) 是一种高度有效且广泛应用的伪随机数生成器，以其极长的周期和卓越的统计质量著称。这种算法以一个质数 (即梅森素数) 为基础，设计了复杂的位运算和数组索引技巧，确保随机数序列具有非常长的周期，同时保持良好的均匀分布特性。在 MinmusOS 系统中，Mersenne Twister 的实现通过维护一个 624 个元素的状态数组来实现，每个元素为 32 位无符号整数。

在初始化阶段，Mersenne Twister 从一个种子开始，利用数组的前一个元素来计算后一个元素的状态，这个过程结合了线性反馈和一些非线性变换。当生成随机数时，如果数组的所有元素都已被使用，则会执行一个名为 “twist” 的操作，这是算法的核心，用于重新生成整个状态数组，从而确保随机数生成的连续性。

梅森旋转算法的一个关键特性是它的 “twist” 操作，它通过特定的位操作和模运算将当前状态数组转换为新的状态数组，这些操作包括位的移动和异或。这确保了算法的输出不会快速重复，从而极大地增加了随机数序列的复杂度和随机性。

```

1 pub struct FibonacciLCG {
2     state1: u32,
3     state2: u32,
4     m: u32,
5 }
6
7 impl FibonacciLCG {
8     pub fn new(seed1: u32, seed2: u32) -> Self {
9         FibonacciLCG {
10             state1: seed1,
11             state2: seed2,
12             m: 2_u32.pow(32),
13         }
14     }
15
16     pub fn next(&mut self) -> u32 {
17         let next_value: u32 = (self.state1.wrapping_add(self.state2)) % self.m;
18         self.state1 = self.state2;
19         self.state2 = next_value;
20         next_value
21     }
22 }

```

代码 6.7 斐波那契线性同余生成器

尽管梅森旋转算法提供了优秀的统计性质和长周期，但它的内存占用相对较高，因此在内存受限的环境中可能不是最佳选择。然而，对于大多数科学计算、仿真和游戏开发等需要高质量随机数序列的应用，梅森旋转算法仍然是非常理想的选择。

### 6.5.6 时间种子生成器 (Time Seed Generator)

时间种子生成器 (Time Seed Generator) (代码 6.9) 是一种基于时间作为初始种子的随机数生成方法。在这种方法中，时间戳通常被用作种子值，以确保每次启动时生成的随机数序列具有唯一性，因为种子值依赖于实际的系统时间，这样每次启动或调用都不同。

在 MinmusOS 中实现的 TimeSeed 结构体采用一个 64 位的时间戳，通过截取其低 32 位作为初始状态。随后，通过一系列的位运算（位移和异或操作），来更新状态并生成新的随机数。这些位运算包括左移 13 位，右移 17 位，再左移 5 位，每次操作都通过异或将变换后的位重新混合，从而产生难以预测的结果。

这种基于时间种子的随机数生成器的主要优势在于其简单性和能够根据不同的时间产生不同的随机序列，使得其非常适合于需要简单随机性的应用，如临时密码生成或会话标识符的创建。然而，由于其依赖于系统时间，其随机性质并不适合于所有类型的应用，特别是那些需要高级随机性如加密应用。

总体而言，时间种子生成器提供了一种快速且方便的方式来生成随机数，但在安全性要求较高的场合可能需要更复杂的随机数生成方法以确保随机序列的不可预测性和安全性。

```
1 pub struct MersenneTwister {
2     state: [u32; 624],
3     index: usize,
4 }
5
6 impl MersenneTwister {
7     pub fn new(seed: u32) -> Self {
8         let mut mt = MersenneTwister {
9             state: [0; 624],
10            index: 624,
11        };
12        mt.state[0] = seed;
13        for i in 1..624 {
14            mt.state[i] = 0x6c078965_u32.wrapping_mul(mt.state[i - 1] ^ (mt.state[i - 1] >>
15                30)).wrapping_add(i as u32);
16        }
17        mt
18    }
19
20    pub fn next(&mut self) -> u32 {
21        if self.index >= 624 {
22            self.twist();
23        }
24        let mut y: u32 = self.state[self.index];
25        y ^= y >> 11;
26        y ^= (y << 7) & 0x9d2c5680;
27        y ^= (y << 15) & 0xefc60000;
28        y ^= y >> 18;
29        self.index += 1;
30        y
31    }
32
33    fn twist(&mut self) {
34        for i in 0..624 {
35            let y: u32 = (self.state[i] & 0x80000000) + (self.state[(i + 1) % 624] &
36                0xffffffff);
37            self.state[i] = self.state[(i + 397) % 624] ^ (y >> 1);
38            if y % 2 != 0 {
39                self.state[i] ^= 0x9908b0df;
40            }
41        }
42    }
}
```

代码 6.8 梅森旋转算法

```

1 pub struct TimeSeed {
2     state: u32,
3 }
4
5 impl TimeSeed {
6     pub fn new(start_time: u64) -> Self {
7         TimeSeed {
8             state: (start_time & 0xFFFFFFFF) as u32,
9         }
10    }
11
12    pub fn next(&mut self) -> u32 {
13        self.state ^= self.state << 13;
14        self.state ^= self.state >> 17;
15        self.state ^= self.state << 5;
16        self.state
17    }
18 }

```

代码 6.9 时间种子生成器

### 6.5.7 线性反馈移位寄存器 (Linear Feedback Shift Register)

线性反馈移位寄存器 (Linear Feedback Shift Register, LFSR) (代码 6.10) 是一种有效的数字逻辑结构，广泛用于生成伪随机数和实现数字通信系统中的信号加密。LFSR 通过移位寄存器的结构，结合选定的反馈位进行线性反馈来生成序列。这种方法的优点在于它可以非常高效地生成具有良好统计性质和较长周期的伪随机序列。

在 MinmusOS 中，LFSR 实现了一个基本的线性反馈移位寄存器，其中包含一个 32 位的状态寄存器。随机数的生成过程包括对状态寄存器进行位移操作，并从寄存器的不同位置取位进行异或运算，得到的结果决定了反馈位。具体到这个实现，反馈位是通过状态寄存器的第 0 位、第 2 位、第 3 位和第 5 位进行异或运算得到的。然后，整个寄存器向右移动一位，新计算出的反馈位被置于寄存器的最高位 (第 31 位)。

这种生成器的一个重要特性是其周期和生成的序列质量依赖于反馈位的选择和初始种子的值。合适的反馈策略可以使得 LFSR 生成的序列达到最大周期，且输出序列在很多应用场合下近似于随机。LFSR 在实际应用中非常灵活，常见于无线通信和加密算法中，例如流密码和硬件加速的随机数生成。

### 6.5.8 组合生成器 (Combined Generator)

组合生成器 (Combined Generator) (代码 6.11) 是一种采用多个随机数生成算法合并输出以增强随机性和复杂性的方法。这种生成器通常结合两种或更多的算法，目的是利用各自算法的优点，同时降低各个算法潜在弱点的影响，以生成更难预测和统计上更均匀的随机数序列。

在 MinmusOS 中实现的 CombinedGenerator 结构体结合了线性同余生成器 (LCG) 和异或移位算

```
1 pub struct LFSR {
2     state: u32,
3 }
4
5 impl LFSR {
6     pub fn new(seed: u32) -> Self {
7         LFSR {
8             state: seed,
9         }
10    }
11
12    pub fn next(&mut self) -> u32 {
13        let bit: u32 = ((self.state >> 0) ^ (self.state >> 2) ^ (self.state >> 3) ^
14            (self.state >> 5)) & 1;
15        self.state = (self.state >> 1) | (bit << 31);
16        self.state
17    }
}
```

代码 6.10 线性反馈移位寄存器

法 (Xorshift32)。这种组合允许利用 LCG 的数学基础和 Xorshift 的高效位运算，通过各自独立的随机序列生成并使用异或 (XOR) 运算合并结果，从而提高整体的随机性。具体操作中，CombinedGenerator 首先使用两个不同的种子值初始化两个独立的生成器，然后在每次调用 next 方法时，分别从这两个生成器获取一个随机数，并通过异或操作合并这两个随机数，输出最终的随机值。

这种组合方法的优势在于它能够有效地抵消单一生成器可能存在的周期性短、模式可预测等问题，同时保持生成速度和效率。此外，通过适当选择和配置内部生成器，可以进一步优化输出序列的随机性和性能。

```
1 pub struct CombinedGenerator {
2     gen1: LCG,
3     gen2: Xorshift32,
4 }
5
6 impl CombinedGenerator {
7     pub fn new(seed1: u32, seed2: u32) -> Self {
8         CombinedGenerator {
9             gen1: LCG::new(seed1),
10            gen2: Xorshift32::new(seed2),
11        }
12    }
13
14    pub fn next(&mut self) -> u32 {
15        let r1: u32 = self.gen1.next();
16        let r2: u32 = self.gen2.next();
17        r1 ^ r2
18    }
19 }
```

代码 6.11 组合生成器

## 6.6 sort 库

lib/src/sort.rs 文件是 MinmusOS 标准运行库的一个组成部分，包含多种排序算法的实现，这些排序算法可以对整数数组进行高效排序，是操作系统或其他系统级软件中常见的基础算法。通过 Rust 语言的功能，这个库确保了代码的安全性和效率，适合在资源受限的环境中使用。

以下是 sort 库中包含的十种排序算法：

- (1) 冒泡排序 (Bubble Sort)
- (2) 选择排序 (Selection Sort)
- (3) 插入排序 (Insertion Sort)
- (4) 合并排序 (Merge Sort)
- (5) 快速排序 (Quick Sort)
- (6) 堆排序 (Heap Sort)
- (7) 希尔排序 (Shell Sort)
- (8) 计数排序 (Counting Sort)
- (9) 桶排序 (Bucket Sort)
- (10) 基数排序 (Radix Sort)

sort 库的实现较为基础，排序算法（代码 6.12）的实现过程在此不再赘述。

```

1 pub fn bubble_sort(arr: &mut [i32]) { ... }
2 pub fn selection_sort(arr: &mut [i32]) { ... }
3 pub fn insertion_sort(arr: &mut [i32]) { ... }
4 pub fn merge_sort(arr: &mut [i32]) { ... }
5 pub fn quick_sort(arr: &mut [i32]) { ... }
6 pub fn heap_sort(arr: &mut [i32]) { ... }
7 pub fn shell_sort(arr: &mut [i32]) { ... }
8 pub fn counting_sort(arr: &mut [i32]) { ... }
9 pub fn bucket_sort(arr: &mut [i32]) { ... }
10 pub fn radix_sort(arr: &mut [i32]) { ... }
```

代码 6.12 lib/src/sort.rs

### 6.6.1 冒泡排序 (Bubble Sort)

冒泡排序是一种简单的排序算法，通过重复遍历待排序列，比较每对相邻元素，如果顺序错误就交换它们。这个过程重复进行，直到没有再需要交换的元素，此时列表已经排序完成。尽管其实现简单，但效率较低，主要用于教学和理解排序原理。

### 6.6.2 选择排序 (Selection Sort)

选择排序是通过不断选择未排序部分的最小元素，并将其放到已排序序列的末尾来实现排序的算法。这个过程重复进行，直到所有元素均排序完毕。该方法不稳定，时间复杂度为  $O(n^2)$ ，适用

于元素数量较少的情况。

### 6.6.3 插入排序 (Insertion Sort)

插入排序将每一个数据插入到其在已排序列中的适当位置。它从第二个元素开始，逐个将未排序的元素插入到已排序部分，直到所有元素都被正确排序。插入排序在接近排序完成的列表上表现良好。

### 6.6.4 合并排序 (Merge Sort)

合并排序是一种有效的排序方法，采用分治策略将数据分解成越来越小的部分，然后合并以排序。该算法将数组分割至单一元素数组，然后将它们重新合并成有序数组。它具有  $O(n \log n)$  的时间复杂度，对大数据集合非常有效。

### 6.6.5 快速排序 (Quick Sort)

快速排序是一种高效的排序算法，通过选取一个“基准”元素并将数组分为比它小的和比它大的两个子数组，然后对这两个子数组再递归地应用相同的过程来实现排序。其平均时间复杂度为  $O(n \log n)$ ，但最坏情况下为  $O(n^2)$ 。

### 6.6.6 堆排序 (Heap Sort)

堆排序是基于优先队列概念的选择排序的一种改进。它使用二叉堆数据结构来管理数据，可以在  $O(\log n)$  时间内找到最大值或最小值。时间复杂度为  $O(n \log n)$ ，适用于大数据量排序。

### 6.6.7 希尔排序 (Shell Sort)

希尔排序是插入排序的一种泛化，也称为减小增量排序算法。它通过将原列表分割成多个子列表，各自独立排序，然后逐步综合这些子列表。这种方法提高了插入排序处理大规模数组时的速度，虽然最坏情况的时间复杂度仍为  $O(n^2)$ 。

### 6.6.8 计数排序 (Counting Sort)

计数排序是一个非基于比较的排序算法，适用于一定范围内的整数排序。通过计算每个元素的出现次数来实现排序。该算法在数据范围不是特别大且数据分布均匀时，能够提供线性的时间复杂度  $O(n)$ 。

### 6.6.9 桶排序 (Bucket Sort)

桶排序是计数排序的扩展，将元素分布到多个桶中，每个桶再分别排序（通常使用其他排序算法或递归应用桶排序）。它最适用于数据均匀分布的场景，可以在平均线性时间内完成排序，即

$O(n)$ 。

### 6.6.10 基数排序 (Radix Sort)

基数排序是通过按位数切割数字，从最低位开始，使用稳定排序算法（如桶排序）逐步排序的方法。适用于整数或字符串排序，尤其是当键值的长度固定时，能够提供近线性的排序时间  $O(nk)$ （其中  $k$  是键值的位数）。

## 6.7 string 库

MinmusOS 实现了一个字符串 string 库，以提供基本的字符串操作功能。这个库覆盖了从基本的字符串拼接到更复杂的比较和搜索操作。

表 6.2 列出了 string 库中的各个函数以及它们的功能描述。

表 6.2 string 库函数

函数名	功能
strlen	计算字符串的长度，直到 0
strcat	将 s2 拼接到 s1 后面
strncat	将 s2 的前 len 个字符拼接到 s1 后面
strcpy	复制 s2 到 s1
strncpy	复制 s2 的前 len 个字符到 s1
strcmp	比较两个字符串
strcasecmp	不区分大小写地比较两个字符串
strncmp	比较两个字符串的前 len 个字符
strcasencmp	不区分大小写地比较两个字符串的前 len 个字符
strupr	将字符串 s 中的小写字母转换为大写
strlwr	将字符串 s 中的大写字母转换为小写
strchr	查找字符 ch 在字符串 s 中的位置
strstr	查找子字符串 substr 在字符串 s 中的位置
strrchr	查找字符 ch 在字符串 s 中的最后位置
strrstr	查找子字符串 substr 在字符串 s 中的最后位置
strrev	将字符串 s 反转

## 7 应用程序实现

在本节（章节 7）中，笔者将详细探讨在 MinmusOS 上实现的一个示例应用程序——汉诺塔解决方案。这个示例应用程序不仅展示了基于 Rust 的系统级编程实践，也具体演示了如何在裸机或类似环境下运行复杂的应用程序。

### 7.1 应用程序构建脚本

apps/hanoi/build.rs 构建脚本（代码 7.1）的主要功能是为 Rust 编译器配置特定的链接器脚本，以便正确地编译和链接汉诺塔应用程序。以下是详细介绍：

(1) 环境变量读取：脚本使用 `env!("CARGO_MANIFEST_DIR")` 来获取当前包的清单（Manifest）目录。这个环境变量是由 Cargo 设置的，指向你的项目的根目录，即包含 `Cargo.toml` 的目录。

(2) 路径拼接：通过 `std::path::Path::new` 将获取到的目录转换为一个路径对象，然后使用 `join` 方法拼接上 `linker.ld`。这样操作是为了构造出链接器脚本的完整路径。

(3) 链接器参数设置：脚本使用 `println!` 输出一个特殊的编译器指令，这条指令会被 Cargo 捕捉并用于配置编译过程。具体来说，`cargo:rustc-link-arg-bins` 告诉 Rust 编译器在编译二进制文件时应该使用指定的链接器脚本。

```
1 fn main() {
2     let local_path = std::path::Path::new(env!("CARGO_MANIFEST_DIR"));
3     println!("cargo:rustc-link-arg-bins==script={}{}", local_path.join("linker.ld").display());
4 }
```

代码 7.1 apps/hanoi/build.rs

在操作系统内核中实现应用程序时，构建脚本起到的关键作用是确保应用程序能够与内核适当链接，以便在内核的上下文中正确执行。使用自定义的链接器脚本，可以精确控制程序的内存布局、符号解析等关键方面，这些都是在裸机或自定义操作系统上运行代码的必要条件。通过这种方式，可以确保应用程序的代码和数据被放置在内核预期的特定位置，使得程序能够在没有标准操作系统支持的环境中运行。

### 7.2 应用程序链接器脚本

应用程序的链接器脚本（代码 7.2）是定义程序在内存中的布局的关键文件。该脚本指定了程序的各个部分应该放在内存的哪个位置，以及如何组织这些部分。

### 7.3 应用程序入口

代码 7.3 是应用程序与操作系统交互的起点，尤其是在裸机环境或自定义操作系统如 MinmusOS 中。这里的实现详细说明了如何初始化程序、处理异常，并确保程序可以在没有标准库支持的环境

```

1  /* 配置程序入口点 */
2  ENTRY(_start)
3
4  /* 配置段的顺序和位置 */
5  SECTIONS {
6      /* 定义起始内存地址 */
7      . = 0x02000000;
8
9      /* 定义应用程序开始标记 */
10     .start_marker :
11     {
12         LONG(0xB16B00B5)
13     }
14
15     /* 定义应用程序起始点 */
16     _app_start = .;
17
18     /* 定义启动段, 包含启动代码的实际入口点 */
19     .start : {
20         *(.start)
21     }
22
23     /* 定义代码段, 包含程序的机器代码 */
24     .text : {
25         *(.text .text.*)
26     }
27
28     /* 定义 BSS 段, 包含程序中未初始化的数据 */
29     .bss : {
30         *(.bss .bss.*)
31     }
32
33     /* 定义只读数据段, 包含常量等不应被程序修改的数据 */
34     .rodata : {
35         *(.rodata .rodata.*)
36     }
37
38     /* 定义数据段, 包含已初始化的全局变量和静态变量 */
39     .data : {
40         *(.data .data.*)
41     }
42
43     /* 配置异常处理信息, 用于支持运行时错误处理 */
44     .eh_frame : {
45         *(.eh_frame .eh_frame.*)
46     }
47     .eh_frame_hdr : {
48         *(.eh_frame_hdr .eh_frame_hdr.*)
49     }
50
51     /* 在内存中设置一个结束标记, 用于标识引导加载器的结束 */
52     .end_marker :
53     {
54         SHORT(0xDEAD)
55     }
56 }
```

代码 7.2 apps/hanoi/linker.ld

下运行。下面是各部分的具体介绍：

```
1  #![no_std]
2  #![no_main]
3
4  use core::panic::PanicInfo;
5
6  fn main() {
7      ...
8  }
9
10 #[no_mangle]
11 #[link_section = ".start"]
12 pub extern "C" fn _start() {
13     main();
14     loop {}
15 }
16
17 #[panic_handler]
18 fn panic(_info: &PanicInfo) -> ! {
19     loop {}
20 }
```

代码 7.3 apps/hanoi/src/main.rs

### (1) 属性指令

- ① **#![no\_std]**: 这条属性禁用 Rust 标准库。因为在裸机或内核模块中，标准库的某些部分（如堆内存分配、线程等）可能不可用。
- ② **#![no\_main]**: 通常 Rust 程序从标准的 main 函数开始执行。这个属性指示编译器程序将不从传统的 main 函数入口开始，这是因为在操作系统或裸机环境中，入口点通常是自定义的。

### (2) 入口函数\_start

- ① **#[no\_mangle]**: 此属性用于防止编译器更改函数名称（即避免名称改编），确保链接时可以正确识别该函数。
- ② **#[link\_section = ".start"]**: 将此函数放置在特定的内存段，这里是.start 段，正如链接器脚本中指定的。
- ③ **pub extern "C"**: 表示这个函数应遵循 C 语言的函数调用约定，这对于与其他语言或运行时环境交互是必要的。
- ④ **main()**: 函数体内调用 **main()**，之后进入无限循环，这样做是为了防止程序执行完毕后继续向下执行到未定义的内存区域。

### (3) Panic 处理

- ① **#[panic\_handler]**: 这个属性定义了一个函数，用于处理程序运行中遇到的 panic 情况（即程序中的不可恢复错误）。
- ② **panic**: 这个函数接收一个关于 panic 的信息对象，返回类型为 **!**，表示这个函数不会返

回（即终止函数）。函数体中的无限循环确保在 panic 发生后，程序不会无序退出或执行未定义操作。

此部分代码是整个程序能够在特定硬件或自定义操作系统上运行的基础。它确保了程序在启动时能够按照预定的方式初始化，同时在运行过程中遇到严重错误时有定义良好的行为。在不使用标准库的情况下，开发者需要手动处理所有的初始化和异常情况。

## 7.4 应用程序实现

这个汉诺塔解决方案应用程序是为 MinmusOS 操作系统内核编写的，完全使用 Rust 语言实现，不依赖于标准库，这意味着它能在裸机或极为简化的环境中运行。在程序的主函数中，首先初始化一个 HanoiTowers 结构体（代码 7.4），这个结构体包含三个数组，分别代表三根柱子，以及记录顶部位置和移动次数的变量。程序开始时，将从 1 到 STACK\_SIZE 的数字依次放入数组 a，代表柱子 A 上从小到大的排列的圆盘。

```

1 struct HanoiTowers {
2     a: [i32; STACK_SIZE],
3     b: [i32; STACK_SIZE],
4     c: [i32; STACK_SIZE],
5     a_top: usize,
6     b_top: usize,
7     c_top: usize,
8     count: usize,
9 }
```

代码 7.4 HanoiTowers 数据结构

```

1 fn main() {
2     let mut towers = HanoiTowers {
3         a: [0; STACK_SIZE],
4         b: [0; STACK_SIZE],
5         c: [0; STACK_SIZE],
6         a_top: 0,
7         b_top: STACK_SIZE,
8         c_top: STACK_SIZE,
9         count: 0,
10    };
11    for i in 0..STACK_SIZE {
12        towers.a[i] = (i + 1) as i32;
13    }
14    println!("Hanoi Tower with {} disks:", STACK_SIZE);
15    print!("#{:>4}      ", towers.count);
16    print_stacks(&towers);
17    move_disks(STACK_SIZE as i32, 'A', 'C', 'B', &mut towers);
18 }
```

代码 7.5 主函数

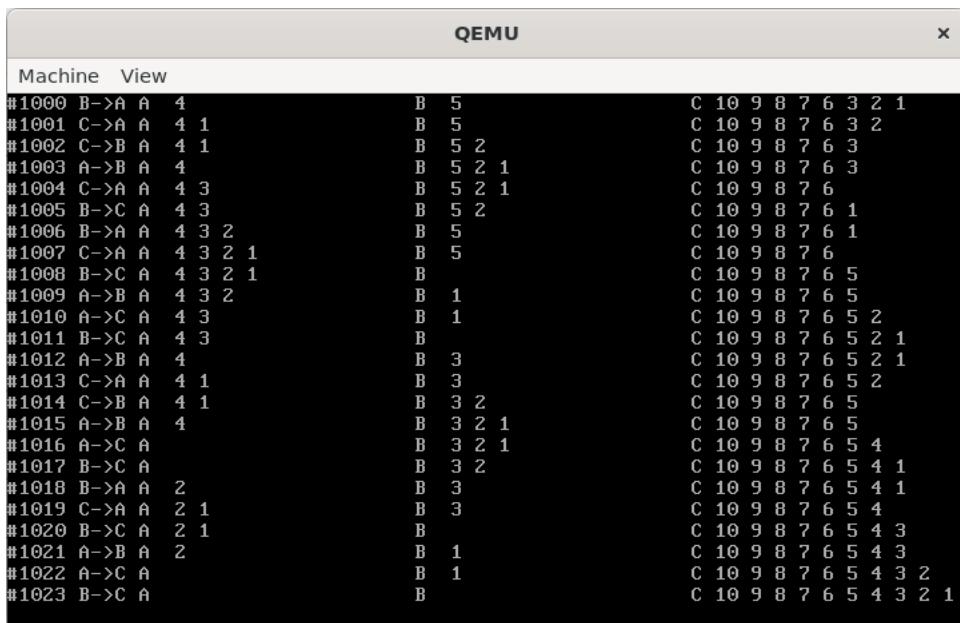
```
1 fn move_disks(n: i32, from: char, to: char, aux: char, towers: &mut HanoiTowers) {
2     if n == 1 {
3         move_one_disk(from, to, towers);
4     } else {
5         move_disks(n - 1, from, aux, to, towers);
6         move_one_disk(from, to, towers);
7         move_disks(n - 1, aux, to, from, towers);
8     }
9 }
10
11 fn move_one_disk(from: char, to: char, towers: &mut HanoiTowers) {
12     towers.count += 1;
13     print!("{:>4} {}->{}", towers.count, from, to);
14     transfer_disk(from, to, towers);
15     print_stacks(towers);
16 }
17
18 fn transfer_disk(from: char, to: char, towers: &mut HanoiTowers) {
19     let (from_stack, to_stack, from_top, to_top) = match (from, to) {
20         ('A', 'B') => (&mut towers.a, &mut towers.b, &mut towers.a_top, &mut towers.b_top),
21         ('A', 'C') => (&mut towers.a, &mut towers.c, &mut towers.a_top, &mut towers.c_top),
22         ('B', 'A') => (&mut towers.b, &mut towers.a, &mut towers.b_top, &mut towers.a_top),
23         ('B', 'C') => (&mut towers.b, &mut towers.c, &mut towers.b_top, &mut towers.c_top),
24         ('C', 'A') => (&mut towers.c, &mut towers.a, &mut towers.c_top, &mut towers.a_top),
25         ('C', 'B') => (&mut towers.c, &mut towers.b, &mut towers.c_top, &mut towers.b_top),
26         _ => return,
27     };
28     if *from_top < STACK_SIZE {
29         let disk: i32 = from_stack[*from_top];
30         from_stack[*from_top] = 0;
31         *from_top += 1;
32         if *to_top > 0 {
33             *to_top -= 1;
34             to_stack[*to_top] = disk;
35         }
36     }
37 }
38
39 fn print_stacks(towers: &HanoiTowers) {
40     print!(" A ");
41     for a in (0..STACK_SIZE).rev() {
42         if a >= towers.a_top {
43             print!("{:>2}", towers.a[a]);
44         } else {
45             print!("   ");
46         }
47     }
48     print!(" B ");
49     ...
50     print!(" C ");
51     ...
52     println!();
53 }
```

代码 7.6 工具函数

代码 7.6 实现了汉诺塔解决方案应用程序的工具函数。程序调用 `move_disks` 函数，这是一个递归函数，负责按照汉诺塔的规则移动圆盘，即每次只移动一个圆盘，并且任何时候较大的圆盘不能位于较小的圆盘之上。每次移动圆盘都会调用 `move_one_disk` 函数，这个函数更新圆盘的位置，记录移动次数，并调用 `print_stacks` 函数打印当前柱子的状态。`transfer_disk` 函数则是处理实际的圆盘从一个数组到另一个数组的转移，确保不违反汉诺塔的移动规则。

程序的入口点是 `_start` 函数，这是为了符合操作系统的启动要求，该函数将直接调用 `main` 函数（代码 7.5），并在执行完毕后进入无限循环，防止程序退出到未定义的代码。异常处理由 `panic_handler` 实现，当程序遇到不可恢复的错误时，它将进入另一个无限循环，确保系统不会无序地崩溃。整体而言，这个程序不仅是对经典汉诺塔问题的一个解决方案实现，也展示了在自定义操作系统中如何处理底层的程序逻辑和异常情况。

图 7.7 展示了汉诺塔解决方案应用程序的运行。



The screenshot shows the QEMU Machine View window. The title bar says "QEMU". The main area is titled "Machine View" and contains two columns of text. The left column lists moves from stack A to stack B, and the right column shows the resulting state of stacks A, B, and C. The moves are numbered #1000 through #1023. The final state is:

Stack	10	9	8	7	6	5	4	3	2	1
C	10	9	8	7	6	3	2			
B	10	9	8	7	6	3	2			
A	10	9	8	7	6	5	4	3	2	1

图 7.7 应用程序汉诺塔解决方案演示

## 8 系统演示

在本节（章节 8）中，笔者将演示 MinmusOS 几个关键功能和组件，旨在展示 MinmusOS 的实际运行状态。以下是具体的演示内容：

- (1) **内核启动演示**: 展示操作系统从引导到完全启动的整个过程。这包括内核初始化硬件设备、设置内存管理和加载系统服务等关键步骤。
- (2) **键盘驱动程序演示**: 演示键盘驱动程序如何处理输入，包括字符的捕获、特殊按键的响应以及键盘中断的处理。
- (3) **命令行解释器 20 条可执行命令演示**: 命令行解释器是用户与操作系统交互的主要界面。本演示将展示 20 条基本命令的执行。
- (4) **命令行解释器错误提示演示**: 演示当用户输入错误或执行不当操作时，命令行解释器如何提供错误反馈和帮助信息，以帮助用户更正操作或理解命令的正确用法。
- (5) **示例文本文件演示**: 这部分将通过操作五个示例文本文件，展示文件系统的读能力。
- (6) **应用程序汉诺塔解决方案演示**: 演示一个简单的汉诺塔解决方案应用程序如何在 MinmusOS 上运行，展示系统处理递归算法和用户交互的能力。
- (7) **异常处理器演示**: 展示操作系统如何处理各种异常情况，如非法内存访问、除零错误以及默认异常情况等。
- (8) **PANIC 处理器演示**: 展示 PANIC 处理器的工作过程，这是系统在遇到无法恢复的错误时的最后手段。展示系统如何响应致命错误，保护数据安全，并尽可能地提供错误信息和恢复选项（这里包括索引越界和手动触发两种 PANIC）。



图 8.1 内核启动演示

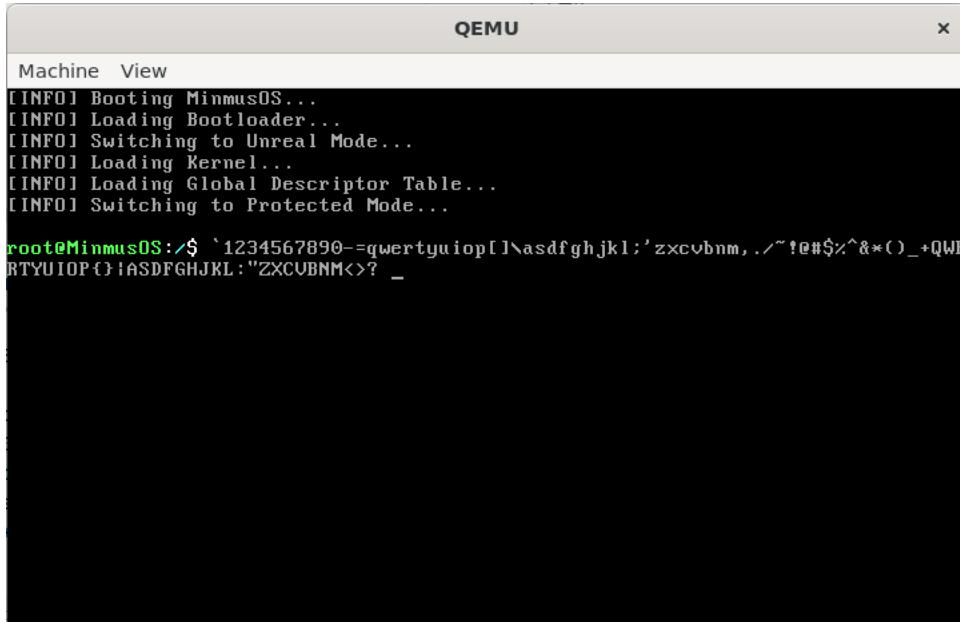


图 8.2 键盘驱动程序演示



图 8.3 help 命令演示

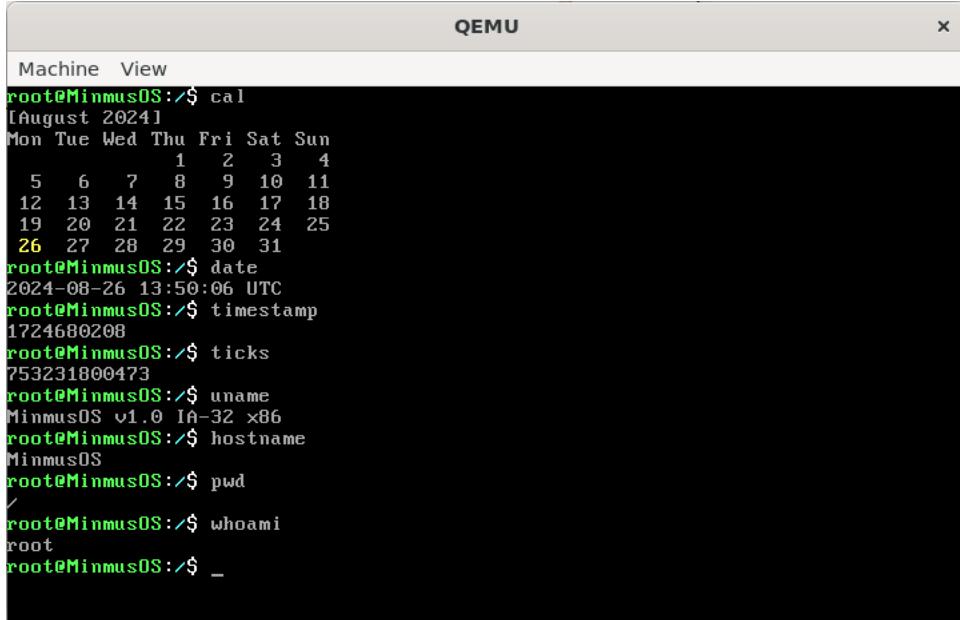


图 8.4 clear 命令演示

A screenshot of a QEMU terminal window titled "QEMU". The window has a menu bar with "Machine" and "View". The terminal prompt is "root@MinmusOS:~\$ ". The screen displays a grid of hex values from 0x00 to 0xFF, each followed by its corresponding color swatch. The colors range from black (0x00) to white (0xFF).

	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	
10	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	0x00	
20	21	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F	0x01	
30	31	32	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F	0x02	
40	41	42	43	45	46	47	48	49	4A	4B	4C	4D	4E	4F	0x03	
50	51	52	53	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F	0x04
60	61	62	63	64	65	67	68	69	6A	6B	6C	6D	6E	6F	0x05	
70	71	72	73	74	75	76	78	79	7A	7B	7C	7D	7E	7F	0x06	
80	81	82	83	84	85	86	87	88	8A	8B	8C	8D	8E	8F	0x07	
90	91	92	93	94	95	96	97	98	9A	9B	9C	9D	9E	9F	0x08	
A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	AB	AC	AD	AE	AF	0x09	
B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	BA	BC	BD	BE	BF	0x0A	
C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	CA	CB	CD	CE	CF	0x0B	
D0	D1	D2	D3	D4	D5	D6	D7	D8	D9	DA	DB	DC	DE	DF	0x0C	
E0	E1	E2	E3	E4	E5	E6	E7	E8	E9	EA	EB	EC	ED	EF	0x0D	
F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	FA	FB	FC	FD	FE	0x0E	
															0x0F	

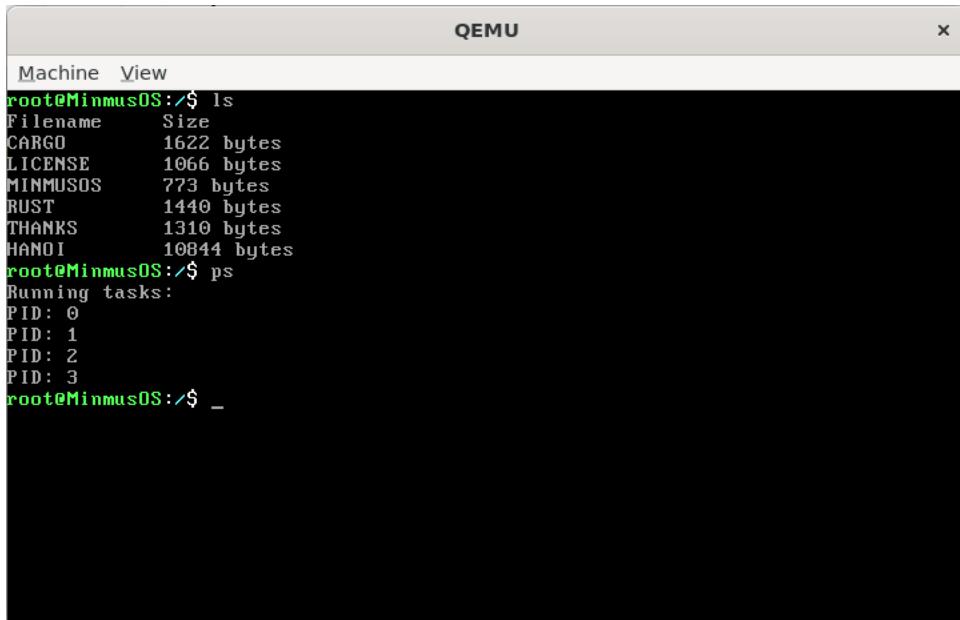
图 8.5 color 命令演示



QEMU

```
Machine View
root@MinmusOS:/$ cal
[August 2024]
Mon Tue Wed Thu Fri Sat Sun
      1  2  3  4
 5  6  7  8  9  10  11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
root@MinmusOS:/$ date
2024-08-26 13:50:06 UTC
root@MinmusOS:/$ timestamp
1724680208
root@MinmusOS:/$ ticks
753231800473
root@MinmusOS:/$ uname
MinmusOS v1.0 IA-32 x86
root@MinmusOS:/$ hostname
MinmusOS
root@MinmusOS:/$ pwd
/
root@MinmusOS:/$ whoami
root
root@MinmusOS:/$ _
```

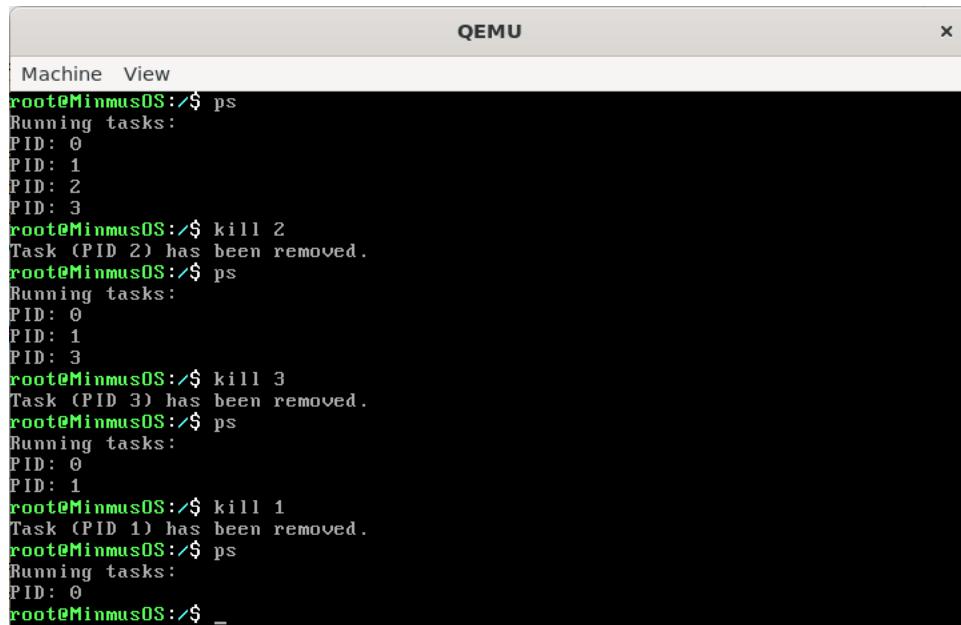
图 8.6 cal、date、timestamp、ticks、uname、hostname、pwd、whoami 命令演示



QEMU

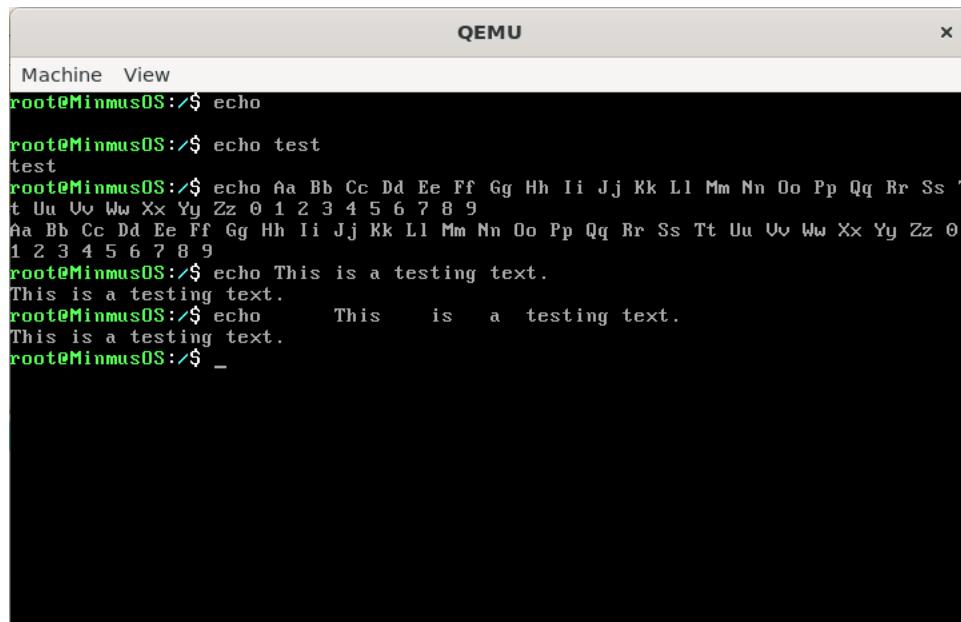
```
Machine View
root@MinmusOS:/$ ls
Filename      Size
CARGO        1622 bytes
LICENSE       1066 bytes
MINMUSOS      773 bytes
RUST         1440 bytes
THANKS        1310 bytes
HANOI         10844 bytes
root@MinmusOS:/$ ps
Running tasks:
PID: 0
PID: 1
PID: 2
PID: 3
root@MinmusOS:/$ _
```

图 8.7 ls、ps 命令演示



```
root@MinmusOS:~$ ps
Running tasks:
PID: 0
PID: 1
PID: 2
PID: 2
PID: 3
root@MinmusOS:~$ kill 2
Task (PID 2) has been removed.
root@MinmusOS:~$ ps
Running tasks:
PID: 0
PID: 1
PID: 3
root@MinmusOS:~$ kill 3
Task (PID 3) has been removed.
root@MinmusOS:~$ ps
Running tasks:
PID: 0
PID: 1
root@MinmusOS:~$ kill 1
Task (PID 1) has been removed.
root@MinmusOS:~$ ps
Running tasks:
PID: 0
root@MinmusOS:~$ _
```

图 8.8 kill 命令演示



```
root@MinmusOS:~$ echo
root@MinmusOS:~$ echo test
test
root@MinmusOS:~$ echo Aa Bb Cc Dd Ee Ff Gg Hh Ii Jj Kk Ll Mm Nn Oo Pp Qq Rr Ss T
t Uu Vv Ww Xx Yy Zz 0 1 2 3 4 5 6 7 8 9
Aa Bb Cc Dd Ee Ff Gg Hh Ii Jj Kk Ll Mm Nn Oo Pp Qq Rr Ss Tt Uu Vv Ww Xx Yy Zz 0
1 2 3 4 5 6 7 8 9
root@MinmusOS:~$ echo This is a testing text.
This is a testing text.
root@MinmusOS:~$ echo      This      is      a      testing      text.
This is a testing text.
root@MinmusOS:~$ _
```

图 8.9 echo 命令演示

```

Machine View
root@MinmusOS:~$ test
Command not found!
root@MinmusOS:~$ cat
Usage: cat <filename>
root@MinmusOS:~$ cat test
File not found!
root@MinmusOS:~$ run
Usage: run <appname>
root@MinmusOS:~$ run test
Application not found!
root@MinmusOS:~$ run license
This file is not a valid executable!
root@MinmusOS:~$ kill
Usage: kill <pid>
root@MinmusOS:~$ kill -1
Please enter a valid PID (1-31).
root@MinmusOS:~$ kill 0
Please enter a valid PID (1-31).
root@MinmusOS:~$ kill 32
Please enter a valid PID (1-31).
root@MinmusOS:~$ kill a
Please enter a valid PID (1-31).
root@MinmusOS:~$ kill 31
Task with PID 31 not found!
root@MinmusOS:~$ _

```

图 8.10 命令行解释器错误提示演示

```

Machine View
root@MinmusOS:~$ cat cargo
Cargo

Cargo is the designated package manager and build system for the Rust programming language, playing a crucial role in managing project dependencies and facilitating consistent, reproducible builds. By using Cargo, developers can easily specify and manage libraries their projects depend on in a clear, concise manner through a Cargo.toml file. Cargo handles downloading these dependencies from the crates.io repository, making sure they are compatible with each other and keeping them updated as necessary.

Beyond dependency management, Cargo also automates Rust project builds, ensuring that the compilation process is both reproducible and optimized. It records precise versions of dependencies in the Cargo.lock file, which helps maintain consistency across development environments and different machines. This system not only streamlines the development process but also significantly enhances project portability and ease of collaboration among multiple developers.

Cargo also plays a pivotal role in the Rust ecosystem by enforcing semantic versioning, which helps manage changes in libraries in a way that prevents compatibility problems. Its rich configuration options enable developers to tailor build processes and dependency resolutions to suit specific needs or environments. Furthermore, Cargo supports workspaces, allowing developers to manage multiple interrelated Rust packages within a single project, simplifying complex project management tasks. This extensive functionality underscores Cargo's importance in maintaining the robustness and efficiency of Rust project development.

root@MinmusOS:~$ _

```

图 8.11 文本文件 cargo 演示

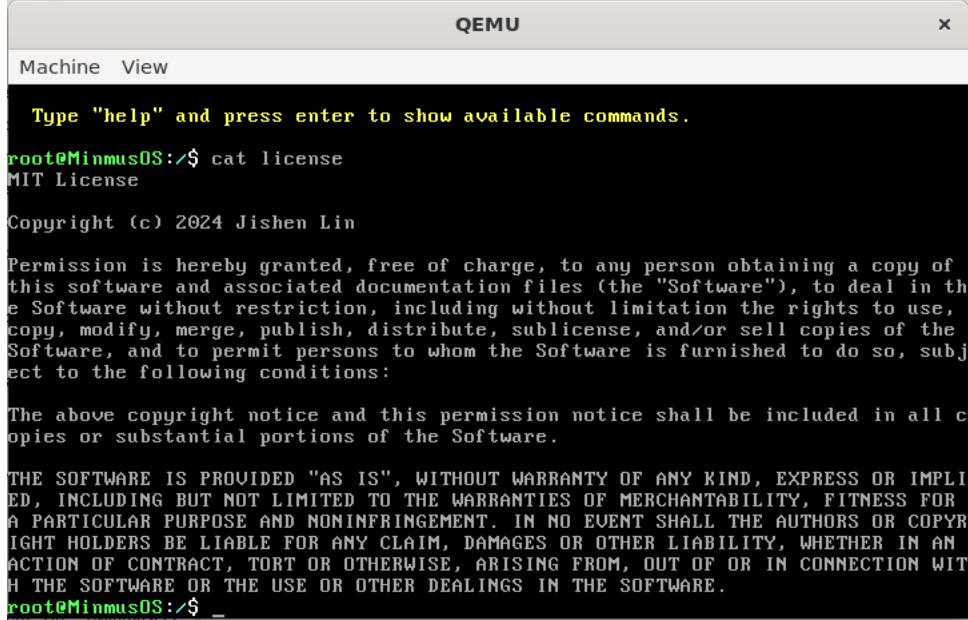


图 8.12 文本文件 license 演示

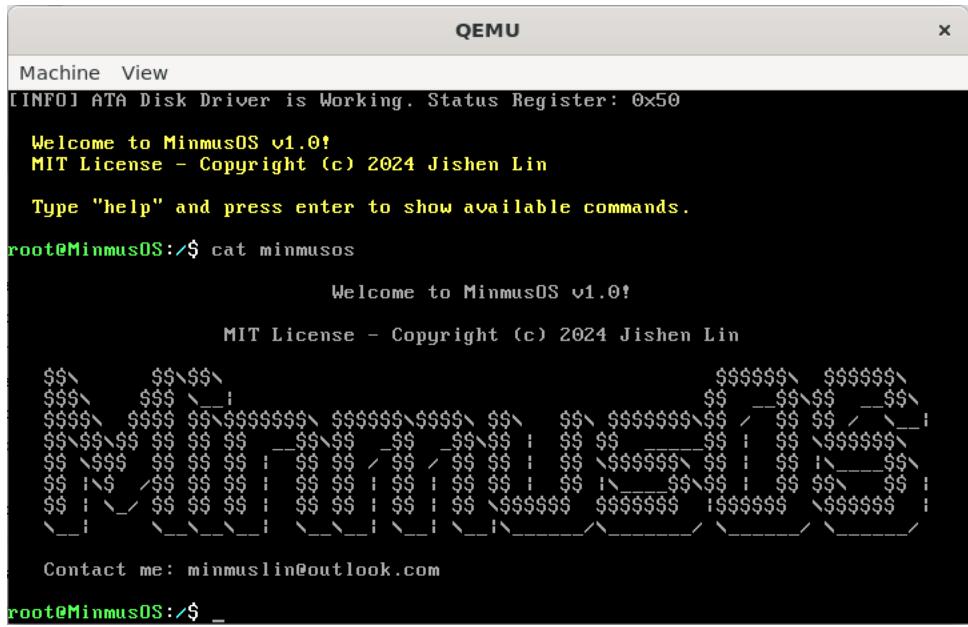
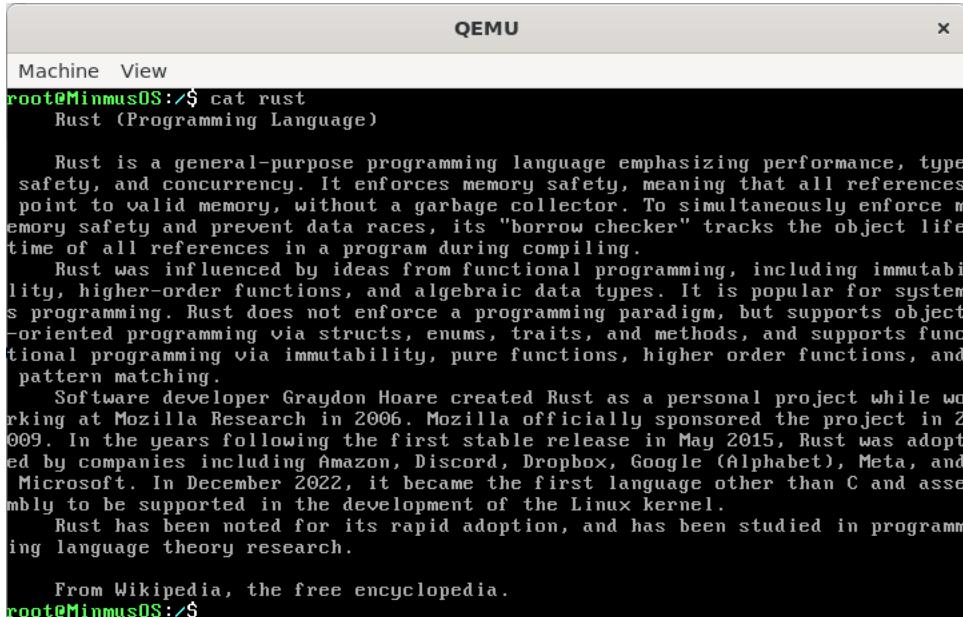


图 8.13 文本文件 minmusos 演示



```

QEMU
Machine View
root@MinmusOS:~$ cat rust
Rust (Programming Language)

Rust is a general-purpose programming language emphasizing performance, type safety, and concurrency. It enforces memory safety, meaning that all references point to valid memory, without a garbage collector. To simultaneously enforce memory safety and prevent data races, its "borrow checker" tracks the object lifetime of all references in a program during compiling.

Rust was influenced by ideas from functional programming, including immutability, higher-order functions, and algebraic data types. It is popular for system's programming. Rust does not enforce a programming paradigm, but supports object-oriented programming via structs, enums, traits, and methods, and supports functional programming via immutability, pure functions, higher order functions, and pattern matching.

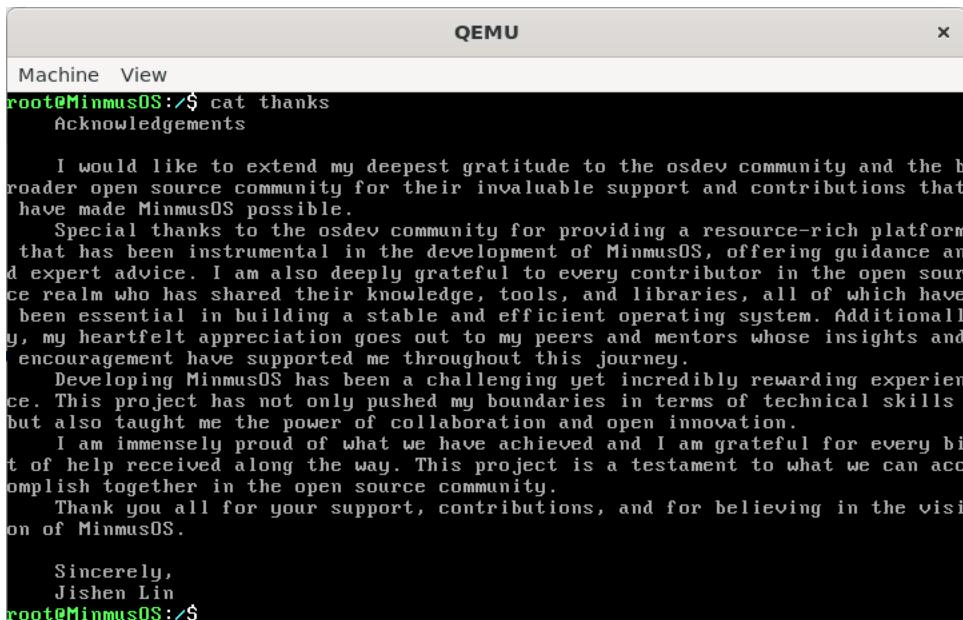
Software developer Graydon Hoare created Rust as a personal project while working at Mozilla Research in 2006. Mozilla officially sponsored the project in 2009. In the years following the first stable release in May 2015, Rust was adopted by companies including Amazon, Discord, Dropbox, Google (Alphabet), Meta, and Microsoft. In December 2022, it became the first language other than C and assembly to be supported in the development of the Linux kernel.

Rust has been noted for its rapid adoption, and has been studied in programming language theory research.

From Wikipedia, the free encyclopedia.
root@MinmusOS:~$ _

```

图 8.14 文本文件 rust 演示



```

QEMU
Machine View
root@MinmusOS:~$ cat thanks
Acknowledgements

I would like to extend my deepest gratitude to the osdev community and the broader open source community for their invaluable support and contributions that have made MinmusOS possible.

Special thanks to the osdev community for providing a resource-rich platform that has been instrumental in the development of MinmusOS, offering guidance and expert advice. I am also deeply grateful to every contributor in the open source realm who has shared their knowledge, tools, and libraries, all of which have been essential in building a stable and efficient operating system. Additionally, my heartfelt appreciation goes out to my peers and mentors whose insights and encouragement have supported me throughout this journey.

Developing MinmusOS has been a challenging yet incredibly rewarding experience. This project has not only pushed my boundaries in terms of technical skills but also taught me the power of collaboration and open innovation.

I am immensely proud of what we have achieved and I am grateful for every bit of help received along the way. This project is a testament to what we can accomplish together in the open source community.

Thank you all for your support, contributions, and for believing in the vision of MinmusOS.

Sincerely,
Jishen Lin
root@MinmusOS:~$ _

```

图 8.15 文本文件 thanks 演示

```
#1000 B->A A 4          B 5           C 10 9 8 7 6 3 2 1  
#1001 C->A A 4 1       B 5           C 10 9 8 7 6 3 2  
#1002 C->B A 4 1       B 5 2          C 10 9 8 7 6 3  
#1003 A->B A 4         B 5 2 1        C 10 9 8 7 6 3  
#1004 C->A A 4 3       B 5 2 1        C 10 9 8 7 6  
#1005 B->C A 4 3       B 5 2          C 10 9 8 7 6 1  
#1006 B->A A 4 3 2     B 5           C 10 9 8 7 6 1  
#1007 C->A A 4 3 2 1   B 5           C 10 9 8 7 6  
#1008 B->C A 4 3 2 1   B 1            C 10 9 8 7 6 5  
#1009 A->B A 4 3 2     B 1            C 10 9 8 7 6 5  
#1010 A->C A 4 3       B 1            C 10 9 8 7 6 5 2  
#1011 B->C A 4         B               C 10 9 8 7 6 5 2 1  
#1012 A->B A 4         B 3             C 10 9 8 7 6 5 2 1  
#1013 C->A A 4 1       B 3             C 10 9 8 7 6 5 2  
#1014 C->B A 4 1       B 3 2            C 10 9 8 7 6 5  
#1015 A->B A 4         B 3 2 1          C 10 9 8 7 6 5  
#1016 A->C A           B 3 2 1          C 10 9 8 7 6 5 4  
#1017 B->C A           B 3 2            C 10 9 8 7 6 5 4 1  
#1018 B->A A 2         B 3             C 10 9 8 7 6 5 4 1  
#1019 C->A A 2 1       B 3             C 10 9 8 7 6 5 4  
#1020 B->C A 2 1       B               C 10 9 8 7 6 5 4 3  
#1021 A->B A 2         B 1             C 10 9 8 7 6 5 4 3  
#1022 A->C A           B 1             C 10 9 8 7 6 5 4 3 2  
#1023 B->C A           B               C 10 9 8 7 6 5 4 3 2 1
```

图 8.16 应用程序汉诺塔解决方案演示

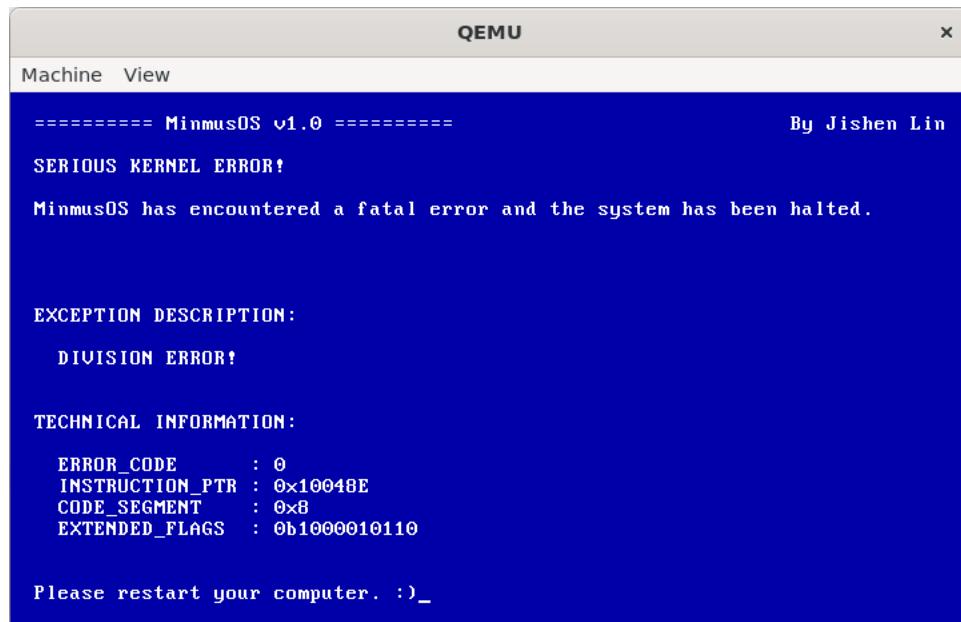


图 8.17 0 号异常演示：除零错误

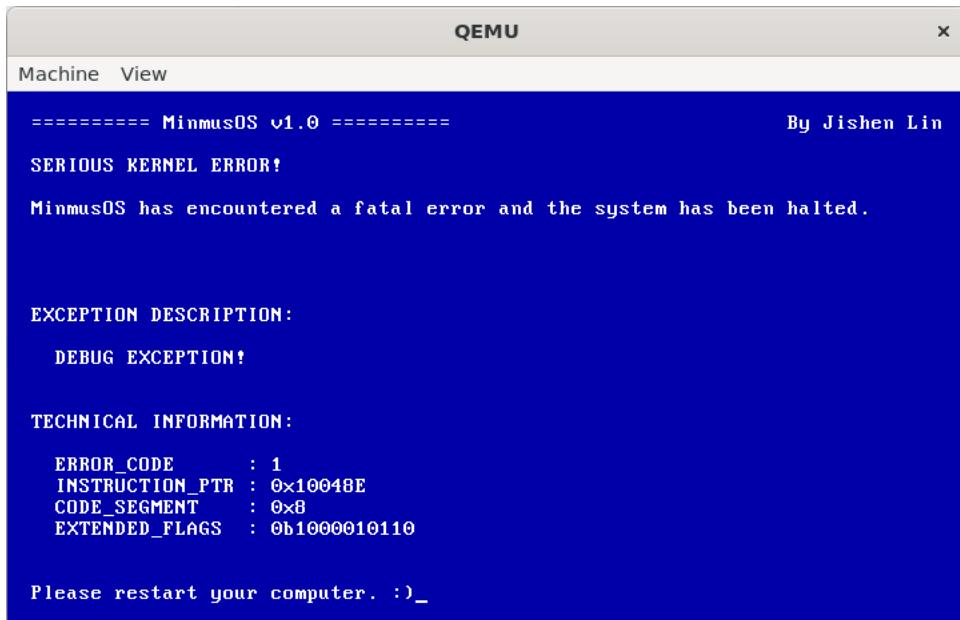


图 8.18 1 号异常演示：调试异常

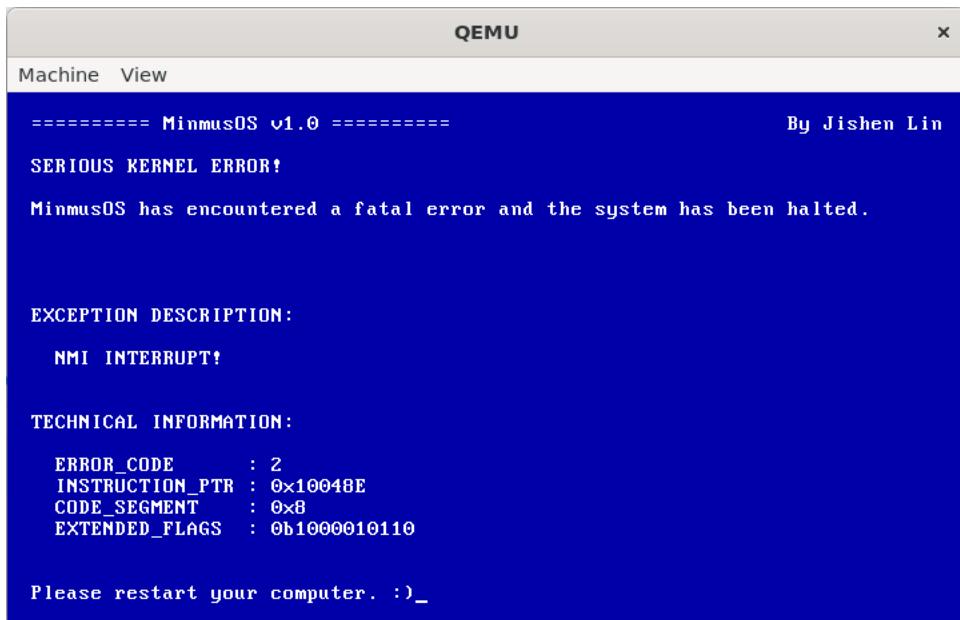


图 8.19 2 号异常演示：NMI 中断

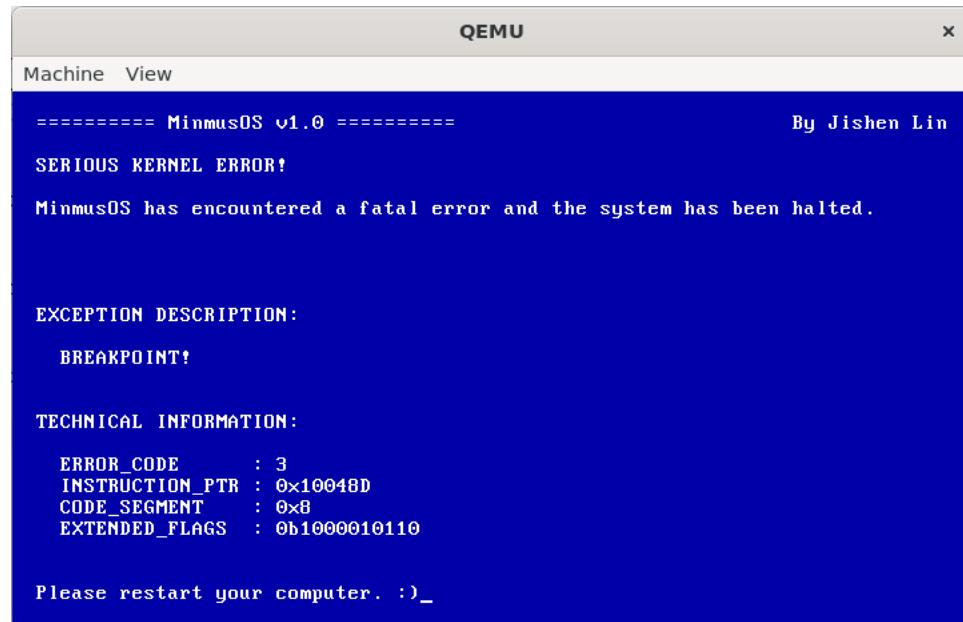


图 8.20 3 号异常演示：断点

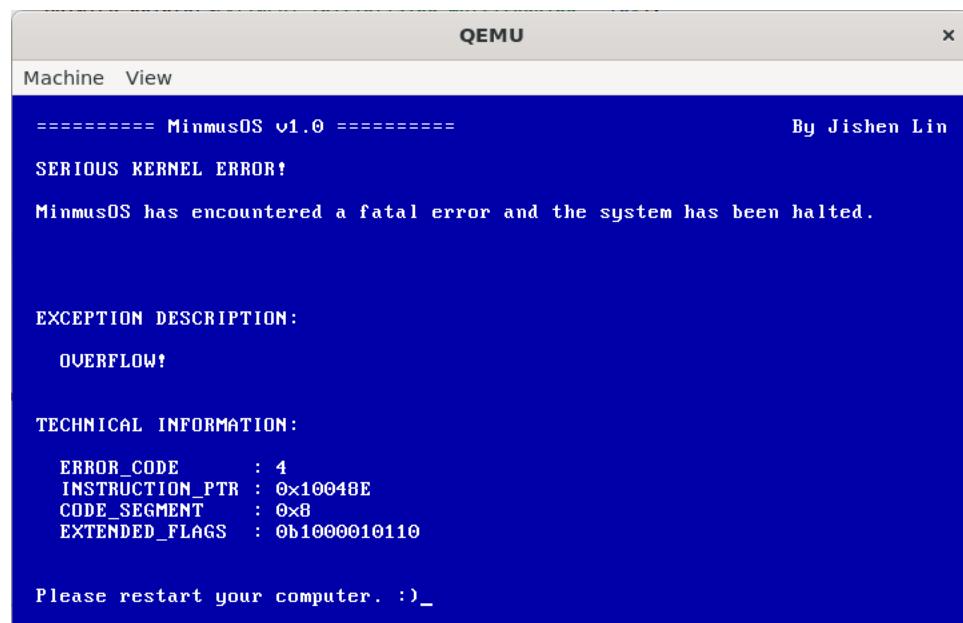


图 8.21 4 号异常演示：溢出

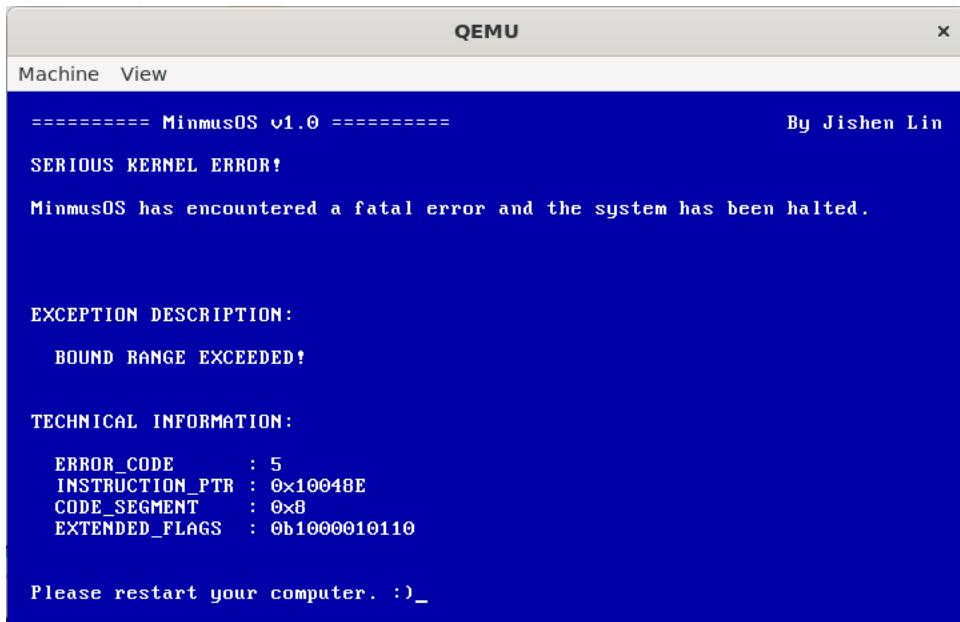


图 8.22 5 号异常演示：BOUND 范围超出

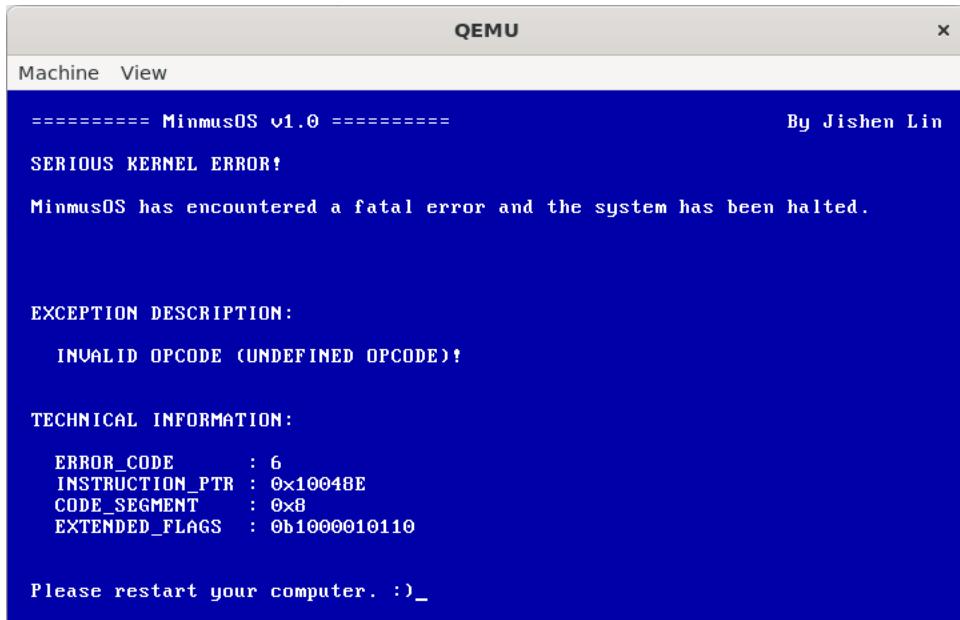


图 8.23 6 号异常演示：无效操作码（未定义操作码）

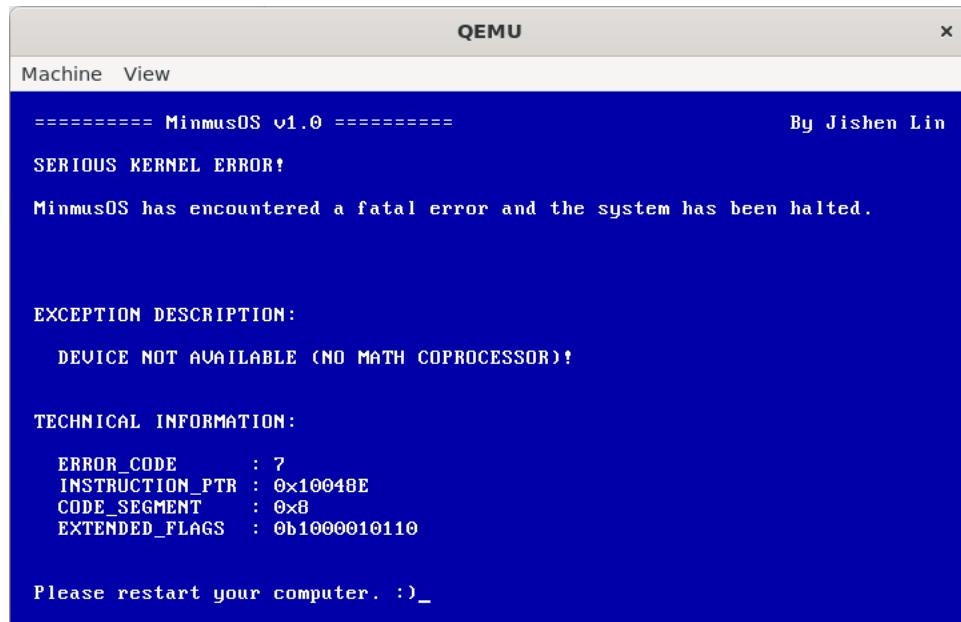


图 8.24 7 号异常演示：设备不可用（无数学协处理器）

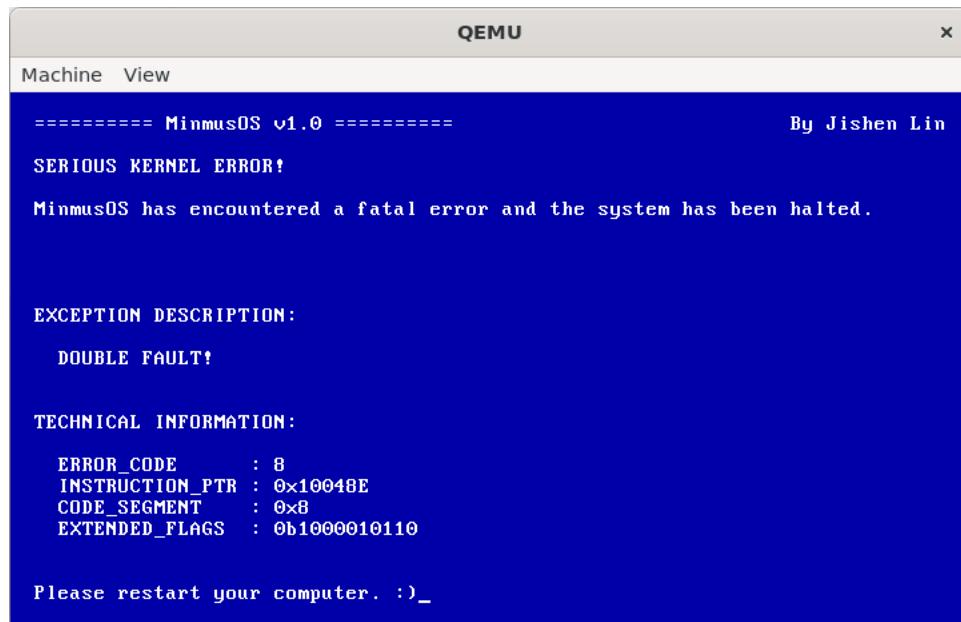


图 8.25 8 号异常演示：双重故障

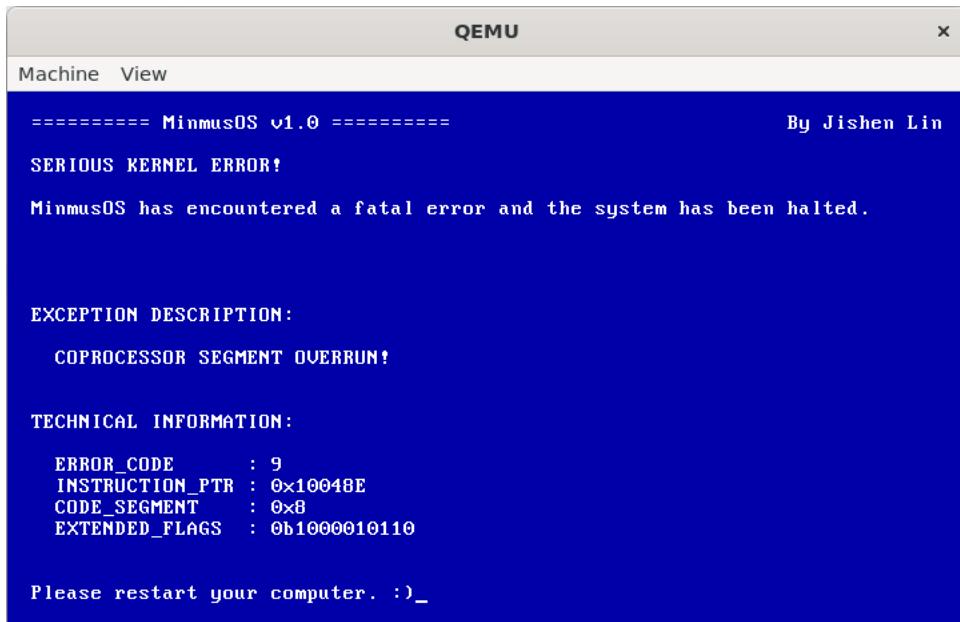


图 8.26 9 号异常演示：协处理器段溢出（保留）

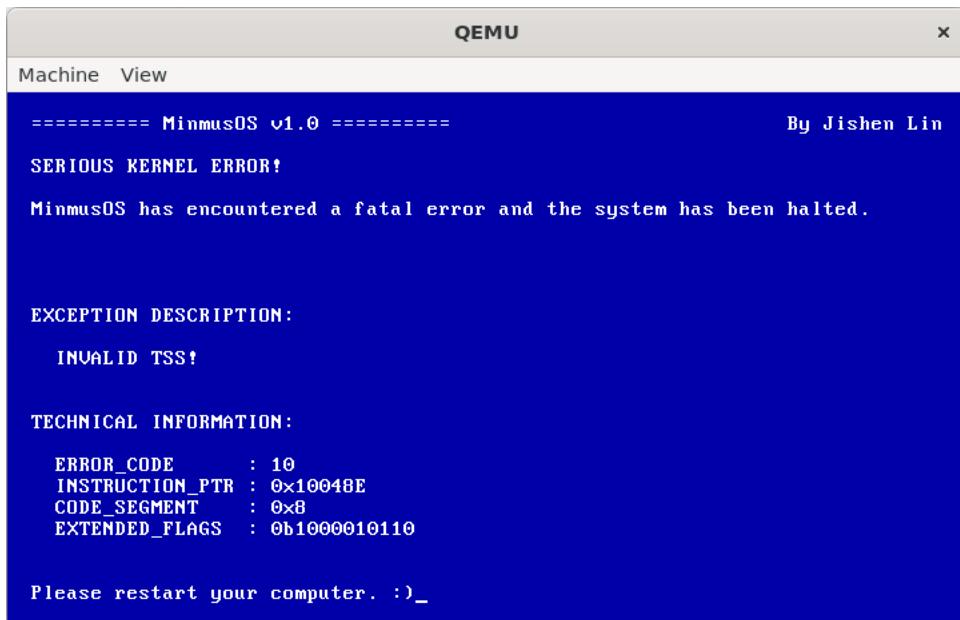


图 8.27 10 号异常演示：无效的 TSS

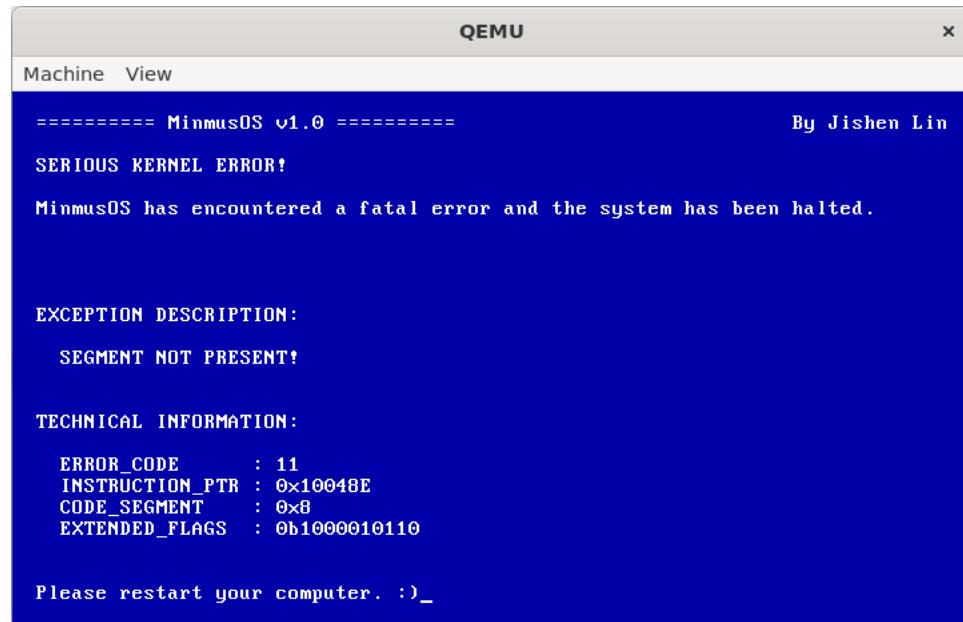


图 8.28 11 号异常演示：段不存在

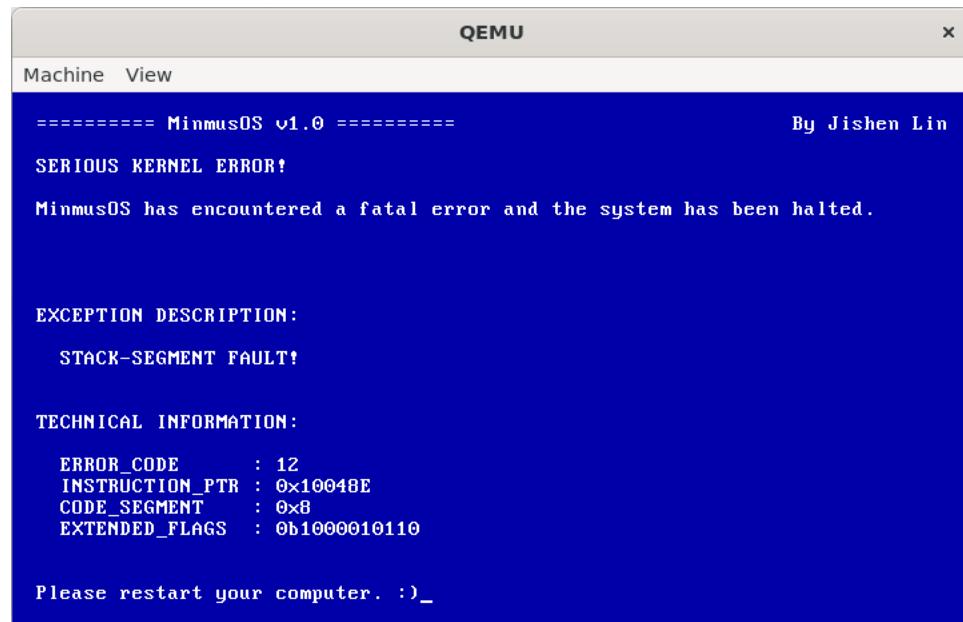


图 8.29 12 号异常演示：栈段故障

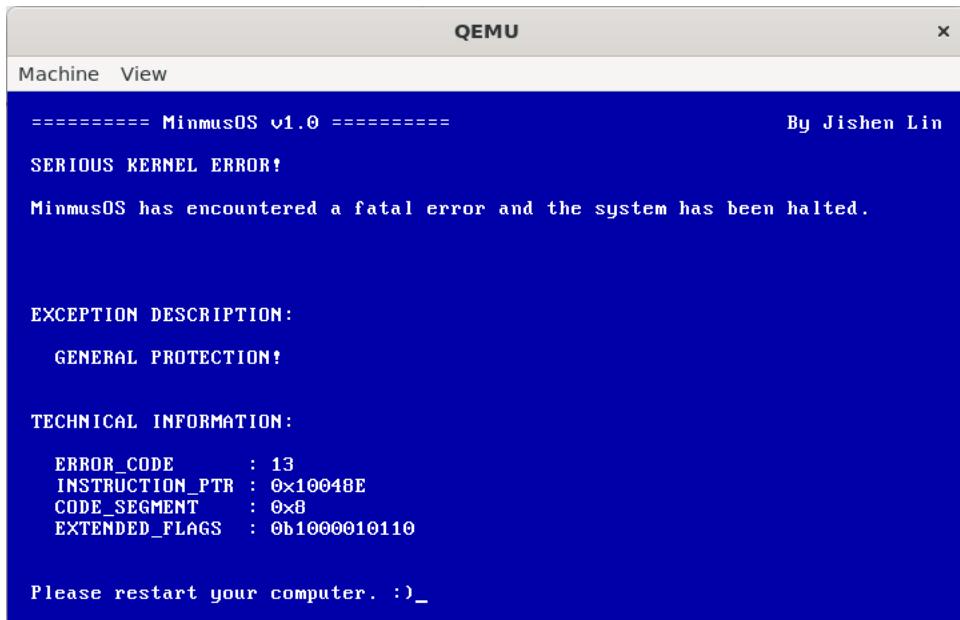


图 8.30 13 号异常演示：通用保护

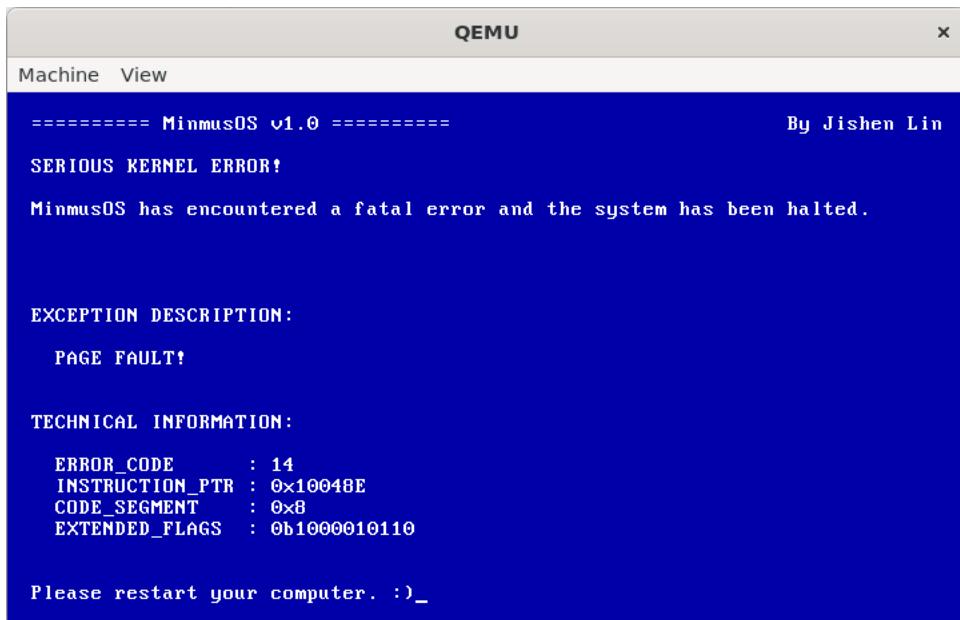


图 8.31 14 号异常演示：页面错误

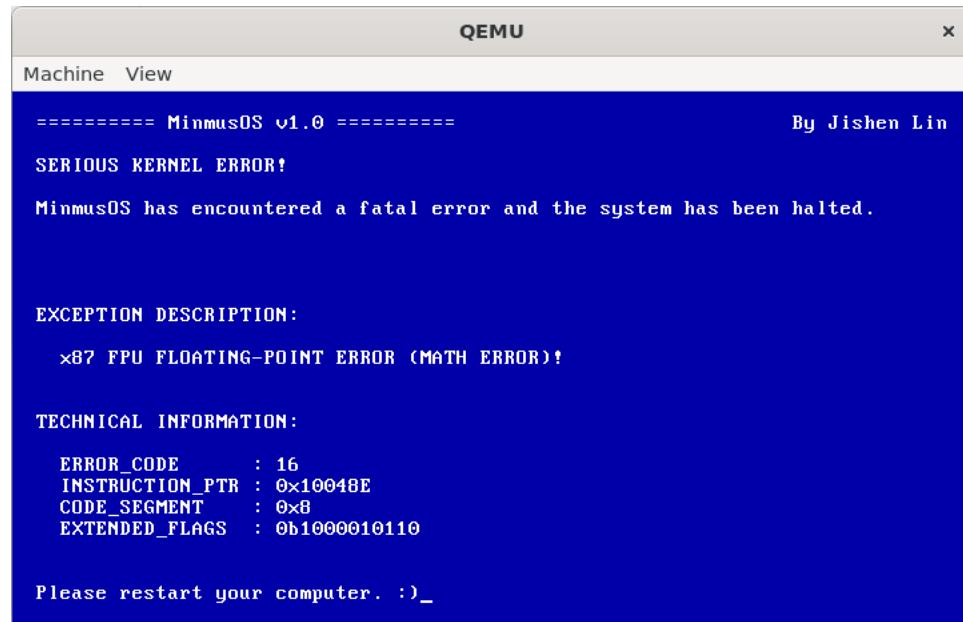


图 8.32 16 号异常演示：x87 FPU 浮点错误（数学故障）

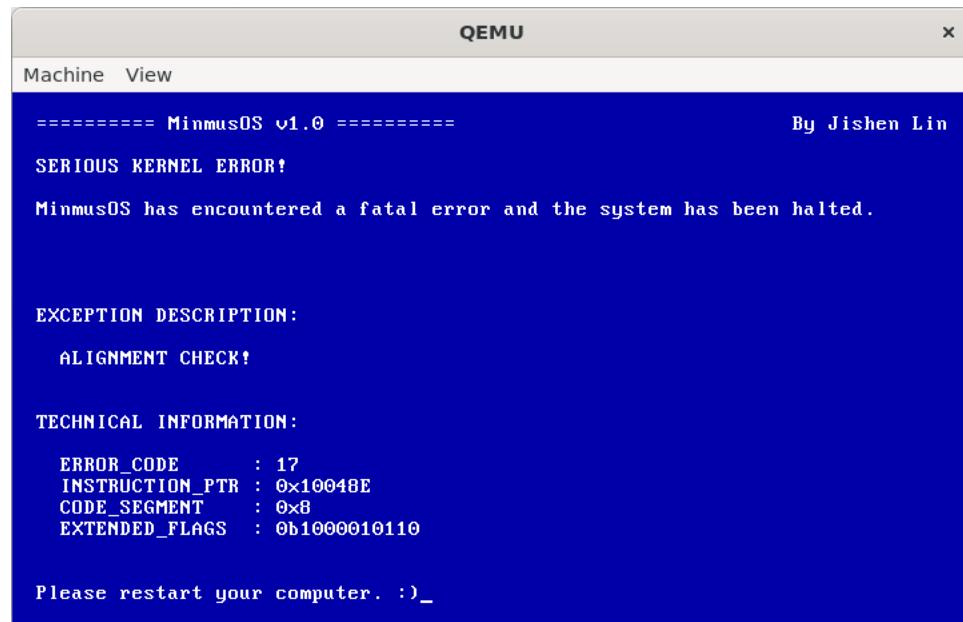


图 8.33 17 号异常演示：对其检查

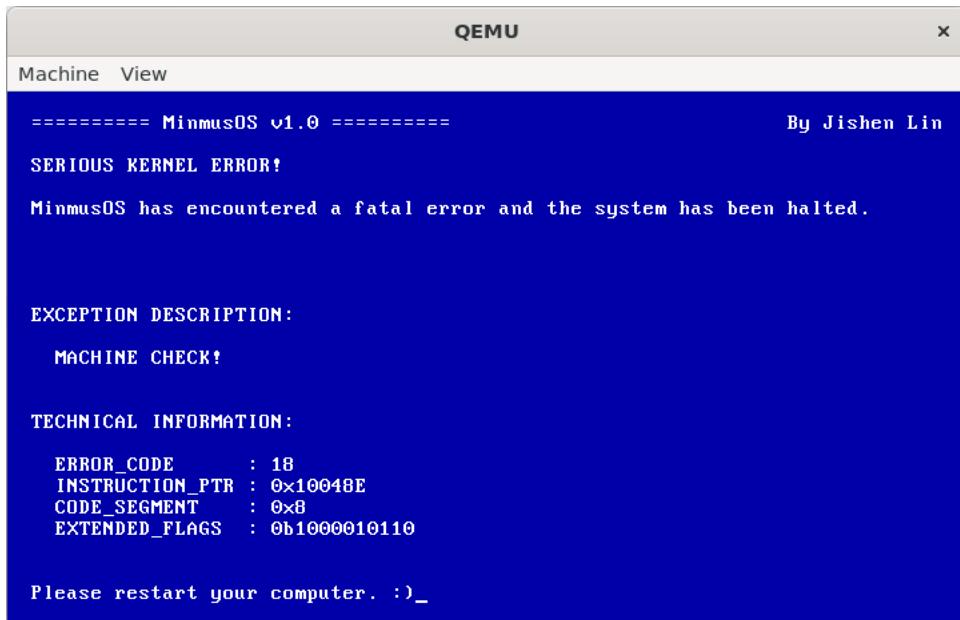


图 8.34 18 号异常演示：机器检查

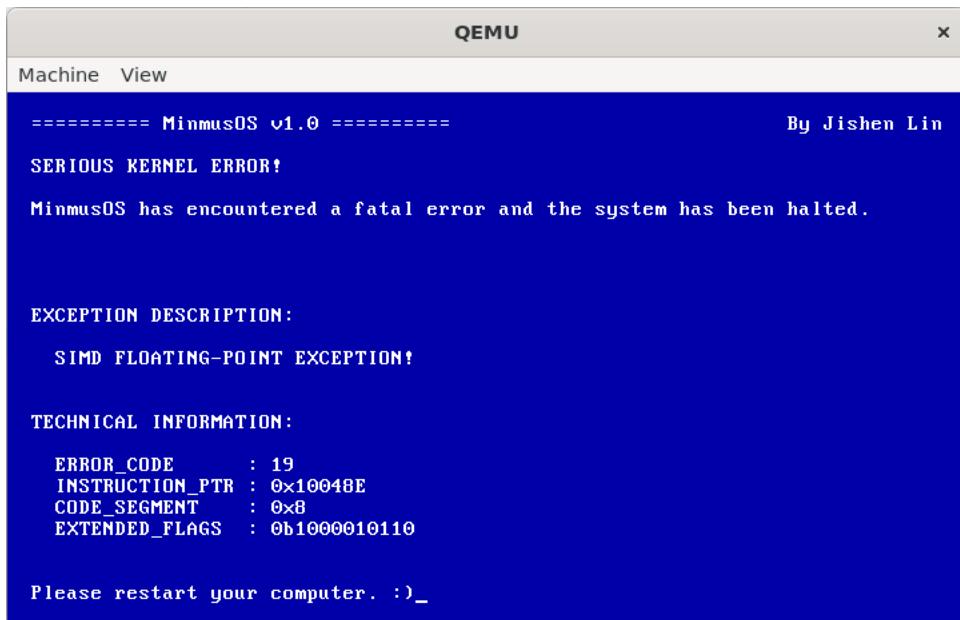


图 8.35 19 号异常演示：SIMD 浮点异常

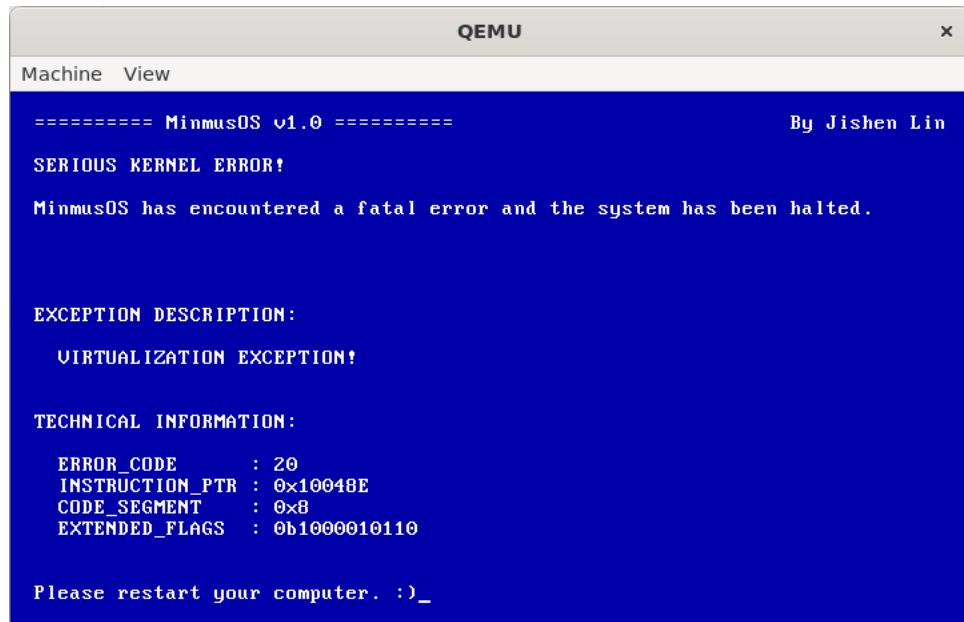


图 8.36 20 号异常演示：虚拟化异常

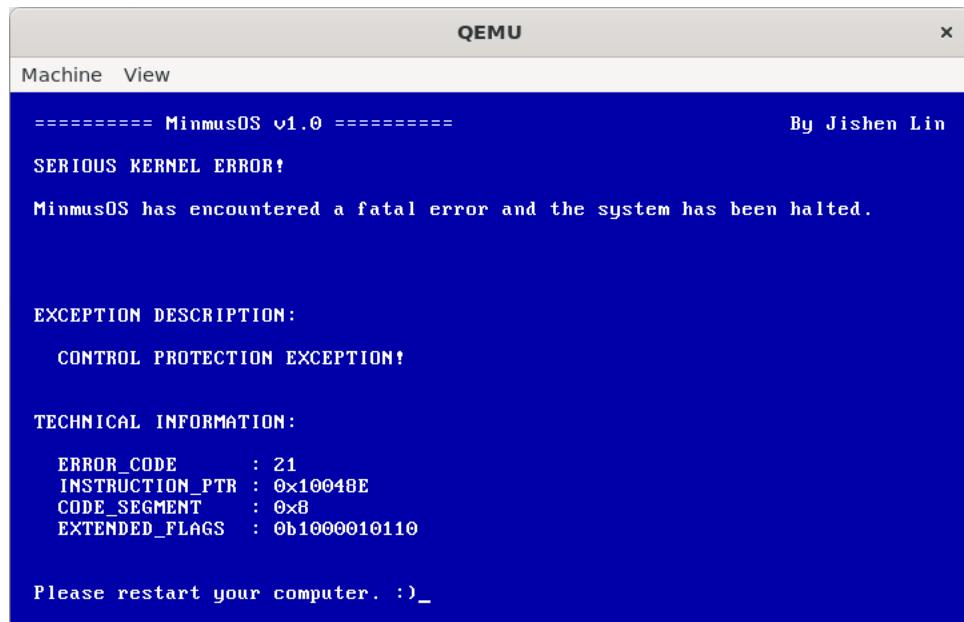


图 8.37 21 号异常演示：控制保护异常

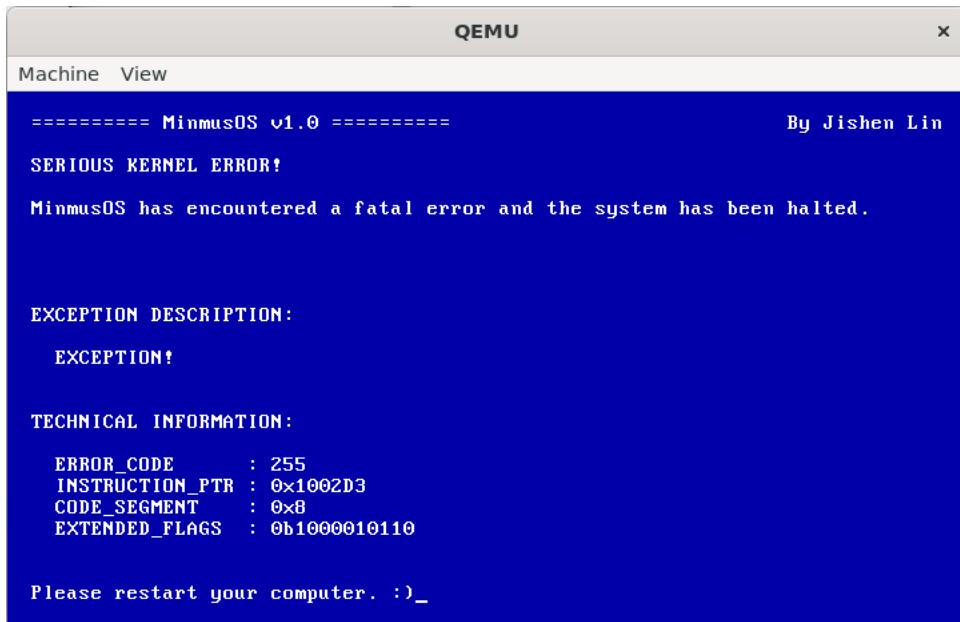


图 8.38 默认异常演示：Intel 保留



图 8.39 PANIC 演示：索引越界

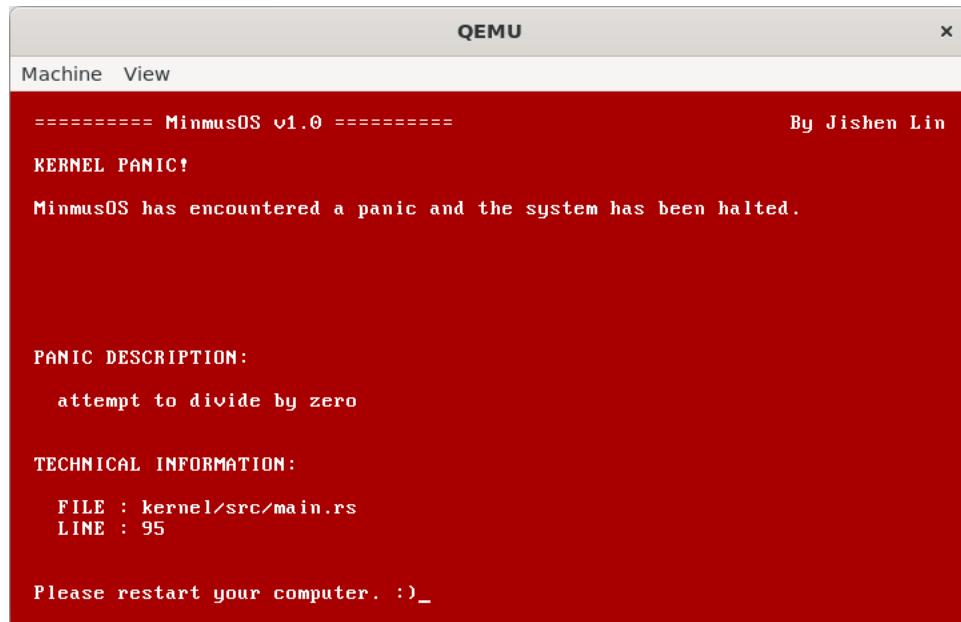


图 8.40 PANIC 演示：除零错误

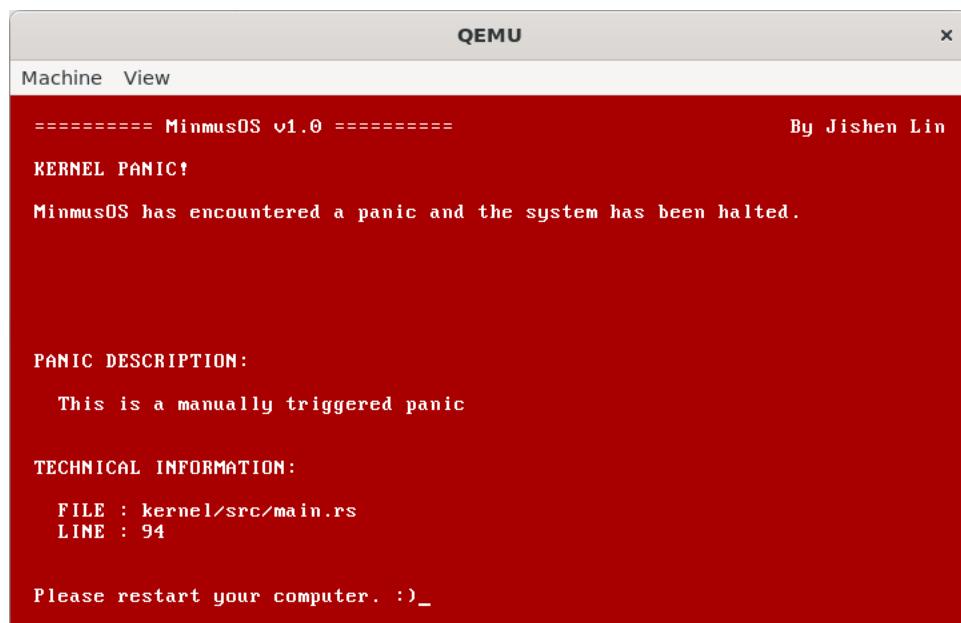


图 8.41 PANIC 演示：手动触发 PANIC

## 9 总结与展望

在本节（章节 9）中，笔者将对 MinmusOS 项目进行总结，回顾在开发 MinmusOS 项目过程中遇到的挑战，并对 MinmusOS 的未来开发进行展望。

### 9.1 项目总结

MinmusOS 通过结合 Rust 语言的内存安全特性和高效的系统设计，展示了一个稳定且功能丰富的操作系统内核。每个部分都以其独特的方式展示了 Rust 语言在操作系统开发中的强大潜力和优势。

MinmusOS 的引导程序是操作系统启动的关键入口，它负责从计算机启动开始到操作系统核心功能载入的整个过程。在该项目中，引导程序首先在实模式下运行，设置必要的 CPU 环境与系统资源，然后将系统切换到保护模式以支持更高级的操作系统功能，如虚拟内存管理。引导程序还负责加载内核到内存中，并转交控制权给内核。这一部分的实现展示了 Rust 语言在处理底层硬件操作时的强大能力和优势，例如通过精细控制内存安全和利用 Rust 的模块化特性来增强代码的可维护性和可靠性。引导过程的成功实施是整个操作系统稳定运行的基础，因此在这一阶段中对 Rust 的内存安全保障特性进行了充分利用，有效地避免了传统 C 语言开发中常见的内存错误和安全漏洞。

MinmusOS 的内核是系统的核心，负责管理计算机的硬件资源和运行环境。内核实现了包括任务调度、内存管理、系统调用处理、中断处理和文件系统管理等多项基本功能。这些功能的实现充分展示了 Rust 语言在系统级编程中的应用潜力，尤其是在并发控制和资源管理方面。内核的设计利用了 Rust 的类型安全和生命周期管理，显著提高了操作系统的稳定性和响应速度。通过精心设计的模块化结构，MinmusOS 内核不仅保证了代码的高内聚低耦合，还易于扩展和维护。此外，内核中实现的文件系统和内存管理策略，也充分利用了 Rust 的错误处理和模式匹配特性，极大地增强了系统对错误状态的处理能力和恢复能力。

项目的标准运行库为 MinmusOS 提供了一套基础的运行时支持，包括数学计算、互斥同步、输出打印、随机数生成、数据排序和字符串处理等功能。这些库的实现不仅丰富了操作系统的功能，也提供了必要的 API 支持，使得用户和开发者可以更容易地开发应用程序和服务。标准运行库的设计和实现采用了 Rust 的泛型和特性（traits），保证了代码的复用性和高性能。例如，通过为不同类型的数据结构实现统一的接口，使得算法的应用更为广泛和灵活。这些库的实现细节体现了 Rust 在保证性能的同时，也极大地提高了代码的安全性和可维护性，为操作系统的稳定运行提供了坚实的基础。

MinmusOS 包括了一些基本应用程序，如汉诺塔解决方案。这部分的实现不仅增加了操作系统的用户友好性，也为系统的实际应用提供了示例。每个应用程序都是独立的 Rust 项目，利用 Rust 的包管理和模块系统进行构建，这不仅确保了开发过程的高效性，也使得应用程序能够充分利用操作系统提供的资源。通过实现和集成这些应用程序，MinmusOS 展示了一个完整的操作系统框架不仅需要强大的核心功能，也需要丰富的应用支持，以满足终端用户的需求。

总的来说，MinmusOS 的开发展示了 Rust 语言在现代操作系统设计中的强大潜力和优势。通过这一项目，我们不仅验证了 Rust 在系统级编程应用的可行性，也为操作系统的未来发展提供了新的思路和方向。

### 9.2 项目挑战

笔者在实现 MinmusOS 的过程中遇到的挑战如下：

(1) **引导程序的实现**：实现 MinmusOS 的引导程序涉及将系统从真实模式切换到保护模式，并加载内核到内存中的复杂过程。这一挑战要求精确地配置硬件和系统参数，如设置全局描述符表 (GDT) 和初始化中断描述符表 (IDT)。引导程序必须在有限的资源和初始化阶段中准确执行，任何错误都可能导致系统启动失败。通过精心设计的启动代码和对硬件细节的深入理解，MinmusOS 成功实现了一个稳定的引导过程，确保了系统的顺利启动和运行。

(2) **内核功能的实现**：MinmusOS 内核的实现面临多项挑战，包括任务调度、内存管理、设备驱动集成以及系统调用处理等。内核必须高效而稳定地管理和调度系统资源，以支持复杂的多任务环境。实现这些功能涉及到对底层硬件的深入操作和对操作系统理论的应用，特别是在并发和资源保护方面。通过采用模块化设计和利用 Rust 的安全特性，MinmusOS 内核能够提供强大的性能和高度的系统安全性。

(3) **标准运行库的实现**：MinmusOS 的标准运行库提供了一系列基本的运行时支持功能，如数学运算、互斥锁、字符串处理等。挑战在于设计一套既通用又高效的库函数，支持跨平台的操作系统功能，同时保持 API 的简洁性和易用性。这些库需要在保证性能的同时，提供必要的错误处理和兼容性支持。通过精细的接口设计和对 Rust 生态系统的深入利用，MinmusOS 的标准运行库成功地为应用开发和系统运行提供了强有力的支持。

(4) **应用程序的实现**：在 MinmusOS 中，应用程序的实现不仅要求功能完备和用户友好，还需要确保运行安全和资源使用高效。挑战在于如何构建一个用户应用程序，同时保持应用与操作系统之间的高效交互。这包括设计一套能够简洁表达复杂操作的 API 和提供充分的系统服务支持。MinmusOS 通过提供一套丰富的 API 和开发工具，成功地支持了用户应用程序的开发和运行，从而极大地丰富了系统的实用性和可扩展性。

### 9.3 项目展望

MinmusOS 预计在未来添加如下功能：

(1) **按需分页**：实施按需分页机制将改善内存管理，使系统在处理内存分配时更加高效。这将允许 MinmusOS 仅在实际需要时才加载页面到物理内存，减少内存浪费并提升响应速度。

(2) **进程模型**：发展更为复杂的进程模型，支持多线程和进程间通信，将使 MinmusOS 能够更有效地管理和调度多任务，从而提升多任务环境下的系统性能和用户体验。

(3) **Linux/POSIX 系统调用**：实现兼容 Linux 和 POSIX 的系统调用将增强 MinmusOS 与现有软

件生态系统的兼容性，使得更多现有的应用程序和工具能够在 MinmusOS 上无缝运行。

(4) **FAT32 文件系统**: 支持 FAT32 文件系统将扩展 MinmusOS 在文件存储和访问方面的能力，允许处理更大的文件和提供更高的存储效率。

(5) **ISO9660 文件系统**: 加入对 ISO9660 文件系统的支持将使 MinmusOS 能够读取 CD-ROM 和其他光存储介质，这对于确保操作系统的多样化媒体支持非常重要。

(6) **ext2 文件系统**: 通过实现 ext2 文件系统，MinmusOS 将能够更好地支持基于 Linux 的环境和应用，提高文件系统的稳定性和数据完整性。

(7) **抢占式内核**: 发展抢占式内核设计将为 MinmusOS 带来更高的响应速度和更优的系统稳定性，特别是在高负载情况下能够更公平地分配处理器资源，避免单个进程或任务独占系统资源。

(8) **高级可编程中断控制器 (APIC)**: APIC 支持更复杂的中断管理功能，如中断优先级和 CPU 核间的中断负载平衡。它能够将中断直接发送到特定的处理器核心，从而提高处理效率和响应速度。此外，APIC 还支持更多的中断向量，允许系统处理更多的独立中断源，这在高负载或大规模 I/O 需求的现代计算环境中尤为重要。

(9) **SATA AHCI 磁盘驱动**: 为了提高对现代硬盘的支持和性能，MinmusOS 将开发 SATA AHCI 磁盘驱动。这将允许系统以更高的数据传输速率和更好的硬盘管理能力运行，同时支持热插拔和原生命令队列 (NCQ) 等高级特性。

(10) **网络通信**: 网络功能的加入将使操作系统能够处理网络通信和互联网连接。这将包括实现 TCP/IP 协议栈和相关的网络服务，如 HTTP 和 FTP，以支持更广泛的网络应用和服务。

(11) **VESA 视频驱动**: MinmusOS 计划引入 VESA 视频驱动来支持更高分辨率和颜色深度的显示模式，从而大幅提升用户的视觉体验。VESA 驱动将允许系统直接与现代显示硬件交互，支持各种通用的图形界面标准，这对于开发图形用户界面至关重要。

(12) **图形用户界面**: 图形用户界面 (GUI) 的引入将标志着 MinmusOS 从命令行界面向更现代、更友好的用户交互方式转变。计划中的 GUI 将包括窗口管理、图形渲染和用户输入处理，为用户提供直观和丰富的操作体验。

## 参考资料

- [1] Silberschatz A., Galvin P. B., Gagne G., Operating System Concepts, Wiley Publishing, 2008.
- [2] Intel® 64 and IA-32 Architectures Software Developer's Manual.
- [3] [https://wiki.osdev.org/MBR\\_\(x86\)](https://wiki.osdev.org/MBR_(x86))
- [4] [https://wiki.osdev.org/Disk\\_access\\_using\\_the\\_BIOS\\_\(INT\\_13h\)](https://wiki.osdev.org/Disk_access_using_the_BIOS_(INT_13h))
- [5] [https://wiki.osdev.org/Real\\_Mode](https://wiki.osdev.org/Real_Mode)
- [6] [https://wiki.osdev.org/Protected\\_Mode](https://wiki.osdev.org/Protected_Mode)
- [7] [https://wiki.osdev.org/Global\\_Descriptor\\_Table](https://wiki.osdev.org/Global_Descriptor_Table)
- [8] <https://wiki.osdev.org/Interrupts>
- [9] [https://wiki.osdev.org/Interrupt\\_Descriptor\\_Table](https://wiki.osdev.org/Interrupt_Descriptor_Table)
- [10] [https://wiki.osdev.org/Interrupt\\_Service\\_Routines](https://wiki.osdev.org/Interrupt_Service_Routines)
- [11] <https://wiki.osdev.org/Exceptions>
- [12] [https://wiki.osdev.org/8259\\_PIC](https://wiki.osdev.org/8259_PIC)
- [13] [https://wiki.osdev.org/PS/2\\_Keyboard](https://wiki.osdev.org/PS/2_Keyboard)
- [14] [https://wiki.osdev.org/ATA\\_PIO\\_Mode](https://wiki.osdev.org/ATA_PIO_Mode)
- [15] [https://wiki.osdev.org/Multitasking\\_Systems](https://wiki.osdev.org/Multitasking_Systems)
- [16] [https://wiki.osdev.org/Context\\_Switching](https://wiki.osdev.org/Context_Switching)
- [17] [https://wiki.osdev.org/System\\_Calls](https://wiki.osdev.org/System_Calls)
- [18] [https://wiki.osdev.org/Memory\\_management](https://wiki.osdev.org/Memory_management)
- [19] <https://wiki.osdev.org/Segmentation>
- [20] <https://wiki.osdev.org/Paging>
- [21] [https://wiki.osdev.org/Memory\\_Allocation](https://wiki.osdev.org/Memory_Allocation)
- [22] [https://wiki.osdev.org/Page\\_Frame\\_Allocation](https://wiki.osdev.org/Page_Frame_Allocation)
- [23] [https://wiki.osdev.org/Memory\\_Management\\_Unit](https://wiki.osdev.org/Memory_Management_Unit)
- [24] [https://wiki.osdev.org/File\\_Management](https://wiki.osdev.org/File_Management)

## 致辞

从零开始实现一个操作系统，是我本科期间一次独特且宝贵的经历。这一路走来，实属不易。

实现操作系统的想法可以追溯到 2024 年寒假，当时我主要学习川合秀实所著的《30 天自制操作系统》。然而，这本书所依赖的工具链过于陈旧，必须使用书中提供的编译器和特有的非标准函数，启动区更是为 2.88MB 软盘设计的。由于没有使用标准编译器，作为初学者的我必须严格按照教程操作，这让我在开发过程中感到束手束脚，无法灵活地增添新功能，这并不是我所追求的。

之后，我转而学习于渊所著的《Orange's: 一个操作系统的实现》，但因其难度过大，最终放弃。

4 月，我在 Bilibili 平台偶然发现了一位宝藏 UP 主 LunaixSky，他发布了一系列视频教程《从零开始自制操作系统》。这位作者的操作系统 LunaixOS 及相关教程完全原创，未参考任何现有的操作系统开发书籍或开源内核代码。为了编写这套教程，他耗费了大量时间和精力钻研技术文档和现行工业标准，力求知识的独创性。但随着教程的深入和项目的频繁重构，我在跟随教程到分页与内核重映射部分时便跟不上了。Git 提交记录和视频教程的不完全对应加大了我的学习难度，让我最终不得不选择放弃。然而，这段学习经历让我受益匪浅，不仅提升了我的技术理解能力，还让我真正领悟到了开发操作系统的魅力，并培养了我查阅 Intel®64 与 IA-32 架构软件开发者手册等官方技术文档的能力。加入作者的 LunaixOS 技术交流群，更让我感受到了开源精神的伟大。

在期末周结束后的小学期，我重拾起这个项目。面对 C 语言中指针满天飞带来的复杂性和潜在的不安全因素，我做出了一个重要决定：将 MinmusOS 的编程语言从 C 语言改为 Rust 语言。Rust 的开发体验相比 C 语言无疑更加出色。C 语言中的许多不安全或未定义行为，往往只能在运行时暴露出问题或被隐藏，而 Rust 则通过其严格的安全性检查，在编译期就能有效避免这些错误。这种安全性和可靠性，极大地提升了我的开发效率。

自此，我不断在 OS Dev Wiki 学习并查阅 Intel IA-32 架构软件开发者手册，逐步完成了引导程序、内核、标准运行库与应用程序的实现。从最初学习和复现别人的代码，到逐渐成长为能够独立实现自定义功能的开发者，甚至为开源社区贡献代码、修复 Bug，我感受到自己编程能力的巨大进步，也体会到了从无到有实现一个项目的成就感。回首整个项目，于我而言，这不仅仅是一项课程设计，更是我技术生涯中的重要里程碑。

在此，我要衷心感谢张惠娟老师和王冬青老师对于操作系统理论的教授，并让我有这个机会从零开始实现一个操作系统。感谢开源社区和 OS Dev Wiki 的贡献者们，是你们的无私分享让我不断增加对操作系统开发的理解。更要感谢一路坚持、没有因为技术难题而放弃的自己。最后，感谢阅读至此的你，愿你也能在自己的学习与探索中找到属于自己的光芒。

2024 年 8 月 27 日  
于同济大学四平路校区