南开大学
Nankai University

南　　开　　大　　学

操作系统大赛“功能赛”赛道

NeoOS——一个基于 Rust 的教学操作系统

学校：南开大学

队伍名称：能润就行

参赛成员 ： 穆禹宸 袁贞芷 李潇逸

选题：PROJECT-0

指导教师：宫晓利

2023 年 6 月 7 日

目录

一、 概述

操作系统是计算机系统中最基础也是最重要的一个部分，其主要工作就是管理计算机系统的硬件资源、提供用户和应用程序的接口，以及提供各种服务。从用户和应用程序的角度来说，操作系统为其提供了可视化、多任务、网络、安全等各种服务；从程序员的角度来说，操作系统主要是指用户登录的界面或者接口；从设计人员的角度来说，操作系统则是指各式各样模块和单元之间的联系。

操作系统是计算机科学领域中非常重要的一门课程。通过操作系统实验，我们可以更加深入地理解操作系统新型的体系结构、基本功能及原理。目前，主流的操作系统实验：国内以清华大学为代表设计了 uCore 实验，后采取 Rust + Risc-V 开发了 rCore [4] 实验；而国外则以 MIT 设计的 xv6 操作系统实验（如 6.828）和 UCB 设计的 Pintos 为代表。这些实验的设计都着眼于操作系统整体功能设计与实现，尤其重视进程调度、内存分配等内容，但是以 uCore 为代表的实验，存在着先实现内存分配，后实现进程调度的问题，这与理论课的一般授课进度是颠倒的；而包括 Pintos 在内的国外实验课，也存在不够重视文件系统等模块的现象（比如 Pintos 的 Lab4 文件系统是可选作业），因此这些都导致了学生可能对操作系统实践之中某些内容的了解匮乏。然而，在目前主流操作系统实验中，已经存在的问题都没有得到较好地解决。

在设计操作系统实验时，我们需要满足实验和理论课的进度尽量同步，以及针对重要的模块——比如文件系统——进行详细的设计，力图在实验进行的同时可以与理论课深度结合从而减小实验压力，而在完成整个实验之后，则可以对操作系统有着深入了解。同时，本项目也结合了 OS Dev wiki [3] 的一些先进理念，在提供完整实验的同时，提供了更加丰富的操作体系实践内容。

二、 项目背景

NeoOS 是一个操作系统课程实验项目，旨在为高校的计算机专业提供高效、可靠的操作系统实验。该项目的目标是设计和实现一个功能齐全的操作系统，具有虚拟内存管理、进程和线程管理、文件系统、网络栈等关键功能。通过参与 NeoOS 项目，学生们可以深入了解操作系统的内部工作原理，学习并应用各种操作系统核心概念和技术。

实现意义：

- 实践操作系统理论知识：NeoOS 项目为学生们提供了一个实践操作系统理论知识的平台。通过参与项目，学生们可以将课堂上学到的操作系统概念和原理应用到实际的代码实现中，加深对操作系统的理解和掌握。
- 锻炼系统设计与编程能力：NeoOS 项目要求学生们设计和实现一个功能齐全的操作系统，这需要他们合理规划系统结构、设计模块接口、处理复杂的数据结构和算法等。通过参与项目，学生们可以锻炼系统设计和编程能力，提升自己的技术水平。
- 学习多任务调度和资源管理：NeoOS 项目涉及多任务调度和资源管理的实现。学生们需要设计和实现进程和线程管理模块，合理调度系统资源，实现任务的并发执行和资源的有效利用。这将使他们深入了解多任务调度和资源管理的原理和技术。
- 理解操作系统内核的组成：NeoOS 项目要求学生们实现操作系统的核心组件，如虚拟内存管理、文件系统、网络栈等。通过实现这些组件，学生们将理解操作系统内核的不同部分如何相互协作，形成一个完整的操作系统环境。
- 团队合作与沟通能力培养：NeoOS 项目通常需要学生们以小组形式合作完成。在项目开发过程中，学生们需要共同协作、分工合作、解决问题，并及时沟通交流。这将培养他们的

团队合作与沟通能力，提高工作效率和协同能力。

NeoOS 作为一个操作系统课程实验项目，为学生们提供了完善的实践操作系统设计和实现的过程。通过完成本项目，学生们可以将理论知识应用到实际项目中，锻炼系统设计与编程能力，深入理解操作系统的内部工作原理，并培养团队合作与沟通能力。

三、 系统设计

(一) 仓库布局

本实验框架代码的目录结构如下:

```
.
├── boot
├── boot_header
├── doc
├── kernel
├── sample_programs
├── screenshots
├── target
├── test
└── tools
```

仓库中有如图所示的 9 个文件夹，较为关键的是 boot、boot_header、kernel 这三个文件夹。下面分别进行介绍。

(二) boot

boot 文件夹的目录结构如下所示:

```
./boot
├── Cargo.toml
└── src
    ├── main.rs
    ├── page_table.rs
    └── utils.rs
```

在 boot 文件夹中共有四个文件。其主要功能是通过 UEFI 启动内核，同时有一些 UEFI 启动器，并进行以下操作:

- 从磁盘读取内核镜像
- 读取物理内存信息，并构建页帧分配器
- 构建初步的页表
- 对内核各个 section、内核栈、BIOS 等物理地址进行映射

(三) boot_header

boot_header 文件夹的目录结构如下所示:

```
./boot_header
```

```
├── Cargo.toml
├── src
│   └── lib.rs
```

在 boot_header 中有两个文件，主要定义了 bootloader 探测到的一些配置信息，这些信息之后将被传递给内核。

```rust

pub struct Header {
    /// The version of the boot p 文件对象为 fs/file.rs，与 unix 一样做了抽象。文件系统的 filesystem 和 Inode 的抽象层使用 trait 设计。rotocol.
    pub version: u8,
    /// The boot flags.
    pub cmdline: *const u8,
    /// The length of the cmdline string.
    pub cmdline_len: u64,
    /// The graphic mode. Must be false because we do not support it but we may add it in the future(?)
    pub enable_graph: bool,
    /// The address of the Root System Description Pointer used in the ACPI programming interface.
    pub acpi2_rsdp_addr: u64,
    /// The physical address to the start of virtual address.
    pub mem_start: u64,
    /// The address of the System Management BIOS.
    pub smbios_addr: u64,
    /// The graphic information.
    pub graph_info: GraphInfo,
    /// The address to the memory mapping provided by the UEFI.
    pub mmap: u64,
    /// The length of the mmap descriptors.
    pub mmap_len: u64,
    /// The kernel entry.
    pub kernel_entry: u64,
    /// The first process.
    pub first_proc: *const u8,
    /// The length of the `first\_proc` string.
    pub first_proc_len: u64,
    /// The argument for the first process.
    pub args: *const u8,
    /// The length of the arguments.
    pub args_len: u64,
    }
```

(四) kernel

kernel 文件夹的目录结构如下所示：

./kernel

```
├── build.rs
├── Cargo.toml
├── linker.ld
│   ├── arch
│   ├── banner.txt
│   ├── debug.rs
│   ├── drivers
│   ├── elf.rs
│   ├── error.rs
│   ├── f32.rs
│   ├── f64.rs
│   ├── fs
│   ├── irq
│   ├── lib.rs
│   ├── logging.rs
│   ├── main.rs
│   ├── memory.rs
│   ├── mm
│   ├── net
│   ├── process
│   ├── signal
│   ├── sync
│   ├── sys
│   ├── syscall
│   ├── time.rs
│   ├── trigger.rs
│   └── utils
└── x86_64.json
```

kernel 是 NeoOS 的核心，下面对整个 kernel 进行简要分析。

1. 内核启动基本流程

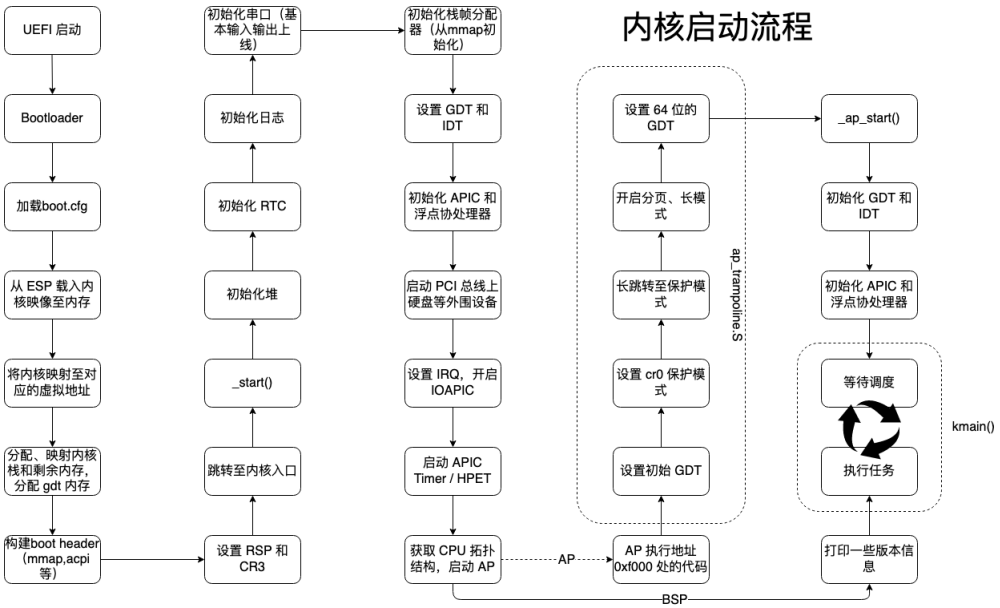

图 1: 内核启动流程

内核启动的流程代码如下所示，这段代码定义于 kernel/src/arch/x86_64/boot/mod.rs，其中的每一步都以函数调用的形式完成。

```rust
/// The entry point of kernel
#[no_mangle]
pub unsafe extern "C" fn _start(header: *const Header) -> ! {
    let header = unsafe { &*(header) };
    // Initialize the heap.
    let heap = init_heap();

    // Initialize RTC for read.
    init_rtc();

    // Initialize logging on the fly.
    // This operation is safe as long as the macro is not called.
    init_env_logger().unwrap();
    // Initialize the serial port for logging.
    init_all_serial_ports();

    kwarn!("logger started!");
    kinfo!("logging level is {}", *LOG_LEVEL);
    kinfo!("heap starts at {:#x}", heap);

    if let Err(errno) = init_kernel_page_table() {
        panic!(
            "failed to initialize kernel page tables! Errno: {:?}",
```

```
            errno
        );
    }
    kinfo!("initialized kernel page tables");
    }
```

接下来，进行内核启动关键步骤的分析。

2. 堆初始化

在文件 kernel/src/memory.rs 中定义的函数 init_heap 用于进行堆初始化。该函数通过指定一块 bss 区域作为初始堆的起始地址，该区域通过 ld 脚本实现，该脚本制定了 bss 所在的虚拟地址。由于在 bootloader 阶段已经把 bss 区域映射到了虚拟地址，所以可以进行直接访问。

```rust
/// The memory can be used only after we have initialized the heap!
pub fn init_heap() -> usize {
    const MACHINE_ALIGN: usize = core::mem::size_of::<usize>();
    const HEAP_BLOCK: usize = KERNEL_HEAP_SIZE / MACHINE_ALIGN;
    static mut HEAP: [usize; HEAP_BLOCK] = [0; HEAP_BLOCK];

    unsafe {
        // Initialize the heap.
        super::ALLOCATOR
            .lock()
            .init(HEAP.as_ptr() as usize, HEAP_BLOCK * MACHINE_ALIGN);
        HEAP.as_ptr() as usize
    }
}
```

3. 物理内存初始化

在文件 kernel/src/arch_x86_64/mm/paging.rs 中定义的函数 init_mm 用于进行物理内存初始化。该函数将 bootloader 读取到的物理内存信息进行存储，之后将这些信息传递给物理页帧分配器用于给之后的内存分配做准备。

```rust
/// Inserts all UEFI mapped memory regions into the bitmap-based frame allocator.
/// It is important for the use of the memory management.
pub fn init_mm(header: &'static Header) -> KResult<()> {
    // Initialize the kernel frame allocator for the user space.
    let mut allocator = LOCKED_FRAME_ALLOCATOR.lock();
    // Reinterpret the memory region.
    let mmap = unsafe {
        core::slice::from_raw_parts(
            phys_to_virt(header.mmap) as *const MemoryDescriptor,
            header.mmap_len as usize,
        )
```

```
    };

    for descriptor in mmap.iter() {
        kdebug!("init_mm(): {:x?}", descriptor);

        if descriptor.ty == MemoryType::CONVENTIONAL {
            let start_frame = descriptor.phys_start as usize / PAGE_SIZE;
            let end_frame = start_frame + descriptor.page_count as usize;
            allocator.insert(start_frame..end_frame)?;
        }
    }

    Ok(())
}
```

4. 终端初始化

在文件 kernel/src/arch_x86_64/intrrupt/mod.rs 中定义的函数 init_interrupt_all 用于进行终端初始化。

```
/// Initialize the trap handler and the interrupt descriptor table.
pub unsafe fn init_interrupt_all() -> KResult<()> {
    // Step 1: This operation should be atomic, so no other interrupts could
    ↪ happten.
    let flags = disable_and_store();
    kinfo!("init_interrupt_all(): disabled interrupts.");
    // Step 2: Set up the global descriptor table.
    init_gdt()?;
    kinfo!("init_interrupt_all(): initialized gdt.");
    // Step 3: Set up the interrupt descriptor table.
    init_idt()?;
    kinfo!("init_interrupt_all(): initialized idt.");
    // Step 4: Set up the syscall handlers.
    init_syscall()?;
    kinfo!("init_interrupt_all(): initialized syscall handlers.");
    // Step 5: Restore the interrupt.
    restore(flags);

    Ok(())
}
```

该函数功能具体通过如下几个步骤实现：

1. disable_and_store：用于暂时关闭中断。
2. init_gdt：初始化全局描述符表。
3. init_idt：初始化中断向量表。

4. init_syscall：初始化 syscall 相关寄存器用以快速 syscall。
5. restore：启动中断。

此外，由于中断涉及权限切换，需要用汇编手动实现，相关代码在同目录下的.S 文件中。

5. CPU 初始化

在文件 kernel/src/arch_x86_64/cpu.rs 中定义的函数 init_cpu 用于进行 CPU 初始化。

```rust
/// Initialize the Advanced Programmable Interrupt Controller.
pub fn init_cpu() -> KResult<()> {
    init_apic()?;

    kinfo!(
        "init_cpu(): {:#x?}",
        LOCAL_APIC.read().get(&cpu_id()).unwrap().get_info(),
    );

    unsafe {
        enable_float_processing_unit();
    }

    Ok(())
}
```

该函数功能具体通过如下几个步骤实现：

1. init_apic：启动 Advanced Programmable Interrupt Handler，用于初始化各个核心的中断处理器。APIC 的具体功能和实现请查询 wiki
2. enable_float_processing_unit：启动浮点单元，启动后可以计算浮点数。

6. PCI 总线初始化

在文件 kernel/src/drivers/pci_bus.rs 中定义的函数用于进行 PCI 总线初始化。

```rust
pub fn init_pci() -> KResult<()> {
    let devices_connected = unsafe { pci::scan_bus(&Ops, CSpaceAccessMethod::IO)
    ↪ };

    for device in devices_connected.into_iter() {
        kinfo!(
            "init_pci(): {:02x}:{:02x}.{} {:#x} {:#x} ({} {}) irq: {}:{:?}",
            device.loc.bus,
            device.loc.device,
            device.loc.function,
            device.id.vendor_id,
            device.id.device_id,
            device.id.class,
```

```
            device.id.subclass,
            device.pic_interrupt_line,
            device.interrupt_pin,
        );

        // Initialize this device.
        init_device(&device)?;
    }

    Ok(())
}
```

PCI 总线上链接了网卡、SATA 设备，主要功能是探测总线上的设备，之后初始化这些设备并分配 MMIO、DMA 等用于访问它们的内存区域。其中的 init_device 用于初始化，定义了硬盘和网卡初始化的相关函数。

7. ACPI 初始化

在文件 kernel/src/arch/x86_64/acpi.rs 中定义的函数用于进行定义 ACPI 初始化。

```
pub fn init_acpi(header: &Header) -> KResult<()> {
    let handler = AcpiHandlerImpl {};
    let table = match unsafe { AcpiTables::from_rsdp(handler,
    ↪ header.acpi2_rsdp_addr as usize) } {
        Ok(table) => table,
        Err(e) => {
            kerror!("init_acpi(): acpi parse kerror: {:?}", e);
            return Err(Errno::EINVAL);
        }
    };
    }
```

ACPI 是硬件启动后生成的一些硬件描述信息，与 APIC 应作区分。应当禁用其中包含的 PIC 信息（也就是 APIC 的前身），记录其中包含的 IRQ 的掩码。同时在此函数中初始化高精度时钟 HEPT（qemu 不支持），并初始化其他 CPU 核心。

8. CPU 其他核心启动

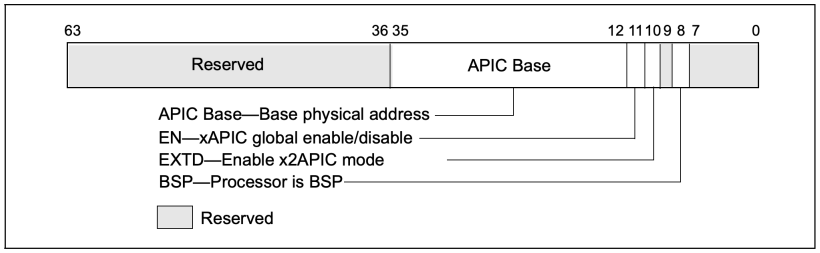

Figure 2-1. IA32_APIC_BASE MSR Supporting x2APIC

NeoOS 是一个有多核心 CPU 操作系统。在多核心 CPU 模型下,默认启动的核心为 Bootstrap Processor（BSP），其他核心为 Application Processor（AP）。AP 的启动时通过 Inter-Processor Interrupt（IPI）。具体实现步骤如下：

1. 发送一次 INIT IPI，告知对应核心应当被唤醒。
2. 发送一次 STARTUP INIT，并等待执行，通过 IPI 给出的跳转指令进行跳转。其相关代码定义于 kernel/src/arch/x86_64/boot/ap_trampoline.S。
3. 其他核心进入初始化并启动成功。

```
__ap_startup:
    cli
    cld

    xor ax, ax
    mov ds, ax
    mov es, ax
    mov ss, ax

    ; Load the address of the page table in advance.
    mov edi, [__ap_trampoline.page_table]
    mov cr3, edi

    lgdt [__gdtr]

    mov eax, cr0
    or eax, 1
    mov cr0, eax

    ; To load correct value of cs we must perform a long jump.
    jmp 0x8:__ap_start_32
```


该文件实现的流程为：实模式-> 平坦模式-> 保护模式-> 设置 64 位 GDT-> 进入 kernel_start。

9. 信号处理

在文件 kernel/src/arch/signal/mod.rs 中定义的函数用于实现和 unix 一样的信号处理机制。

10. 进程与线程

在 kernel/src/process 文件夹中定义的代码用于实现进程和线程机制。在该目录下，mod.rs 是进程实现代码，thread.rs 是线程实现代码，scheduler.rs 实现了简单的先进先出的线程调度器。

进程进程和线程大体上很相近，区别如下：

- 线程共享内存，但具有单独的寄存器。
- 线程共享文件对象。
- 线程有单独的信号处理机制。

- 进程统一管理虚拟内存。

进程和线程的虚拟内存段的分配单位是 arena，每段内存会有一个 callback 来辅助实现内存页的映射、换进换出、缺页中断处理等等。ELF 文件映射通过 kernel/src/elf.rs 实现。

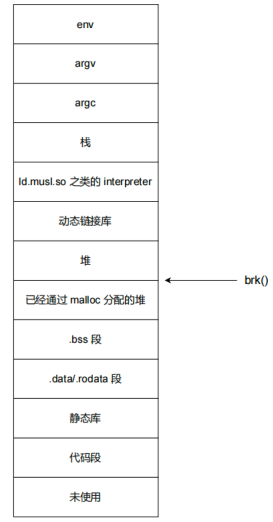

图 2: 进程虚拟和线程的虚拟内存管理

11. 文件系统与虚拟文件系统

文件系统使用了 LINUX-APFS 作为支持 [1] [2]。在 kernel/src/process 文件夹中定义了实现文件系统的相关代码，实现了以下功能：

- Apple Filesystem（只读，内存写）
- rCore Simple Filesystem（支持读写）
- /dev 设备虚拟文件系统（fs/devfs）
- /proc 进程相关文件系统，可用于获取内存映射等进程对应的元数据（fs/proc）

文件对象为 fs/file.rs，与 unix 一样做了抽象。文件系统的 filesystem 和 Inode 的抽象层使用 trait 设计。fs 文件夹结构如下所示：

```
./kernel/src/fs
├── apfs
├── devfs
├── epoll.rs
├── file.rs
├── mod.rs
├── proc
└── sfs
```

图 3: 文件系统层次结构

12. 网络栈

在 kernel/src/net 文件夹中定义的代码用于实现网络栈。具体实现了以下功能：

- TCP socket
- UDP socket
- Raw socket

网络的 socket 使用 trait 实现，底层实现采用了 smoltcp 第三方库 [5]。硬件层自 rcore 的 crate 修改而来，代码实现于 kernel/src/drivers/ isomorphic_drivers/net/ethernet/intel/e1000.rs 文件。

图 4: 网络层次结构

（五） NeoOS 内核组件说明

1. 虚拟内存管理

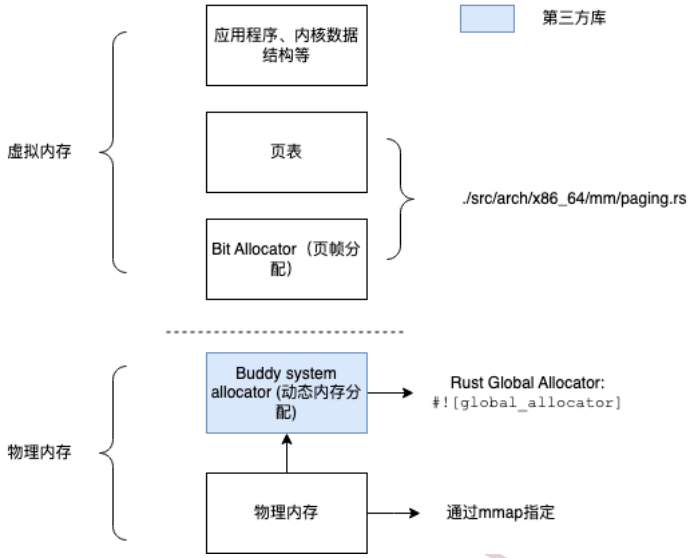

图 5: 内存管理

x86_64 架构下的页表相关代码定义于 kernel/src/x86_64/mm/paging.rs 文件中，内核管理用户虚拟内存定义于 kernel/memory.rs 和 kernel/src/mm/mod.rs 和 callback.rs 文件中。

2. APIC

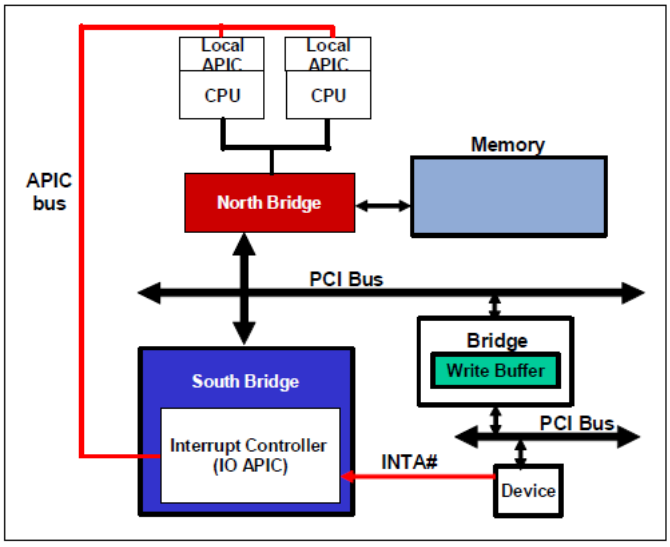

图 6: APIC 架构图

相关代码定义于 kernel/src/arch/x86_64/apic 文件中。其中，IOAPIC 负责将 IRQ 和中断分发给对应核心；LAPIC 是每个 CPU 核心所对应的 APIC，Intel 目前所采用的是 x2APIC 架构，在此之前则采用 xAPIC 架构。

3. 同步原语

相关代码定义于 kernel/src/sync。相关代码实现了如下机制：

- 基于自旋的互斥锁（mutex.rs），支持屏蔽中断和允许中断。
- 快速用户空间的互斥锁 (futex.rs)。
- Raw Mutex：暂时未在 NeoOS 中使用，可用来实现类似条件变量的机制，之后将进制挂起到 parking lot。
- Rust 中实现同步以及所有权限的一些问题。

四、 帮助文档

本项目在项目仓库的首页提供一个Readme.md 文件，提供了关于项目的简单介绍。
至于实验手册，我们提供了中文、英文两种版本，分别放置于项目仓库目录之下。

五、 总结与未来展望

总而言之，目前我们已经完整地实现了一个操作系统，包括内核和文件系统以及网络栈等部分，这些已经完成的内容已经足够我们设计一个操作系统实验了。然而，目前的项目仍然存在一些不足，比如目前我们所设想的实验步骤仅仅是我们在开发这个项目时的开发步骤，这与一个实验设计的步骤并不能够严格匹配。同时，这个项目目前也没有设计出足够多的测试样例和测试点，这也有待我们进一步设计和尝试。同时，目前这个实验所涉及的难度没有区分度，并没有区分出必做、选作等内容，也许这也是一个完善的实验未来可以提供的方向。

因此，我们对项目提出如下展望，这些内容将在未来进一步开发：

- 实现更多的多核调度算法
- 完善网络栈的更多技术细节
- 进一步丰富文件系统，使其支持更多系统
- 针对不同功能加入足够的测试样例
- 编写更加详细的实验文档，特别是在设计理念部分还需要更多内容
- 将整个项目划分为 5-10 个小的 Lab，使其层层递进、环环相扣
- 设计更多有区分度的实验，将其分为必做和选作
- 整体优化，并在 Github 上托管开源的实验指导书，并考虑设计课程网站

参考文献

[1] Apfsprogs, 2023. https://github.com/linux-apfs/apfsprogs.

[2] linux-apfs, 2023. https://github.com/linux-apfs/linux-apfs-rw.

[3] Os dev wiki, 2023. https://wiki.osdev.org/Expanded_Main_Page.

[4] rcore 社区, 2023. http://rcore-os.cn/rCore-Tutorial-Book-v3/index.html.

[5] smoltcp 文档, 2023. https://docs.rs/smoltcp/latest/smoltcp.