華東師範大學
EAST CHINA NORMAL UNIVERSITY

proj0 – 面向操作系统课程的操作系统竞赛和实验

题目二：适合本校特点的实验指导教程

基于Rust与xv6的操作系统教学方案设计与实现

汇报队伍：SEIOS 队员：郑子攸，叶晨皓，周恒如 指导老师：郭建

Contents

总览

華東師範大學
EAST CHINA NORMAL UNIVERSITY

華東師範大學
EAST CHINA NORMAL UNIVERSITY

第一部分

项目介绍

1.1 项目简介

華東師範大學
EAST CHINA NORMAL UNIVERSITY

- Rust语言是目前操作系统研究的重要方向，但**现有的操作系统课程以C语言实现为主**
- 本项目旨在填补基于Rust语言的教学操作系统课程目前存在的空白，**提供一种同时学习Rust与操作系统的可行路径**
- 本项目实现了一套操作系统实践课程，主要完成了如下工作
  - 系统内核架构分析与核心模块文档注释编写
  - 学生实验设计，实验指导手册及参考实现编写
  - 实验平台设计，自动化评测学生实验，提供内核调试功能
  - Rust语言重写用户程序，实现用户与内核线程

1.2 参考情况与增量工作说明

華東師範大學
EAST CHINA NORMAL UNIVERSITY

- 本项目基于xv6-riscv-rust的系统内核与MIT 6.S081课程内容实现，增量工作如下
- 对内核核心模块（内存管理，进程管理，文件系统）**编写了完善的文档注释**
- **使用Rust重写xv6的用户程序**，并设计基于Rust语言用户程序的实验
- **基于Rust语言实现了用户线程库**，用于设计多线程实验，填补xv6中无法进行多线程实验的空缺
- **对内核核心模块新增线程管理和共享变量管理**，完善xv6在内核多线程上的不足
- **实现了学生实验平台**，提供自动化评测、内核调试、反汇编输出等功能
- **完成12个实验设计与实验指导手册及参考实现的编写**，并提供实验阅读材料

華東師範大學
EAST CHINA NORMAL UNIVERSITY

第二部分

前期调研

2.1 调研内容

華東師範大學
EAST CHINA NORMAL UNIVERSITY

学校/项目	编程语言	实验难度	工具链和平台	实验内容与教学价值概要
MIT (xv6)	C	覆盖完善	RISC-V (QEMU模拟器)	覆盖完整操作系统功能（系统调用、内存管理、中断、文件系统等），实验内容深入全面，适合高年级本科生，强调OS完整性与工程实践能力。
清华大学 (rCore)	Rust	中等	RISC-V (QEMU, K210开发板)	从零构建类Unix OS，全面覆盖内核启动、内存/进程管理、文件系统及驱动；适合初学者，以Rust语言强调内存安全与现代OS理念。
中科院软件所 (RVOS)	C	较低	RISC-V (QEMU模拟器、 Docker环境)	实验聚焦于核心OS机制（引导启动、任务管理、中断处理等），难度梯度小，适合初学者建立OS基本概念与实践技能。
南京大学 (JYY OS)	C	较高	抽象机平台 (QEMU模拟 x86/RISC-V)	涉及深入的OS理论和实践（线程/进程管理、文件系统设计与实现），实验强度高、理论性强，适合高年级本科生与研究生深入研究。
华中科大 (PKE)	C	中等	RISC-V (Spike精确模拟, QEMU与FPGA硬件)	分模块逐步构建“代理内核”，覆盖特权级切换、内存管理、设备驱动与进程调度；强调软硬结合的工程实践，适合本科生掌握实际应用场景下的内核开发能力。

2.2 主要相似工作 – MIT (xv6)

華東師範大學
EAST CHINA NORMAL UNIVERSITY

- MIT 6.828 课程通过一系列以 xv6 操作系统为平台的实验，系统地覆盖了现代操作系统的核心功能，包括用户程序与系统调用接口、异常与中断处理、虚拟内存与页表管理、内存优化策略（如惰性分配与写时复制）、线程与同步机制、文件系统结构以及设备驱动与网络通信。
- 每个实验从实际功能出发，要求学生亲自阅读、修改和扩展内核代码，循序渐进地构建完整的操作系统理解模型。这种“由简入深”的实践方式不仅锻炼了学生系统级编程能力，更加深了其对操作系统设计思想、内核机制和性能权衡的掌握，是操作系统教育中极具代表性的训练体系。这门课程共包含11组实验。

2.2 主要相似工作 – MIT (xv6)

華東師範大學
EAST CHINA NORMAL UNIVERSITY

- **Lab1: Utilities** 实现常见 UNIX 工具掌握用户态程序构建与系统调用接口，奠定操作系统基础编程能力。
- **Lab2: System Calls** 添加新系统调用深入理解系统调用机制及用户态与内核态之间的交互与信息传递。
- **Lab3: Page Tables** 多级页表打印与内核页表隔离，深入掌握虚拟地址映射与页表管理策略。
- **Lab4: Traps** 实现调用栈回溯与用户级定时器功能，理解 trap 机制与用户态中断控制流切换。
- **Lab5: xv6 Lazy Page Allocation** 实现按需分配机制，掌握缺页异常处理与虚拟内存管理的延迟映射策略。
- **Lab6: Copy-on-Write Fork for xv6** 改造 fork() 为写时复制机制，深入掌握内存共享优化与页错误处理技术。
- **Lab7: Multithreading** 构建用户级线程库并进行并发测试，掌握线程切换、同步机制及并发程序设计。
- **Lab8: Locks** 实现并测试多核锁机制，强化对内核同步、竞态与死锁问题的理解与解决能力。
- **Lab9: File System** 扩展 xv6 文件系统支持大文件与符号链接，掌握文件索引结构与路径解析机制。
- **Lab10: Mmap** 实现内存映射文件访问，深入理解文件 I/O 与虚拟内存系统的集成原理。
- **Lab11: Networking** 开发网卡驱动与基础网络协议支持，掌握内核网络栈、DMA 通信与中断处理机制。

2.3 主要相似工作 – 清华大学 (rCore)

華東師範大學
EAST CHINA NORMAL UNIVERSITY

- rCore 操作系统实验通过覆盖日志输出、系统调用、安全检查、进程调度、虚拟内存、文件系统、进程通信、并发同步及图形界面等核心模块，系统地培养学生用 Rust 开发操作系统的能力，强化其对内核结构和抽象机制的理解。
- 实验内容具备较强的工程实践性，能帮助学生掌握 Rust 在内核中的高安全性与高性能用法。然而，其入门门槛较高，难度递进陡峭，且实验文档在关键实现细节与调试指导上略显不足，容易使初学者在复杂模块中卡顿，适合有一定操作系统基础知识的专业学生进行学习。

2.3 主要相似工作 – 清华大学 (rCore)

華東師範大學
EAST CHINA NORMAL UNIVERSITY

- **Lab1：彩色化 LOG** 通过实现彩色日志输出和内存布局显示，掌握操作系统的输出机制与调试可视化设计。
- **Lab2：sys_write 安全检查** 通过为 sys_write 添加地址合法性检查，理解系统调用中的内存隔离与安全边界。
- **Lab3：获取任务信息** 实现任务状态查询系统调用，掌握任务控制块扩展与内核任务监控机制。
- **Lab4：重写 sys_get_time** 在适配虚拟内存的基础上实现 mmap/munmap 系统调用，深入理解内存映射与权限控制。
- **Lab5：进程创建&&stride 调度算法** 通过实现新进程创建方式与 stride 调度算法，掌握进程生命周期管理与调度策略设计。
- **Lab6：硬链接** 实现硬链接与 inode 引用计数机制，深入理解文件系统中的元数据管理与目录结构映射。
- **Lab7：进程通信：邮箱** 通过构建邮箱机制实现进程间通信，掌握内核缓冲区管理与非阻塞通信接口设计。
- **Lab8：银行家算法** 通过用户态同步模型与 eventfd 系统调用实现，掌握并发控制与异步事件通知机制。
- **Lab9：支持图形显示的应用** 实现 GPU 驱动与帧缓冲设备，掌握图形显示路径与图形界面下的设备驱动开发方法。

華東師範大學
EAST CHINA NORMAL UNIVERSITY

第三部分

技术方案

3.1 内核与文档注释

- **基于Rust语言的系统内核**
  - 系统内核绝大部分使用Rust语言实现，小部分使用RISC-V汇编实现
  - 系统结构与xv6保持一致，提供相同的系统调用接口，由内存管理、进程管理和文件系统等模块组成

- **内核文档注释**
  - 使用Rust文档注释格式，**为系统内核核心模块（内存，进程，文件）的所有函数、结构体以及全局变量编写注释**
  - 通过执行cargo doc指令，能够自动生成HTML格式的内核文档，便于学生快速查找阅读
  - 函数的文档注释内容完善，**包含功能说明，参数，返回值，可能出错的情况**
  - **unsafe函数还额外包含安全性说明**
  - 文档的编写是由手工编写与大模型生成相结合的，所有文档均经过人工查验，确保其内容的正确性

3.2 Rust用户程序与实验平台

華東師範大學
EAST CHINA NORMAL UNIVERSITY

- **Rust语言用户程序以及多线程库**
  - 本项目**使用Rust语言重写了xv6的用户程序**
  - 支持在内核上**同时运行C语言与Rust语言编写的用户程序**
  - 基于Rust语言实现的用户程序进一步**实现用户多线程库以及内核多线程库**
  - 用于设计多线程实验，**填补xv6中无法进行多线程实验的空缺**
- **学生实验平台**
  - 设计并实现学生实验平台，**能够自动化评分，提供内核调试以及反汇编输出功能**，帮助学生学习操作系统
  - 本项目参考了xv6的自动化测试脚本，使用Python实现了对学生实验的全自动评测与分数计算
  - 该脚本将评测点与评测库分离，**能够自动收集评测点，灵活调整评测方案**
  - 自动编译系统内核并重置文件系统，能够与QEMU-GDB通信，追踪内核输出，保证评测有效性

3.3 实验指导手册与参考实现

華東師範大學
EAST CHINA NORMAL UNIVERSITY

- **实验指导手册与参考实现**
  - 实验指导手册提供了**实验背景、实验题干、实现思路以及具体提示**，可以通过向学生提供不同的内容以调整实验难度
  - 实验的设计综合了MIT 6.828课程近五年的所有实验，参考了其中难度合适的实验，并设计了部分原创实验
  - 每个实验都**提供了参考实现，以及参考实现的代码解析**
  - 参考实现均经过实机运行，能够在测试脚本下得到满分

華東師範大學
EAST CHINA NORMAL UNIVERSITY

第四部分

系统内核架构

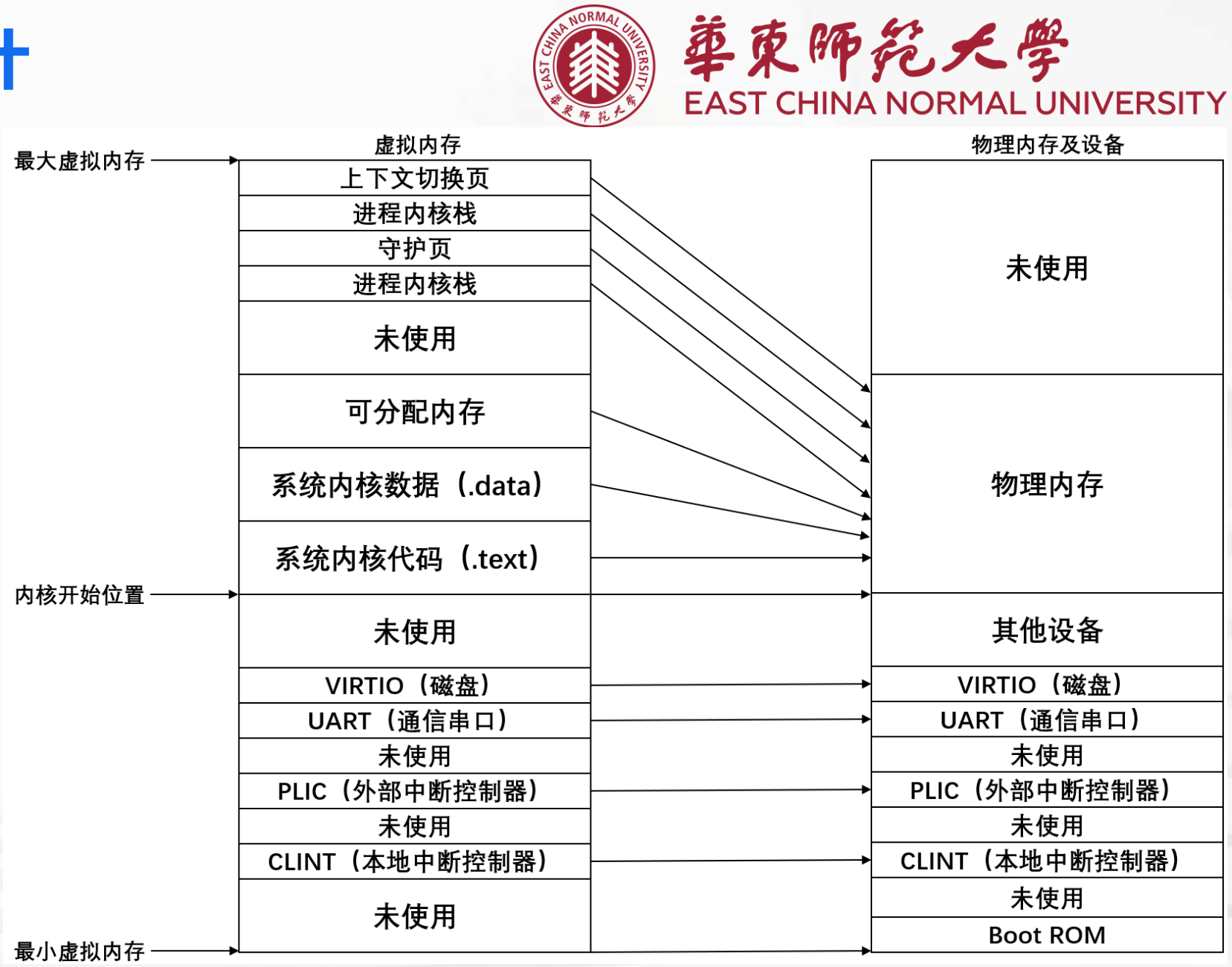
4.1 内核整体架构

華東師範大學
EAST CHINA NORMAL UNIVERSITY

- 内存管理模块：物理内存管理，虚拟内存管理
- 进程管理模块：中断处理，进程调度，进程控制，进程同步
- 文件系统模块：物理文件系统，虚拟文件系统

<table>
<tr><td rowspan="4">内核</td><td colspan="3">系统调用接口</td></tr>
<tr><td>内存管理模块</td><td>进程管理模块</td><td>文件管理模块</td></tr>
<tr><td>虚拟内存管理</td><td>进程控制与同步</td><td>虚拟文件系统</td></tr>
<tr><td>物理内存管理</td><td>中断处理与进程调度</td><td>物理文件系统</td></tr>
<tr><td>硬件驱动</td><td>RISC-V 页表</td><td>中断控制器</td><td>串口与磁盘驱动</td></tr>
</table>

4.2 内存布局设计

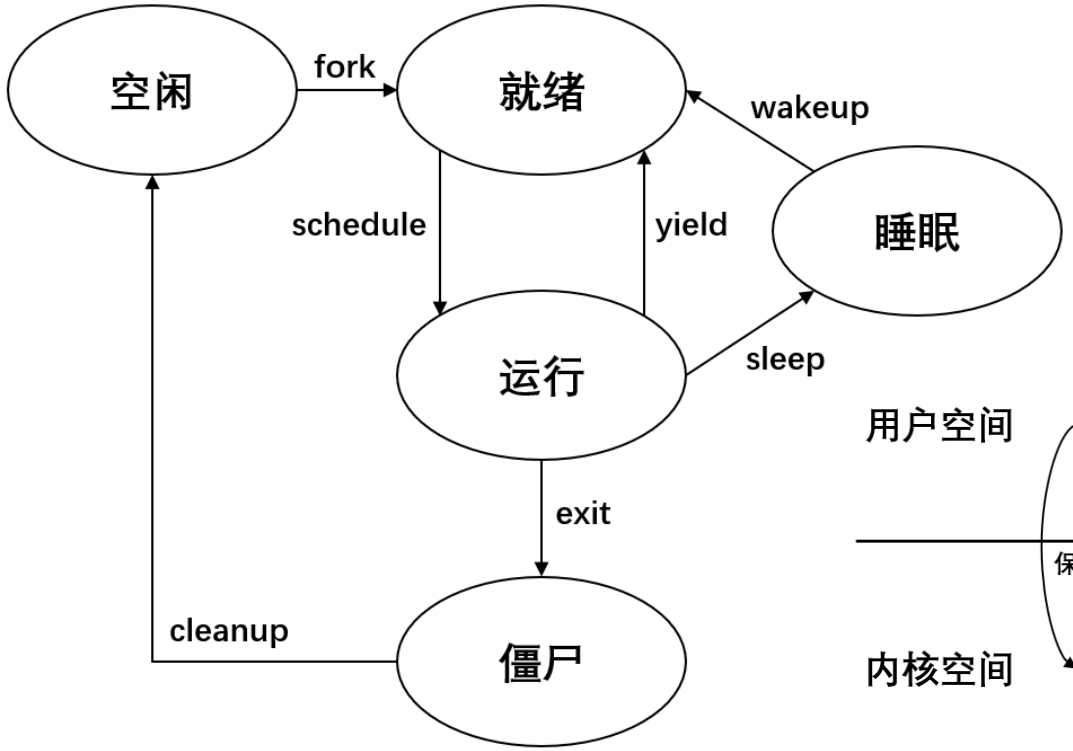


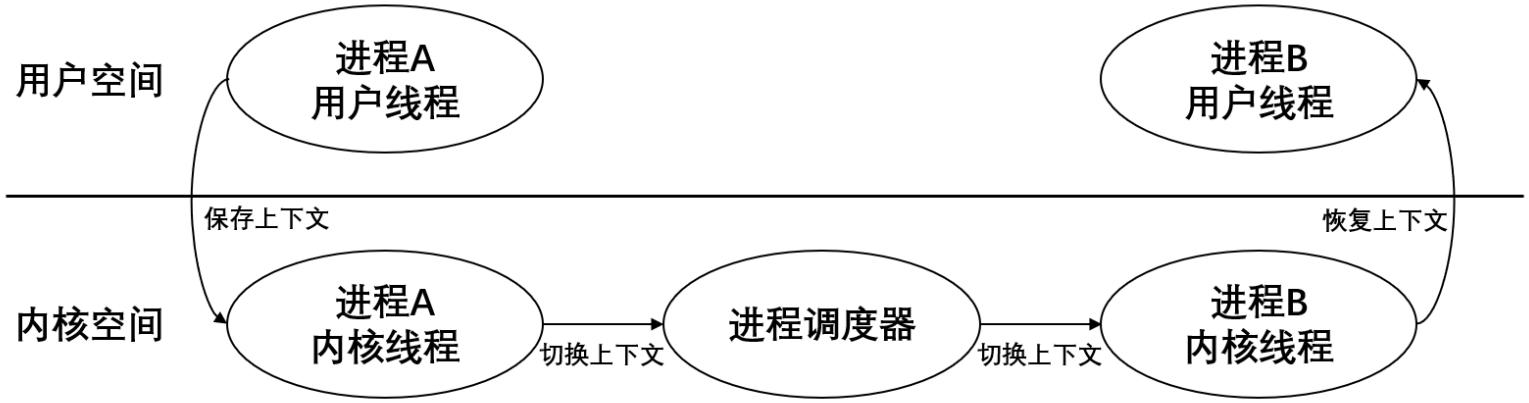
- 对硬件设备和系统内核采用直接映射，简化内核态内存访问的复杂性
- 进程内核栈以及上下文切换页存放在虚拟地址最高处，便于用固定偏移量寻址

4.3 进程管理设计

华東師範大學 EAST CHINA NORMAL UNIVERSITY

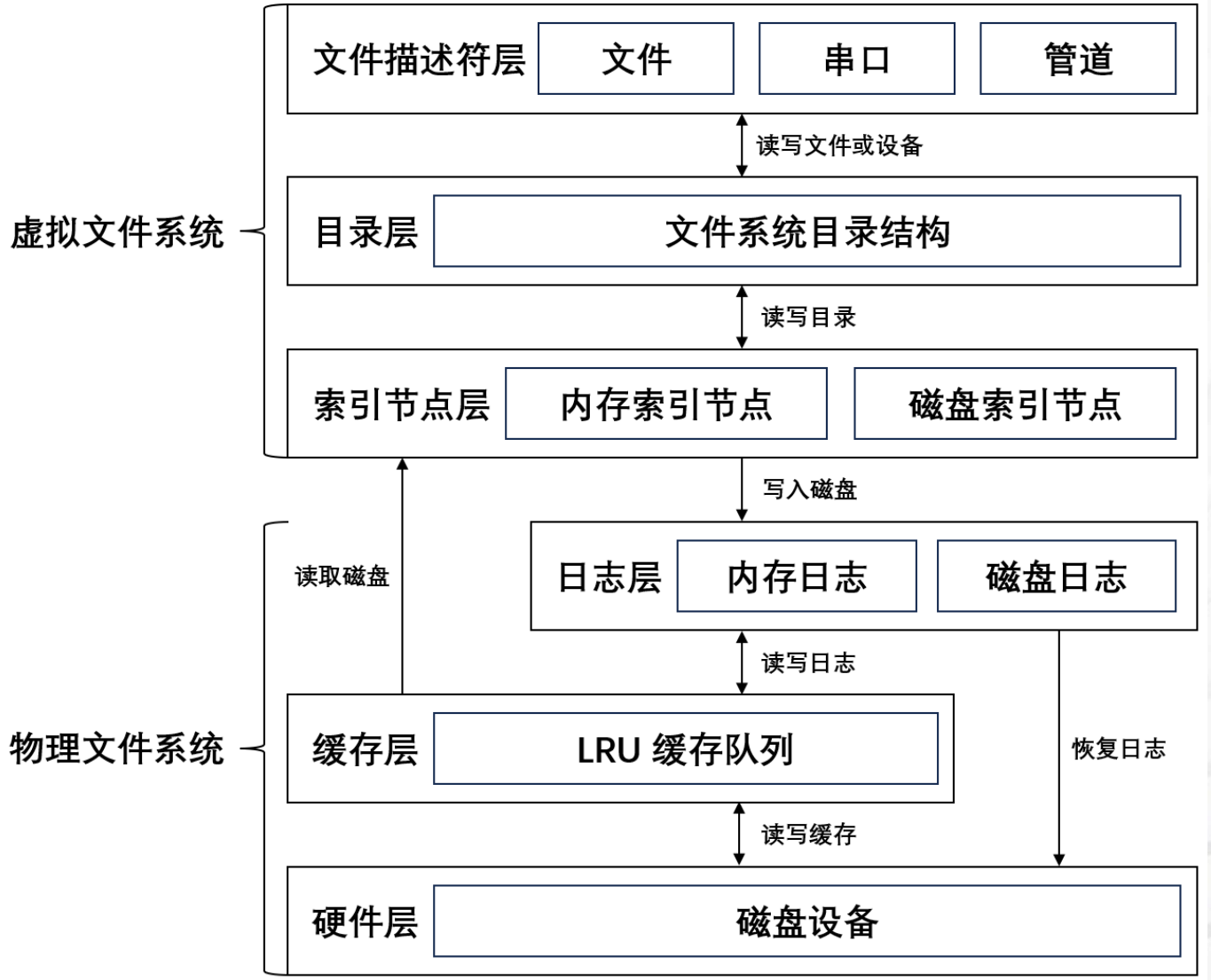
- 用户程序通过fork，wait等系统调用接口进行进程控制
- 进程在空闲，就绪，运行，睡眠，僵尸状态间切换
- 进程切换时首先进入其内核线程，再通过进程调度器切换至其他进程





4.4 文件系统设计

- 文件系统采用六层设计
- 硬件层，缓存层，日志层形成物理文件系统
- 索引节点层，目录层，文件描述符层形成虚拟文件系统



華東師範大學
EAST CHINA NORMAL UNIVERSITY

第五部分

内核文档注释

5.1 文档注释的意义

華東師範大學
EAST CHINA NORMAL UNIVERSITY

- 在本项目中，我们针对基于 Rust 语言实现的类 xv6 教学操作系统的内核代码，**围绕内存管理、进程管理、文件系统三个核心模块，系统性地补充了文档注释。**
- 注释内容涵盖**函数功能说明、调用流程解释、参数与返回值说明**，并根据具体语义补充了**可能的错误场景与安全性提示。**该工作对于操作系统课程的教学与学生的实验实践具有显著意义。
- **意义一：降低理解操作系统内核的难度**
  - 内核代码复杂且耦合度高，初学者难以直接理解调用逻辑。
  - 调用链中存在大量未知函数，容易造成阅读断层与认知障碍。
  - 文档注释提供函数功能说明与上下文解释，有助于快速理解整体逻辑。
  - Rust 引入的所有权与生命周期机制进一步增加理解难度。
  - 注释中对 Rust 类型系统特性的解释，有效降低语言本身的学习门槛。

5.1 文档注释的意义

華東師範大學
EAST CHINA NORMAL UNIVERSITY

- **意义二：提升内核调试的效率**
  - 内核态调试难度大，传统工具在处理底层问题时效果有限。
  - 实验中常见的 bug 多涉及汇编、中断或内存操作，定位困难。
  - 注释中提供异常场景与错误提示，有助于学生快速建立调试思路，提升调试效率，减少实验挫败感。
- **意义三：帮助学生进行内核开发，高效完成实验内容**
  - 传统教学偏重接口使用，学生难以深入掌握内核实现细节，注释提升了内核代码的可读性与可维护性。
  - 为学生提供可理解、可修改的开发基础，鼓励学生从完成实验转向探索与优化内核，培养开发者思维。
- **意义四：增加对 Rust 语言以及 Unsafe 代码的理解**
  - Rust 提供内存与并发安全保障，但内核开发中仍需使用 `unsafe`。
  - 注释中指出安全风险与语言层面的防御机制，帮助学生建立系统级编程中的安全意识。
  - 强化操作系统与嵌入式领域开发的基础能力。

5.2 文档注释的结构

華東師範大學
EAST CHINA NORMAL UNIVERSITY

- 本项目在实现内核文档时，参照上述标准库文档实现，根据不同函数的特性，包含了不同的文档段落
- 对于普通函数，文档内容包含
  - **函数功能说明**：使用简短的语言简述函数的功能，帮助学习者快速了解函数功能
  - **流程解释**：详细说明函数的执行流程及其具体实现，帮助无法读懂具体代码的学习者理解细节
  - **参数与返回值说明**：解释函数的输入与输出值
  - **可能的错误场景**：对Option，Result出现失败变体，或函数可能发生panic的情况进行说明，帮助学习者了解出错的可能情况
- 对于Unsafe函数，额外添加
  - **安全性说明**：详细说明该Unsafe函数需要人为保证的前提条件，否则可能造成系统进入不安全的状况
  - 有助于帮助学生快速了解使用Rust语言进行操作系统开发时需要注意的Unsafe代码使用问题

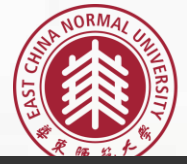
5.3 文档注释的覆盖范围

華東師範大學
EAST CHINA NORMAL UNIVERSITY

- 本项目的文档注释已经覆盖
- **内存管理模块**
- **进程管理模块**
- **文件系统模块**
- **并发控制、中断处理、系统启动、设备驱动等关键代码**

```
├── driver
│   ├── console.rs
│   ├── mod.rs
│   ├── uart.rs
│   └── virtio_disk.rs
├── fs
│   ├── bio.rs
│   ├── block.rs
│   ├── file
│   │   ├── mod.rs
│   │   └── pipe.rs
│   ├── inode.rs
│   ├── log.rs
│   ├── mod.rs
│   └── superblock.rs
├── lib.rs
├── main.rs
├── mm
│   ├── addr.rs
│   ├── kalloc.rs
│   ├── kvm.rs
│   ├── list.rs
│   ├── mod.rs
│   └── pagetable.rs
├── plic.rs
├── printf.rs
├── process
│   ├── context.rs
│   ├── cpu.rs
│   ├── mod.rs
│   ├── proc
│   │   ├── elf.rs
│   │   ├── mod.rs
│   │   └── syscall.rs
│   └── trapframe.rs
├── rmain.rs
├── sleeplock.rs
├── spinlock.rs
├── start.rs
└── trap.rs
```

5.4 文档注释的效果 – 进程控制块



華東師範大學
EAST CHINA NORMAL UNIVERSITY

Type 'S' or '/' to search, '?' for more options...

xv6_rust::process::proc

Struct Proc

Source

Settings Help Summary

```
pub struct Proc {
    index: usize,
    pub excl: SpinLock<ProcExcl>,
    pub data: UnsafeCell<ProcData>,
    pub killed: AtomicBool,
}
```

进程结构体，代表操作系统内核中的一个进程实体。

该结构体封装了进程在进程表中的索引， 进程状态的排它锁保护数据（`ProcExcl`）， 进程私有数据（`ProcData`）， 以及进程是否被杀死的原子标志。

通过该结构体，操作系统能够管理进程调度、状态更新和资源访问的并发安全。

Fields

`index: usize`

进程在进程表中的索引，唯一标识该进程槽位。

`excl: SpinLock<ProcExcl>`

进程排它锁保护的状态信息，包括状态、pid、等待通道等。

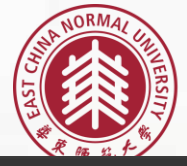
`data: UnsafeCell<ProcData>`

进程私有数据，包含内存、上下文、文件描述符等，通过 UnsafeCell 实现内部可变性。

`killed: AtomicBool`

标识进程是否被杀死的原子布尔变量，用于调度和信号处理。

5.4 文档注释的效果 – Syscall函数



華東師範大學
EAST CHINA NORMAL UNIVERSITY

xv6_rust
0.1.0

Proc

Fields
- data
- excl
- index
- killed

Methods
- abondon
- arg_addr
- arg_fd
- arg_i32
- arg_raw
- arg_str
- check_abondon
- fetch_addr
- fetch_str
- fork
- new
- sleep
- syscall
- user_init
- yielding

Trait

```
pub fn syscall(&mut self)
```
Source

功能说明

处理当前进程发起的系统调用请求。根据 TrapFrame 中寄存器 a7 指定的系统调用号， 调用对应的系统调用处理函数，并将返回结果写回寄存器 a0。

流程解释

1. 使能中断，允许系统中断处理。
2. 通过不安全代码获取当前进程的 TrapFrame 指针，读取系统调用号 `a7`。
3. 调用 `tf.admit_ecall()`，完成系统调用的相关状态处理（如跳过指令等）。
4. 使用 `match` 匹配系统调用号，调用对应的系统调用实现函数。
5. 若系统调用号非法，调用 `panic!` 抛出异常，终止内核执行。
6. 将系统调用执行结果写入 TrapFrame 的返回寄存器 `a0`， 成功返回实际结果，失败返回 -1（以 `usize` 格式存储）。

参数

- `&mut self`：当前进程的可变引用，用于访问其 TrapFrame 和调用系统调用实现。

返回值

- 无返回值，系统调用结果通过 TrapFrame 的 `a0` 寄存器返回给用户态。

可能的错误

- 系统调用号非法时，会导致内核 panic，内核崩溃或重启。
- 各个系统调用具体实现可能返回错误，统一映射为返回值 -1。

安全性

- 使用了 `unsafe` 获取 TrapFrame 裸指针，假设指针有效且唯一所有权。
- 该函数应在内核上下文且进程排他访问时调用，避免数据竞争。
- 系统调用执行过程中可能包含更底层的 `unsafe`，调用此函数时需确保整体安全环境。

5.5 文档注释的效果

華東師範大學
EAST CHINA NORMAL UNIVERSITY

- 目前，本项目的文档注释已经基本覆盖内存管理模块、进程管理模块和文件系统模块，使用tokie统计注释行数，目前文档注释总量为4710行。

Language	Files	Lines	Code	Comments	Blanks
GNU Style Assembly	6	390	341	6	43
C	21	4956	4197	225	534
C Header	12	640	455	97	88
Makefile	1	127	94	7	26
Markdown	3	198	0	177	21
Perl	1	38	31	2	5
Python	2	664	514	40	110
TOML	2	28	22	1	5
Rust	49	6978	5557	294	1127
|- Markdown	47	5909	0	4710	1199
(Total)		12887	5557	5004	2326
Total	97	14019	11211	849	1959

華東師範大學
EAST CHINA NORMAL UNIVERSITY

第六部分

Rust语言用户程序与线程库

6.1 Rust语言用户程序

華東師範大學
EAST CHINA NORMAL UNIVERSITY

- 在本项目中，操作系统的整体架构与 MIT xv6 保持一致，采用相同的系统调用接口与用户态启动方式。
- 为进一步统一语言生态、提升教学一致性，已经使用 Rust 重写大部份测试用例，构建一个全栈基于 Rust 的教学操作系统与测试平台，帮助学生理解Rust语言安全性。

6.2 用户程序库基础结构

華東師範大學
EAST CHINA NORMAL UNIVERSITY

由于在操作系统基础上编写用户程序，无法直接使用Rust的标准库。

SEIOS用户程序库（在no_std环境下）结合操作系统系统调用接口、alloc官方库，core官方库和部份第三方库进行进行实现。

- 用户程序入口
  - 通过link.ld进行用户代码链接。 通过ENTRY(_start)来指定用户程序入口函数 。
  - OUTPUT_ARCH(riscv)指定目标架构为RISC-V。
  - ENTRY(_start)指定程序的入口点为_start符号（用户程序启动代码）。
  - BASE_ADDRESS = 0x10000表示程序加载的基地址为0x10000。
  - .text 段包含所有函数代码。 .rodata 段存储字符串常量（如日志信息）。 .data 和 .bss 分别存储初始化和未初始化的全局变量。

6.2 用户程序库基础结构

華東師範大學
EAST CHINA NORMAL UNIVERSITY

- 用户栈中的堆内存分配
  - 基于buddy_system_allocator库实现的用户栈中内存分配器初始化配置
- _start
  - 1.初始化用户栈中的堆内存
  - 2.从内核初始化的进程用户栈中获取参数argc: usize, argv: usize（内核参数设置符合函数调用规范） seios操作系统内核已经将参数压入用户栈中，以下start_中将符合C语言标准的参数转换为符合Rust语言标准的参数（处理字符串）
  - 3.执行用户编写的主函数fn main(argc:usize, argv:&[&str]) -> i32
  - 4.退出，执行exit系统调用，保存返回值并退出进程。

6.3 系统调用接口

華東師範大學
EAST CHINA NORMAL UNIVERSITY

SYSCALL_FORK(syscall_id = 1) 功能：创建当前进程的一个子进程，复制父进程的内存、TrapFrame、打开文件、当前工作目录等信息，并将子进程状态设置为可运行
SYSCALL_EXIT(syscall_id = 2) 功能：终止当前进程，释放其资源，并向父进程传递退出状态
SYSCALL_WAIT(syscall_id = 3) 功能：等待任意子进程终止，并获取其退出状态
SYSCALL_PIPE(syscall_id = 4) 功能：创建一个管道，用于进程间通信（IPC）
SYSCALL_READ(syscall_id = 5) 功能：从文件描述符读取数据。
SYSCALL_KILL(syscall_id = 6) (未实现sig：信号编号) 功能：向指定进程发送信号
SYSCALL_EXEC(syscall_id = 7) 功能：加载并执行新程序，替换当前进程的地址空间
SYSCALL_FSTAT(syscall_id = 8) 功能：获取文件描述符对应的文件状态信息。
SYSCALL_CHDIR(syscall_id = 9) 功能：更改当前进程的工作目录。
SYSCALL_DUP(syscall_id = 10) 功能：复制文件描述符

6.3 系统调用接口

華東師範大學
EAST CHINA NORMAL UNIVERSITY

SYSCALL_GETPID(syscall_id = 11) 功能： 获取当前进程的 PID
SYSCALL_SBRK(syscall_id = 12) 功能： 调整进程的堆内存大小
SYSCALL_SLEEP(syscall_id = 13) 功能： 使进程休眠指定时间
SYSCALL_UPTIME(usize = 14) 功能： 获取系统启动后的时钟周期数
SYSCALL_OPEN(usize = 15) 功能： 打开或创建文件
SYSCALL_WRITE(usize = 16) 功能： 向文件描述符写入数据。
SYSCALL_MKNOD(usize = 17) 功能： 创建设备文件或特殊文件
SYSCALL_UNLINK(usize = 18) 功能： 删除文件链接
SYSCALL_LINK:(usize = 19) 功能： 创建硬链接
SYSCALL_MKDIR(usize = 20) 功能： 创建目录
SYSCALL_CLOSE:(usize = 21) 功能： 关闭文件描述符
SYSCALL_GETMTIME(usize = 22) 功能： 获取riscv处理器的计时器寄存器

6.3 系统调用接口

華東師範大學
EAST CHINA NORMAL UNIVERSITY

SYSCALL_THREAD_CREATE:(usize = 24) 功能： 创建新线程
SYSCALL_THREAD_COUNT:(usize = 25) 功能： 获取当前进程中的线程数
SYSCALL_THREAD_WAITTID: (usize = 26)功能： 等待指定线程结束
SYSCALL_GETTID: (usize = 27) 功能： 获取当前线程ID
SYSCALL_GET_TASK_EXITSTATUS: (usize = 28) 功能： 获取线程的推出值
SYSCALL_SEMAPHORE_CREATE: (usize = 29) 功能： 创建一个简易信号量
SYSCALL_SEMAPHORE_UP: (usize = 30) 功能： 信号量V操作(释放)
SYSCALL_SEMAPHORE_DOWN: (usize = 31) 功能： 信号量P操作(获取)

6.4 用户线程库

華東師範大學
EAST CHINA NORMAL UNIVERSITY

- 用户空间线程库（User-Level Thread Library）是一种在用户态实现线程管理的机制。
- 与操作系统内核管理的线程（内核线程）不同，用户线程的创建、调度、同步等操作完全由库在用户空间处理，无需频繁陷入内核，从而减少上下文切换的开销。
- 这可以算是一个简易的内核，可用于教学，便于学生理解内核切换任务的过程。

6.5 用户线程库实现

華東師範大學
EAST CHINA NORMAL UNIVERSITY

- 用户线程包含：
  - 上下文信息（寄存器、栈指针、返回地址等）
  - 状态（空闲、运行、可调度、等待）
  - 控制信息（含唯一编号、栈空间、及管理指针）
- 线程调度器结构
  - 管理所有用户线程
  - 记录当前正在执行的线程
  - 维护任务间的等待依赖（等待特定或任意线程完成）
- 线程切换机制
  - 线程通过主动让出执行权参与调度
  - 使用上下文切换技术保存和恢复寄存器状态
  - 避免无效的切换（如切换到自身或空闲线程）
  - 调度器寻找状态为“可调度”的下一个线程执行

```rust
pub struct Task {
    pub id: usize,
    pub stack: Vec<u8>,
    pub ctx: TaskContext,
    pub state: TaskState,
    pub r_ptr: u64
}
```

```rust
pub struct Runtime {
    tasks: Vec<Task>,
    current: usize,
    waits: Vec<usize>
}
```

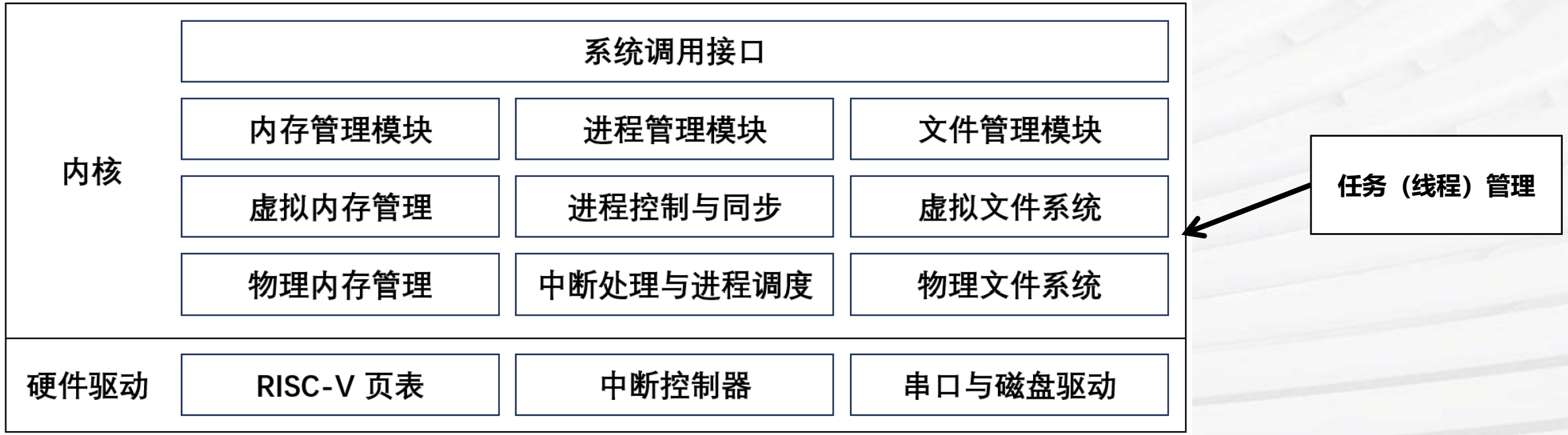
6.5 用户线程库实现

華東師範大學
EAST CHINA NORMAL UNIVERSITY

- 线程创建与启动
  - 从空闲任务池中选择任务槽位
  - 初始化其上下文、栈指针与传入参数
  - 设置线程初始状态为可调度，等待调度器启动执行
- 线程同步机制
  - 线程可以选择挂起，等待特定线程或任意线程完成
  - 其他线程可通过发送信号唤醒被等待的线程
  - 实现了线程间的简单同步依赖关系
- 线程结束与资源回收
  - 执行完毕后进入“可重用”状态
  - 自动唤醒所有等待当前线程完成的其他线程
  - 返回控制权给调度器，继续执行其他任务

6.6 在内核整体架构新增内核线程

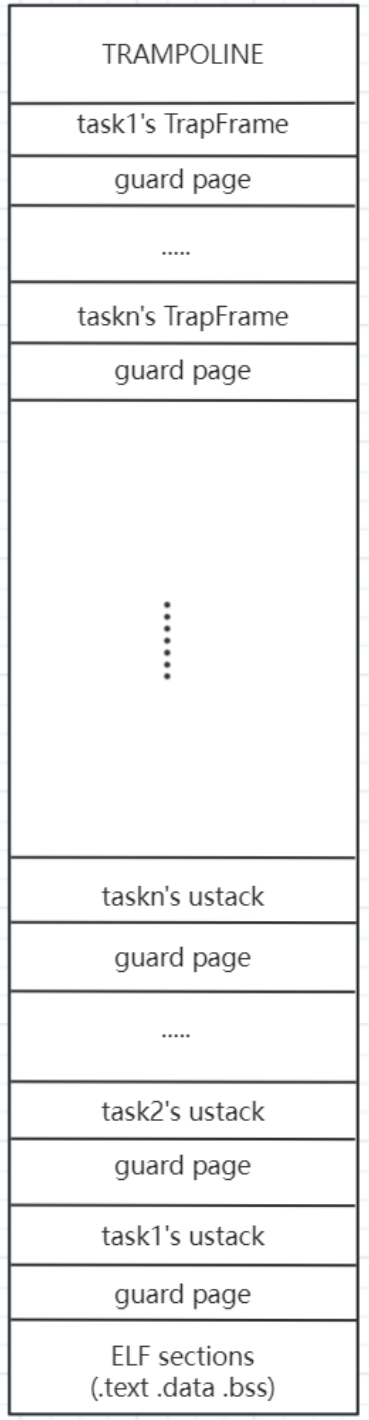
由于xv6没有实现内核线程管理功能，让操作系统缺少了并行和使用共享资源的功能，并可能导致多线程教学实验的缺失。在初赛阶段实现用户线程，在用户态实现线程管理存在一个固有局限：线程调度器无法主动抢占当前线程的执行权，必须等待运行中的线程主动释放CPU资源后才能进行线程切换。而操作系统运行在内核态，如果进一步扩展线程管理功能，就可以利用时钟中断机制直接中断当前用户线程的执行，从而实现更高效的线程调度和切换。



6.7 内核线程的作用

- 利用RISC-V的多核扩展(SMP)实现真正的并行处理
- 让多任务可以共享内存，能让程序实现更高效的协作和通信
- 细粒度调度替代整个进程的切换，提升CPU利用率

6.8 进程内存布局


TRAMPOLINE
task1's TrapFrame
guard page
.....
taskn's TrapFrame
guard page
taskn's ustack
guard page
.....
task2's ustack
guard page
task1's ustack
guard page
ELF sections
(.text .data .bss)

6.9 内核线程的实现

将Process结构体转换为Task（包括主任务和子任务）结构体。新增 is_child_task、tasks、parnet和semaphore_list成员变量。

is_child_task：区分主线程和子线程。

tasks：保存进程中所有创建的线程

parents：保存指向父线程的指针

semaphore_list：保存进程中的信号量

```rust
pub struct Process {
    /// 进程在进程表中的索引，唯一标识该进程槽位。
    index: usize,
    /// 进程排它锁保护的状态信息，包括状态、pid、等待通道等。
    pub excl: SpinLock<ProcExcl>,
    /// 进程私有数据，包含内存、上下文、文件描述符等，通过 UnsafeCell 实现内部可变性。
    pub data: UnsafeCell<ProcData>,
    /// 标识进程是否被杀死的原子布尔变量，用于调度和信号处理。
    pub killed: AtomicBool,
}
```

→

```rust
pub struct Task {
    /// 在任务表中的索引，唯一标识该进程（主线程）槽位。
    index: usize,
    /// 是否为子线程
    pub is_child_task: bool,
    /// 任务排它锁保护的状态信息，包括状态、pid、等待通道等。
    pub excl: SpinLock<TaskExcl>,
    /// 任务私有数据，包含内存、上下文、文件描述符等，通过 UnsafeCell 实现内部可变性。
    pub data: UnsafeCell<TaskData>,
    /// 子线程
    pub tasks: SpinLock<Vec<Option<*mut Task>>>,
    /// 父主线程
    pub parent: Option<*mut Task>,
    /// 锁
    pub semaphore_list: SpinLock<Vec<Option<Semaphore>>>,
    /// 标识（主线程）是否被杀死的原子布尔变量，用于调度和信号处理。
    pub killed: AtomicBool,
}
```

6.10 内核线程的创建

内核创建新线程

1.线程ID分配

2.与进程共享Pagetable

3.子线程TrapFrame分配

4.子线程User Stack的分配

5.子线程TrapFrame和Context的设置、子线程设置入口地址和参数和sp指针

6. 将任务加入可执行FIFO队列。

```rust
fn thread_create(&mut self, entry: usize, arg: usize) -> Result<usize, ()> {
    // 将所有创建的线程挂在主线程下
    let main_task: &mut Task = if self.is_child_task {
        unsafe { self.parent.unwrap().as_mut().unwrap() }
    } else {
        self
    };
    let pos: usize = main_task.tasks.lock().len() + 1;
    assert!(pos < MAX_TASKS_PER_PROC);
    let main_task_ptr: *const Task = main_task as *const Task;
    let main_task_data: &mut TaskData = main_task.data.get_mut();
    // 准备任务结构体和Context
    let child_task: &mut Task = unsafe { PROC_MANAGER.alloc_task(parent: main_task_ptr).ok_or(err: ())? };
    child_task.is_child_task = true;
```

6.10 新增关于内核线程的系统调用

- [x] **SYSCALL_THREAD_CREATE(usize = 24)**
  功能：在当前进程中创建新的线程。
  参数：
    entry: 线程入口点地址
    arg: 传递给线程的参数
  返回值：
    (tid)≥0：成功，子线程的线程号。
    -1：错误。

- [x] **SYSCALL_THREAD_COUNT(usize = 25)**
  功能：获取当前进程中的线程数。
  参数：
    无。
  返回值：
    (count)≥1：。

- [x] **SYSCALL_THREAD_WAITTID(usize = 26)**
  功能：等待指定的线程结束。
  参数：
    tid: 要等待的线程ID
  返回值：
    0： 成功。
    -1： 失败。
- [x] **SYSCALL_GETTID(usize = 27)**
  功能：获取当前线程ID。
  参数：
    无。
  返回值：
    (pid)≥1： 当前线程pid。

- [x] **SYSCALL_GET_TASK_EXITSTATUS(usize = 28)**
  功能：获取线程的推出状态。
  参数：
    tid: 要等待的线程ID
  返回值：
    exit_code： 线程的结束值。
- [x] **SYSCALL_SEMAPHORE_CREATE(usize = 29)**
  功能：创建信号量。
  参数：
    count: 信号量初始值
  返回值：
    id： 信号量ID。
    -1： 创建失败。
- [x] **SYSCALL_SEMAPHORE_UP(usize = 30)**
  功能：信号量V操作(释放)。
  参数：
    sem_id: usize 信号量ID
  返回值：
    0： 成功。
    -1： 失败。
- [x] **SYSCALL_SEMAPHORE_DOWN(usize = 31)**
  功能：信号量P操作(获取)。
  参数：
    sem_id: usize 信号量ID
  返回值：
    0： 成功。
    -1： 失败。

6.10 内核线程的系统调用展示

```rust
#![no_std]
#![no_main]
use syscall_riscv::{sys_gettid, sys_semaphore_create, sys_semaphore_down, sys_semaphore_up, sys_thread_waittid};
use user_rust_lib::{exit, kernel_thread::thread_create};
use core::ptr::addr_of_mut;
#[macro_use]
extern crate user_rust_lib;
static mut COUNTER: u32 = 0;
const SEM_SYNC: usize = 0;
fn first(arg:usize){
    let a: *mut u32 = addr_of_mut!(COUNTER);
    for i: i32 in 0..1000{
        sys_semaphore_down(pos: SEM_SYNC);
        unsafe { a.write_volatile( val: *a +1 ) };
        sys_semaphore_up(pos: SEM_SYNC);
    }
    exit(exit_code: 0)
}

fn second(arg:usize){
    let a: *mut u32 = addr_of_mut!(COUNTER);
    for i: i32 in 0..1000{
        sys_semaphore_down(pos: SEM_SYNC);
        unsafe { a.write_volatile( val: *a +1 ) };
        sys_semaphore_up(pos: SEM_SYNC);
    }
    exit(exit_code: 0)
}

#[no_mangle]
▶ Run | ⚙ Debug
fn main(argc:usize, argv:&[&str]) -> i32 {
    assert_eq!(sys_semaphore_create(1) as usize, SEM_SYNC);
    // create threads
    let tid1: isize = thread_create(f: first, arg: 17);
    let tid2: isize = thread_create(f: second, arg: 17);

    let code: isize = sys_thread_waittid(tid: tid1 as usize);
    let code: isize = sys_thread_waittid(tid: tid2 as usize);
    println!("count:{}",unsafe { COUNTER });
    0
}
```

```
$ sem
count:2000
```

6.10 内核线程后续待改进

- 在内核线程代码编写上缺少RAII思想，需要程序员手动回收资源，可能会导致内存泄露和重复使用。
- 将资源管理与对象生命周期绑定在一起的思想主要体现在 **所有权（Ownership）系统** 和 **RAII（Resource Acquisition Is Initialization）** 模式中。这是 Rust 内存安全和资源安全的核心设计理念。
- 后续需要编译器自动通过析构函数（Droptrait）确保资源释放。

華東師範大學
EAST CHINA NORMAL UNIVERSITY

第七部分

学生实验平台

7.1 实验平台功能说明

華東師範大學
EAST CHINA NORMAL UNIVERSITY

指令	功能	具体说明
`make qemu`	启动内核	- 该指令用于启动内核，执行用户程序与内核进行交互 - 执行该指令后，将会自动编译系统内核，用户程序，并生成初始化文件系统，最终启动QEMU
`make qemu-gdb`	启动调试	- 该指令用于调试内核，首先启动QEMU并等待GDB连接，连接成功后按照GDB的指令运行内核 - 首先开启两个终端，在第一个终端中执行该指令，等待内核编译完成后启动QEMU
`make grade`	自动评分	- 该指令将执行自动化评分脚本，评测当前实验的完成情况，并给出最后总分 - 具体流程为首先清空所有已编译结果，并重新编译内核，启动评分脚本，完成评分并给出最终结果 - 若某个评测点失败，将会产生一个xv6.out文件记录失败的内核输出
`make asm`	输出汇编	- 该指令会执行objdump对内核进行反汇编，输出内核的汇编指令 - 输出结果在kernel.S文件中，其中主要包含函数名称，指令地址以及指令内容 - 可用于内核异常时根据异常指令地址定位出错函数
`make clean`	重置平台	- 该指令将清空所有编译结果，将实验环境初始化 - 包括用户程序编译结果，内核编译结果，文件系统镜像 - 同时还会清除`make asm`生成的反汇编结果，以及`make grade`生成的错误日志

7.2 测试用例设计

- **综合性回归测试：usertests**
  - 含约 60 个子测试用例，覆盖进程管理、内存管理、文件系统、系统调用边界条件等基础功能模块。
  - 可用于持续验证核心功能的完整性，确保学生在实现新功能时未破坏已有行为逻辑。
  - 作为教学系统的“回归测试基准”，可嵌入自动化测试流程。
- **实验对应专项测试程序**
  - 每个实验新增机制配套一个专门测试程序。
  - 专项测试程序能更精确地验证功能实现是否符合实验设计目标，补充通用测试的不足。
  - 构成“覆盖广 + 针对强”的测试闭环设计，提升教学准确性与系统鲁棒性。

7.2 测试用例设计

華東師範大學
EAST CHINA NORMAL UNIVERSITY

- 下面简单介绍移植自xv6的60个测试用例的一部分：

```
void copyin(char *s);          // 测试将无效指针传递给从用户空间读取数据的系统调用
void copyout(char *s);         // 测试将无效指针传递给向用户空间写入数据的系统调用
void copyinstr1(char *s);      // 测试将非法字符串指针传递给系统调用
void copyinstr2(char *s);      // 测试当参数字符串长度恰好等于内核缓冲区大小时的处理
void copyinstr3(char *s);      // 测试字符串参数跨越用户空间最后一页边界时的行为
void rwsbrk();                 // 测试进程归还内存后对已释放地址执行读写操作的处理
void truncate1(char *s);       // 测试使用 O_TRUNC 截断文件后已打开文件描述符读取行为是否正确
void truncate2(char *s);       // 测试文件被截断后对原文件描述符执行写操作是否正确失败
void truncate3(char *s);       // 测试多进程同时对同一文件进行截断和写入操作的情况
void iputtest(char *s);        // 测试进程切换到新目录后删除该目录时内核对引用计数的处理
void exitiputtest(char *s);    // 测试进程退出时能否正确释放其当前工作目录的引用（iput）
void openiputtest(char *s);    // 测试尝试写打开目录出错时内核是否正确回收 inode
void opentest(char *s);        // 测试打开存在的文件成功和打开不存在的文件失败的情况
```

7.3 测试脚本设计

華東師範大學
EAST CHINA NORMAL UNIVERSITY

- 自动化测试脚本通过执行 make grade 命令触发测试流程，评测的整体过程包括以下几个关键步骤：
- **1. 清理环境并重置文件系统镜像：**首先清除之前编译生成的中间文件，并将文件系统镜像恢复为干净状态，以确保测试在一致的初始环境下运行。
- **2. 重新编译内核与用户程序：**执行完整的 make 构建流程，编译最新的内核和用户态测试程序，并生成包含测试程序的初始文件系统镜像。
- **3. 通过QEMU启动系统并执行测试程序：**启动带有 GDB 调试端口的 QEMU 模拟器，进入 xv6 shell 后，根据测试用例预设内容自动执行测试命令（包括实验相关测试程序与通用的用户程序测试）。
- **4. 采集QEMU输出并进行内容匹配：**通过与 GDB 建立连接，实时捕获 xv6 运行期间的标准输出，并与预期的输出模式进行匹配校验。
- **5. 统计得分并输出测试结果：**根据每个测试点是否通过来累计得分，最终输出整体得分情况与详细的测试通过/失败信息，供用户评估实验完成度。

7.4 测试点例子

華東師範大學
EAST CHINA NORMAL UNIVERSITY

- 一个修改过的测试点示例如下所示，该测试点用于backtrace功能的测试，测试流程是首先在内核中执行bttest，获取内核输出的几个栈地址，然后在内核的可执行文件中使用addr2line工具查找这几个栈地址是否是目标文件中的正确地址。

```
@test(10, "backtrace test")
def test_backtracetest():
    r.run_qemu(shell_script([
        'bttest'
    ]))
    a2l = addr2line()
    matches = re.findall(BACKTRACE_RE, r.qemu.output, re.MULTILINE)
    assert_equal(len(matches), 3)
    files = ['syscall.rs', 'mod.rs', 'trap.rs']
    for f, m in zip(files, matches):
        result = subprocess.run([a2l, '-e', 'target/riscv64gc-unknown-none-elf/debug/xv6-rust', m],
stdout=subprocess.PIPE)
        if not f in result.stdout.decode("utf-8"):
            raise AssertionError('Trace is incorrect; no %s' % f)
```

7.5 测试脚本运行效果

華東師範大學
EAST CHINA NORMAL UNIVERSITY

- 运行一个自动测试脚本，若所有测试均通过的情况如下所示，可以看到各个测试点的通过情况和分数汇总：

```
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 4.22s
make[1]: Leaving directory '/workspaces/os-competition/xv6-rust'
== Test backtrace test == backtrace test: OK (2.6s)
== Test running alarmtest == (2.5s)
== Test   alarmtest: test0 ==
  alarmtest: test0: OK
== Test   alarmtest: test1 ==
  alarmtest: test1: OK
== Test   alarmtest: test2 ==
  alarmtest: test2: OK
== Test usertests == usertests: OK (224.2s)
Score: 79/79
```

7.5 测试脚本运行效果

- 若存在学生实验不符合要求的情况，测试点不通过的输出如下所示：

```
== Test backtrace test == backtrace test: FAIL (2.9s)
    got:
      0
    expected:
      3
    QEMU output saved to xv6.out.backtracetest
== Test running alarmtest == (26.3s)
== Test   alarmtest: test0 ==
  alarmtest: test0: FAIL
    ...
         test1 failed: too few calls to the handler
         test2 start
         ..........................................................................
..........................................................................
..........................................................................
..........................................................
         test2 failed: alarm not called
         $ qemu-system-riscv64: terminating on signal 15 from pid 81378 (make)
    MISSING '^test0 passed$'
```

華東師範大學
EAST CHINA NORMAL UNIVERSITY

第八部分

实验指导与参考实现

8.1 实验设计

- **实验一：进行系统调用**
- 实现一个用户程序，进行sleep系统调用，sleep 程序应当根据用户指定的时间单位 tick 暂停执行。tick 是由内核定义的一种时间单位，表示来自定时器芯片的两次中断之间的时间间隔。
- **实验二：打乒乓**
- 编写一个用户程序，使用系统调用，通过一对管道在两个进程之间“乒乓”传递一个字节。父进程应向子进程发送一个字节；子进程打印进程编号 ，然后通过管道将字节写回给父进程，并退出。父进程应从子进程读取该字节，打印进程编号 ，然后退出。
- **实验三：并发素数筛**
- 编写一个使用管道的并发素数筛选程序。使用 pipe 和 fork 来构建一个流水线。第一个进程将连续自然数依次写入这条流水线。对于每一个素数，需要安排一个进程，通过管道从左边的邻居读取数据，再通过另一个管道向右边的邻居写数据。最终输出所有素数。
- **实验四：文件查找**
- 编写一个用户程序，实现在一个指定文件系统目录下查找具有指定名称的文件。该用户程序应该接受两个参数，即查找目录以及文件名称。并对输入目录进行递归查找，找到每一个子目录下出现的符合名称的文件。

8.1 实验设计

- **实验五：系统调用跟踪**
- 为内核增加一个系统调用追踪功能，通过新增的 trace(mask) 系统调用按位指定需要追踪的系统调用类型。内核在被追踪的系统调用即将返回时，打印出该进程的 PID、系统调用名及其返回值，从而辅助调试后续实验。
- **实验六：加速系统调用**
- 在每个进程创建时，将一个只读页映射到用户空间的 USYSCALL 虚拟地址，并在该页起始位置写入包含当前进程 PID 的 struct usyscall 结构体。用户空间通过 ugetpid() 访问该映射页即可获取进程 PID，实现无需陷入内核的高效用户态系统信息查询。
- **实验七：打印页表**
- 实现一个 vmprint(pagetable_t) 函数，用于按指定格式打印页表内容，并在 exec.c 中对第一个进程（PID 为 1）在返回前调用该函数输出其页表。该功能用于加深对多级页表结构的理解，并通过页表打印测试验证实现正确性。
- **实验八：检测某一页是否被访问过**
- 实现 pgaccess() 系统调用，用于报告一组用户页是否被访问过。它接收起始虚拟地址、页数以及用于写入访问结果的用户空间位掩码缓冲区，返回值按位表示每页的访问状态，有助于实现如页面置换等高级功能。

8.1 实验设计

- **实验九：回溯调用栈**
- 实现一个 backtrace() 函数，用于输出当前内核栈的调用地址，并在 sys_sleep 中调用该函数进行回溯测试。通过 addr2line 工具可将输出地址解析为源代码位置，从而实现内核调试中常用的调用路径追踪功能。
- **实验十：周期性任务调用**
- 该实验要求为内核增加定时报警机制，使进程在占用一定 CPU 时间后自动触发用户定义的处理函数，实现类似用户级中断的效果。该机制可用于周期性任务或资源限制场景，正确实现需通过 alarmtest 和 usertests 验证。
- **实验十一：内存懒分配**
- 该实验要求学生实现的懒加载内存分配，通过修改sbrk()系统调用，使其不立即分配物理内存，而是延迟到访问页面时由内核处理页面错误并分配内存。需确保懒加载机制正确处理各种内存分配、访问和错误情况。
- **实验十二：优先级调度**
- 该实验要求学生为内核添加优先级调度功能。需要学生添加一个系统调用以设置进程优先级，并修改系统调度函数算法，使其能够选择当前系统中优先级最高的进程执行，从而实现优先级调度功能。

8.1 实验设计

華東師範大學
EAST CHINA NORMAL UNIVERSITY

- **实验十三：多线程中进程的Pagetable的内存管理**
- 该实验要求学生实现在同一页表下，对进程中所有线程进行内存资源分配和回收，包括线程用户栈、线程TrapFrame、线程TRAMPOLINE。
- **实验十四：semaphore实现**
- 在多线程操作系统中，互斥锁（semaphore）是保证共享资源安全访问的核心同步机制。本实验要求基于Rust和类Unix系统调用，实现两种不同行为的用户态互斥锁。第一种为semaphore自旋锁，第二种semaphore阻塞锁。确保后续能正确地进行多线程安全累加计

8.2 实验指导示例 – 实验七

華東師範大學
EAST CHINA NORMAL UNIVERSITY

实验要求

为了帮助你更直观地理解 RISC-V 的页表结构，同时也便于将来的调试工作，你的任务是：**实现一个函数，用于打印当前进程的页表内容**。你需要定义一个名为 `vmprint()` 的函数。该函数接受一个 `pagetable` 类型的参数，并以特定格式打印该页表的内容（格式要求将在后续实验说明中详细介绍）。这个打印函数的目标是清晰地展示虚拟页与物理页的映射关系，以及页表的层次结构。

为验证你的实现，请在 `syscall.rs` 文件中，在`sys_exec`函数插入如下代码：

```
let guard = self.excl.lock();
if guard.pid == 1 {
    let data = self.data.get_mut();
    data.pagetable.as_ref().unwrap().vm_print(0);
}
drop(guard);
```

这段代码会在第一个用户进程执行时自动打印其页表信息。

只要你能够通过 `make grade` 中与页表打印相关的测试用例（即 `pte printout` 测试），就可以获得该实验部分的**满分**。

8.2 实验指导示例 – 实验七

華東師範大學
EAST CHINA NORMAL UNIVERSITY

实验现象

现在当你启动时，应该会看到类似如下的输出，描述的是**第一个进程在刚刚完成** `sys_exec()` **调用后**的页表内容：

```
page table 0x80409000
..0: pte 0x20103401 pa 0x8040d000
.. ..0: pte 0x20103801 pa 0x8040e000
.. .. ..0: pte 0x20103801 pa 0x8040e000
..255: pte 0x20102801 pa 0x8040a000
.. ..511: pte 0x20102c01 pa 0x8040b000
.. .. ..511: pte 0x20102c01 pa 0x8040b000
```

打印输出的第一行显示的是传入 `vmprint` 函数的参数（即页表的起始地址）。随后每一行对应一个有效的页表项（PTE），包括那些指向更深层级页表页的中间页表项。

每一行页表项之前会有若干个 `" .."` 缩进，缩进的数量表示该页表项所在页表在页表树中的层级深度（顶层为 0，越往下层缩进越多）。

每一行页表项的输出内容包括：

- 该页表项在当前页表页中的索引；
- 页表项的控制位（如有效位、用户位、读写权限等）；
- 从页表项中提取出的物理页地址。

8.2 实验指导示例 – 实验七

華東師範大學
EAST CHINA NORMAL UNIVERSITY

注意：**不要打印无效的页表项**（即 `valid` 位未置位的 PTE）。

在上面的输出示例中，顶层页表页中映射了第 0 项和第 255 项；对于第 0 项指向的下一级页表中，仅第 0 项被映射；而该页表中的第 0 项再进一步指向底层页表，其中第 0、1、2 项被映射。

你实现的代码可能会打印出与示例中不同的物理地址，但映射的项数以及它们对应的虚拟地址索引应当是一致的。

实验提示

- 你可以将 `vmprint()` 函数的实现放在 `mm/pagetable.rs` 文件中；
- 在实现过程中，可以使用 `const/riscv.h` 文件末尾定义的一些宏，这些宏能帮助你处理页表项的位操作；
- 可以参考函数 `walk` 的实现思路，它同样是递归地遍历多级页表结构；
- 在Rust的`println`宏中使用`{:x}`打印十六进制数

8.3 参考实现示例 – 实验七

華東師範大學
EAST CHINA NORMAL UNIVERSITY

实现页表打印函数 `vm_print()`

文件路径：`src/mm/pagetable.rs`

```
pub fn vm_print(&self, level: usize)
{
    if INIT_VM_PRINT.swap(false, core::sync::atomic::Ordering::Relaxed) {
        println!("page table {:p}", &(self.data));
    }
    for (idx, pte) in self.data.iter().enumerate() {
        if pte.is_valid() {
            if level == 0 {
                println!("..{}: pte {:#x} pa {:p}", idx, pte.data, pte.as_page_table());
                (unsafe { &*pte.as_page_table() }).vm_print(1);
            }
            if level == 1 {
                println!(".. ..{}: pte {:#x} pa {:p}", idx, pte.data, pte.as_page_table());
                (unsafe { &*pte.as_page_table() }).vm_print(2);
            }
            if level == 1 {
                println!(".. .. ..{}: pte {:#x} pa {:p}", idx, pte.data, pte.as_page_table());
            }
        }
    }
}
```

解释：

该函数递归打印多级页表结构，采用缩进标识树的深度（`..`，`.. ..` 等）。其中：

- 顶层页表打印头部；
- 只打印有效的 PTE；
- 每级递归调用自身，逐层深入；
- 缩进由 `level` 决定。

8.3 参考实现示例 – 实验七

華東師範大學
EAST CHINA NORMAL UNIVERSITY

系统调用返回前触发页表打印

文件路径：`src/process/proc/syscall.rs`

```
let guard = self.excl.lock();
if guard.pid == 1 {
    let data = self.data.get_mut();
    data.pagetable.as_ref().unwrap().vm_print(0);
}
drop(guard);
```

解释：

在系统调用结束之前检查当前进程是否为 PID 1（即第一个用户进程），如果是，则调用其页表的 `vm_print()` 函数。

- 使用 `lock` 获取排他访问；
- 获取进程页表；
- 调用递归打印；
- `drop(guard)` 手动释放锁，保持良好资源管理。

总结

華東師範大學
EAST CHINA NORMAL UNIVERSITY

- 本项目实现了一套基于 Rust 的教学操作系统实验，包含完整的文档注释体系、Rust语言用户程序与多线程库、学生实验平台、实验指导手册与参考实现。
- 填补了当前Rust语言入门级操作系统实验的空白，帮助没有操作系统基础的学生从C语言过渡，并掌握Rust语言与基础操作系统知识。
- 本项目文档完善且结构清晰，便于根据各校操作系统课程需求，自行增加实验内容

華東師範大學
EAST CHINA NORMAL UNIVERSITY

谢谢！
Thanks for listening!

汇报队伍：SEIOS 队员：郑子攸，叶晨皓，周恒如 指导老师：郭建