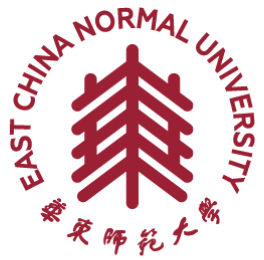


基于Rust与xv6的操作系统教学方案设计与实现

比赛信息

2025年全国大学生计算机系统能力大赛 - 操作系统设计赛(全国) - OS功能挑战赛道

- **赛题：** 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
- **队名：** SEIOS
- **院校：** 华东师范大学
- **指导老师：** 郭建，张敏
- **参赛队员：** 郑子攸，叶晨皓，周恒如

项目简介

- 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
- 本项目基于xv6-riscv-rust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注释编写
  - Rust语言重写用户程序及测试用例
  - 自动化评测方案移植
  - 实验指导手册及参考实现编写
- 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重写部分xv6的用户程序以及测试用例，计划后续使用Rust重写所有测试用例
- 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
- 移植xv6的评测方案，实现了Rust环境下的学生实验全自动评测
- 参考xv6的学生实验，已完成6个实验的移植与实验指导手册及参考实现的编写，计划后续完成全部14个实验移植

目标

- 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
- 为系统内核中的所有函数添加文档注释，实现注释100%覆盖，提升代码的可读性
- 针对操作系统的不同部分设计难度由浅入深的实验题，帮助学生逐步学习Rust语言与操作系统
- 设计完善的测试用例与测试脚本，实现自动化评测，简化实验检查步骤并提高评分公平性
- 编写实验指导手册，提供每一个实验题的题干、提示、参考实现以及详细的实验解答

前期调研

前期调研阶段我们收集了多个现有的操作系统实验课程，其中包含：

1. MIT - 6.828（xv6）
2. 清华大学 - rCore-Tutorial-Book 第三版
3. 中科院软件所 - 循序渐进，学习开发一个 RISC-V 上的操作系统
4. 南京大学 - 操作系统：设计与实现
5. 华中科技大学 - 操作系统实验（riscv-pke）

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。

本节中我们详细描述与我们的工作密切相关的前两个项目，MIT的xv6和清华大学的rCore。二者各自在培养学生系统级编程能力与深入理解操作系统机制方面取得了良好效果，但也存在一定的不足。xv6实验指导文档详尽，实验设计循序渐进且难度合理，但因其基于C语言实现，未充分利用现代编程语言提供的内存安全特性，学生可能因此难以深刻体会和掌握更先进、更安全的编程范式；清华大学的rCore基于Rust语言开发，天然具备高安全性与高性能的优势，却在实验难度控制和指导文档上存在一些不足，实验难度梯度较为陡峭，且在关键实现细节与调试指导方面的文档较少，使得初学者容易在复杂实验环节中遇到困难，更适合有操作系统基础的学生进行学习。因此，一个同时兼具Rust语言内存安全特性、实验难度逐级递增、且提供完善实验指导的教学操作系统，将更有效地帮助学生掌握现代操作系统设计与实现技能，我们的工作旨在达成这一目标。

MIT - 6.828（xv6）

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，下面分别简述各组实验所涵盖的内容。

- **Lab1: xv6 and Unix Utilities**

这个实验通过让学生在 xv6 上实现五个常见 UNIX 工具（如 `sleep`, `pingpong`, `primes`, `find`, `xargs`），系统地介绍了 xv6 用户态程序的编译运行流程及基本的系统调用接口。学生将在 `util` 分支上编写这些命令，学习如何通过 `fork`、`pipe`、`exec`、`read`、`write`、`sleep` 等系统调用与内核交互，并用 `Makefile` 把程序纳入文件系统镜像中进行测试。这一过程训练了学生对操作系统调用流程的理解、用户态与内核态的接口掌握，以及基础的并发与 I/O 编程能力，为后续深层次操作系统机制的探索打下坚实的基础。

- **Lab2: System Calls**

在这个实验中，学生需要在 xv6 中添加两个新系统调用 —— `trace` 用于控制并打印用户进程及其子进程的系统调用跟踪，以及 `sysinfo` 用于返回系统空闲内存和进程数量等信息 —— 通过修改用户调用 stub、系统调用号表、`syscall()` 处理函数，以及在 `proc` 结构中添加状态字段，该实验有效锻炼了学生对系统调用机制、用户态与内核态交互路径、参数传递、以及内核数据统计与复制技术等操作系统核心能力的理解与实现。

- **Lab3: Page Tables**

在本实验中，学生首先实现了 `vmprint` 函数，直观打印 RISC-V 多级页表结构，锻炼对虚拟地址与物理地址映射机制的理解；接着，为每个进程创建独立的内核页表并在进程调度时切换，掌握内核态与用户态的页表隔离逻辑；最后，优化 `copyin` 和 `copyinstr`，通过给内核页表添加用户空间映射，让内核能够直接访问用户内存，无需手工译码地址，提高地址访问效率。该实验让学生深入理解多级页表、页表遍历、上下文切换及页表映射策略，是构建现代虚拟内存管理能力的重要训练。

- **Lab4: Traps**

在这个实验中，学生在 xv6 中实现了两个关键功能：首先编写 `backtrace()`，利用 RISC-V 汇编中的帧指针机制，在 panic 或调试时打印内核的调用栈；接着添加 `sigalarm()` 和 `sigreturn()` 两个系统调用，通过修改 trap 处理逻辑，使得当进程运行达到设定时钟节拍后，会在用户态中断当前执行流转而执行用户自定义处理函数，并在 handler 返回时恢复原本上下文继续执行。整个实验不仅加强了学生对

trap、上下文保存/恢复、中断处理、及用户态-内核态控制流切换的理解，也训练了他们在真实环境中设计和实现用户级中断/定时机制的能力，进一步加深了对操作系统核心机制的掌握。

- **Lab5: xv6 Lazy Page Allocation**

在本实验中，学生需修改 xv6，使得 `sbrk()` 不再立即分配物理页，而仅调整进程的虚拟地址空间大小；当用户程序首次访问尚未映射的页时，会触发缺页异常，内核在异常处理（`usertrap()`）中检测到该情况（`scause` 为 load/store page-fault），随后调用 `kalloc()` 分配物理页并用 `mappages()` 建立映射。还需要修改 `uvmunmap()` 和 `fork()` 等函数，以跳过尚未映射的页，支持缩减和进程复制。学生通过这一实验深入理解虚拟内存机制、异常处理、页表结构和按需分配策略，掌握了现代操作系统中对内存的高效和智能管理方法。

- **Lab6: Copy-on-Write Fork for xv6**

在本实验中，学生的任务是将 xv6 的 `fork()` 系统调用改造成“写时复制”（COW）版本，即子进程最初不会拷贝父进程的所有物理内存页，而是共享所有用户页，且将它们设置为只读（清除PTE_W位）。仅当父或子尝试写入共享页时，硬件会触发页错误（store page-fault），内核在 `usertrap()` 中捕获该异常，分配新的物理页复制原内容，并将出错进程的页表更新为可写 PTE，从而局部执行复制与写操作。为此，还需引入物理页的引用计数机制，仅在最后一个引用被释放时才真正调用 `kfree()` 释放物理内存。此外，修改 `copyout()` 方法确保内核式的写入操作也能触发 COW 逻辑。学生通过此实验深入掌握页表权限控制、缺页异常机制、分页优化策略，以及如何在操作系统中实现按需资源复制和管理共享内存，极大提升了其理解和应用操作系统虚拟内存与性能优化能力。

- **Lab7: Multithreading**

在本实验中，学生将为 xv6 实现一个用户级线程库（`uthread.c` 和 `uthread_switch.S`），完成线程创建、切换与调度逻辑，使多个线程能在单个进程内协作执行，并通过 `uthread` 测试程序验证多线程调度机制。随后，学生在宿主系统上利用 POSIX `pthread` 库，编写多线程的哈希表并发测试（`ph.c`），分析并解决并发访问带来的错误，再通过加锁机制（互斥锁、条件变量、屏障）提高并发安全性与性能（`ph_safe` 与 `ph_fast` 测试）。该实验不仅培养了学生对线程上下文切换及寄存器状态保存/恢复机制的理解，还通过实战加深了其对并发同步原语（如锁、条件变量、屏障）及竞争条件与性能优化的掌握，是操作系统并发编程能力的重要训练。

- **Lab8: Locks**

在本实验中，学生通过分析并完善 xv6 内核中的自旋锁（spinlock）和睡眠锁（sleeplock）机制，学习如何在多处理器和中断环境中实现正确的同步与互斥，任务包括修改和测试锁的获取与释放逻辑，消除可能出现的竞态条件和死锁问题，从而确保关键内核资源在并发访问下的安全性与一致性；这一实验不仅帮助学生深入理解操作系统中多核同步原语的设计思想，还提升了他们识别并解决实际并发问题的能力，是理解操作系统并发控制与调度机制的关键环节。

- **Lab9: File System**

在本实验中，学生被要求在 xv6 文件系统中添加对大型文件和符号链接的支持：首先扩展 inode 的块结构，引入双重间接块（doubly-indirect block），使可管理文件大小从原先的 268 块提升至约 65 803 块；然后实现 `symlink(target, path)` 系统调用，创建符号链接并在 `open()` 路径解析中正确处理软链接和深度检测，从而模拟 Unix 风格的链式追踪机制。该实验深入锻炼了学生对磁盘块管理、间接块索引逻辑、文件系统元数据更新、路径查找机制以及系统调用扩展的理解，同时提升了他们在复杂数据结构和 I/O 操作下设计健壮内核功能的能力。

- **Lab10: Mmap**

在本实验中，学生为 xv6 增加了内存映射（`mmap`/`munmap`）功能，使用户程序能够将文件内容直接映射到其虚拟地址空间。该实验内容包括实现系统调用接口、在进程页表中建立映射项、处理缺页异常以加载文件页，以及支持写回文件的同步机制。通过此过程，学生深刻理解了虚拟内存与文件 I/O 的结合方式，掌握了页表管理、按需加载与写时回写策略，实现高效的用户程序内存访问模型；这不仅强化了他们对内存管理、文件系统交互及异常处理机制的掌握，还培养了构建高性能映射机制和内核扩展能力，是操作系统高级功能设计的重要训练。

- **Lab11: Networking**

在本实验中，学生需要实现 xv6 的 E1000 网卡驱动，使之能够通过描述符环（descriptor rings）与硬件进行 DMA 通信，完成 `e1000_transmit()` 和 `e1000_recv()` 两个函数的填充，以及对接网络栈（如 `net_rx()`）处理数据包。具体任务包括初始化描述符环、管理 mbuf 缓存、处理中断通知、实现数据包入列与出列，并在用户态实现基本的 UDP socket 接口。通过这一实验，学生将深入理解设备驱动编程、内存映射 I/O、硬件中断响应机制、DMA 环境下的并发访问控制，并掌握操作系统如何在内核中构建高效且可靠的网络通信路径，这为系统层面处理网络协议和硬件交互提供了极具实战意义的训练。

清华大学 - rCore-Tutorial-Book 第三版

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。

- **Lab1：彩色化 LOG**

本实验旨在引导学生在实现裸机 `hello world` 输出的基础上，进一步完善操作系统的日志系统，具体要求包括实现彩色化输出的日志宏（如 `info!`、`debug!`、`error!` 等），用于内核调试信息的分类与可视化展示。通过利用 ANSI 转义序列控制终端颜色，学生不仅理解了串口输出机制，还提升了对内核输出机制与调试信息组织的掌握。实验过程中还要求输出操作系统各段的内存布局（如 `.text`、`.data` 等），帮助学生进一步了解程序的静态结构与内存映射关系。该实验在技术上虽简单，但通过引入日志抽象与可配置输出机制，逐步培养学生对内核可观察性、可调试性设计的意识与工程实践能力，是深入理解操作系统开发流程的重要起点。

- **Lab2：sys_write 安全检查**

本实验聚焦于操作系统内核中系统调用的安全性检查，要求学生在实现 `sys_write` 系统调用后，加入对用户传入地址合法性的验证，确保其只能访问本进程自身的内存空间，防止非法访问其他进程或内核空间的数据。尽管当前尚未启用虚拟内存机制，该实验通过手动实现地址范围检查，引导学生理解进程隔离与内核可信计算边界的重要性。通过这一实践，学生不仅加深了对系统调用机制的掌握，也培养了操作系统安全性设计的基本意识，体现了“保护内核与用户空间边界”的关键原则，是构建健壮操作系统的基础步骤之一。

- **Lab3：获取任务信息**

本实验要求学生在已有多任务系统基础上，新增系统调用 `sys_task_info`，用于查询指定任务的运行状态、系统调用使用统计及总运行时间等关键信息。学生需要理解并操作任务控制块（TCB），扩展其结构以记录系统调用次数及任务时间，进而实现内核中任务状态与行为的可观测性。通过实现这一任务信息查询机制，学生不仅熟悉了系统调用参数传递与内核态信息访问的流程，也加深了对任务调度、进程控制块管理和内核数据结构扩展的理解，培养了构建可监控、可管理操作系统的能力。

- **Lab4：重写 sys_get_time**

本实验主要包括两个任务：一是重写 `sys_get_time` 系统调用以适配引入虚拟内存后的内核架构，二是实现简化版的内存映射与解除映射系统调用 `mmap` 和 `munmap`。学生需处理虚拟地址空间的管理问题，如页对齐检查、权限位解析、映射冲突检测等，从而深入理解虚拟内存分配、权限控制及页表维护等操作系统核心机制。通过这些实践，学生不仅掌握了虚拟内存的基础原理和应用接口，还学会在用户态与内核态之间进行资源隔离与安全管理，强化了其对现代操作系统内存管理子系统的整体认知与实现能力。

- **Lab5：进程创建&&stride 调度算法**

本实验通过实现新的系统调用 `spawn` 或调度算法 `stride`，引导学生深入理解操作系统中进程创建与调度的核心机制。在“实验练习1”中，学生需自行设计并实现 `sys_spawn`，以支持直接加载和执行用户程序，跳过传统的 `fork + exec` 模式，从而理解现代进程创建中资源复制与程序加载的分离思路；此外，还涉及进程状态管理与系统调用参数验证等关键内核逻辑。而“实验练习2”则要求学生实现 stride 调度算法和 `sys_set_priority` 系统调用，以支持按优先级比例公平分配 CPU 时间，锻炼其对调度策略实现与时间片更新逻辑的掌握。两个练习分别从进程生命周期管理和调度器设计出发，培养学生独立设计系统调用接口、管理内核状态并分析公平性与效率的能力，体现了操作系统工程中“控制、资源与抽象”三大核心主题的实际应用。

- **Lab6：硬链接**

本实验要求学生在已有文件系统基础上实现硬链接机制，具体包括三个系统调用：`sys_linkat` 创建硬链接、`sys_unlinkat` 取消链接、以及 `sys_stat` 查询文件状态。实验的核心在于管理多个目录项指向同一个 inode，同时正确维护 inode 的引用计数（nlink）并更新元数据结构。学生需要理解文件系统中“目录项—inode—数据块”三层结构的分离与映射关系，掌握链接的建立与解除机制，并实现与 POSIX 接口兼容的调用行为。通过本实验，学生将掌握操作系统中文件系统管理的底层细节，包括 inode 生命周期管理、元信息同步与文件状态抽象表示，增强对持久化存储资源共享与隔离机制的系统性理解。

- **Lab7：进程通信：邮箱**

本实验要求学生实现基于邮箱机制的进程间通信（IPC），通过两个系统调用 `mailwrite` 和 `mailread` 实现“数据报”式通信：每个进程拥有一个环形缓冲区组成的邮箱，支持最多 16 条报文、每条最大 256 字节的消息收发，遵循 FIFO 顺序且不记录发送来源。学生需处理内核中邮箱队列的管理、边界条件的判断（如邮箱满、读缓冲不足）、内存访问安全性（如缓冲区有效性）等关键问题。通过本实验，学生不仅加深了对用户态与内核态之间数据传递机制的理解，还掌握了非阻塞通信接口设计、缓冲区管理、系统调用实现等系统级技能，提升了对操作系统中进程隔离与共享资源通信机制的整体把握。

- **Lab8：银行家算法**

本实验围绕“并发同步”这一操作系统核心主题展开，分为两个任务：一是模拟银行家算法背景下的分数更新过程，要求在用户态实现一个高并发、安全无死锁的评分系统；二是内核态实现 `eventfd` 系统调用，支持事件通知机制。在第一个任务中，学生需设计生产者-消费者模型，合理加锁防止并发冲突，并用选手分数和作为一致性验证手段，深化对经典同步问题（如哲学家问题）的理解。第二个任务要求学生扩展系统调用接口，实现一个支持计数器读写、阻塞与非阻塞、信号量与非信号量模式的通用事件描述符，从而熟悉 Linux 式异步事件通知机制。整体上，本实验通过跨用户态与内核态的同步机制实现，训练学生在并发模型设计、线程安全保障、系统接口抽象等方面的综合系统能力。

- **Lab9：支持图形显示的应用**

本实验旨在引导学生实现操作系统对图形显示的支持，要求在内核中集成 `virtio-gpu` 设备驱动，并通过实现设备文件 `/dev/fb0` 提供用户态应用访问显存的接口。学生需完成字符设备到图形帧缓冲的扩展，掌握设备驱动的注册、内核与用户空间的显存映射机制以及系统调用接口的设计与实现。最终目标是成功运行一个基于图形显示的“贪吃蛇”游戏应用，从而使学生在实际操作中理解设备驱动开发、图形渲

染路径和文件系统集成等关键技术，提升其构建具备图形界面的嵌入式操作系统能力。

技术方案

本项目的组成部分包含基于Rust语言的系统内核、内核文档注释、用户程序与测试用例、自动化测试脚本、实验指导手册与参考实现。下面分别说明这些部分的技术方案与参考内容。

基于Rust语言的系统内核

- 本项目的主要工作在于教学方案设计，因此没有重新实现类xv6的内核，内核部分参考了xv6-riscv-rust的实现
- 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

Rust语言用户程序、测试用例以及用户多线程库

- 本项目使用Rust语言重写了部分xv6的用户程序以及测试用例，
  - 后续计划使用纯Rust语言替代C语言用户程序
- 本项目通过用户程序实现对系统内核的测试，具体执行方法为在`shell`中执行`usertests`
- 由于系统调用接口保持一致，目前并行存在Rust语言与C语言版本的用户程序
- 基于Rust语言实现的用户程序进一步实现用户多线程库
  - 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

自动化测试脚本

- 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

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
- 针对每个不同实验均编写的具体的实验指导，详细内容在后文中完整展现
- 每个实验都提供了参考实现，以及参考实现的代码解析，详细内容在后文中完整展现
    - 参考实现均经过实机运行，能够在测试脚本下得到满分

系统内核架构

系统内核整体架构

系统内核部分可粗略分割为硬件驱动与内核两部分，整体架构如图所示

<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>

硬件驱动

- **页表驱动**：基于RISC-V硬件实现页表机制，负责地址翻译
- **中断控制器驱动**：处理和分发来自硬件设备的中断请求
- **通信串口驱动**：提供终端与系统交互的基础通信功能
- **磁盘驱动**：支持系统与磁盘之间的数据交换，实现磁盘I/O操作

内核

该模块是内核的核心，抽象硬件驱动功能并通过系统调用提供给用户程序。其包括三个子模块：

1. **内存管理模块**

- **物理内存管理**：实现物理内存页的分配、释放与管理。
- **虚拟内存管理**：负责维护虚拟地址与物理地址之间的映射，提供隔离与安全保障。

2. **进程管理模块**

- **中断处理**：响应外部设备或软件产生的中断请求。
- **进程调度**：实现抢占式优先级调度，采用轮转算法分配时间片。
- **进程控制**：提供进程的创建、终止、阻塞与唤醒等控制功能。
- **进程同步**：支持信号量、互斥锁、条件变量与屏障等同步机制，保障并发安全。

3. **文件系统模块**

- **物理文件系统**：直接与磁盘驱动交互，管理存储介质上的实际数据。
- **虚拟文件系统**：实现文件系统的抽象层，提供统一接口，层次化分为：
    - 硬件层（磁盘设备驱动）
    - 缓存层（提高访问性能）
    - 日志层（保障数据一致性与恢复）
    - 索引节点层（管理文件元数据）
    - 目录层（维护文件树结构）
    - 文件描述符层（管理文件操作句柄）

内核代码树如下所示：

```
├── asm
│   ├── entry.S             // 汇编启动入口代码，设置堆栈并跳转到内核主函数
│   ├── init.c              // 用于生成用户态初始化代码的C语言源文件
│   ├── initcode.S          // 用户初始化程序(init)的汇编代码，嵌入到内核镜像中
│   ├── kernelvec.S         // 内核中断向量表定义，处理中断和异常入口
│   ├── swtch.S             // 上下文切换的汇编实现，保存/恢复CPU状态
│   └── trampoline.S        // 用户态与内核态切换的跳板页代码，含sret指令
├── consts
│   ├── driver.rs           // 驱动模块所用常量，例如设备寄存器基址等
│   ├── fs.rs               // 文件系统相关常量定义，如磁盘块大小等
│   ├── memlayout.rs        // 内核虚拟地址空间布局与物理内存布局定义
│   ├── mod.rs              // consts模块总入口，统一导出子模块
│   ├── param.rs            // 全局系统参数，例如最大进程数、内核栈大小
│   └── riscv.rs            // RISC-V 架构相关常量，例如CSR寄存器号等
├── driver
│   ├── console.rs          // 控制台输出驱动，依赖UART输出
│   ├── mod.rs              // driver模块入口，统一导出各设备驱动
│   ├── uart.rs             // UART串口驱动，实现字符收发
│   └── virtio_disk.rs      // VirtIO磁盘驱动，处理块设备的读写操作
├── fs
│   ├── bio.rs              // 块缓冲区管理，缓存磁盘块读写
│   ├── block.rs            // 磁盘块分配器，管理空闲块
│   ├── file
│   │   ├── mod.rs          // file模块入口，统一导出文件与管道接口
│   │   └── pipe.rs         // 管道文件实现，实现无名管道的读写
│   ├── inode.rs            // 索引节点实现，表示文件的元数据
│   ├── log.rs              // 日志模块，支持原子文件系统操作
│   ├── mod.rs              // fs模块入口，统一导出文件系统各子模块
│   └── superblock.rs       // 超级块结构与加载逻辑，描述文件系统全局信息
├── ld
│   └── kernel.ld           // 链接脚本，控制内核各段的内存布局
├── lib.rs                  // 内核库模块入口，包含通用宏、panic等定义
├── main.rs                 // 内核主函数，启动各子系统并进入第一个进程
├── mm
│   ├── addr.rs             // 虚拟地址与物理地址处理工具
│   ├── kalloc.rs           // 内核物理页分配器，实现简单的空闲页管理
│   ├── kvm.rs              // 内核页表初始化与映射操作
│   ├── list.rs             // 链表工具，支持双向链表实现
│   ├── mod.rs              // mm模块入口，统一导出内存管理模块
│   └── pagetable.rs        // 页表实现，包含页表项结构与映射函数
```

```
├── plic.rs                 // PLIC 外部中断控制器驱动，实现中断使能与查询
├── printf.rs               // 内核 printf 实现，格式化字符串输出函数
├── process
|   ├── context.rs          // 上下文结构体，保存用户进程寄存器状态
|   ├── cpu.rs              // CPU状态与调度器实现，包含每核的调度器状态
|   ├── mod.rs              // process模块入口，导出进程调度与管理功能
|   ├── proc
|   |   ├── elf.rs          // ELF文件加载器，将用户程序加载进内存
|   |   ├── mod.rs          // proc子模块入口，导出进程创建、退出等功能
|   |   └── syscall.rs      // 系统调用处理器，提供用户态系统服务接口
|   └── trapframe.rs        // 陷入帧结构体，保存陷入内核时的用户态上下文
├── register
|   ├── clint.rs            // CLINT 定时中断控制器驱动，管理mtime中断
|   ├── mie.rs              // mie CSR 寄存器访问封装，控制中断使能位
|   ├── mod.rs              // register模块入口，导出CSR访问相关接口
|   ├── mstatus.rs          // mstatus CSR 操作，管理特权级状态
|   ├── satp.rs             // satp CSR 操作，管理页表根指针
|   ├── scause.rs           // scause CSR，分析异常或中断来源
|   ├── sie.rs              // sie CSR 操作，软件中断控制
|   ├── sip.rs              // sip CSR 状态寄存器封装，查询中断状态
|   └── sstatus.rs          // sstatus CSR 操作，管理S模式状态标志
├── rmain.rs                // 多核入口函数，初始化每个CPU的调度器与陷入向量
├── sleeplock.rs            // 睡眠锁实现，适合I/O等待的互斥保护
├── spinlock.rs             // 自旋锁实现，用于短时临界区互斥保护
├── start.rs                // 内核入口函数，启动第一个核的执行流程
└── trap.rs                 // 中断与异常处理主逻辑，分派不同的trap类型
```

内存管理模块

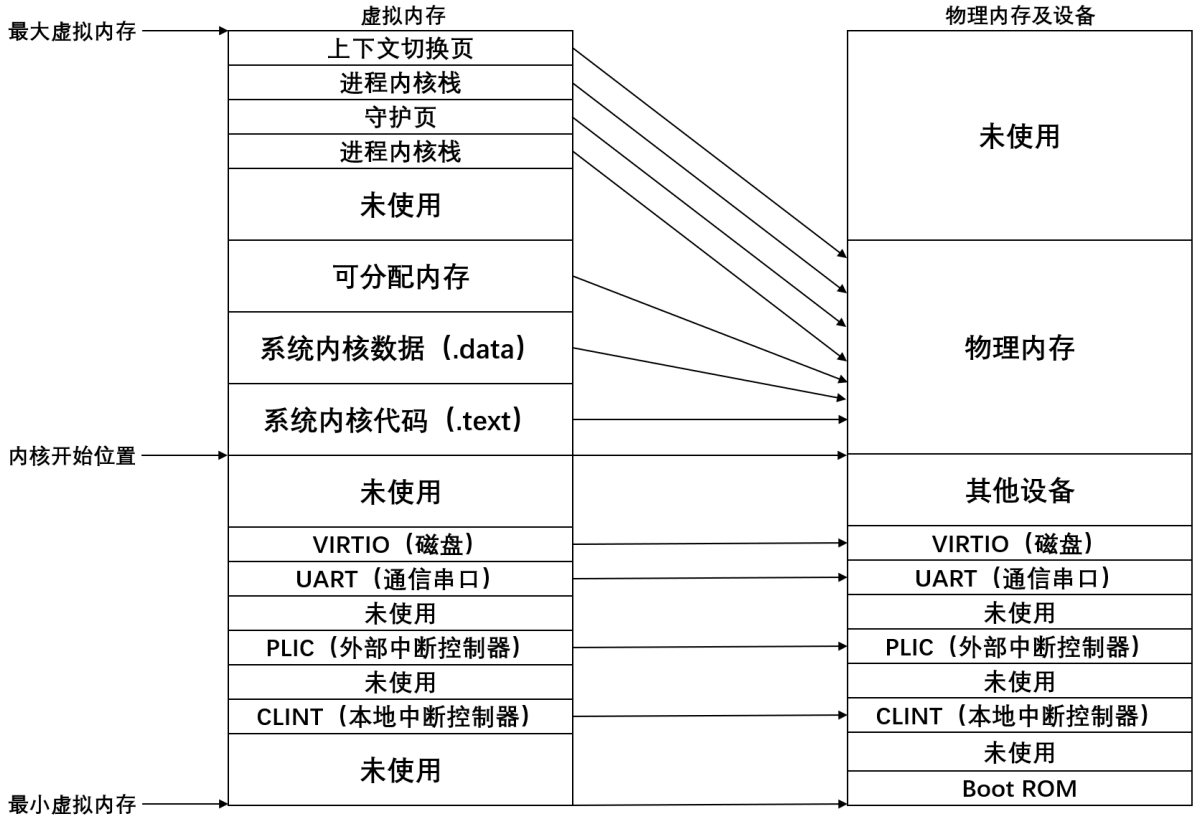
物理内存管理

物理内存管理负责对物理内存进行有效组织和分配，以确保内存使用效率与安全性。其整体架构包括以下内容：

1. **物理内存布局**
   - 以内核代码段末尾为起点，内核数据段后的所有物理内存视作可分配内存池，用于系统和用户程序的动态内存请求。
2. **分页管理机制**
   - 将物理内存分成固定大小的页面，每个页面由链表节点管理。链表用于维护空闲页面的状态，以支持高效的页面分配和回收。
3. **伙伴算法动态分配**
   - 基于分页机制，采用伙伴（Buddy）算法进行动态内存分配。
   - 每个内存块的大小均为2的幂次，每个块拥有一个地址相邻、大小相同的伙伴块。
   - 内存块处于两种状态之一：
     - **空闲状态**：可以被分配。
     - **已分配状态**：正在被系统或用户程序使用。
4. **内存分配逻辑**
   - 请求内存时：
     - 请求的内存大小向上取整为2的幂次。

        - 查找空闲内存块链表，若不存在匹配块，则递归地分割更大的内存块，直到大小合适的空闲块出现。
        - 将匹配的内存块标记为已分配状态并提供给请求方。
    - 释放内存时：
        - 内存释放后递归检查伙伴块的状态，若伙伴块也为空闲状态，则将其合并为更大的空闲块，循环执行直至无法再合并。
5. **Rust 标准库兼容性**
    - 实现 Rust `alloc` 库中的 `GlobalAlloc` 接口，提供通用的内存分配（`alloc`）和释放（`dealloc`）函数。
    - 借助 Rust 标准库中的数据结构，提升内核与用户程序的开发效率及内存安全性。

虚拟内存管理



虚拟内存管理模块的目标是通过页表机制将虚拟地址空间有效映射到物理内存空间，实现进程隔离和内存保护。虚拟内存的映射方案如图所示。具体结构包括：

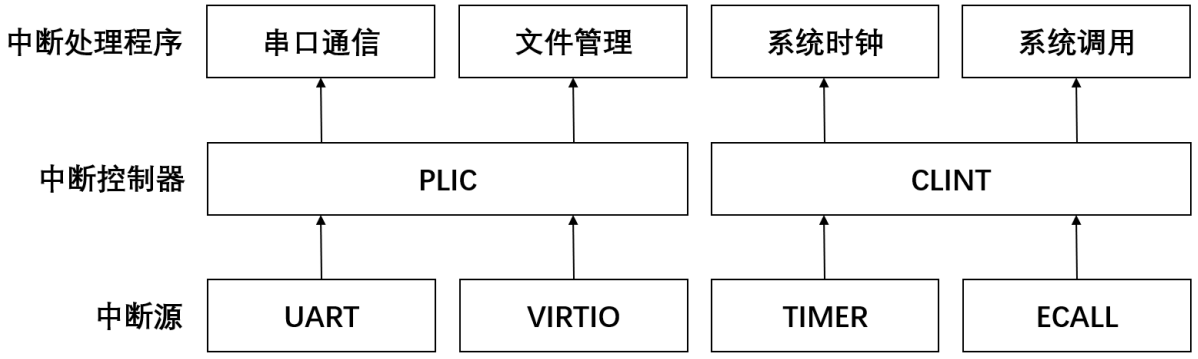
1. **内核与用户虚拟内存空间**
    - 内核与用户进程各自拥有独立的虚拟内存地址空间，通过页表映射到物理内存。
2. **内核页表布局**
    - **上下文切换页（位于虚拟地址空间顶部）**
        - 供进程上下文切换使用，允许不同进程在相同虚拟地址保存和恢复上下文信息，以简化上下文切换流程。
    - **进程内核栈区域**
        - 位于上下文切换页之下，专为每个进程的内核线程提供独立的内核栈空间，保证用户线程与内核线程数据的有效隔离，避免相互干扰。
    - **内核代码与数据映射区域**

        - 对内核自身代码和数据的直接映射，使得内核能够高效访问内存。
3. **设备寄存器地址映射**
    - 设备寄存器以特定的内存地址形式被访问，这些地址并不位于标准物理内存范围内。
    - 本架构采用**直接映射**方案，将设备寄存器的物理地址直接映射到内核页表中虚拟地址相同的位置，以方便内核访问硬件设备。
4. **页表管理与数据传递**
    - 页表管理模块负责不同虚拟地址空间之间的映射管理，提供高效的数据传递和共享机制，确保内存隔离的同时支持必要的通信。

进程管理模块

中断处理机制

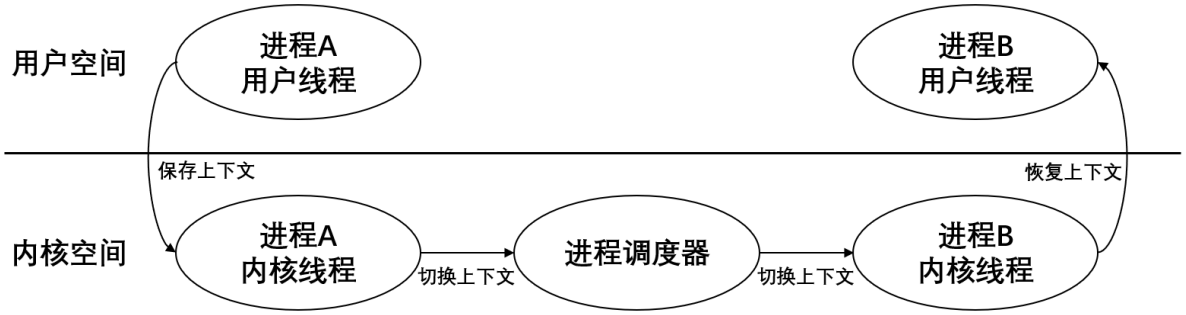
中断处理机制负责接收并统一处理多种中断源的请求，处理流程如图所示，具体设计如下：



1. **支持的中断源**
    - **外部中断**（PLIC控制）：
        - 通信串口（UART）
        - 磁盘设备（基于VIRTIO接口）
    - **本地中断**（CLINT控制）：
        - 定时器（TIMER）
        - 环境调用（ECALL）
2. **中断分发逻辑**
    - 所有中断均由统一的中断处理函数进行捕获处理。
    - 通过Rust的`match`表达式实现对不同中断原因的模式匹配，将对应的中断请求转发到指定模块（如进程调度模块或设备驱动模块）。
    - 使用match表达式的通配符处理所有未明确定义的情况，统一输出警告信息以保证确定性与安全性。

进程调度机制

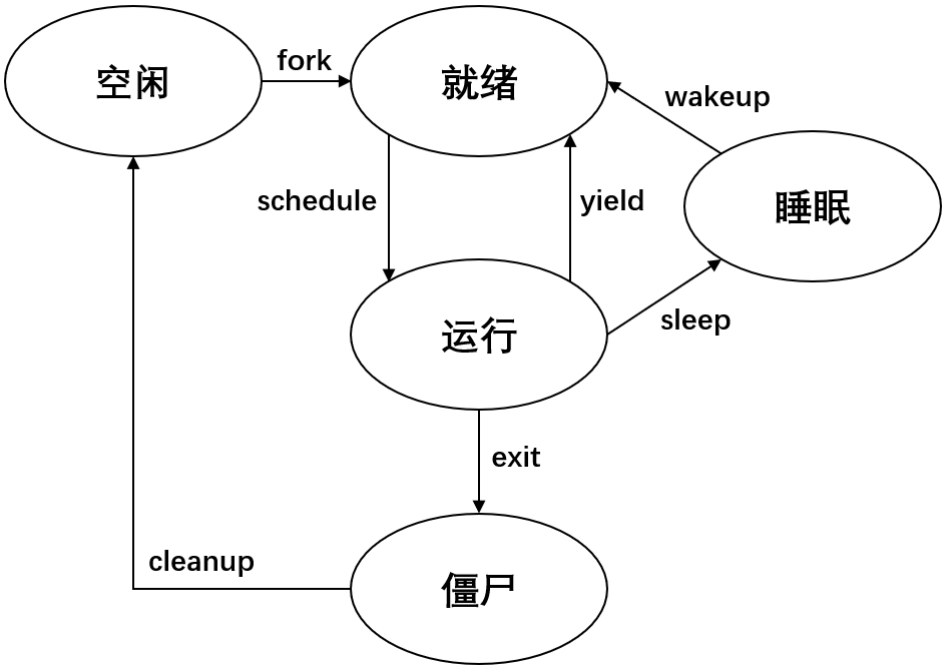
进程调度器负责不同进程之间的上下文切换，切换的流程如图所示，其设计如下：



1. **线程模型**
    - 每个进程包含两个线程：
        - **用户线程**：运行在RISC-V的用户模式（U-mode），每个用户线程拥有独立的页表。
        - **内核线程**：运行在RISC-V的内核模式（S-mode），共享相同内核页表但使用独立的内核栈。
2. **调度过程**
    - 进程调度触发条件包括：
        - 时间片耗尽
        - 进程主动放弃CPU
        - 进程等待资源进入睡眠状态
    - 以进程A切换到进程B为例，调度过程为：
        1. 保存进程A用户线程上下文，切换至A的内核线程。
        2. 保存进程A内核线程上下文，切换到调度器上下文。
        3. 调度器选择下一个运行的进程B。
        4. 加载进程B内核线程上下文。
        5. 最终恢复进程B用户线程上下文，完成一次完整的上下文切换。
3. **多核支持与并发安全**
    - 所有CPU核心共享一个全局进程队列，调度器需访问该队列。
    - 利用Rust的所有权机制与智能指针（如`Arc<Mutex<T>>`），确保每个核心在访问进程队列前必须获取独占的写权限，以避免数据竞争和并发错误。

进程控制机制

进程控制机制负责管理进程生命周期中的状态转换，确保操作系统运行效率最大化，进程状态切换如图所示，具体包括：



1. **进程状态定义**
   - **空闲状态（Idle）**：系统启动时所有进程的初始状态。
   - **就绪状态（Ready）**：等待CPU执行。
   - **运行状态（Running）**：占有CPU进行运行。
   - **睡眠状态（Sleeping）**：等待资源就绪（如锁、磁盘I/O、串口输入等）。
   - **僵尸状态（Zombie）**：进程终止后，尚未被父进程回收资源。
2. **进程生命周期管理**
   - 系统初始化完成后，启动**init进程**用于回收孤儿进程资源。
   - 创建进程（`fork`调用）时：
     - 将父进程页表和物理页面复制给子进程。
     - 子进程进入**就绪状态**。
   - 进程执行过程中：
     - 如果时间片耗尽或主动让出CPU，则返回**就绪状态**。
     - 若等待资源则进入**睡眠状态**，直到资源就绪再返回就绪状态。
   - 进程终止（`exit`调用）后：
     - 进入**僵尸状态**。
     - 父进程负责回收其僵尸子进程资源。
     - 若父进程已终止，则父进程被设置为**init进程**。
3. **进程控制块（PCB）实现**
   - 使用Rust中的`Option`枚举封装进程控制块指针：
     - 若父进程不存在（如已退出），访问PCB将返回`Option::None`。
     - 强制程序员显式处理进程不存在的情况，避免空指针引用导致的安全问题。

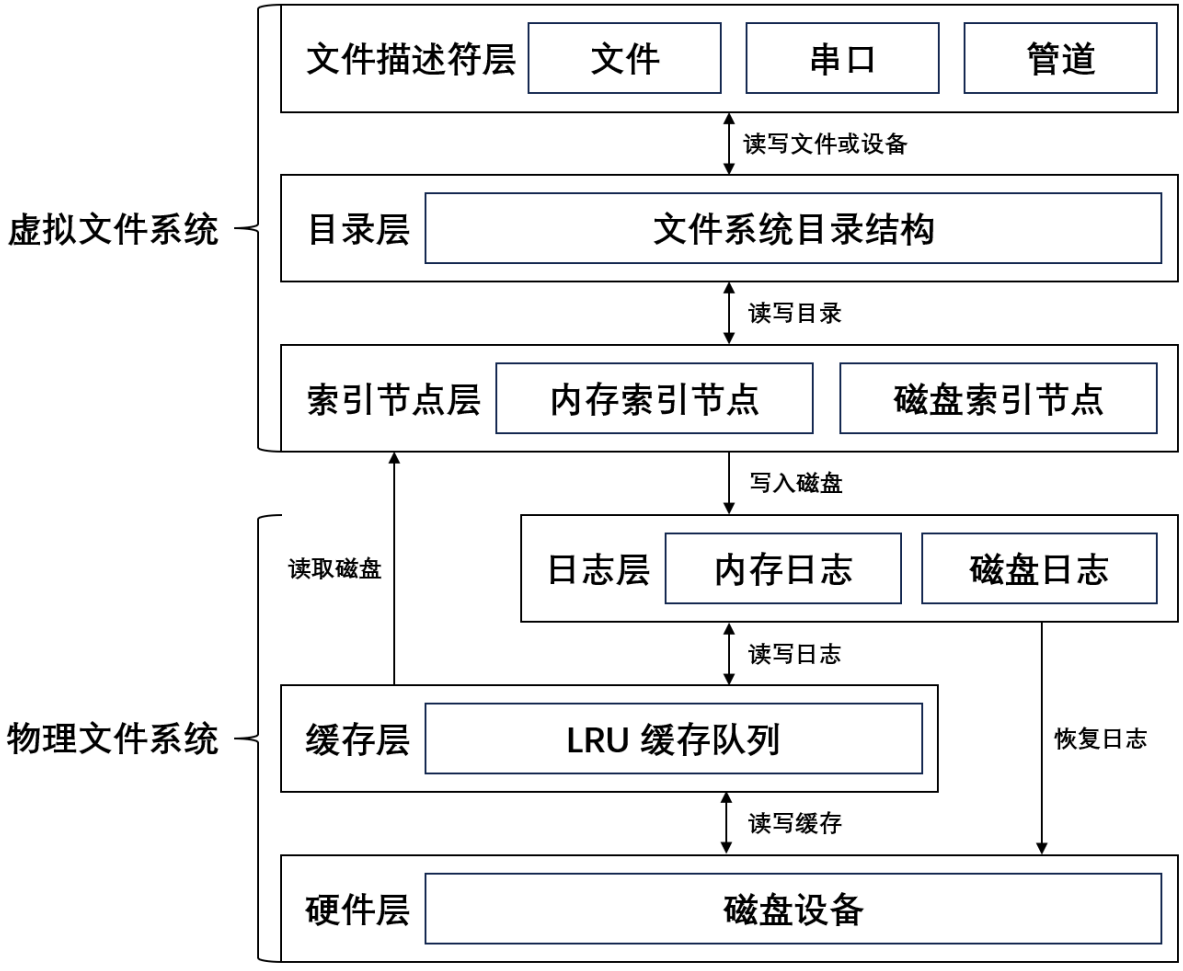
进程同步机制

进程同步机制使用锁来保护临界区，避免数据竞争问题，包含两种类型的锁：

1. **自旋锁（Spinlock）**
   - 基于RISC-V架构提供的原子指令`amoswap`实现。
   - 使用Rust标准库`core::sync::atomic::AtomicBool`提供封装，简单实现自旋锁的加锁和释放。
2. **睡眠锁（Sleeplock）**
   - 在自旋锁的基础上，为了防止CPU资源浪费和死锁问题，设计了睡眠锁机制。
   - 睡眠锁申请过程：
     1. 若锁已被占用，申请锁的进程释放所有已持有的锁并进入睡眠状态。
     2. 锁被释放时，所有等待此锁的睡眠进程被唤醒，并重新尝试获取锁和恢复执行。
   - 保证了高效的CPU资源利用和并发安全性。

文件系统模块

文件系统由物理文件系统与虚拟文件系统两部分组成，其结构与交互模式如图所示：



物理文件系统

1. 硬件层

- 负责与磁盘设备的底层交互：
  - 提供磁盘的初始化、读写操作。

- 初始化过程将文件系统结构载入磁盘，从最低地址开始分别为：
  - **启动块**：负责磁盘启动。
  - **超级块：记录文件系统元数据。
  - **日志块**：记录磁盘写操作日志。
  - **索引节点块：记录文件及目录的元数据。
  - **位图块**：记录磁盘数据块使用情况。
  - **数据块区域：存储文件实际数据。

---

2. 缓存层

- 使用环形队列实现磁盘数据缓存：
  - 采用LRU（最近最少使用）替换算法。
  - 缓存队列由Rust中的`Option`枚举包裹缓存块数组构成。
  - 维护两个可变引用（head 和 tail），分别标记队列中最新和最旧的缓存块，实现高效的LRU驱逐策略。
  - 获取缓存块时：
    - 若有空闲块，直接使用；
    - 若无空闲块，驱逐最旧的未引用缓存块；
    - 若缓存队列满且无法驱逐，则返回`Option::None`。
  - Rust借用检查保证缓存块引用时的数据安全与并发安全。

---

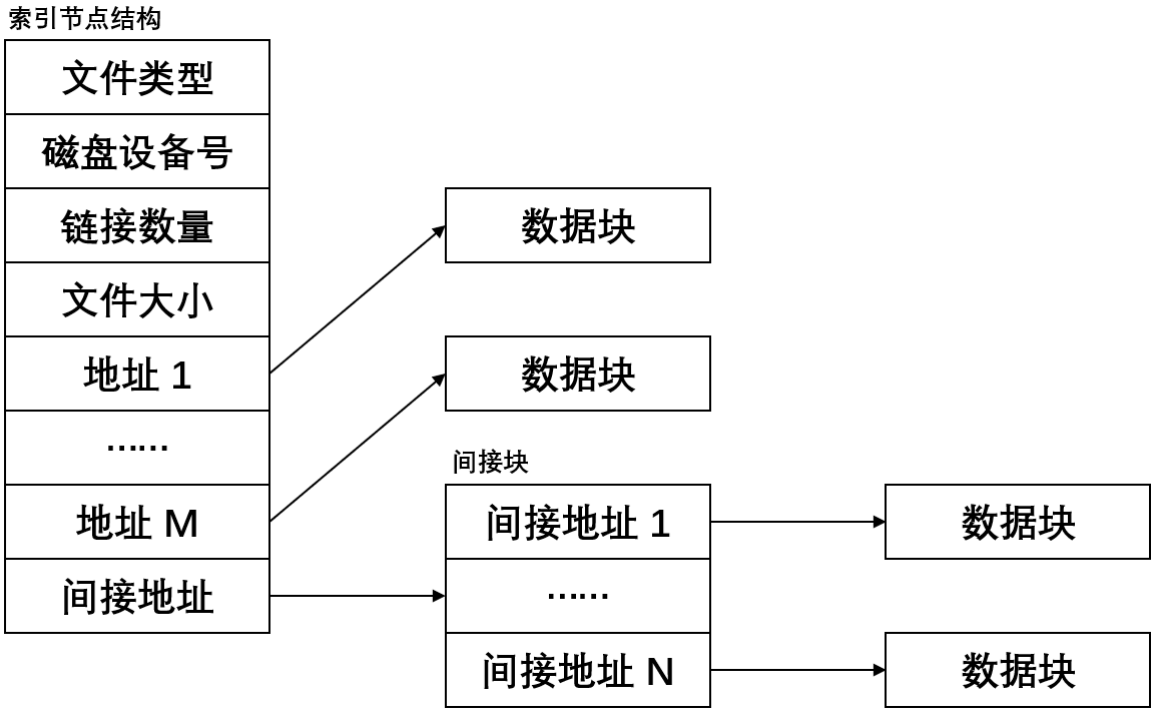
3. 日志层

- 日志层用于记录所有磁盘的写操作，确保文件系统操作的一致性与持久性：
  - 系统启动时检查日志块状态，若发现未完成的日志操作（异常关机），则根据日志恢复磁盘数据。
  - 日志写入流程：
    1. 检查日志空间是否足够，若不足则进程睡眠等待；
    2. 进程声明开始文件操作，写入日志区；
    3. 操作完成，声明文件操作结束；
    4. 当无其他未完成文件操作时，日志层执行组提交，将日志内容写入磁盘日志区；
    5. 完整提交后再将数据安装到相应磁盘位置。
  - 若系统异常停止：
    - 如果日志未完整提交，所有更改回滚；
    - 如果日志完整提交，所有更改恢复并持久化。

虚拟文件系统

虚拟文件系统基于物理文件系统提供的接口，为应用程序提供统一抽象接口。

1. 索引节点层



- 使用索引节点（Inode）存储文件和目录的元数据信息，索引节点的具体结构如图所示，包括：
  - 文件类型（普通文件或目录）；
  - 磁盘设备号（文件所在磁盘设备位置）；
  - 链接数量（文件被链接的目录个数）；
  - 文件大小（文件占用磁盘块数）；
  - 直接块地址与间接块地址：
    - **直接块地址**：直接存储文件数据块地址，适合小文件快速读取；
    - **间接块地址**：存储指向数据块地址的指针，需要两次磁盘访问，支持较大的文件。
- 采用Rust所有权模型：
  - 每个进程获取索引节点拷贝，拥有独立所有权；
  - 在更新索引节点时，进程必须获取对应索引节点锁以保证并发安全。

---

2. 目录层

- 使用树状结构管理文件与目录：
  - 通过解析输入的目录字符串，目录层准确定位目标索引节点；
  - 提供目录查找与定位功能，使文件系统结构清晰直观。

---

3. 文件描述符层

- 提供文件访问的最高层抽象：
  - 每个进程维护一个数组记录已打开文件的索引节点引用；
  - 使用数组编号访问对应文件，提供统一的读写接口。
- 特殊设计——设备抽象：
  - 将设备（如串口）抽象为文件进行访问，提供统一的读写接口；
  - 设备初始化时将读写函数注册到系统设备列表；

- 进程读写文件描述符时：
  - 若目标为设备，调用设备的读写函数；
  - 若目标为普通文件，调用普通文件读写函数。

内核文档注释

意义

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。

降低理解操作系统内核的难度

操作系统内核模块通常耦合度高、代码复杂，且涉及大量底层细节。对初学者而言，直接阅读内核源码往往困难重重，尤其在调用链中存在大量未知函数时，极易陷入理解断层。补充的文档注释提供了关键函数的功能说明与调用上下文，有助于学生在阅读某一段代码时快速构建整体逻辑，提升对系统运行机制的认知效率。此外，Rust 语言本身引入了所有权、生命周期等类型系统机制，增加了代码的理解门槛。文档注释中对这些特性的适当解释，进一步缓解了学生在理解 Rust 内核实现时的困难。

提升内核调试的效率

在操作系统实验过程中，调试常常是最大瓶颈之一。由于运行环境处于内核态，传统的调试手段（如 `print` 调试或 GDB 工具）常常无法有效定位问题，特别是在涉及汇编代码、中断处理或低级内存操作时更为明显。通过在注释中加入对函数执行中可能出现的异常场景与错误触发条件的提示，学生在实验中遇到 bug 时能够借助文档迅速建立排查思路，从而有效提升调试效率，减少挫败感。

帮助学生参与内核开发

在传统操作系统教学中，学生往往停留在接口使用和概念理解阶段，难以深入系统性地掌握内核的实现原理与工程结构。本项目通过详尽注释增强了代码的可读性与可维护性，为学生提供了可参与、可修改、可扩展的学习基础。在此基础上，学生不仅能够完成实验任务，更有能力对内核行为进行主动探索与优化，实现从“阅读者”向“开发者”的角色转变。

增加对Unsafe代码的理解

Rust 的核心优势在于其对内存安全、并发安全等系统级错误的编译期保障机制。本项目在文档注释中补充了函数潜在的安全风险与语言层面的防御机制，帮助学生在理解系统功能的同时建立起面向安全的编程意识。该能力对于从事操作系统、嵌入式系统等领域的开发具有重要价值。

实现

内核核心模块的文档注释完全遵循Rust文档注释规范，根据rustdoc文档所指导的示例如下，每个文档注释都应该包含：

- **简要说明**：开头一两句话概括作用
- **参数列表**（# Parameters）

- **返回值说明**（# Returns）
- **错误说明**（# Errors / # Paincs）
- **示例代码**（# Examples）

下面给出一个来自Rust标准库的例子，`std::env::args()` 函数，其文档如下：

```
Returns the arguments which this program was started with (normally passed
via the command line).

The first element is traditionally the path of the executable, but it can be
set to arbitrary text, and may not even exist. This means this property should
not be relied upon for security purposes.

On Unix systems shell usually expands unquoted arguments with glob patterns
(such as `*` and `?`). On Windows this is not done, and such arguments are
passed as-is.

# Panics

The returned iterator will panic during iteration if any argument to the
process is not valid unicode. If this is not desired,
use the [`args_os`] function instead.

# Examples

```
use std::env;

// Prints each argument on a separate line
for argument in env::args() {
    println!("{}", argument);
}
```

[`args_os`]: ./fn.args_os.html
```

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

本项目采用手工编写与大模型优化的方式编写文档注释，具体实现流程如下：

1. 首先阅读需要编写文档的代码，并找出其在内核中被调用的位置
2. 编写其功能说明、参数、返回值、以及可能的错误场景
3. 将上述内容以及函数代码输入到大模型中，命令其检查并完善函数的流程解释
4. 若函数是Unsafe的，则额外添加其安全性说明
5. 人工检查所有内容，并添加到函数顶部
6. 执行`cargo doc --no-deps --document-private-items`,利用rustdoc工具产生HTML版本的文档

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，具体的文件与函数覆盖如下所示：

```
.
├── fs
|   ├── bio.rs
|   ├── block.rs
|   ├── file
|   |   ├── mod.rs
|   |   └── pipe.rs
|   ├── inode.rs
|   └── log.rs
├── mm
|   ├── kalloc.rs
|   ├── kvm.rs
|   └── pagetable.rs
├── process
|   ├── cpu.rs
|   ├── mod.rs
|   ├── proc
|   |   ├── elf.rs
|   |   └── mod.rs
|   └── trapframe.rs
└── rmain.rs
```

使用tokie统计注释行数，目前文档注释总量为3435行。

```
===============================================================================
 Language            Files        Lines         Code     Comments       Blanks
===============================================================================
 GNU Style Assembly      6          390          341            6           43
 C                      23         5123         4325          251          547
 C Header               12          644          459           97           88
 Makefile                1          130           97            7           26
 Markdown                3          198            0          177           21
 Perl                    1           40           33            2            5
 Python                  2          672          514           47          111
 TOML                    2           28           22            1            5
-------------------------------------------------------------------------------
 Rust                   49         7018         5648          275         1095
 |- Markdown            45         4223            0         3435          788
```

```
 (Total)                          11241         5648         3710         1883
================================================================================
 Total                    99         14243        11439          863         1941
================================================================================
```

示例

内存管理模块中，`BuddySystem`的结构体定义如下

```rust
/// 伙伴系统内存分配器的核心结构。
///
/// `BuddySystem` 是内核堆分配器的底层实现，采用经典的伙伴系统算法，
/// 将堆内存划分为以 2 的幂为大小的块，通过分裂和合并操作进行内存分配与回收。
///
/// 本结构记录了伙伴系统管理的内存范围、分配状态以及每个块大小等级的元信息，
/// 并在初始化完成后提供线程安全的内存操作接口（通常由 [`KernelHeap`] 封装）。
///
/// 初始化通过 [`BuddySystem::init`] 方法完成，
/// 分配与释放分别通过 `alloc()` 和 `dealloc()` 实现。
pub struct BuddySystem {
    /// 伙伴系统管理的起始物理地址（页对齐）。
    base: usize,

    /// 伙伴系统管理的实际结束地址（页对齐），不包括该地址本身。
    actual_end: usize,

    /// 支持的块大小等级数量，对应 log2(最大块数) + 1。
    /// 每个等级的块大小为 `2^k * LEAF_SIZE`。
    nsizes: usize,

    /// 标记该结构是否已完成初始化。
    /// 防止重复初始化，若已初始化再次调用 `init()` 将 panic。
    initialized: bool,

    /// 每个块大小等级对应的分配状态和空闲链表信息。
    /// 该字段为指向 `BuddyInfo` 切片的裸指针，需在初始化过程中手动构造并写入。
    /// 使用 `MaybeUninit` 包装，以避免未初始化内存的 UB。
    infos: MaybeUninit<*mut [BuddyInfo]>,
}
```

内存管理模块中，`GlobalAlloc`接口的实现如下所示

```rust
/// 实现 `GlobalAlloc` 接口以支持全局堆分配。
///
/// 本实现使 `KernelHeap` 可以作为 Rust 全局分配器使用，
/// 支持内核中 `Box`、`Vec` 等标准堆分配类型的底层内存管理。
///
/// `alloc` 和 `dealloc` 方法会加锁访问内部的 [`BuddySystem`] 分配器，
/// 从而保证在多核环境下的线程安全。
unsafe impl GlobalAlloc for KernelHeap {
    /// # 功能说明
    ///
    /// - `alloc`：根据指定的内存布局分配堆空间；
```

```
    ///
    /// # 参数
    ///
    /// - `layout`：一个 [`Layout`] 对象，指定分配或释放的内存大小与对齐；
    /// # 返回值
    ///
    /// - `alloc` 返回一个满足给定 `layout` 的裸指针；若内存不足返回空指针；
    unsafe fn alloc(&self, layout: Layout) -> *mut u8 {
        self.0.lock().alloc(layout)
    }

    /// # 功能说明
    ///
    /// - `dealloc`：释放由 `alloc` 分配的堆空间，并尝试进行伙伴合并；
    /// # 参数
    ///
    /// - `layout`：一个 [`Layout`] 对象，指定分配或释放的内存大小与对齐；
    /// - `ptr`：待释放的内存指针，必须来自之前分配的有效地址；
    unsafe fn dealloc(&self, ptr: *mut u8, layout: Layout) {
        self.0.lock().dealloc(ptr, layout)
    }
}
```

进程管理模块中，进程控制块及其私有数据块的实现如下

```
/// 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。
pub struct Proc {
    /// 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,
}

/// 进程私有数据结构，保存进程运行时的核心信息。
///
/// 该结构体仅在当前进程运行时访问，或在持有 [`ProcExcl`] 锁的其他进程（例如 fork）
/// 初始化时访问。包含内核栈指针、内存大小、上下文、打开的文件、用户页表等私有资源。
pub struct ProcData {
    /// 进程内核栈的起始虚拟地址。
    kstack: usize,
    /// 进程使用的内存大小（字节数）。
    sz: usize,
    /// 进程上下文（寄存器状态等），用于上下文切换。
    context: Context,
```

```
    /// 进程名称，最长16字节，通常用于调试和显示。
    name: [u8; 16],
    /// 进程打开的文件数组，元素为可选的引用计数智能指针。
    open_files: [Option<Arc<File>>; NFILE],
    /// 指向 TrapFrame 的裸指针，保存用户态寄存器临时值等信息。
    pub tf: *mut TrapFrame,
    /// 进程的用户页表，管理用户地址空间映射。
    pub pagetable: Option<Box<PageTable>>,
    /// 进程当前工作目录的 inode。
    pub cwd: Option<Inode>,
}
```

进程管理模块中，fork函数的实现如下

```
    /// # 功能说明
    /// 创建当前进程的一个子进程（fork），
    /// 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。
    ///
    /// # 流程解释
    /// 1. 获取当前进程的私有数据引用 `pdata`。
    /// 2. 通过 `PROC_MANAGER.alloc_proc()` 分配一个新的子进程，
    ///    若失败则返回错误 `Err(())`。
    /// 3. 获取子进程的排它锁 `cexcl` 和私有数据 `cdata`。
    /// 4. 复制父进程的用户内存到子进程页表，调用 `uvm_copy`。
    ///    若复制失败，清理子进程相关资源，返回错误。
    /// 5. 设置子进程的内存大小 `sz` 与父进程一致。
    /// 6. 复制 TrapFrame（用户寄存器状态），并将子进程的返回值寄存器 `a0` 设为 0。
    /// 7. 克隆父进程的打开文件数组和当前工作目录。
    /// 8. 复制父进程名称到子进程。
    /// 9. 记录子进程的进程 ID（pid）。
    /// 10. 设置子进程的父进程为当前进程。
    /// 11. 将子进程状态置为 `RUNNABLE`，表示可调度。
    /// 12. 返回子进程的进程 ID。
    ///
    /// # 参数
    /// - `&mut self`：当前进程的可变引用，用于访问自身私有数据和状态。
    ///
    /// # 返回值
    /// - `Ok(usize)`：子进程的进程 ID（pid）。
    /// - `Err(())`：分配子进程或复制内存失败时返回错误。
    ///
    /// # 可能的错误
    /// - 子进程分配失败（如进程表满），返回 `Err(())`。
    /// - 复制父进程内存失败时，清理子进程并返回错误。
    /// - 若 TrapFrame 指针无效，`unsafe` 操作可能导致未定义行为。
    ///
    /// # 安全性
    /// - 使用了多处 `unsafe` 操作，包括裸指针复制和子进程数据访问，
    ///   假设指针有效且内存分配正确。
    /// - 调用者需保证进程状态和私有数据在调用时无并发冲突。
    /// - 子进程资源清理确保不产生内存泄漏和悬挂指针。
    fn fork(&mut self) -> Result<usize, ()> {
        let pdata = self.data.get_mut();
        let palarm = self.alarm.get_mut();
```

```
        let child = unsafe { PROC_MANAGER.alloc_proc().ok_or(())? };
        let mut cexcl = child.excl.lock();
        let cdata = unsafe { child.data.get().as_mut().unwrap() };
        let calarm = unsafe { child.alarm.get().as_mut().unwrap() };

        // clone memory
        let cpgt = cdata.pagetable.as_mut().unwrap();
        let size = pdata.sz;
        if pdata.pagetable.as_mut().unwrap().uvm_copy(cpgt, size).is_err() {
            debug_assert_eq!(child.killed.load(Ordering::Relaxed), false);
            child.killed.store(false, Ordering::Relaxed);
            cdata.cleanup();
            calarm.cleanup();
            cexcl.cleanup();
            return Err(())
        }
        cdata.sz = size;

        // clone trapframe and return 0 on a0
        unsafe {
            ptr::copy_nonoverlapping(pdata.tf, cdata.tf, 1);
            ptr::copy_nonoverlapping(palarm.alarm_frame, calarm.alarm_frame, 1);
            cdata.tf.as_mut().unwrap().a0 = 0;
        }

        // clone opened files and cwd
        cdata.open_files.clone_from(&pdata.open_files);
        cdata.cwd.clone_from(&pdata.cwd);

        // copy process name
        cdata.name.copy_from_slice(&pdata.name);

        let cpid = cexcl.pid;

        drop(cexcl);

        unsafe { PROC_MANAGER.set_parent(child.index, self.index); }

        let mut cexcl = child.excl.lock();
        cexcl.state = ProcState::RUNNABLE;
        drop(cexcl);

        Ok(cpid)
    }
```

文件系统模块中，日志Log结构体定义如下

```
/// 用于记录和管理文件系统日志的核心结构体。
///
/// `Log` 结构体实现了一个简化的事务性日志机制，模仿 xv6 中的 write-ahead log，
/// 以确保多步文件系统更新的原子性与崩溃恢复能力。
/// 它记录了日志区域的位置、事务状态以及当前事务中涉及的块号，
/// 并通过配套的操作函数（如 `commit`, `recover`, `write_head` 等）
/// 提供日志写入与回滚功能。
///
```

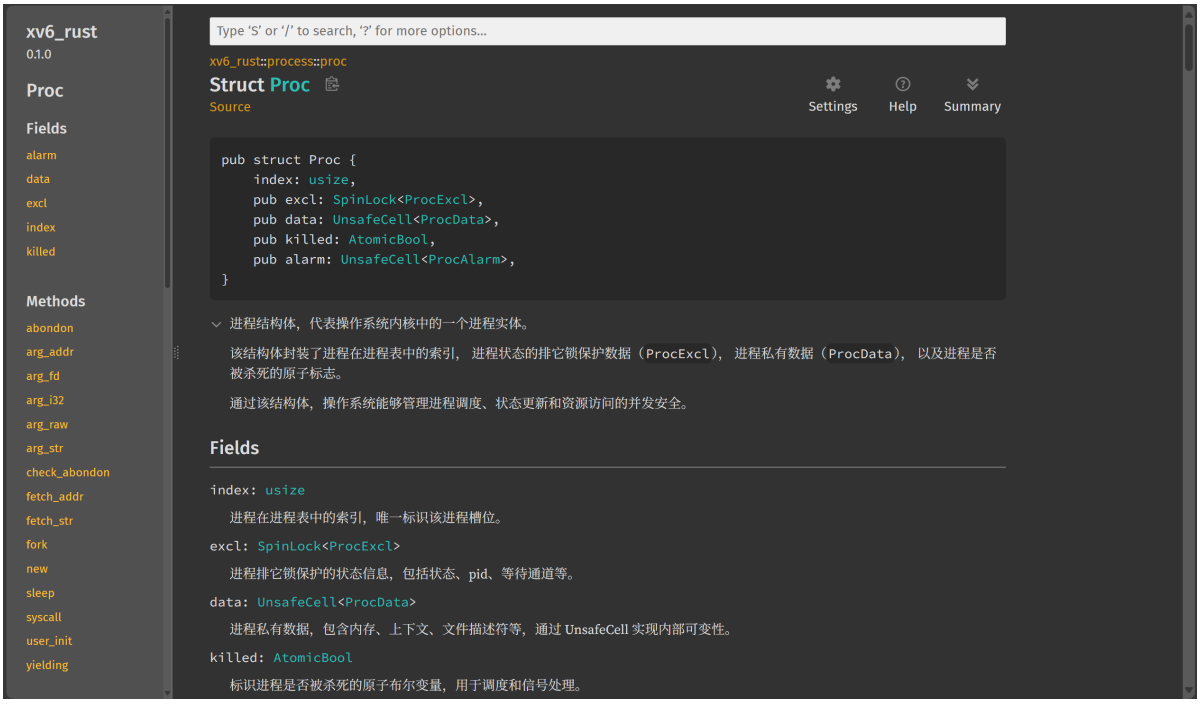
```rust
/// 该结构体由全局唯一的 [`LOG`] 实例持有，并封装在 [`SpinLock`] 中，
/// 确保在并发环境中操作的同步安全。
pub struct Log {
    /// 日志区在磁盘中的起始块号（由超级块中读取）
    start: u32,
    /// 日志区域中可用块的数量（包括日志头块和数据块）
    size: u32,
    /// 所在磁盘设备的编号
    dev: u32,
    /// 当前正在进行的文件系统操作数（事务嵌套层数）
    outstanding: u32,
    /// 指示日志系统是否正在提交事务，
    /// 为 true 时禁止新的文件系统操作进入
    committing: bool,
    /// 当前事务的日志头，记录了修改的块号及数量
    lh: LogHeader,
}
```

文件系统模块中，commit提交日志函数的定义如下：

```rust
    /// 提交日志，将日志中的数据正式写入文件系统。
    ///
    /// # 功能说明
    /// 此函数用于在文件系统操作完成后将本次事务对应的日志内容写入磁盘。
    /// 它代表事务的提交阶段，确保所有缓存在日志区的修改操作被复制到它们原本的磁盘位置。
    /// 提交过程结束后，还会清空日志头，表明日志区已可用于下一次事务。
    ///
    /// # 流程解释
    /// 1. 检查 `committing` 标志是否已设置，若未设置则触发 panic；
    /// 2. 若当前日志长度 `lh.len` 大于 0，执行以下步骤：
    ///     - 调用 [`write_log`]：将缓存中的原始数据块复制到日志块区域；
    ///     - 调用 [`write_head`]：将日志头写入磁盘，标志着事务正式提交；
    ///     - 调用 [`install_trans`]：将日志块中的内容写回到它们的原始位置；
    ///     - 调用 [`empty_head`]：清空日志头，表示日志区可复用。
    ///
    /// # 参数
    /// 无参数。操作对象为当前 `Log` 实例。
    ///
    /// # 返回值
    /// 无返回值。函数通过副作用修改磁盘内容和内部状态。
    ///
    /// # 可能的错误
    /// - 如果调用时未设置 `committing = true`，将触发 panic；
    /// - 若日志头损坏或越界可能导致复制错误；
    /// - 过程中调用的 I/O 函数（如 `bread`, `bwrite`）失败可能影响日志提交完整性；
    /// - 内部 `unsafe` 指针操作若不正确，可能引发未定义行为。
    ///
    /// # 安全性
    /// 本函数为 `unsafe`，调用者必须确保：
    /// - 当前已设置 `committing = true`，表示事务提交阶段；
    /// - 调用过程中未持有锁，以避免潜在的休眠导致死锁；
    /// - 调用路径控制良好，不允许并发或递归提交。
    pub unsafe fn commit(&mut self) {
        if !self.committing {
            panic!("log: committing while the committing flag is not set");
```

```
        }
        // debug_assert!(self.lh.len > 0);      // it should have some log to
commit
        if self.lh.len > 0 {
            self.write_log();
            self.write_head();
            self.install_trans(false);
            self.empty_head();
        }
    }
```

最终产生的HTML格式文档如下所示



Rust语言用户程序与用户线程库

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。在配合。通过用户程序以系统调用的方式与内核交互，可以有效地验证系统各项功能的正确性。
为进一步统一语言生态、提升教学一致性，已经使用 `Rust` 重写部分测试用例，并已实现初步成功运行。未来计划将全部用户程序迁移为 Rust 实现，构建一个全栈基于 Rust 的教学操作系统与测试平台。

用户程序库基础结构

由于在Seios操作系统基础上编写用户程序，无法直接使用Rust的标准库。
本用户程序库（在`no_std`环境下）结合Seios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。

rust用户程序

1. 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`extern "C" fn _start(argc: usize, argv: usize) -> !`。

```
// src/link.ld
```

```
OUTPUT_ARCH(riscv)
ENTRY(_start)

BASE_ADDRESS = 0x10000;

SECTIONS
{
    . = BASE_ADDRESS;
    .text : {
        *(.text.entry)
        *(.text .text.*)
    }
    . = ALIGN(4K);
    .rodata : {
        *(.rodata .rodata.*)
        *(.srodata .srodata.*)
    }
    . = ALIGN(4K);
    .data : {
        *(.data .data.*)
        *(.sdata .sdata.*)
    }
    .bss : {
        *(.bss .bss.*)
        *(.sbss .sbss.*)
    }
    /DISCARD/ : {
        *(.eh_frame)
        *(.debug*)
    }
}
```

1. `OUTPUT_ARCH(riscv)` 指定目标架构为RISC-V。
2. `ENTRY(_start)` 指定程序的入口点为_start符号（用户程序启动代码）。
3. `BASE_ADDRESS = 0x10000` 表示程序加载的基地址为0x10000。
4. `.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。

2. 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

```
use buddy_system_allocator::LockedHeap;

const USER_HEAP_SIZE: usize = 0x10000;

static mut HEAP_SPACE: [u8; USER_HEAP_SIZE] = [0; USER_HEAP_SIZE];

#[global_allocator]
static HEAP: LockedHeap = LockedHeap::empty();
```

`#[global_allocator]`

将此分配器注册为Rust的全局分配器，所有标准库类型（如Vec、Box）的内存分配会通过它进行。

后续可以结合 `sbrk` 系统调用，升级为内核中的堆内存分配

3. `_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）

```
#[no_mangle]
#[link_section = ".text.entry"]
pub extern "C" fn _start(argc: usize, argv: usize) -> ! {
    unsafe {
        HEAP.lock()
            .init(HEAP_SPACE.as_ptr() as usize, USER_HEAP_SIZE);
    }
    let mut v: Vec<&'static str> = Vec::new();
    for i in 0..argc{
        let str_start = unsafe {
            ((argv + i * core::mem::size_of::<usize>()) as * const
usize).read_volatile()
        };
        let len = (0usize..)
            .find(|i| unsafe{((str_start + *i) as *const u8).read_volatile()
== 0})
            .unwrap();
        v.push(
            core::str::from_utf8(
                unsafe {
                    core::slice::from_raw_parts(str_start as *const u8, len)
                }
            )
            .unwrap()
        );
    }
    exit(main(argc,v.as_slice()));
}
```

使用 `#[linkage = "weak"]` 进行默认 `main` 函数的设置，确保用户编写main函数

```
#[linkage = "weak"]
#[no_mangle]
fn main(_argc:usize, _argv:&[&str]) -> i32 {
    panic!("Cannot find main!");
}
```

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。

基础系统调用

1. syscall(riscv处理器)
   在RISCV处理器上的系统调用，此操作系统最多支持**6个**参数。
   通过rust语言内嵌汇编实现，使用`ecall`来触发内核陷入。

```rust
// src/syscall_riscv/src/lib.rs

/// the syscall on RISCV chips which supports 6 parameters
pub fn syscall(id: usize, args: [usize; 6]) -> isize {
    let mut ret: isize;
    unsafe {
        core::arch::asm!(
            "ecall",
            inlateout("x10") args[0] => ret,
            in("x11") args[1],
            in("x12") args[2],
            in("x13") args[3],
            in("x14") args[4],
            in("x15") args[5],
            in("x17") id
        );
    }
    ret
}
```

2. 系统调用详细信息
   系统调用的接口进行封装。

- [x] **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。
  参数：无。
  返回值：
  child_pid(>0)：父进程中返回子进程的 PID（Process ID），用于后续管理子进程
  0：子进程中返回 0，用于确认当前是子进程
  -1：错误则返回 -1。可能的错误原因是：复制过程中出现错误

- [x] **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。
  参数：
  status：退出状态码（通常 0 表示成功，非 0 表示错误）。
  返回值：无（进程终止，不会返回）。

- [x] **SYSCALL_WAIT(syscall_id = 3)**
  功能：等待任意子进程终止，并获取其退出状态。
  参数：
  addr：指向存储退出状态的指针（可为 NULL）。
  返回值：
  child_pid(>0)：成功返回终止的子进程 PID。
  -1：错误（如无子进程或信号中断）。

- [x] **SYSCALL_PIPE(syscall_id = 4)**
  功能：创建一个管道，用于进程间通信（IPC）。
  参数：pipefd[2]：数组，用于返回管道的读写端文件描述符（pipefd[0] 读，pipefd[1] 写）。
  返回值：
  0：成功。
  -1：错误。
- [x] **SYSCALL_READ(syscall_id = 5)**
  功能：从文件描述符读取数据。
  参数：
  fd：文件描述符。
  buf：存储数据的缓冲区指针。
  count：要读取的字节数。
  返回值：
  count(>=0)：实际读取的字节数。
  -1：错误。
- [x] **SYSCALL_KILL(syscall_id = 6)** (未实现sig：信号编号)
  功能：向指定进程发送信号。
  参数：
  pid：目标进程的 PID。
  返回值：
  0：成功。
  -1：错误。
- [x] **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。
  参数：
  path：可执行文件路径。
  argv：命令行参数数组（以 NULL 结尾）。
  返回值：
  N/A：成功不返回（进程被替换）
  -1：失败无法启动进程。
- [x] **SYSCALL_FSTAT(syscall_id = 8)**
  功能：获取文件描述符对应的文件状态信息。
  参数：
  fd：文件描述符。
  stat：指向 `struct stat` 的指针。
  返回值：
  0：成功。
  -1：错误。
- [x] **SYSCALL_CHDIR(syscall_id = 9)**
  功能：更改当前进程的工作目录。
  参数：
  path：目标目录路径。
  返回值：
  0：成功。
  -1：错误。

- [x] **SYSCALL_DUP(syscall_id = 10)**
  功能：复制文件描述符。
  参数：
    oldfd：原文件描述符。
  返回值：
    fd(≥0)：新的文件描述符。
    -1：错误。
- [x] **SYSCALL_GETPID(syscall_id = 11)**
  功能：获取当前进程的 PID。
  参数：无。
  返回值：
    pid(≥0)：当前进程的 PID。
- [x] **SYSCALL_SBRK(syscall_id = 12)**
  功能：调整进程的堆内存大小。
  参数：
    increment：内存增量（字节）。
  返回值：
    heap_addr：成功返回新的堆顶地址
    -1：错误（如内存不足）。
- [x] **SYSCALL_SLEEP(syscall_id = 13)**
  功能：使进程休眠指定时间。
  参数：
    ticks：休眠的时钟周期数。
  返回值：
    无
- [x] **SYSCALL_UPTIME(usize = 14)**
  功能：获取系统启动后的时钟周期数。
  参数：无。
  返回值：
    ticks：系统启动后运行时间时钟周期数。
- [x] **SYSCALL_OPEN(usize = 15)**
  功能：打开或创建文件。
  参数：
    path：文件路径。
    flags：打开模式（如 O_RDONLY）。
  返回值：
    fd(≥0)：文件描述符。
    -1：错误。
- [x] **SYSCALL_WRITE(usize = 16)**
  功能：向文件描述符写入数据。
  参数：
    fd：文件描述符。
    buf：数据缓冲区指针。
    count：要写入的字节数。
  返回值：
    count(≥0)：实际写入的字节数。
    -1：错误。

- [x] **SYSCALL_MKNOD(usize = 17)**
  功能：创建设备文件或特殊文件。
  参数：
  path：设备文件的路径。
  major：主设备号（标识设备类型）。
  minor：次设备号（标识具体设备实例）。
  返回值：
  0：成功。
  -1：错误。
- [x] **SYSCALL_UNLINK(usize = 18)**
  功能：删除文件链接。
  参数：
  path：文件路径。
  返回值：
  0：成功。
  -1：错误。
- [x] **SYSCALL_LINK:(usize = 19)**
  功能：创建硬链接。
  参数：
  oldpath：原文件路径。
  newpath：新链接路径。
  返回值：
  0：成功。
  -1：错误。
- [x] **SYSCALL_MKDIR(usize = 20)**
  功能：创建目录。
  参数：
  path：目录路径。
  返回值：
  0：成功。
  -1：错误。
- [x] **SYSCALL_CLOSE:(usize = 21)**
  功能：关闭文件描述符。
  参数：
  fd：文件描述符。
  返回值：
  0：成功。
  -1：错误。
- [x] **SYSCALL_GETMTIME(usize = 22)**
  功能：获取riscv处理器的计时器寄存器。
  参数：
  无。
  返回值：
  (mtime)≥0：时间戳。
  -1：错误。

错误处理

为用户程序实现的 `panic` 处理函数，使用rust提供的属性注释 `#[panic_handler]` 来定义当程序发生不可恢复错误（panic）时的处理行为。该标记该函数为全局panic处理器，替代标准库的默认实现。当程序触发 `panic!` 时，此函数会被调用。

该错误处理函数先输出错误发生的信息,再调用 `kill(getpid())` 终止当前进程（ `getpid()` 获取当前进程ID）。 `unreachable!()` 宏表示代码不会执行到这里，用于告诉编译器该分支不可达，帮助编译器进行优化
`PanicInfo`：包含panic的位置（文件、行号、列号）和错误消息。
``

对于用户程序的错误，简单输出

```
    #[panic_handler]
    fn panic_handler(panic_info: &core::panic::PanicInfo) -> ! {
        // 打印 panic 位置（文件 + 行号）
        if let Some(location) = panic_info.location() {
            println!(
                "Panicked at {}:{}:{}",
                location.file(),
                location.line(),
                location.column()
            );
        }
        // 打印 panic 消息（如果有）
        println!("Error: {}", panic_info.message());
        kill(getpid());
        unreachable!()
    }
```

用户线程库(在用户空间实现)

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。
并且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。

这里在用户创建的线程中的使用 `yield_task` 主动让出处理器资源和使用 `guard` 退出任务。
以下用户线程示例

```
let r_ptr = runtime.init();

    let args1 = MyType::new(12, "ych");
    let args2 = MyType::new(17, "kss");
    runtime.spawn(|r_ptr, args | {
        println!("TASK 1 STARTING");
        let id = 1;
        let arg =  args as *const MyType;

        let para = unsafe {*arg};
        wait_task(r_ptr);
        for i in 0..4 {
            println!("task: {} counter: {} arg:{}", id, i, para.str);
            yield_task(r_ptr);
```

```
        }
        println!("TASK 1 FINISHED");
        guard(r_ptr);
    },&args1 as *const MyType as u64);
    runtime.spawn(|r_ptr, args| {
        println!("TASK 2 STARTING");
        let id = 2;
        let arg =  args as *const MyType;

        let para = unsafe {*arg};

        for i in 0..8 {
            println!("task: {} counter: {} arg:{}", id, i, para.str);
            yield_task(r_ptr);
        }
        for i in 0..8 {
            println!("task: {} counter: {} arg:{}", id, i, para.str);
        }
        println!("TASK 2 FINISHED");
        signal_task(r_ptr, 1);
        guard(r_ptr);
    },&args2 as *const MyType as u64);
    runtime.run();
```

运行结果

```
TASK 0 (Runtime) STARTING
TASK 1 STARTING
TASK 2 STARTING
task: 2 counter: 0 arg:kss
task: 2 counter: 1 arg:kss
task: 2 counter: 2 arg:kss
task: 2 counter: 3 arg:kss
task: 2 counter: 4 arg:kss
task: 2 counter: 5 arg:kss
task: 2 counter: 6 arg:kss
task: 2 counter: 7 arg:kss
task: 2 counter: 0 arg:kss
task: 2 counter: 1 arg:kss
task: 2 counter: 2 arg:kss
task: 2 counter: 3 arg:kss
task: 2 counter: 4 arg:kss
task: 2 counter: 5 arg:kss
task: 2 counter: 6 arg:kss
task: 2 counter: 7 arg:kss
TASK 2 FINISHED
task: 1 counter: 0 arg:ych
task: 1 counter: 1 arg:ych
task: 1 counter: 2 arg:ych
task: 1 counter: 3 arg:ych
TASK 1 FINISHED
stackful_coroutine PASSED
```

1. 用户空间线程上下文结构

```rust
pub struct TaskContext {
    pub x1: u64,  //ra: return address
    pub x2: u64,  //sp(s0)
    pub x8: u64,  //fp
    pub x9: u64,  //s1
     //x18-27  通用寄存器
    pub x18: u64,
    pub x19: u64,
    pub x20: u64,
    pub x21: u64,
    pub x22: u64,
    pub x23: u64,
    pub x24: u64,
    pub x25: u64,
    pub x26: u64,
    pub x27: u64,
    pub nx1: u64,   // new return address
    pub r_ptr: u64  // self ptr 指向RUNTIME的指针
    pub params: u64 // 传入参数的 args 的地址
}
```

2. 用户空间线程状态
   用户空间线程目前已实现`Available`该任务槽位未被占用，空闲状态；
   `Sleep`任务处于挂起状态，等待信号；
   `Ready`任务处于可运行状态，等待调度器调度执行；
   `Running`当前任务正在 CPU 上运行。

```rust
pub enum TaskState {
    Available,
    Sleep,
    Ready,
    Running,
}
```

3. 用户空间线程结构
   用户态线程库的线程控制块 (Task Control Block) 结构体设计

```rust
pub struct Task {
    pub id: usize,          // 线程id
    pub stack: Vec<u8>,     // 线程栈
    pub ctx: TaskContext,   // 线程上下文
    pub state: TaskState,   // 线程状态
    pub r_ptr: u64          // 线程管理器指针
}
```

   `id`: 线程唯一标识符
   `stack`: 动态分配的线程栈
   `ctx`: 保存寄存器上下文的结构体
   `state`: 线程状态
   `r_ptr`: 线程管理器指针

4. 用户空间线程管理者
   用于用户线程的管理，切换线程

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

`tasks`：管理所有任务，Task 包含栈、上下文和状态。
`current`：标记当前运行的任务（索引）。
`waits`： 各任务需要等待其他任务的id（`wait[i]`的值为`0`表示没有需要等待的任务；`wait[i]`的值为`x`{x = 1,2,...,MAX_TASKS},表示需要等待任务号为x发出signal信号）；`wait[i]`的值为`usize::MAX`表示需要等待任意任务发出signal信号）。

5. 用户空间线程切换核心
用户态线程切换汇编，符合riscv架构指令二进制代码遵循严格的函数调用规范（Application Binary Interface, ABI），核心包括寄存器使用约定、参数传递规则、栈帧管理三部分。
通过switch的汇编保存需要被调用者保存(x8-x9, x18-x27)，x1(ra)存储返回地址寄存器和x2(sp)栈指针寄存器。

```
// src/thread/switch.S
.global switch
switch:
    sd x1, 0x00(a0)
    sd x2, 0x08(a0)
    sd x8, 0x10(a0)
    sd x9, 0x18(a0)
    sd x18, 0x20(a0)
    sd x19, 0x28(a0)
    sd x20, 0x30(a0)
    sd x21, 0x38(a0)
    sd x22, 0x40(a0)
    sd x23, 0x48(a0)
    sd x24, 0x50(a0)
    sd x25, 0x58(a0)
    sd x26, 0x60(a0)
    sd x27, 0x68(a0)
    sd x1, 0x70(a0)
    sd x11, 0x80(a0)

    ld x1, 0x00(a1)
    ld x2, 0x08(a1)
    ld x8, 0x10(a1)
    ld x9, 0x18(a1)
    ld x18, 0x20(a1)
    ld x19, 0x28(a1)
    ld x20, 0x30(a1)
    ld x21, 0x38(a1)
    ld x22, 0x40(a1)
    ld x23, 0x48(a1)
    ld x24, 0x50(a1)
    ld x25, 0x58(a1)
    ld x26, 0x60(a1)
    ld x27, 0x68(a1)
    ld t0, 0x70(a1)
    ld x10, 0x78(a1) #set r_ptr
```

```
    ld x11, 0x80(a1) #set param
    jr t0
```

了加载Runtime的地址，确保后续`guard`和`yield_task`进行任务管理。

```
    ld x10, 0x78(a1) #set r_ptr for the `guard` and the `yield_task`
```

加载任务的参数给自定义函数使用。

```
    ld x11, 0x80(a1)  #set args parameter
```

用户线程切换核心，使用`t_yield`进行任务调度

```
#[inline(never)]
fn t_yield(&mut self) -> bool {
    let mut pos = self.current;
    let mut temp = 0usize;
    while self.tasks[pos].state != TaskState::Ready {
        pos += 1;
        if pos == self.tasks.len() {
            pos = 1;
            if temp == 1 {
                pos = 0;
            }
            temp = 1;
        }
        if pos == 0 && pos == self.current {
            return false;
        }

    }

    if self.tasks[self.current].state != TaskState::Available {
        self.tasks[self.current].state = TaskState::Ready;
    }

    self.tasks[pos].state = TaskState::Running;
    let old_pos = self.current;
    self.current = pos;
    if old_pos == pos {
        return  self.tasks.len() > 0
    }

    unsafe {
        switch(&mut self.tasks[old_pos].ctx, &self.tasks[pos].ctx);
    }

    self.tasks.len() > 0
}
```

以上是任务切换过程

```
    let mut pos = self.current;
```

```
    let mut temp = 0usize;
    while self.tasks[pos].state != TaskState::Ready {
        pos += 1;
        if pos == self.tasks.len() {
            pos = 1;
            if temp == 1 {
                pos = 0;
            }
            temp = 1;
        }
        if pos == 0 && pos == self.current {
            return false;
        }
    }
```

依次在任务列表中寻找状态为`Ready`的任务。在对于Runtime的初始任务(id=0)的IDLE任务，切换时跳过，避免不必要的上下文切换开销。

```
    if old_pos == pos {
        return  self.tasks.len() > 0
    }
```

在发现新任务和旧任务的相同时，不切换，避免不必要的上下文切换开销。

6. 用户线程主动yield切换
用于给主动让出当前任务的执行权，切换到其他就绪任务。

```
// src/thread/mod.rs
pub fn yield_task(r_ptr: u64) {
    unsafe {
        let rt_ptr = r_ptr as *mut Runtime;
        (*rt_ptr).t_yield();
    };
}
```

7. 线程结束
用于线程退出处理，用户自定义函数运行完毕后，返回任务管理器，并且唤醒其他任务，将当前任务状态设置为`Available`。

```
// src/thread/mod.rs
pub fn guard(r_ptr: *const Runtime) {
    unsafe {
        let rt_ptr = r_ptr as *mut Runtime;
        (*rt_ptr).t_return();
    };
}
```

```rust
```

impl Runtime{
...
fn t_return(&mut self) {
for (index, value) in self.waits.iter_mut().enumerate() {

```
            if *value == usize::MAX || *value == self.current {
                *value = 0;
                self.tasks[index].state = TaskState::Ready;
            }
        }
        if self.current != 0 {
            self.tasks[self.current].state = TaskState::Available;
            self.t_yield();
        }
    }
    ...
}
```

8. 线程创建
   找到空闲任务，设置其栈和上下文。
   设置任务入口函数f和栈指针（需对齐）和任务参数。
   `fn spawn(&mut self, f: fn(*const Runtime, u64), params: u64) -> usize`需要传入一个自定义的`fn(*const Runtime, u64)`的函数指针和一个自定义参数的地址`u64`，返回值是分配的任务的`tid`。

```
// src/thread/mod.rs
pub fn spawn(&mut self, f: fn(*const Runtime, u64), params: u64) -> usize {
    let available = self
        .tasks
        .iter_mut()
        .find(|t| t.state == TaskState::Available)
        .expect("no available task.");

    let size = available.stack.len();
    unsafe {
        let s_ptr = available.stack.as_mut_ptr().offset(size as isize);

        let s_ptr = (s_ptr as usize & !7) as *mut u8;

        available.ctx.x1 = guard as u64; //ctx.x1  is old return address
        available.ctx.nx1 = f as u64; //ctx.nx1 is new return address
        available.ctx.x2 = s_ptr.offset(-32) as u64; //cxt.x2 is sp
        available.ctx.r_ptr = available.r_ptr;
        available.ctx.params = params;        // pointer to user's custom
parameter
    }
    available.state = TaskState::Ready;

    available.id
}
```

9. 等待其他任务
   由于任务有同步性的需求，实现等待其他任务完成的功能。
   `waittid_task`：等待指定的任务完成，需要指定任务号（id）；
   `wait_task`：等待任务其他任务完成。

```
pub fn waittid_task(r_ptr: *const Runtime, id: usize) {
    unsafe {
        let rt_ptr = r_ptr as *mut Runtime;
        (*rt_ptr).t_wait(id);
    };
}
pub fn wait_task(r_ptr: *const Runtime){
    unsafe {
        let rt_ptr = r_ptr as *mut Runtime;
        (*rt_ptr).t_wait(0);
    };
}
```

`t_wait`是具体实现，将任务的状态设置为`Sleep`挂起状态，将，并让出处理器资源，切换到其他任务。根据`id`参数确实当前任务是等待指定任务还是任意任务。

```
    fn t_wait(&mut self, id: usize) {
        if self.current != 0 {
            self.tasks[self.current].state = TaskState::Sleep;
            if id == 0 {
                self.waits[self.current] = usize::MAX;
            } else {
                self.waits[self.current] = id;
            }
            self.t_yield();
        }
    }
```

10. 唤醒指定任务

    通过`signal_task`来唤醒指定任务。

```
pub fn signal_task(r_ptr: *const Runtime, id: usize) {
    unsafe {
        let rt_ptr = r_ptr as *mut Runtime;
        (*rt_ptr).t_signal(id);
    };
}
```

```
    fn t_signal(&mut self, id: usize){
        if self.waits[id] == usize::MAX {
            self.tasks[id].state = TaskState::Ready;
            self.waits[id] = 0;
        }else if self.waits[id] == self.current {
            self.tasks[id].state = TaskState::Ready;
            self.waits[id] = 0;
        }
        self.t_yield();
    }
```

11. 获取当前任务id

```
pub fn gettid_task(r_ptr: *const Runtime)-> usize{
    unsafe {
        let rt_ptr = r_ptr as *mut Runtime;
        (*rt_ptr).t_gettid()
    }
}
```

用户线程库(结合内核线程实现)

由于SEIOS只实现了进程结构，没有线程，导致任务管理繁琐。线程将提供更高效地切换效率；提供更灵活的任务同步机制；由于创建线程不需要复制整个用户地址空间中的数据，一定程度减少内存的开销。SEIOS目前正在实现内核线程。这部分工作还在进行中。

1. 线程Context结构

```
#[repr(C)]
#[derive(Copy,Clone)]
pub struct TaskContext {
    ra: usize,
    sp: usize,

    // callee-saved
    s0: usize,
    s1: usize,
    s2: usize,
    s3: usize,
    s4: usize,
    s5: usize,
    s6: usize,
    s7: usize,
    s8: usize,
    s9: usize,
    s10: usize,
    s11: usize,
}
```

2. 线程号（pid）分配器
   简单的回收并重新分配pid。在`RecycleAllocator`中`recycled`为空时，根据`current`新分配一个pid；在`RecycleAllocator`中`recycled`不为空时，从`recycled`取出一个值进行分配。

```
use alloc::vec::Vec;

pub struct RecycleAllocator{
    current: usize,
    recycled: Vec<usize>
}

impl RecycleAllocator {
    pub fn new() -> Self {
        RecycleAllocator { current: 0, recycled: Vec::new() }
    }

    pub fn tid_alloc(&mut self) -> usize {
        if let Some(pid) = self.recycled.pop() {
```

```
            pid
        }
        else {
            self.current += 1;
            self.current - 1
        }
    }
    pub fn tid_dealloc(&mut self, tid: usize) {
        debug_assert!(tid < self.current);
        debug_assert!(
            !self.recycled.iter().any(|i| *i ==  tid),
            "pid {} has been deallocated!",
            tid
        );
        self.recycled.push(tid);
    }
}
```

3. 线程切换

线程切换的结构也是进行函数调用，修改部分寄存器。

通过switch的汇编保存需要被调用者保存(x8-x9, x18-x27)，x1(ra)存储返回地址寄存器和x2(sp)栈指针寄存器。

```
.altmacro
.macro SAVE_SN n
    sd s\n, (\n+2)*8(a0)
.endm
.macro LOAD_SN n
    ld s\n, (\n+2)*8(a1)
.endm
    .section .text
    .globl __switch
__switch:
    # __switch(
    #     current_task_cx_ptr: *mut TaskContext,
    #     next_task_cx_ptr: *const TaskContext
    # )
    # save kernel stack of current task
    sd sp, 8(a0)
    # save ra & s0~s11 of current execution
    sd ra, 0(a0)
    .set n, 0
    .rept 12
        SAVE_SN %n
        .set n, n + 1
    .endr
    # restore ra & s0~s11 of next execution
    ld ra, 0(a1)
    .set n, 0
    .rept 12
        LOAD_SN %n
        .set n, n + 1
    .endr
    # restore kernel stack of next task
    ld sp, 8(a1)
```

```
    ret
```

4. 线程结构

```
pub struct ProcessControlBlock {
    pub pid: usize,
    pub is_zombie: bool,
    pub page_table: PageTable,
    pub parent: Option<Weak<ProcessControlBlock>>,
    pub children: Vec<Arc<ProcessControlBlock>>,
    pub exit_code: i32,
    pub fd_table: [Option<Arc<File>>; NFILE],
    pub signals: SignalFlags,
    pub tasks: Vec<Option<Arc<TaskControlBlock>>>,
}
```

5. 关于内核线程的系统调用

```
const SYSCALL_THREAD_CREATE: usize;   //线程创建
const SYSCALL_GETTID: usize;          //获取线程号
const SYSCALL_WAITTID: usize;         //等待其他线程
```

自动化评测方案

测试用例

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。因此，我们成功地将 xv6 的用户程序 `usertests` 以及各实验对应的测试程序（如 `alarmtest`、`bttest` 等）移植到了该 Rust 实现的类 xv6 教学操作系统上。通过用户程序以系统调用的方式与内核交互，可以有效地验证系统各项功能的正确性。

具体而言，`usertests` 作为一个综合性测试程序，涵盖了约 60 个测试用例，广泛测试了进程管理、内存管理、文件系统、系统调用边界条件等基础功能，能够全面评估系统的核心模块是否实现完整。与此同时，针对每一项扩展实验所引入的新内核机制，我们也分别移植了对应的测试程序，如用于验证定时中断与处理机制的 `alarmtest`，确保新增功能的实现具备明确的验证手段。

这种双层次的测试体系具备显著优势：一方面，`usertests` 可以作为回归测试手段，判断学生是否在实现新功能时引入了对原有内核行为的破坏；另一方面，各实验专用测试程序则能精准评估学生实现的功能是否符合实验要求，形成互为补充、相互验证的测试闭环。通过这种方式，既保障了教学实验的可控性与规范性，也提升了项目整体的健壮性与可维护性。

当前，这些测试程序仍主要以 C 语言实现。为进一步统一语言生态、提升教学一致性，我们已开始尝试以 Rust 重写部分测试用例，并已实现初步成功运行。未来计划将全部用户程序迁移为 Rust 实现，构建一个全栈基于 Rust 的教学操作系统与测试平台。

下面简单介绍移植自xv6的60个测试用例：

```
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
void writetest(char *s);      // 测试创建文件并多次写入后能够完整读取出所有数据
void writebig(char *s);       // 测试向文件写入 MAXFILE 个块的数据并正确读取验证文件大小
void createtest(char *s);     // 测试创建大量小文件后再将它们全部删除的行为
void dirtest(char *s);        // 测试创建目录、进入目录然后退出并删除目录的基本操作
void exectest(char *s);       // 测试 exec 调用继承文件描述符及输出重定向是否正常工作
void pipe1(char *s);          // 测试父子进程通过管道传输数据的读写是否正确
void killstatus(char *s);     // 测试使用 wait 获取被 kill 信号终止的子进程退出状态是否为-1
void preempt(char *s);        // 测试多进程并发运行下的抢占调度以及 kill/wait 同步行为
void exitwait(char *s);       // 测试频繁 fork/exit/wait 场景下退出状态和值是否匹配
void reparent(char *s);       // 测试父进程退出时孤儿进程能否正确重新由 init 进程收养
void twochildren(char *s);    // 测试两个子进程几乎同时退出时父进程 wait 的正确性
void forkfork(char *s);       // 测试并发进行多级 fork 操作是否引发锁竞争问题
void forkforkfork(char *s);   // 测试进程反复 fork 直到资源耗尽时系统能否正常处理
void reparent2(char *s);      // 测试大量进程退出与重新收养子进程过程中是否出现死锁
void mem(char *s);            // 测试在大量内存分配和释放后能否成功再次分配内存
void sharedfd(char *s);       // 测试父子进程共享同一文件描述符写入时文件偏移和锁同步是否正常
void fourfiles(char *s);      // 测试多进程同时写入不同文件时的块分配和数据完整性
void createdelete(char *s);   // 测试多进程并发在同一目录创建和删除文件的行为
void unlinkread(char *s);     // 测试文件被删除后已打开的文件描述符仍可正常读写数据
void linktest(char *s);       // 测试创建硬链接、删除源文件以及各种非法链接操作的处理
void concreate(char *s);      // 测试多进程同时对同一文件执行创建、链接和删除操作的稳定性
void linkunlink(char *s);     // 测试父子进程并发执行文件创建、链接和删除是否产生死锁
void bigdir(char *s);         // 测试目录包含大量条目（使用间接块）时创建和删除链接的正确性

void subdir(char *s);         // 测试嵌套目录的创建、删除以及通过 “..” 返回父目录等操作是否正确
void bigwrite(char *s);       // 测试执行超出日志容量的大规模写入操作时文件系统的表现
void manywrites(char *s);     // 测试多进程同时向文件执行写入以检测磁盘驱动的死锁问题
void bigfile(char *s);        // 测试写入大文件并以分段方式读取验证数据完整性
void fourteen(char *s);       // 测试目录和文件名长度达到 DIRSIZ (14 字节) 限制时的创建行为
void rmdot(char *s);          // 测试删除当前目录 “.” 和父目录 “..” 项是否被禁止
void dirfile(char *s);        // 测试将普通文件用作目录时诸如 chdir、创建子文件等操作的错误处理
void iref(char *s);           // 测试多次创建进入目录并使用空文件名操作时 inode 引用计数是否正确释放
void forktest(char *s);       // 测试 fork 在进程/内存耗尽时能否优雅失败并正确回收资源
void sbrkbasic(char *s);      // 测试 sbrk 请求过多内存时返回正确失败值及小增量分配的正确性

void sbrkmuch(char *s);       // 测试将地址空间扩展到很大并能正确释放和重新分配内存页
void kernmem(char *s);        // 测试用户进程尝试读取内核内存时内核能否将其正确终止
void sbrkfail(char *s);       // 测试耗尽内存后失败的分配是否释放资源以及后续分配能否成功
void sbrkarg(char *s);        // 测试通过 sbrk 分配的内存可用于文件读写和管道操作
void validatetest(char *s);   // 测试系统调用对超出有效内存范围的字符串参数的验证
void bsstest(char *s);        // 测试未初始化的全局 BSS 段数据是否默认清零
```

```
void bigargtest(char *s);     // 测试 exec 的参数总大小超过一页时能否安全失败且不破坏内存

void fsfull();                // 测试将文件系统写满时的行为（可能触发内核 panic，仅用于验证）
void argptest(char *s);       // 测试 read 系统调用传入非法缓冲区地址和长度时的错误处理
void stacktest(char *s);      // 测试用户栈下方是否存在未映射的保护页以捕获栈溢出
void pgbug(char *s);          // 测试向 exec 和 pipe 传入异常指针时内核是否安全处理（验证指针转换漏洞）
void sbrkbugs(char *s);       // 测试进程将内存缩小到零或非常小等边界情况时是否触发内核错误

void badwrite(char *s);       // 测试使用非法用户缓冲区写文件在删除文件后是否有磁盘块遗留
void badarg(char *s);         // 测试 exec 调用包含非法参数指针时内核能否避免崩溃
void execout(char *s);        // 测试在内存耗尽的情况下执行 exec 是否能安全失败（无内核崩溃）
```

测试脚本

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。

测试脚本的实现分为两部分，所有实验通用的测试库`gradelib.py`与每个实验单独设计的测试点`grade-lab-*`，我们沿用了xv6中的通用测试库，并针对Rust语言环境修改了每个实验单独的测试点文件，使得测试程序能够正确运行。整套测试脚本的技术细节如下：

测试框架使用 `@test(points, title, parent)` 装饰器来注册测试用例。支持设置父子依赖关系：只有当父测试通过时，子测试才会被执行。每个测试都会记录执行状态、输出结果及得分情况，通过包装函数 `run_test()` 管理依赖执行、异常捕获、结果打印与得分统计，确保测试顺序合理、输出清晰。测试通过 `Runner` 类封装 QEMU 启动流程，自动执行 `make qemu-gdb` 以构建调试版内核并启动 QEMU，随后通过 GDB 客户端连接调试端口。测试期间，Runner 注册监视器、执行模拟器并监听输出事件；当检测到条件满足或达到超时，便安全终止模拟器并释放资源，整个流程实现“编译-运行-监控-终止”的自动闭环。

测试中的监视器用于驱动交互和监控行为，如：`save()` 保存输出日志、`stop_breakpoint()` 设置断点、`stop_on_line()` 在输出匹配时中止测试、`shell_script()` 自动执行命令脚本。监视器可组合使用，实现灵活的输入驱动与行为捕捉，是实现自动化测试的关键机制。测试过程中，QEMU 所有输出都会被实时捕获并保存在 `runner.qemu.output` 中。通过 `r.match()` 或 `assert_lines_match()` 可断言输出是否符合预期，支持检查特定行出现与否，或确保禁止某些行存在。断言失败会输出格式化的差异对比，帮助快速定位问题。此机制是验证内核行为正确性的核心手段。

每个测试用例执行后，系统会更新已获得分数 `TOTAL` 和总分值 `POSSIBLE`。测试通过时得分累加，失败则只统计总分。最终 `run_tests()` 打印总成绩，若有失分则返回非零退出码。通过 `end_part()` 可实现阶段性计分。整体机制确保得分过程透明、结果明确。

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。若正确则通过该测试点，具体的实验解释详见下一章。

```
@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)
```

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
```

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$'
== Test   alarmtest: test1 ==
  alarmtest: test1: 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 '^\.?test1 passed$'
== Test   alarmtest: test2 ==
  alarmtest: test2: 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 '^\.?test2 passed$'
== Test usertests == usertests: OK (279.4s)
Score: 19/79
make: *** [Makefile:129: grade] Error 1
```

实验指导手册与参考实现

目前已完成11个实验的移植、修改与实现，分别是：

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。

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。

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。

实验指导手册

实验零：实验平台操作说明

本实验指导手册基于Rust语言开发的类xv6架构操作系统内核编写。下面介绍实验平台以及系统内核的操作方法，请根据说明进行模仿，确保执行的结果与说明一致。

实验仓库操作方法

根据实验内容的不同，多个实验被分散到不同的仓库分支下，实验序号与分支名称的对应关系如下：

实验序号	仓库分支名称
1，2，3，4	lab-util
5，6	lab-syscall
7，8	lab-pagetable
9，10	lab-trap
11	lab-lazyalloc
12	lab-schedule

在执行某一序号的实验时，首先执行`git checkout 仓库分支名称`，切换到对应的分支后再进行实验。

实验平台操作方法

实验平台的操作通过Makefile实现，下面介绍本实验平台的Makefile允许执行的操作：

`make qemu`

- 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
- 本指令不会检测到内核代码的修改，对内核代码进行修改后，请先执行`make clean`

正确执行该指令后应该看到如下输出

```
qemu-system-riscv64 -machine virt -bios none -kernel target/riscv64gc-unknown-
none-elf/debug/xv6-rust -m 3G -smp 3 -nographic -drive
file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-
device,drive=x0,bus=virtio-mmio-bus.0

xv6-rust is booting

KernelHeap: available physical memory [0x80054028, 0x88000000)
  buddy system: useful memory is 0x7fab000 bytes
  buddy system: leaf size is 16 bytes
  buddy system: free lists have 24 different sizes
  buddy system: alloc 0x300490 bytes meta data
  buddy system: 0x55000 bytes unavailable
KernelHeap: init memory done
hart 2 starting
hart 1 starting
```

```
file system: checking logs
file system: no need to recover
file system: setup done
init: starting sh
$
```

`make qemu-gdb`

- 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

QEMU启动后输出如下

```
*** Now run 'gdb' in another window.
qemu-system-riscv64 -machine virt -bios none -kernel target/riscv64gc-unknown-
none-elf/debug/xv6-rust -m 3G -smp 3 -nographic -drive
file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-
device,drive=x0,bus=virtio-mmio-bus.0 -S -gdb tcp::26000
```

- 在第二个终端中执行`gdb-multiarch`（或任何可用的gdb），待gdb启动后执行`source .gdbinit`

此时GDB窗口输出如下

```
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04.2) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
...
(gdb) source .gdbinit
The target architecture is set to "riscv:rv64".
warning: No executable has been specified and target does not support
determining executable automatically.  Try using the "file" command.
0x0000000000001000 in ?? ()
(gdb)
```

至此，QEMU与GDB连接成功，可以在GDB中执行调试指令以运行内核，具体调试办法详见**内核调试方法**

`make asm`

- 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

以系统启动位置的汇编代码为例，该文件结构如下

```
0000000080000000 <_entry>:
    80000000:   0002c117            auipc   sp,0x2c
    80000004:   00010113            mv      sp,sp
    80000008:   6509                lui     a0,0x2
    8000000a:   f14025f3            csrr    a1,mhartid
    8000000e:   0585                addi    a1,a1,1
    80000010:   02b50533            mul     a0,a0,a1
    80000014:   912a                add     sp,sp,a0
    80000016:   00001097            auipc   ra,0x1
    8000001a:   114080e7            jalr    276(ra) # 8000112a <start>
```

`make grade`

- 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
- 由于评分脚本也依赖QEMU与GDB的连接机制，请在执行评分前关闭所有已启动的QEMU
- 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

一次正常的评分结果如下

```
== Test sleep, no arguments == sleep, no arguments: OK (8.6s)
== Test sleep, returns == sleep, returns: OK (1.2s)
== Test pingpong == pingpong: OK (1.4s)
== Test primes == primes: OK (1.4s)
== Test find, in current directory == find, in current directory: OK (1.5s)
== Test find, recursive == find, recursive: OK (2.4s)
== Test xargs == xargs: OK (2.6s)
Score: 100/100
```

`make clean`

- 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

运行后将执行如下指令

```
rm -rf kernel.S
cargo clean
rm -f user/*.o user/*.d user/*.asm user/*.sym \
user/initcode user/initcode.out fs.img \
mkfs/mkfs .gdbinit xv6.out \
user/usys.S \
user/_cat user/_echo user/_forktest user/_grep user/_init user/_kill user/_ln
user/_ls user/_mkdir user/_rm user/_sh user/_stressfs user/_usertests user/_grind
user/_wc user/_zombie user/_sleep user/_pingpong user/_primes user/_find
user/_xargs
```

系统内核操作方法

系统内核支持两个快捷键

- `Ctrl+P`：输出内核中正在运行的进程列表
- `Ctrl+A+X`：停止内核运行，关闭QEMU

在执行`make qemu`后，系统内核会启动，并自动执行shell用户程序，该程序输出一个dollar符号并等待用户选择用户程序进行执行。

shell 是一个非常简洁的用户态程序，它为操作系统提供基本的用户交互接口。在内核中，它的源代码位于 `user/sh.c` 文件，是用户运行的第一个可交互程序。

命令解析与执行

- Shell 使用 `fork + exec` 模式执行用户输入的命令。
- 支持内建命令（如 `cd`）和外部程序（如 `ls`, `cat`, `sh`, `echo`）。
- 输入示例：

```
$ echo hello
$ ls /bin
```

- 对于外部程序，shell 会：
  1. `fork` 创建子进程；
  2. 在子进程中调用 `exec` 替换为对应的用户程序；
  3. 父进程 `wait` 直到子进程结束。

I/O 重定向（< 和 >）

- 允许将输入或输出重定向到文件。
- 示例：

```
$ echo hello > out.txt
$ cat < out.txt
```

- 实现方式：
  - shell 在解析命令时检测 `<` 或 `>`；
  - 使用 `open()` 打开对应文件；
  - 使用 `dup()` 将标准输入（fd=0）或标准输出（fd=1）重定向到该文件描述符。

管道支持（|）

- 支持使用 `|` 连接多个命令的输出与输入。
- 示例：

```
$ ls | grep foo
```

- 实现方式：
  - shell 创建管道（`pipe(fd)`）；
  - fork 两个子进程，一个写管道，一个读管道；

- 使用 `dup()` 将 stdout/stdin 重定向为管道端点。

内建命令：`cd`

- xv6 shell 支持内建的 `cd` 命令改变当前工作目录。
- 示例：

```
$ cd /bin
```

- 特殊性：
  - `cd` 不能用 `exec` 实现，因为目录更改必须在当前进程（shell）中生效。
  - 因此，shell 检测到 `cd` 后直接调用 `chdir()` 而不 fork。

系统内核调试方法

常用 GDB 调试命令

1. 显示源码布局

```
layout src
```

- 显示源代码窗口。
- 按 `Ctrl+L` 可刷新界面。
- 使用 `layout split` 可同时显示代码和汇编。

---

2. 设置断点

```
b rust_main
b usertrap
b trap.rs:42
```

- 在指定函数或文件行号设置断点。
- 可以使用 `info break` 查看当前所有断点。
- 使用 `delete` 删除断点。

---

3. 运行与继续执行

```
c         # continue，继续运行直到下一个断点
```

- 如果你设置了断点并执行 `continue`，GDB 会在命中断点时停止。

---

4. 单步调试

```
s         # step：单步进入函数
n         # next：单步执行，不进入函数
fin       # finish：运行直到当前函数返回
```

- `s`（step）适合函数内部逐行观察。
- `n`（next）在调用函数时会跳过函数体。
- `fin`（finish）会跑到当前函数结束，常用于快速跳出。

---

5. 变量与表达式查看

```
p var_name              # 打印变量值
p/x 0x80000000          # 以十六进制打印值
p *(int*)0x80000000     # 以 C 样式解引用地址
```

- `x/4x`：查看内存，十六进制模式。
- `x/s`：查看字符串内容。
- `display`：持续显示变量。
- `set var foo = 3`：更改变量值。

---

6. 查看调用栈与函数

```
bt       # backtrace：查看调用栈
frame 1  # 切换到栈帧 1
info registers
```

- `bt` 是诊断内核 panic 和 trap 问题的重要工具。
- 可以在每一层 frame 中使用 `list`、`p` 查看局部变量。

---

7. 内核结构调试示例

```
p *myproc        # 查看当前进程结构体
p myproc->trapframe
p myproc->pagetable
```

如果你在 `proc.rs` 等处设置断点并想调试当前 `Proc` 的状态，这种方式非常有用。

---

8. 查看内存

```
x/16x 0x80000000      # 查看从物理地址 0x80000000 开始的 16 个字
x/4i $pc              # 查看当前 PC 附近的指令
```

实验一：进行系统调用

实验要求

为内核实现 UNIX 程序 `sleep`；你的 `sleep` 程序应当根据用户指定的时间单位 **tick** 暂停执行。**tick** 是由 xv6 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。你的代码应写在文件 `user/sleep.c` 中。**同时请尝试使用Rust语言实现相同的功能，并将代码写在 `user_rust/src/bin/sleep_rust.rs` 中。**

实验提示

- 查看 `user/` 目录中的其他程序（例如 `user/echo.c`、`user/grep.c` 和 `user/rm.c`），了解如何获取传递给程序的命令行参数。
- 如果用户忘记传入参数，`sleep` 程序应当打印一条错误信息。
- 命令行参数是以字符串形式传入的；你可以使用 `atoi`（见 `user/ulib.c`）将其转换为整数。
- 使用系统调用 `sleep`。
- 确保你的 `main` 函数调用了 `exit()` 以正确退出程序。
- 将你的 `sleep` 程序添加到 Makefile 中的 `UPROGS` 列表中；完成后，运行 `make qemu` 将编译你的程序，你就可以在 xv6 的 shell 中运行它了。
- Rust版本的用户程序只需要执行`sleep`调用即可，无需传入参数

实验现象

```
init: starting sh
$ sleep 10
（输出下一个提示符前停顿一会）
$
```

实验二：打乒乓

实验要求

编写一个程序，使用 UNIX 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（每个方向一个管道）。父进程应向子进程发送一个字节；子进程应打印 `"<pid>: received ping"`（其中 `<pid>` 是子进程的进程 ID），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 `"<pid>: received pong"`，然后退出。你的实现应保存在文件 `user/pingpong.c` 中。**同时请尝试使用Rust语言实现相同的功能，并将代码写在`user_rust/src/bin/pingpong_rust.rs`中。**

实验提示

- 使用 `pipe` 创建一个管道。
- 使用 `fork` 创建一个子进程。
- 使用 `read` 从管道中读取数据，使用 `write` 向管道中写入数据。
- 使用 `getpid` 获取调用进程的进程 ID。
- 将该程序添加到 Makefile 中的 `UPROGS` 列表中。
- xv6 上的用户程序只能使用有限的一组库函数。你可以在 `user/user.h` 中查看这些函数的列表；它们的源代码（系统调用除外）位于 `user/ulib.c`、`user/printf.c` 和 `user/umalloc.c` 中。
- Rust版本的实现应当与C语言版本功能保持一致

实验现象

```
init: starting sh
$ pingpong
3: received ping
2: received pong
$ pingpong_rust
5: received ping
4: received pong
$
```

实验三：并发素数筛

实验要求

编写一个使用管道（pipes）的并发版素数筛选程序。这个想法来自 Unix 管道的发明者 Doug McIlroy。这个网页Bell Labs and CSP Threads的图片和相关文字解释了如何实现它。你的C语言版本实现应写在文件 `user/primes.c` 中。**同时尝试使用Rust语言实现相同的功能，写在 `user_rust/src/bin/primes_rust.rs` 中。**

你的目标是使用 `pipe` 和 `fork` 来构建一个流水线（pipeline）。第一个进程将数字 2 到 35 依次写入这条流水线。对于每一个素数，你需要安排一个进程：它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。由于内核对文件描述符和进程数量有限制，第一个进程可以在数字 35 时停止。

实验提示

- 要小心关闭进程中不需要的文件描述符，否则在第一个进程到达数字 35 之前，你的程序可能会因为资源耗尽而导致内核无法运行。
- 一旦第一个进程处理到数字 35，它应当等待整个流水线中的所有进程终止，包括所有的子进程、孙进程等。因此，主进程 `primes` 只有在所有输出完成并且所有其他 `primes` 进程都退出之后才能退出。
- 提示：当管道的写端被关闭时，`read` 会返回 0。
- 最简单的做法是直接将 32 位（4 字节）的整数写入管道，而不是使用格式化的 ASCII 输入/输出。
- 只有在需要时，才创建流水线中的新进程。
- 在 `Makefile` 的 `UPROGS` 中添加该程序。

实验效果

```
init: starting sh
$ primes
prime 2
prime 3
prime 5
prime 7
prime 11
prime 13
prime 17
prime 19
prime 23
prime 29
prime 31
$ primes_rust
prime 2
```

```
prime 3
prime 5
prime 7
prime 11
prime 13
prime 17
prime 19
prime 23
prime 29
prime 31
$
```

实验四：文件查找

实验内容

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。本实验只需要使用C语言实现，应写在文件 `user/find.c` 中。

实验提示

- 查看 `user/ls.c` 了解如何读取目录内容。
- 使用递归，使 `find` 能够深入子目录进行查找。
- 不要递归进入 `"."` 和 `".."` 目录。
- 文件系统的更改会在多次运行 QEMU 时保留；如果你想要一个干净的文件系统，可以运行 `make clean` 然后再运行 `make qemu`。
- 你需要使用 C 风格字符串。可以参考《K&R C 语言程序设计》一书，例如第 5.5 节。
- 注意：在 C 语言中，不能使用 `==` 来比较字符串（不像 Python 那样）；应使用 `strcmp()` 函数来进行字符串比较。
- 在 `Makefile` 的 `UPROGS` 中添加该程序。

实验效果

```
init: starting sh
$ echo > b
$ mkdir a
$ echo > a/b
$ find . b
./b
./a/b
$
```

实验五：系统调用跟踪

实验要求

在本次实验中，你将为内核添加一个 **系统调用跟踪功能**，这个功能在你调试后续实验时将会非常有帮助。

你需要实现一个新的系统调用：`trace`。该系统调用用于控制追踪行为，它接受一个参数：一个整数类型的“**掩码（mask）**”。这个掩码的每一位代表一个系统调用，若某一位被设置为 1，则表示需要追踪对应编号的系统调用。

例如，若希望追踪 `fork` 系统调用，一个程序可以调用：

```
trace(1 << SYS_fork);
```

其中，`SYS_fork` 是 `include/syscall.h` 中定义的系统调用编号。

你需要对内核进行修改，使其在每个系统调用即将返回结果时，判断是否需要输出追踪信息。若该系统调用的编号在当前掩码中被设置，则输出一行追踪信息。追踪信息包括：

- 当前进程的 PID（进程号）
- 系统调用的名称
- 返回值

注意：你不需要打印系统调用的参数。

此外，`trace` 系统调用只对当前调用它的进程及该进程随后通过 `fork` 创建的子进程生效，对其他进程没有影响。

实验现象

我们提供了一个名为 `trace` 的用户态程序，用于在启用追踪功能的情况下运行另一个程序（见 `user/trace.c`）。完成实验后，你应该会看到类似如下的输出结果：

```
/* 示例1 */
$ trace 32 grep hello README
2: syscall read -> 1023
2: syscall read -> 966
2: syscall read -> 70
2: syscall read -> 0
/* 示例2 */
$ trace 2147483647 grep hello README
3: syscall trace -> 0
3: syscall exec -> 3
3: syscall open -> 3
3: syscall read -> 1023
3: syscall read -> 966
3: syscall read -> 70
3: syscall read -> 0
3: syscall close -> 0
/* 示例3 */
$ grep hello README
/* 示例4 */
$ trace 2 usertests forkforkfork
usertests starting
5: syscall fork -> 6
test forkforkfork: 5: syscall fork -> 7
7: syscall fork -> 8
8: syscall fork -> 9
8: syscall fork -> 10
9: syscall fork -> 11
```

```
8: syscall fork -> 12
10: syscall fork -> 13
......
......
OK
5: syscall fork -> 71
ALL TESTS PASSED
```

在上面的几个示例中，trace 的行为如下：

1. **第一个示例中**，trace 仅追踪了 `read` 系统调用。数字 32 是 `1 << SYS_read` 的结果，表示只追踪编号为 `SYS_read` 的系统调用。
2. **第二个示例中**，trace 运行 grep 时追踪了所有的系统调用。数值 `2147583647` 的二进制形式是低 31 位全为 1，表示开启所有系统调用的追踪。
3. **第三个示例中**，程序未启用追踪功能，因此不会有任何追踪输出。
4. **第四个示例中**，运行 usertests 中的 `forkforkfork` 测试，追踪的是所有子进程中的 `fork` 系统调用。

只要你的程序运行行为与上述描述一致（尽管具体的进程 ID 可能会有所不同），就说明你的实现是正确的。

实现提示

1. **在 Makefile 中添加可执行目标**
   在 `Makefile` 中的 `UPROGS` 列表中添加一项：`$(USER)/_trace\`，用于编译用户态的 `trace` 程序。
2. **构建系统调用接口原型**
   运行 `make qemu` 后你会发现系统无法编译 `user/trace.c`，因为你还没有为新的系统调用添加用户态的接口。这部分工作包括：
   - 在 `user/user.h` 中添加 `trace` 的函数声明；
   - 在 `user/usys.pl` 中添加 `trace` 的名字，用于自动生成汇编包装代码；
   - 在 `include/syscall.h` 中为 `trace` 分配一个新的系统调用编号。

   `Makefile` 会调用 `user/usys.pl` 脚本，它会生成 `user/usys.S` 文件，这个文件包含系统调用的用户态包装函数，它们会通过 RISC-V 的 `ecall` 指令切换到内核态。
3. **实现内核中的系统调用处理逻辑**
   修复用户程序编译后，再次运行 `trace 32 grep hello README`，你会发现程序仍然报错，因为此时 `trace` 系统调用尚未在内核中实现。
   - 在 `src/process/proc/syscall.rs` 中添加一个新的 `sys_trace()` 函数；
   - 该函数需要将传入的 `mask` 参数存储到当前进程的 `proc` 结构体中（见 `src/process/proc/mod.rs`），用于后续判断是否追踪；
   - 获取系统调用参数可以调用 `src/process/proc/syscall.rs` 中已有的参数提取函数，具体用法可以参考 `syscall.rs` 中的其他系统调用实现。
4. **在进程派生时复制追踪设置**
   修改 `src/process/proc/mod.rs` 中的 `fork()` 实现，使得父进程的追踪掩码能正确地复制给子进程。

5. **在 syscall 分发函数中实现打印逻辑**
   修改 `src/process/proc/syscall.rs` 中的 `syscall()` 函数，实现在系统调用返回前判断是否启用了追踪，如果是，则打印追踪信息。你需要：
   - 添加一个系统调用名称字符串数组，方便通过系统调用编号进行查找；
   - 在正确的位置打印进程 ID、系统调用名称和返回值等信息。

实验六：加速系统调用

实验要求

在某些操作系统（如 Linux）中，为了提升系统调用的性能，会通过在用户态与内核之间共享只读内存区域的方式来加速特定系统调用。这样一来，在执行这些系统调用时，就无需频繁陷入内核态（kernel crossing），从而减少开销。你的任务是：为 `getpid()` 系统调用实现这种优化机制。

- 在每个新进程创建时，在其地址空间中映射一个只读页面，该页面的虚拟地址为 `USYSCALL`（定义在 `memlayout.h` 中）。
- 在该页面的起始位置，放置一个名为 `struct usyscall` 的结构体（同样定义在 `memlayout.h` 中），并初始化其中的字段，使其保存当前进程的 PID。
- 用户态已经提供了 `ugetpid()` 函数，它会自动从 `USYSCALL` 映射读取数据。因此，无需你修改用户态的相关代码。
- 如果在运行 `pgtbltest` 时，`ugetpid` 的测试用例能够通过，你将获得本部分实验的满分。

实验提示

- 你可以在 `src/process/proc/mod.rs` 文件中的 `alloc_proc_pagetable()` 函数中完成页面映射操作。在该函数中，构建的是进程的页表结构，因此此处是插入只读共享页面映射的合适位置。
- 请设置合适的权限位，确保用户态只能读取该页面。由于该页面仅用于暴露进程的只读信息（如 PID），必须严格禁止写权限，防止用户空间修改其内容。
- 你可能会发现 `map_pages()` 函数很有用。该函数用于在页表中插入一段虚拟地址到物理地址的映射，并设置权限，是实现该功能的关键工具函数。
- 不要忘了在 `alloc_proc()` 中分配并初始化这个页面。`alloc_proc()` 是创建新进程时进行资源初始化的地方，你需要在这里为 `usyscall` 结构体分配一页物理内存，并将当前进程的 PID 写入结构体中。
- 别忘了在 `cleanup()` 中释放该页面。当进程被销毁时，其使用的资源需要被回收，包括你在 `alloc_proc()` 中分配的只读共享页面，避免内存泄漏。

实验七：打印页表

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**。

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。

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

实验八：检测某一页是否被访问过

实验内容

一些垃圾回收器（即自动内存管理的一种形式）可以通过知道哪些页面在运行中被访问过（包括读或写）来优化其性能。在本实验的这一部分中，你需要为内核添加一个新功能，利用 RISC-V 页表中的访问标志（access bit），检测用户空间中哪些页面被访问过，并将这些信息报告给用户程序。在 RISC-V 架构下，每当处理器遇到一次 TLB 未命中（即缺页）时，硬件的页表遍历器（page walker）会将相应的访问位（Accessed bit）标记在页表项（PTE）中。

你的任务是实现一个新的系统调用 `pgaccess()`，它能够返回哪些页面已经被访问过。这个系统调用需要三个参数：

1. 要检测的第一个用户页的起始虚拟地址；
2. 要检测的页面数量；
3. 一个用户地址，指向结果存储缓冲区，该缓冲区以位掩码（bitmask）的形式表示每页的访问情况（每页用一位表示，第一页对应最低有效位）。

当你实现的 `pgaccess()` 成功通过 `pgtbltest` 测试程序中的 `pgaccess` 测试用例时，即可获得本实验部分的满分。

实验提示

- 从实现 `sys_pgaccess()` 函数入手，放在 `process/proc/syscall.rs` 中。
- 你需要使用 `arg_addr()` 和 `arg_i32()` 来解析系统调用的参数（分别用于获取用户传入的地址和整数）。
- 对于输出的位掩码，建议在内核中先使用一个临时缓冲区，填充完结果后，再通过 `copy_out()` 将其拷贝回用户空间。
- 可以为最多可检测的页面数量设置一个上限，简化实现逻辑。
- `mm/pagetable.rs` 中的 `walk()` 函数对于查找特定虚拟地址对应的页表项（PTE）非常有用，推荐使用。
- 你需要在 `const/riscv.h` 中定义 `PTE_A`，即页表项中的访问标志（Accessed Bit）。该标志的具体值可以参考 RISC-V 架构手册进行确定。
- 检查完访问位是否被设置后，**一定要清除该位（PTE_A）**。否则下一次调用 `pgaccess()` 时将无法判断访问位是否是本次新产生的，因为访问位会一直保持为1。
- 在调试页表时，`vmprint()` 函数可能会很有帮助，它可以打印出当前进程的页表结构信息。

实验九：回溯调用栈

实验内容与现象

在调试过程中，**回溯调用栈（Backtrace）** 是一项非常有用的工具。它可以列出从错误发生点向上传递的函数调用栈信息，帮助我们定位程序是如何一步步走到当前出错位置的。

在本实验中，你需要实现一个 `backtrace()` 函数，并将其写在 `src/printf.rs` 中。然后，在 `sys_sleep` 函数中插入对该函数的调用。接着运行用户态测试程序 `bttest`，该程序内部会调用 `sys_sleep`。

如果你的实现正确，程序的输出将类似于以下内容（其中地址可能略有不同）：

```
backtrace:
0x800202f0
0x800220fa
0x80025b32
```

测试完成后，退出 QEMU。在终端中使用如下命令，将这些地址转换为对应的源代码行号（注意根据工具链选择命令）：

```
addr2line -e target/riscv64gc-unknown-none-elf/debug/xv6-rust
0x800202f0
0x800220fa
0x80025b32
Ctrl + D
```

输出应类似如下内容，表示每个调用栈返回地址对应的源文件和行号：

```
/workspaces/xv6-rust/src/process/proc/syscall.rs:289
/workspaces/xv6-rust/src/process/proc/mod.rs:548
/workspaces/xv6-rust/src/trap.rs:86
```

这个功能的实现依赖于编译器在每个函数栈帧中保留的**帧指针（frame pointer）**，它存储了**调用者的栈帧地址**。你需要利用这些帧指针在栈中逐层向上遍历，并打印出每一层栈帧中保存的返回地址（即调用该函数的位置）。

实验提示

- 编译器会将当前函数的帧指针（frame pointer）保存在寄存器 `s0/fp` 中。为了获取该值，可以使用如下代码：

  ```
  let mut fp: usize;
  unsafe{
      core::arch::asm!("mv {}, fp", out(reg) fp);
  }
  ```

  通过内联汇编将寄存器 `fp` 的值提取出来，并返回。你需要在 `backtrace()` 中使用这段代码，以获取当前栈帧的起始地址。

- Rust编译器可能会优化 `fp` 寄存器，为了能够正确读到该寄存器，请确保 `.cargo/config.toml` 中的rustflags包含如下参数

  ```
  "-C", "force-frame-pointers=yes",
  ```

- 根据 RISC-V 调用约定：
  - 当前函数的返回地址保存在帧指针偏移 `-8` 的位置；
  - 上一帧的帧指针保存在偏移 `-16` 的位置；
    利用这一布局，可以沿着帧指针向上传递，逐层还原函数调用栈。
- 内核为每个线程分配一页大小的内核栈，地址为页对齐。你需要按页面对其的计算栈的上下边界，用于判断回溯过程何时终止，避免越界访问。

- 当你验证 `backtrace()` 能正确输出函数调用栈后，可以在 `src/printf.rs` 的 `panic()` 函数中加入对 `backtrace()` 的调用。这样在内核 panic 时，系统会自动打印调用栈信息，有助于分析和调试错误原因。

实验十：周期性任务调用

实验内容

本实验要求你添加一个新特性：让进程在使用 CPU 的过程中能被周期性地通知。这一机制对于计算密集型的程序来说非常有用，比如希望限制自身 CPU 使用时间，或者在持续计算的同时需要周期性地执行某些操作的程序。更广义上讲，你将实现一种初级形式的**用户级中断/异常处理机制**。类似的机制可以被用于在应用层处理如页错误（page fault）等异常，因此具有重要的教学意义。只要你的实现能通过 `alarmtest` 和 `usertests` 的测试用例，就说明你的方案是正确的。

你需要添加一个新的系统调用：

```
int sigalarm(int interval, void (*handler)());
```

当一个应用程序调用 `sigalarm(n, fn)` 后，系统应当在该程序每消耗 `n` 个时钟周期（tick）的 CPU 时间后，自动调用一次它提供的函数 `fn`。函数 `fn` 执行完毕后，用户程序应当**从原先中断的位置继续执行**。

> 注意：xv6 中的 “tick” 是由硬件定时器中断产生的时间单位，并不代表真实世界中的某个精确时间长度。这个时间间隔在系统中是固定的。

若应用程序调用 `sigalarm(0, 0)`，表示关闭之前注册的定时处理函数，系统应停止生成周期性通知。

你会在仓库中找到一个测试程序：`user/alarmtest.c`。请将其添加到 Makefile 的 `UPROGS` 列表中。在你实现 `sigalarm` 和 `sigreturn` 系统调用之前，该程序无法成功编译（`sigreturn` 系统调用用于从处理函数 `fn` 返回时恢复原上下文）。确保这两个系统调用的实现都完成后，再运行测试程序。

`alarmtest.c` 的测试函数 `test0()` 中，调用了 `sigalarm(2, periodic)`，即请求系统每过 2 个 tick 自动调用一次 `periodic()` 函数。随后该程序进入自旋状态。你可以查看 `user/alarmtest.asm` 中生成的汇编代码，这对于调试实现逻辑非常有帮助。

实验现象

若实验的实现是正确的，在shell中执行alarmtest会得到如下输出：

```
$ alarmtest
test0 start
............alarm!
test0 passed
test1 start
..alarm!
alarm!
.alarm!
.alarm!
.alarm!
..alarm!
.alarm!
..alarm!
.alarm!
..alarm!
```

```
test1 passed
test2 start
.......alarm!
test2 passed
```

实验提示

test 0:

首先，你需要修改内核，使其能够在用户进程中触发你注册的 alarm handler 函数。在这一阶段，`test0` 的目标是：当计时器中断触发时，能打印出 `"alarm!"`。目前不需要关心 `"alarm!"` 输出之后程序是否崩溃——如果能输出就说明你成功了一半。

- 你需要修改 `Makefile`，确保 `user/alarmtest.c` 能够被作为用户程序进行编译。
- 在 `user/user.h` 中添加如下声明：

  ```
  int sigalarm(int ticks, void (*handler)());
  int sigreturn(void);
  ```

- 更新以下文件以支持新的系统调用：
  - `user/usys.pl`（生成 `usys.S`）
  - `include/syscall.h`
  - `process/proc/syscall.rs`
- 暂时可以让 `sys_sigreturn()` 直接返回 `0`，无需真正恢复上下文。
- 在 `struct proc`（定义于 `process/proc/mod.rs`）中添加字段来保存：
  - 报警周期 `interval`
  - 报警处理函数 `handler` 的地址
- 同时，你还需要增加一个字段记录自上一次报警以来的 tick 计数（或者距离下次报警还有多少 tick）。
  可在 `alloc_proc()`（定义在 `process/proc/mod.rs`）中初始化这些字段。
- 每一次时钟中断都会通过 `user_trap()`（位于 `src/trap.rs`）来处理。
  你只需要在设备类型为定时器中断时更新 alarm tick，例如：

  ```
  ScauseType::IntSSoft => {
     if CpuManager::cpu_id() == 0 {
         clock_intr();
     }
      ...
  }
  ```

- 如果进程没有注册处理函数，或周期为 0，则不应触发报警。
  注意：`handler` 的地址可能为 0（例如，在 `alarmtest.asm` 中 `periodic` 函数地址为 0）。
- 你需要在 `user_trap()` 中加入逻辑：当 alarm 到期后，应将用户态的下一条指令设置为 `handler` 函数地址，使得返回用户态后执行该函数。
- 在 RISC-V 上，trap 返回用户态执行的地址由 `sepc` 控制。
- 如果你能看到 `"alarm!"` 的输出，就说明 test0 成功。

---

test 1/2:

此时你可能会遇到以下问题：程序在输出 `"alarm!"` 后崩溃、输出 `"test1 failed"`，或直接退出却未显示 `"test1 passed"`。这些错误说明你**尚未正确恢复用户态进程的中断现场。**

要完成 `test1` 和 `test2`，你需要确保：

1. **handler 函数返回后，用户程序能继续从中断位置继续执行。**
2. **被中断前的所有寄存器状态都被正确恢复。**
3. **每次 handler 执行完毕后重新启动周期计数器，使得报警能持续触发。**

设计要求：

- 我们已经为你做出一个关键设计决策：**用户级 handler 必须在结束时调用 `sigreturn()` 系统调用。**
  查看 `alarmtest.c` 中的 `periodic()` 实现可以参考其调用方式。
- 这意味着你可以在 `user_trap()` 中保存中断现场，并在 `sys_sigreturn()` 中恢复这些状态，从而使程序能够从被打断的地方无缝继续执行。

实现提示：

- 你需要保存并恢复用户进程的大量寄存器。
  - **提示**：不只是 `a0-a7`，你还可能需要保存 `ra`、`sp`、`s0-s11`、`t0-t6` 等。
- 在 `usertrap()` 中，当定时器中断触发报警时，应保存全部用户寄存器到 `proc` 中的某个缓存结构里（例如 `trapframe_backup`），以备恢复。
- 在 `sigreturn()` 中，将 `trapframe_backup` 中保存的寄存器信息写回 `trapframe`，从而恢复现场。
- 要防止 handler 嵌套调用：如果上一次 handler 尚未执行完成，内核不能再次触发新的 handler 调用。
  - `test2` 会检测这一点。
- 当你成功通过 `test0`、`test1` 和 `test2` 后，**请务必运行 `usertests` 来验证你未破坏其它内核部分的行为正确性。**

实验十一：内存懒分配

实验背景

操作系统与页表硬件配合时，可以使用懒分配来优化用户空间堆内存的管理。xv6中的应用程序通过 `sbrk()` 系统调用请求堆内存。在给定的内核中，`sbrk()` 会分配物理内存并将其映射到进程的虚拟地址空间。当请求内存较大时，内核分配和映射内存的过程可能需要较长时间。例如，一个千兆字节包含262,144个4096字节的页面；即使每个分配操作很便宜，这样的请求依然需要大量的分配。此外，一些程序分配的内存超过实际需要的内存（例如实现稀疏数组），或提前分配内存以便后续使用。为了让 `sbrk()` 在这些情况下能更快完成，先进的内核会采用懒分配的方式。也就是说，`sbrk()` 不会立即分配物理内存，而是记住哪些用户地址已经分配，并将这些地址在用户页表中标记为无效。当进程第一次尝试使用任何懒分配的内存页面时，CPU会触发一个页面错误，内核通过分配物理内存、清零并映射它来处理该错误。

实验内容

步骤一：移除 `sbrk()` 中的内存分配

你的第一个任务是删除`sbrk()`系统调用实现中的页面分配，位于`proc/mod.rs`中。`sbrk()`系统调用通过增加`n`字节来扩展进程的内存大小，然后返回新分配区域的起始位置（即原来的大小）。你的新实现应该只是将进程的大小（`ProcData.sz`）增加`increment`，并返回原来的大小，而无需进行内存分配。因此，你需要删除对`uvm_alloc()`的调用（但你仍然需要增加进程的大小！）。

完成修改后，编译启动系统内核，并执行`echo hi`，可以看到如下效果：

```
init: starting sh
$ echo hi
scause 0xf
sepc=0x12a4 stval=0x4008
```

步骤二：实现内存懒分配

修改`trap.rs`中的代码，以响应来自用户空间的页面错误，通过在错误地址处映射一个新分配的物理内存页面，然后返回用户空间让进程继续执行。你应该在产生"usertrap(): ..."消息的`printf`调用之前添加你的代码。根据需要修改其他内核代码，以确保`echo hi`能够正常运行。

实验提示

- 通过检查`usertrap()`中的`r_scause()`是否为13或15来判断是否为页错误。
- 使用`r_stval()`获取引发页面错误的虚拟地址。
- 参考中`uvm_alloc()`的代码，因为`sbrk()`通过`growproc()`调用它。需要调用`kalloc()`和`mappages()`。
- 使用`pg_round_down()`将错误的虚拟地址舍入到页面边界。
- 修改`uvm_unmap() uvm_copy() try_clone()`，避免在某些页面未映射时触发panic。

可参考的更优实现

- 处理负的`sbrk()`参数。
- 如果进程在一个虚拟内存地址上发生页面错误，并且该地址高于任何通过`sbrk()`分配的地址，终止该进程。
- 正确处理`fork()`中的父子进程内存拷贝。
- 处理进程将从`sbrk()`获得的有效地址传递给系统调用（如`read`或`write`）的情况，但该地址的内存尚未分配。
- 正确处理内存不足的情况：如果`kalloc()`在页面错误处理程序中失败，终止当前进程。
- 处理位于用户栈下方的无效页面错误。

说明：受限于Rust语言更强的内存安全设计，本实验只需要通过lazytests用户程序测试即可

一个可以接受的设计运行效果如下：

```
$ lazytests
lazytests starting
running test lazy alloc
test lazy alloc: OK
running test out of memory
test out of memory: OK
ALL TESTS PASSED
```

实验十二：优先级调度

实验背景

原始 xv6 调度器按“简单轮转（round-robin）”工作：在进程表中寻找 RUNNABLE 进程，依次让它们运行，每个进程获得大致公平的时间片。该策略实现简单、可预测，但无法区分“重要任务”和“普通任务”。优先级调度为每个进程分配一个整数优先级，**数值越大优先级越高。**调度器应始终把 CPU 分配给**就绪队列中优先级最高**的进程；若有多个并列最高，可在它们之间继续做轮转或按到达顺序选择。

实验内容

本实验将引导你为 **xv6-rust** 内核加入**基于优先级的进程调度。**完成后，内核会总是选择**当前系统中优先级最高**的可运行（RUNNABLE）进程执行；你还将添加系统调用，允许进程在用户态**设置/查询自身优先级。**

- **实现优先级调度**：修改 xv6-rust 的调度器，使其不再简单轮转，而是始终选择**优先级数值更大**的 RUNNABLE 进程运行。
- **为进程添加优先级字段**：在进程结构中加入 `priority` 字段，所有新建进程使用**0**作为默认优先级（推荐优先级范围为0-255）。
- **新增系统调用** `setpri` **/** `getpri`：实现 `setpri(int new_priority)`（设置**当前进程**优先级）与 `getpri()`（返回**当前进程**优先级）。

实现方法

- 进程结构（`Proc`）与调度循环（`schedule()`）位于 `process/` 模块中。
- 系统调用：在 syscall 分发表中注册编号与处理函数，用户态通过封装/汇编入口触发 ecall。
- 本实验的核心是：给 `Proc` 增加 `priority`，修改调度器“挑选下一个进程”的策略，并新增 `setpri` / `getpri`。

实验提示

1. 在进程结构中加入 `priority` 字段

- 在 `struct ProcExcl` 中添加 `priority: usize`。
- 优先级范围建议0 - 255，并在**创建新进程**时初始化，把 `priority` 设为默认值，建议为 0。
- 确认所有新建进程（init、sh、子进程）都能拿到默认优先级。
- Rust 结构新增字段后，记得更新相应的构造/初始化代码。

---

2. 修改调度器：按优先级选择 RUNNABLE 进程

- 把现有“遇到 RUNNABLE 就选”的轮转逻辑改为“**在所有 RUNNABLE 中挑最高优先级**”
- 在调度循环中扫描进程表：记录**当前看到的最高优先级**及其进程；
- 若存在并列最高，可直接取第一位（简单做法），或在并列者之间做**等优先级的轮转**
- 找到候选后进行上下文切换；若无 RUNNABLE 进程，进入 idle（如 `wfi`）或继续自旋。
- 你可以将“谁来选”的策略下沉到如 `alloc_runnable()` 这样的辅助函数中，使 `scheduler()` 逻辑保持简洁。

---

3. 实现系统调用 `setpri` / `getpri`

- `int setpri(int new_priority)`：

    - 仅作用于**当前调用进程**。
    - 读取旧优先级，检查 `new_priority` 是否在合法范围；合法则写入，不合法则可直接返回旧值或返回错误码（自定）。
    - **建议返回旧优先级**，以便用户态了解变更前的值。
    - **建议行为**：如果该调用**降低了当前进程的优先级**，且系统中存在更高优先级的 RUNNABLE 进程，可在返回前调用 `yield()` 主动让出 CPU，让更高优先级尽快运行
- `int getpri(void)`：
    - 返回当前进程的 `priority`。

参考实现

实验一：进行系统调用

C语言实现

```c
int
main(int argc, char *argv[])
{
  // 如果命令行参数数量不是 2（程序名 + 1 个参数），说明用户未传入 sleep 时长
  if (argc != 2) {
    // 向标准输出（fd = 1）打印错误信息
    write(1, "Error Argument\n", 16);
  }
  else {
    // 将字符串参数转换为整数，作为 sleep 的时间（单位为 tick）
    sleep(atoi(argv[1]));
  }

  // 程序执行完毕，调用 exit 正常退出
  exit(0);
}
```

Rust语言实现

```rust
#![no_std]                  // 告诉编译器不链接 Rust 标准库（std），适用于内核/嵌入式/操作系统环境
#![no_main]                 // 禁用 Rust 默认的入口点 `fn main()`，因为我们自己定义了 main 函数

use user_rust_lib::task::sleep;  // 从用户态库中引入 sleep 函数，封装了系统调用

// 引入宏定义，例如封装的 `println!`、`syscall!` 等，来自 user_rust_lib 库
#[macro_use]
extern crate user_rust_lib;

#[no_mangle]                // 禁止 Rust 对函数名进行名称改编，以便让操作系统正确识别 main 函数符号
fn main(argc: usize, argv: &[&str]) -> i32 {
    // 调用 sleep 函数让进程睡眠 10 个 tick（xv6 中的时间单位）
    sleep(10);
```

```
    // 返回 0 表示正常退出
    0
}
```

实验二：打乒乓

C语言实现

```
int
main(int argc, char *argv[])
{
  int p1[2], p2[2];                 // 创建两个管道：p1 用于子进程 -> 父进程，p2 用于父进程 -> 子进程
  char foo[10];                     // 用于接收管道中传输的数据的缓冲区

  pipe(p1);                         // 初始化管道 p1
  pipe(p2);                         // 初始化管道 p2

  if(fork() == 0) {                 // 创建子进程；子进程执行此分支
    // 子进程逻辑
    if(read(p2[0], foo, sizeof(foo)) != 0) {  // 从父进程的写端读取消息（通过 p2[0]）
        printf("%d: received ping\n", getpid()); // 打印“收到 ping”的消息
        write(p1[1], "CHILD", 6);   // 将字符串 "CHILD" 通过 p1 写给父进程
    }
    exit(0);                        // 子进程结束
  }
  else {
    // 父进程逻辑
    write(p2[1], "PARENT", 7);      // 将字符串 "PARENT" 写入 p2，发送给子进程
    if(read(p1[0], foo, sizeof(foo)) != 0) {  // 从子进程读取返回的信息
        wait(0);                    // 等待子进程结束
        printf("%d: received pong\n", getpid()); // 打印“收到 pong”的消息
    }
    exit(0);                        // 父进程结束
  }
}
```

Rust语言实现

```
pub fn main() -> i32 {
    let mut p1 = [0; 2]; // 定义第一个管道 p1：用于 子 -> 父 的通信（子写父读）
    let mut p2 = [0; 2]; // 定义第二个管道 p2：用于 父 -> 子 的通信（父写子读）
    let mut buf = [0u8; 10]; // 定义接收缓冲区，用于存放管道中读取的消息

    pipe(&mut p1); // 创建管道 p1（将两个文件描述符写入 p1）
    pipe(&mut p2); // 创建管道 p2（将两个文件描述符写入 p2）

    if fork() == 0 {
        // 子进程执行逻辑

        // 从 p2 的读端读取来自父进程的数据
        if read(p2[0] as isize, &mut buf) != 0 {
```

```
            // 如果成功读取，则打印收到 ping 的消息，包含子进程的 PID
            println!("{}: received ping", getpid());

            // 通过 p1 的写端向父进程发送 "CHILD" 消息
            user_rust_lib::file::write(p1[1] as isize, b"CHILD");
        }

        exit(0) // 子进程结束
    } else {
        // 父进程执行逻辑

        // 通过 p2 的写端向子进程发送 "PARENT" 消息
        user_rust_lib::file::write(p2[1] as isize, b"PARENT");

        // 从 p1 的读端接收来自子进程的响应
        if read(p1[0] as isize, &mut buf) != 0 {
            // 等待子进程退出，传入一个 dummy 的指针
            wait(&mut 0);

            // 打印收到 pong 的消息，包含父进程的 PID
            println!("{}: received pong", getpid());
        }

        exit(0); // 父进程结束
    }
}
```

实验三：并发素数筛

C语言实现

```
void
pipeline(int *in_pipe){                 // 递归构建管道处理链，每个进程负责一个素数筛选
  int out_pipe[2];                      // 用于传递未被当前素数过滤的数值给下一个进程的管道
  int buf;                              // 缓冲区，用于存储读取到的整数
  int head;                             // 当前进程负责筛选的素数
  pipe(out_pipe);                       // 创建新的输出管道

  if(read(in_pipe[0], &head, 4)){       // 从输入管道读取一个整数作为“当前素数”
    if(head >= 35){                     // 若当前数 >= 35，终止递归构建
      return;
    }
  }
  else{                                 // 如果读不到数据，说明已经结束，直接返回
    return;
  }

  if(fork() == 0){                      // 子进程负责处理下一个素数
    close(out_pipe[1]);                 // 子进程不写，只读，所以关闭写端
    pipeline(out_pipe);                // 递归进入下一层，处理下一个素数
  }
  else{                                 // 父进程：筛选非当前素数的数值并写入输出管道
    printf("prime %d\n", head);         // 打印当前进程筛选出的素数
    while(read(in_pipe[0], &buf, 4)){// 持续从输入管道读取整数
```

```
      if(buf % head != 0){            // 如果不能被当前素数整除，传给下一个进程
        write(out_pipe[1], &buf, 4);
      }
    }
    close(in_pipe[0]);                // 输入处理完毕，关闭输入管道
    close(out_pipe[1]);               // 输出也完成，关闭输出管道
    wait(0);                          // 等待子进程结束，确保输出顺序一致
  }
  return;
}

int
main(int argc, char *argv[])
{
  int in_pipe[2];                     // 主进程使用的初始输入管道
  pipe(in_pipe);                      // 创建主进程的输出管道

  if(fork() == 0){                    // 子进程负责启动 pipeline 递归处理
    close(in_pipe[1]);                // 子进程只读关闭写端
    pipeline(in_pipe);                // 启动筛选流程
  }
  else{
    close(in_pipe[0]);                // 父进程只写，关闭读端
    for(int i = 2; i < 35; i++){      // 将 2~34 写入到管道中，作为待筛选的整数序列
      write(in_pipe[1], &i, 4);
    }
    close(in_pipe[1]);                // 写入完毕后关闭写端，表示写完了
    wait(0);                          // 等待子进程处理完所有筛选
  }
  exit(0);                            // 程序结束
}
```

Rust语言实现

```
#[no_mangle]                          // 防止函数名被 Rust 编译器修改，保证可以被内核识别为
`main`
pub fn main(){
    let mut in_pipe = [0u32; 2];      // 创建一个整数数组用于 pipe，作为主进程写入和子进程读
取的管道
    let _ = pipe(&mut in_pipe);       // 创建 pipe，in_pipe[0] 为读端，in_pipe[1] 为写
端

    if fork() == 0 {                  // 创建子进程处理筛选流程
        let _ = close(in_pipe[1] as isize);  // 子进程不写，只读，关闭写端
        pipeline(in_pipe);                   // 启动递归的管道处理函数
    } else {
        let _ = close(in_pipe[0] as isize);  // 父进程不读，只写，关闭读端
        for i in 2..35 {                     // 将 2~34 的整数写入管道
            let n = i as u32;                // 强制转换为 u32 类型
            let _ = user_rust_lib::file::write(in_pipe[1] as isize,
&n.to_ne_bytes()); // 写入整数的字节表示
        }
        let _ = close(in_pipe[1] as isize);  // 所有数字写入完毕后关闭写端
        let _ = wait(&mut 0);                // 等待子进程执行完毕
```

```
    }

    exit(0);                                    // 主进程退出
}
fn pipeline(in_pipe: [u32; 2]) {
    let mut out_pipe = [0u32; 2];               // 输出管道，用于传递剩余待筛选的数字
    let _ = pipe(&mut out_pipe);                // 创建新管道

    let mut buf = [0u8; 4];                     // 字节缓冲区，用于读取整数
    let mut head = 0u32;                        // 当前进程筛选的素数

    if read(in_pipe[0] as isize, &mut buf) == 0 {
        return;                                 // 如果读取失败（管道关闭），直接返回
    }
    head = u32::from_ne_bytes(buf);             // 将读到的 4 字节转换为整数
    if head >= 35 {                             // 若超出范围，不再筛选，直接返回
        return;
    }

    if fork() == 0 {                            // 为下一个素数创建新进程
        let _ = close(out_pipe[1] as isize); // 子进程只读，不写，关闭写端
        pipeline(out_pipe);                     // 递归进入下一层处理
    } else {
        println!("prime {}", head);             // 输出当前发现的素数
        loop {
            let nread = read(in_pipe[0] as isize, &mut buf); // 从输入管道持续读取数
据
            if nread == 0 {
                break;                          // 如果读取结束（管道写端关闭），退出循环
            }
            let val = u32::from_ne_bytes(buf);                // 将字节转换为整数
            if val % head != 0 {                              // 若不能被当前素数整除
                let _ = user_rust_lib::file::write(out_pipe[1] as isize,
&val.to_ne_bytes()); // 写入输出管道
            }
        }
        let _ = close(in_pipe[0] as isize);    // 输入处理完毕，关闭读端
        let _ = close(out_pipe[1] as isize);   // 输出完成，关闭写端
        let _ = wait(&mut 0);                   // 等待子进程结束，保证输出顺序
    }
}
```

实验四：文件查找

参考实现

```
// 返回路径中最后一级文件名（带处理长度限制）
char*
fmtname(char *path)
{
  static char buf[DIRSIZ+1];      // 用于存放格式化后的文件名
  char *p;

  // 从路径末尾向前查找最后一个 '/'，以获取文件名
```

```
  for(p=path+strlen(path); p >= path && *p != '/'; p--)
    ;

  p++;  // 指向文件名开始的字符

  if(strlen(p) >= DIRSIZ)         // 若文件名已经足够长（无需补齐）
    return p;

  // 若文件名较短，拷贝到缓冲区并补齐 \0
  memmove(buf, p, strlen(p));
  memset(buf+strlen(p), '\0', DIRSIZ-strlen(p));
  return buf;
}

// 递归查找以 path 为根目录下所有名为 name 的文件
void
find(char *path, char *name)
{
  char buf[512], *p;              // buf 用于拼接路径，p 用于遍历目录项
  int fd;                         // 打开目录用到的文件描述符
  struct dirent de;               // 目录项结构体
  struct stat st;                 // 文件状态结构体

  // 尝试打开路径
  if((fd = open(path, 0)) < 0){
    fprintf(2, "find: cannot open %s\n", path);
    return;
  }

  // 获取路径对应的文件状态
  if(fstat(fd, &st) < 0){
    fprintf(2, "find: cannot stat %s\n", path);
    close(fd);
    return;
  }

  // 根据类型分类处理（这里只处理目录）
  switch(st.type) {
    case T_DIR:
      strcpy(buf, path);          // 将当前路径拷贝到 buf 中
      p = buf+strlen(buf);        // p 指向路径末尾
      *p++ = '/';                 // 在路径末尾添加斜杠准备拼接文件名

      // 遍历目录项
      while(read(fd, &de, sizeof(de)) == sizeof(de)){
        if(de.inum == 0 || !strcmp(de.name,".") || !strcmp(de.name,".."))
          continue;               // 跳过空项和 .、..

        memcpy(p, de.name, DIRSIZ); // 拼接完整路径
        p[DIRSIZ] = 0;              // 添加字符串结尾

        int fd_sub = open(buf, 0);  // 打开拼接后的路径
        struct stat st_sub;
        fstat(fd_sub, &st_sub);     // 获取该路径的 stat 信息

        if(st_sub.type == T_FILE){ // 如果是文件，比较文件名是否匹配
```

```
          if(!strcmp(name, fmtname(buf))){
            printf("%s\n", buf);   // 如果匹配，打印路径
          }
        }
        else if(st_sub.type == T_DIR) {
          find(buf, name);         // 是目录，递归进入查找
        }
        close(fd_sub);             // 关闭子路径文件描述符
      }
    break;
  }

  return;
}

int
main(int argc, char *argv[])
{
  find(argv[1], argv[2]);          // 调用 find，从指定路径开始查找指定文件名
  exit(0);                         // 程序结束
}
```

实验五：系统调用跟踪

添加系统调用号定义

文件：`include/syscall.h`

```
#define SYS_trace  22
```

解释：
此处添加了 `trace` 系统调用的编号。
其中 `SYS_trace` = 22 是本实验的核心新增系统调用，用于控制系统调用追踪掩码。

为进程结构体添加 trace 掩码字段

文件：`src/process/proc/mod.rs`

```
pub tracemask: usize,
```

初始化时设置为 0：

```
tracemask: 0,
```

解释：
添加 `tracemask` 字段用于记录当前进程启用了哪些系统调用的追踪。它是一个位图掩码，按位表示系统调用是否开启追踪。

系统调用处理逻辑中添加追踪输出

文件：`src/process/proc/mod.rs`

```
22 => self.sys_trace(),  // 注册 trace 调用处理

// syscall 返回后输出追踪信息
let guard = self.excl.lock();
let pid = guard.pid;
let tracemask = self.data.get_mut().tracemask;
if tracemask & (1 << a7) != 0 {
    println!("{}: syscall {} -> {}", pid, syscall::SYSCALL_NAME[a7], tf.a0 as
isize);
}
drop(guard);
```

解释：

- 为 syscall 分发函数添加了 `SYS_trace` 的处理路径；
- 在系统调用返回前判断当前进程的 `tracemask` 中是否包含当前系统调用（通过位掩码判断 `1 << a7`）；
- 若需要追踪，打印出进程号、系统调用名称和其返回值。

---

fork 过程复制 trace 掩码

文件：`src/process/proc/mod.rs`

```
cdata.tracemask.clone_from(&pdata.tracemask);
```

解释：
在创建子进程时，将父进程的 `tracemask` 值拷贝到子进程，实现“追踪设置可继承”的语义。符合题目要求：trace 只影响当前进程及其子进程。

系统调用名称数组与 trait 注册

文件：`src/process/proc/syscall.rs`

```
pub static SYSCALL_NAME: [&str; 24] = ["", "fork", "exit", "wait", ..., "trace",
"sysinfo"];
fn sys_trace(&mut self) -> SysResult;
```

解释：

- 定义了一个全局系统调用名称数组，用于在追踪输出中打印系统调用名字；
- 在 trait 中注册了 `sys_trace()` 系统调用的声明。

实现 trace 系统调用逻辑

文件：`src/process/proc/syscall.rs`

```rust
fn sys_trace(&mut self) -> SysResult {
    let input_mask = self.arg_i32(0);
    if input_mask < 0 {
        Err(())
    } else {
        (*self.data.get_mut()).tracemask = input_mask as usize;
        Ok(0)
    }
}
```

解释:
实现了 `trace(mask)` 的核心逻辑:

- 读取用户传入的参数 `mask`;
- 若合法，则设置到当前进程的 `tracemask` 字段中;
- 返回 0 表示设置成功。

实验六：加速系统调用

添加共享页虚拟地址常量 `USYSCALL`

文件路径: `src/consts/memlayout.rs`

```rust
pub const USYSCALL: ConstAddr = TRAPFRAME.const_sub(PGSIZE);
```

解释:
新增 `USYSCALL` 常量，用于指定一块在用户地址空间可访问的只读共享页面地址。该地址位于 `TRAPFRAME` 前一页，为后续共享只读数据（如 PID）提供映射依据。

在页表中映射 USYSCALL 页

文件路径: `src/mm/pagetable.rs`

```rust
pub fn alloc_proc_pagetable(trapframe: usize, usyspage: usize) ->
Option<Box<Self>> {
    ...
    pagetable
        .map_pages(
            VirtAddr::from(USYSCALL),
            PGSIZE,
            PhysAddr::try_from(usyspage).unwrap(),
            PteFlag::R | PteFlag::U,
        )
        .ok()?;
    ...
}
```

解释:
为用户进程的页表添加 `USYSCALL` 到物理页的映射，并设置为只读用户权限。这允许用户态程序无需系统调用即可读取如 PID 等信息，从而加速系统调用。

释放页表时取消 USYSCALL 映射

文件路径： `src/mm/pagetable.rs`

```
self.uvm_unmap(USYSCALL.into(), 1, false);
```

解释：
在销毁进程页表时，移除对 USYSCALL 页面的映射，防止悬空映射和潜在资源泄漏。

分配并初始化 UsysPage 页面

文件路径： `src/process/mod.rs`

```
pd.up = unsafe { RawSinglePage::try_new_zeroed().ok()? as *mut UsysPage };
(unsafe { &mut *pd.up }).pid = new_pid as u32;

match PageTable::alloc_proc_pagetable(pd.tf as usize, pd.up as usize) {
    ...
}
```

解释：
在创建进程时分配一页用于 `UsysPage` 的内存，并将当前进程的 PID 写入。该页面将映射到用户态，使得用户程序可通过读取共享内存的方式获得进程信息。

更新 ELF 加载路径的页表创建调用

文件路径： `src/process/proc/elf.rs`

```
match PageTable::alloc_proc_pagetable(pdata.tf as usize, pdata.up as usize) {
```

解释：
确保在通过 ELF 加载用户程序时也传入 `UsysPage` 指针，使页表初始化函数能正确映射该共享页。

在进程控制块中添加 up 字段

文件路径： `src/process/proc/mod.rs`

```
pub up: *mut UsysPage,
```

解释：
在进程的数据结构 `ProcData` 中新增 `up` 字段，用于保存 `UsysPage` 页面的指针，使该页面在整个进程生命周期中可访问并可管理。

释放 UsysPage 页面

文件路径： `src/process/proc/mod.rs`

```
let up = self.up;
self.up = ptr::null_mut();
if !up.is_null() {
    unsafe { RawSinglePage::from_raw_and_drop(up as *mut u8); }
}
```

解释：
在进程退出时回收为 `UsysPage` 分配的物理内存，确保无资源泄漏。

定义用户共享结构体 `UsysPage`

文件路径： `src/process/trapframe.rs`

```
#[repr(C)]
#[derive(Debug)]
pub struct UsysPage {
    pub pid: u32,
}
```

解释：
定义 `UsysPage` 结构体，该结构用于在共享页面中存放只读信息（当前仅包含 PID），用户程序可以通过读取该结构体以免去一次系统调用。

实验七：打印页表

引入原子布尔类型用于只打印一次页表

文件路径：`src/mm/pagetable.rs`

```
use core::sync::atomic::AtomicBool;
```

解释：
为了控制只在第一次调用 `vm_print` 时打印页表头部信息，使用了 `AtomicBool` 类型的全局标志变量 `INIT_VM_PRINT`，具备线程安全性。

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。

注意：最内层的判断 `if level == 1` 出现了两次，应考虑修正为 `level == 2`。

全局初始化标志

文件路径：`src/mm/pagetable.rs`

```
static INIT_VM_PRINT: AtomicBool = AtomicBool::new(true);
```

解释:
只允许打印页表头部一次，避免多次重复输出 `"page table ..."` 字样。

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。

实验八：检测某一页是否被访问过

添加 PTE 的访问位判断与清除函数

文件路径： `src/mm/pagetable.rs`

```
#[inline]
pub fn is_access(&self) -> bool {
    (self.data & (PteFlag::A.bits())) > 0
}

#[inline]
pub fn clear_access(&mut self) {
    self.data &= !PteFlag::A.bits()
}
```

解释：
新增两个方法用于检测和清除页表项中的访问位（Accessed bit）：

- `is_access()` 用于判断该页是否被访问过（读或写）；
- `clear_access()` 在检测之后清除访问位，为下次访问检测提供可能性。

这是实现 `pgaccess` 系统调用的核心，需依赖 PTE 中的 A 位来记录访问状态。

将 `walk_mut` 改为公有以便系统调用访问页表项

文件路径： `src/mm/pagetable.rs`

```
- fn walk_mut(&mut self, va: VirtAddr) -> Option<&mut PageTableEntry> {
+ pub fn walk_mut(&mut self, va: VirtAddr) -> Option<&mut PageTableEntry> {
```

解释：
为了在系统调用 `sys_pgaccess` 中访问用户页表项，需要将原本私有的 `walk_mut()` 方法设为公有（`pub`）。该函数用于获取虚拟地址对应的 PTE 的可变引用，是遍历用户地址空间时的关键接口。

为系统调用表增加 `pgaccess` 调度分支

文件路径： `src/process/proc/mod.rs`

```
+ 22 => self.sys_pgaccess(),
```

解释：
将系统调用号 `22` 映射到对应处理函数 `sys_pgaccess()`，是将用户空间调用正确路由到内核处理逻辑的必要步骤。该修改确保用户可以通过 syscall 机制访问新实现的功能。

在 Syscall trait 中声明 `sys_pgaccess` 接口

文件路径： `src/process/proc/syscall.rs`

```rust
fn sys_pgaccess(&mut self) -> SysResult;
```

解释：
在 `Syscall` trait 中新增对 `sys_pgaccess` 方法的声明，确保进程结构体 `Proc` 的实现中包含该系统调用接口。这是 Rust trait 系统中统一行为定义的一部分。

实现 `pgaccess` 系统调用主体逻辑

文件路径： `src/process/proc/syscall.rs`

```rust
fn sys_pgaccess(&mut self) -> SysResult {
    let mut start_va = self.arg_addr(0);
    let page_num = self.arg_i32(1);
    let ret_pa = self.arg_addr(2);
    let mut ret: usize = 0;
    let pgt = self.data.get_mut().pagetable.as_mut().unwrap();
    for i in 0..page_num {
        let pte = pgt.walk_mut(unsafe { VirtAddr::from_raw(start_va) }).unwrap();
        if pte.is_access() {
            ret |= 1 << i;
        }
        pte.clear_access();
        start_va += PGSIZE;
    }
    let _ = pgt.copy_out(&ret as *const usize as *const u8, ret_pa, size_of::
<usize>());
    Ok(0)
}
```

解释：
该系统调用的核心功能实现：

- 读取用户传入的起始虚拟地址、页数和返回地址；
- 遍历每一页，对应的 PTE 若访问位为 1，则在结果 bitmask 中设置对应位；
- 每次检测后清除该页的访问位；
- 将 bitmask 结果写回用户空间。

实验九：回溯调用栈

添加 Backtrace 函数

文件路径: `src/printf.rs`

```rust
pub fn backtrace() {
    let pgt = unsafe {
CPU_MANAGER.my_proc().data.get_mut().pagetable.as_mut().unwrap() };
    let mut fp: usize;
    unsafe {
        core::arch::asm!("mv {}, fp", out(reg) fp);
    }
    let mut ra: usize;
```

```
    let barrier: usize = (fp + crate::consts::PGSIZE - 1) & !
(crate::consts::PGSIZE - 1);
    println!("backtrace:");
    while fp < barrier {
        ra = unsafe { kvm_pa(VirtAddr::from_raw(*((fp - 8) as *const usize))) }
as usize;
        println!("0x{:x}", ra);
        fp = unsafe { *((fp - 16) as *const usize) };
    }
}
```

解释:
该部分新增了一个 `backtrace()` 函数，用于回溯内核调用栈并打印出每一帧中的返回地址。利用 RISC-V 的 `s0` 寄存器（即帧指针 `fp`）获取当前函数的栈帧指针，并按照固定偏移读取保存的返回地址和上一层帧指针，逐层回溯。`kvm_pa` 用于虚拟地址转物理地址，`barrier` 限制在单页内栈空间，避免栈外非法访问。

在系统调用中插入回溯

文件路径: `src/process/proc/syscall.rs`

```
crate::printf::backtrace();
```

解释:
将 `backtrace()` 的调用插入至 `sys_sleep` 系统调用的开头，使得每次用户调用 `sleep` 时，内核会打印当前的调用栈信息。这是该实验的测试入口，配合 `bttest` 测试程序使用，用于验证调用栈是否能正确回溯输出。原有错误调用被替换为正确的全局函数路径。

实验十：周期性任务调用

添加 ProcAlarm 结构体以支持报警状态

文件路径: `src/process/proc/mod.rs`

```
pub struct ProcAlarm {
    pub interval: usize,
    pub past_tick: usize,
    pub handler_addr: isize,
    pub alarm_frame: *mut TrapFrame,
    pub handler_called: bool,
}

impl ProcAlarm {
    const fn new() -> Self { ... }

    pub fn cleanup(&mut self) { ... }
}
```

解释:
定义了一个新的结构体 `ProcAlarm`，用于保存每个进程的定时器报警相关信息，包括周期间隔、处理函数地址、处理函数是否调用、保存 TrapFrame 的副本等。`cleanup()` 方法用于资源清理，防止内存泄漏。

为进程添加 alarm 字段

文件路径：`src/process/proc/mod.rs`

```
pub struct Proc {
    ...
    pub alarm: UnsafeCell<ProcAlarm>,
}
impl Proc {
    pub const fn new() -> Self {
        ...
        alarm: UnsafeCell::new(ProcAlarm::new()),
    }
}
```

解释：
在 `Proc` 结构中添加 `alarm` 字段，以存储每个进程的报警配置。使用 `UnsafeCell` 是为了在拥有不可变引用时进行内部可变性修改。

添加两个系统调用：sigalarm 和 sigreturn

文件路径：`src/process/proc/syscall.rs`

```
fn sys_sigalarm(&mut self) -> SysResult {
    let interval = self.arg_i32(0) as usize;
    let handler = self.arg_addr(1) as isize;
    let p = self.alarm.get_mut();
    p.interval = interval;
    p.handler_addr = handler;
    Ok(0)
}

fn sys_sigreturn(&mut self) -> SysResult {
    let pa = self.alarm.get_mut();
    let pd = self.data.get_mut();
    unsafe { core::ptr::copy_nonoverlapping(pa.alarm_frame, pd.tf, 1) };
    pa.handler_called = false;
    Ok(0)
}
```

解释：

- `sigalarm`：保存用户传入的 tick 周期和处理函数地址。
- `sigreturn`：恢复中断时保存的 trapframe，使用户进程可以从中断现场继续执行。

在 fork 过程中复制 alarm 状态

文件路径：`src/process/proc/mod.rs`

```
let palarm = self.alarm.get_mut();
let calarm = unsafe { child.alarm.get().as_mut().unwrap() };
...
ptr::copy_nonoverlapping(palarm.alarm_frame, calarm.alarm_frame, 1);
```

解释:
确保 `fork()` 后子进程也能继承父进程的 alarm 状态，包括 `alarm_frame` 内容，保持行为一致。

在 alloc_proc 过程分配 alarm_frame

文件路径: `src/process/mod.rs`

```
let pa = p.alarm.get_mut();
...
pa.alarm_frame = unsafe { RawSinglePage::try_new_zeroed().ok()? as *mut TrapFrame
};
```

解释:
为每个进程初始化 `alarm_frame`，用于保存中断时的 trapframe 备份。

在清理过程中释放 alarm_frame

文件路径: `src/process/mod.rs`

```
let child_alarm = unsafe { self.table[i].alarm.get().as_mut().unwrap() };
...
child_alarm.cleanup();
```

解释:
避免 `alarm_frame` 在进程回收时产生内存泄漏。

注册系统调用号

文件路径: `src/process/proc/mod.rs`

```
22 => self.sys_sigalarm(),
23 => self.sys_sigreturn(),
```

解释:
将 `sigalarm` 和 `sigreturn` 添加到系统调用表中，系统调用号为 22 和 23。

---

trap 处理中注入用户 handler

文件路径: `src/trap.rs`

```
if STARTED.load(Ordering::SeqCst) {
    let pa = unsafe { CPU_MANAGER.my_proc().alarm.get_mut() };
    let pd = unsafe { CPU_MANAGER.my_proc().data.get_mut() };
    if pa.interval > 0 {
        pa.past_tick += 1;
    }
    if (pa.past_tick == pa.interval) && (!pa.handler_called) && (pa.interval > 0)
{
        pa.past_tick = 0;
        unsafe { core::ptr::copy_nonoverlapping(pd.tf, pa.alarm_frame, 1) };
        (unsafe {&mut *pd.tf }).epc = pa.handler_addr as usize;
        pa.handler_called = true;
    }
}
```

解释:
在每次定时器中断时，检查是否需要调用 alarm handler。如果是，则备份当前 trapframe 并跳转到用户自定义 handler。设置 `handler_called = true` 防止 re-entry。

实验十一：内存懒分配

文件路径：`src/mm/pagetable.rs`

1. 修改 `try_clone` 函数中的错误处理

```
return Err(());
```

解释:
修改了 `try_clone` 函数的错误处理方式，原来是使用 `panic!`，现在改为返回 `Err(())`。这种方式避免了程序的崩溃，而是通过返回错误让调用者处理。

---

2. 创建 `null` 变量以避免空指针错误

```
let mut null = PageTableEntry { data: 0 };
```

解释:
在 `unmap` 函数中新增了 `null` 变量，确保在没有找到有效页表条目时，能够安全地跳过处理。

---

3. 替代 `unwrap` 以增加空值检查

```
let pte_option = unsafe { self.walk_mut( VirtAddr::from_raw(ca)) };
if let Some(inn) = pte_option {
    pte = inn;
}
else {
    return;
}
```

解释:
这段修改用于 `unmap` 函数中，替代原来直接使用 `unwrap()` 的方式，增加了对 `walk_mut` 返回结果的安全处理，避免了潜在的程序崩溃。

---

4. 修改 `uvm_copy` 函数中的错误处理

```
return Ok(());
```

解释:
在 `uvm_copy` 函数中，如果在尝试克隆页表条目时失败，现在直接返回 `Ok(())`，防止继续执行不必要的操作。

文件路径：`src/process/proc/mod.rs`

1. 注释掉 `uvm_alloc` 调用

```
// self.pagetable.as_mut().unwrap().uvm_alloc(old_size, new_size)?;
```

解释:
注释掉了 `uvm_alloc` 调用，这表明开发者正在排查或修改内存分配逻辑，避免在当前上下文中调用该方法。

文件路径：`src/register/scause.rs`

1. 增加 `PageFault` 枚举成员

```
PageFault,
```

解释:
在 `ScauseType` 枚举中新增了 `PageFault` 成员，用于表示页面错误。这是为了支持页面故障类型的处理。

---

2. 修改 `get_scause` 函数来处理页面故障

```
13 => ScauseType::PageFault,
15 => ScauseType::PageFault,
```

解释:
在 `get_scause` 函数中，增加了对 `13` 和 `15` 的处理，这两个值分别代表页面缺失（Page Fault）。这表明程序现在能处理来自硬件的页面故障中断。

文件路径：`src/trap.rs`

1. 增加页面故障处理逻辑

```
ScauseType::PageFault => {
    //println!("!!");
    let layout = Layout::from_size_align(4096, 4096).expect("Invalid layout");
```

```
    // 分配 4KB 的内存空间
    let ptr = GlobalAlloc::alloc(&KERNEL_HEAP, layout) as *mut u8;
    if ptr.is_null() {
        p.killed = true.into();
    }
    p.data.get_mut().pagetable.as_mut().unwrap().map_pages(
        VirtAddr::from_raw(pg_round_down(stval::read())),
        4096,
        PhysAddr::from_raw(ptr as usize),
        PteFlag::R | PteFlag::W | PteFlag::U);
}
```

解释:
在 `user_trap` 函数中增加了处理页面故障的逻辑。如果捕获到页面故障（`ScauseType::PageFault`），会分配 4KB 的内存空间，并将其映射到进程的页表中，确保程序能够继续执行。若内存分配失败，则将进程标记为死亡。

---

2. 在 `kerneltrap` 中增加页面故障的接口

```
ScauseType::PageFault => {

}
```

解释:
在 `kerneltrap` 函数中增加了对页面故障的捕获，虽然具体实现未完善，但为后续处理页面故障留下了接口。

实验十二：优先级调度

1. 系统调用接口

```
// 设置当前进程优先级的系统调用
fn sys_setpri(&mut self) -> SysResult {
    // 读取第 0 个系统调用参数（用户传入的新优先级，i32）
    let input_pri = self.arg_i32(0);

    // 检查取值范围：只允许 0..=255
    if input_pri < 0 || input_pri > 255 {
        // 超出范围：给出提示（不修改优先级）
        println!("Priority: 0 - 255");
    } else {
        // 合法：加锁拿到进程的可变引用，写入新的优先级（转为 usize）
        // 注意：lock() 返回的守卫是临时值，语句结束后自动释放锁
        self.excl.lock().priority = input_pri as usize;
    }

    // 主动让出 CPU：让调度器立刻重新评估（若当前优先级被降低，其他更高优先级进程可接管）
    self.yielding();

    // 返回 0 表示调用成功（即使越界提示也返回 0，按你当前语义）
    Ok(0)
}
```

```rust
// 获取当前进程优先级的系统调用
fn sys_getpri(&mut self) -> SysResult {
    // 加锁读取当前进程的优先级（守卫在语句结束时自动释放）
    let pri = self.excl.lock().priority;

    // 将优先级作为返回值带回用户态
    Ok(pri)
}
```

2. 调度函数

```rust
// 从进程表中选择“优先级数值最大”的可运行进程：两阶段：先只读选出候选，再加锁占用，避免借用冲突与竞态
fn alloc_runnable(&mut self) -> Option<&mut Proc> {
    // Phase 1：只读扫描，找出 priority 最大的 RUNNABLE
    let mut best_idx: Option<usize> = None;    // 当前最佳候选在表中的下标
    let mut best_prio: usize = 0;              // 已见到的最高优先级（数值越大优先级越高）

    for (i, p) in self.table.iter().enumerate() { // 只读遍历进程表；用 iter() 而非 iter_mut() 以便后续可变借用
        let guard = p.excl.lock();                // 短暂加锁读取该进程的状态/优先级
        if matches!(guard.state, ProcState::RUNNABLE) { // 仅考虑 RUNNABLE 的进程
            let prio = guard.priority;            // 读取该进程的优先级
            match best_idx {                      // 与当前最佳候选比较
                None => {
                    best_idx = Some(i);           // 首个候选：记录下标
                    best_prio = prio;             // 记录其优先级
                }
                Some(j) => {
                    // 更高优先级则替换；同优先级时用更小的下标打破平局，保证确定性
                    if prio > best_prio || (prio == best_prio && i < j) {
                        best_idx = Some(i);       // 更新最佳下标
                        best_prio = prio;         // 更新最佳优先级
                    }
                }
            }
        }
        // guard 在此作用域结束自动释放（短持锁，避免长时间占用）
    }

    let idx = best_idx?;                          // 若没有任何 RUNNABLE，直接返回 None

    // Phase 2：二次加锁占用（RUNNABLE→ALLOCATED）
    let p = &mut self.table[idx];                 // 现在对选中的条目做可变借用（不会与上面的 iter() 冲突）
    let mut guard = p.excl.lock();                // 再次加锁，准备修改其状态
    if matches!(guard.state, ProcState::RUNNABLE) {// 二次确认：防止在两阶段之间状态被他处改变
        guard.state = ProcState::ALLOCATED;       // 从 RUNNABLE 占用出来；稍后 scheduler 会把它设为 RUNNING
        drop(guard);                              // 释放锁（也可交给作用域自动释放）
        return Some(p);                           // 返回被选中的进程可变引用
```

```
    } else {
        drop(guard);                                    // 候选已不再 RUNNABLE（可能被抢走/睡眠/退出），释放锁
        None                                            // 放弃本次选择；由调用者决定是否重试
    }
}
```

总结

本项目构建了一个基于 Rust 的教学操作系统实验平台，参考 xv6 的系统结构与接口，结合 `xv6-riscv-rust` 实现的内核，设计了完整的文档注释体系、用户程序与测试用例、自动化测试脚本、实验指导手册与参考实现。内核采用 Rust 编写并配套标准化注释文档，支持通过 `cargo doc` 自动生成阅读材料；用户程序与测试机制兼容 xv6，逐步向 Rust 迁移；测试脚本支持自动编译、运行与评分，保障评测的准确性和教学效率；实验设计结合 MIT 6.828 内容，提供分层提示与参考代码，适应不同教学需求，形成一套可持续发展的操作系统教学支撑体系。

参考资料

MIT - 6.828（xv6）

清华大学 - rCore-Tutorial-Book 第三版

中科院软件所 - 循序渐进，学习开发一个 RISC-V 上的操作系统

南京大学 - 操作系统：设计与实现

华中科技大学 - 操作系统实验（riscv-pke）

xv6-riscv-rust

Ko-oK-OS

mit-pdos/xv6-riscv: Xv6 for RISC-V

Stephen Marz: Blog

OSDev Wiki

Writing an OS in Rust