西北工業大學

NORTHWESTERN POLYTECHNICAL UNIVERSITY

NPUcore-Canary 设计文档

作者：杨茂靓

指导老师：张羽

2026 年 1 月 23 日

目录

一、 概述

1.1 系统简介

NPUcore-Canary 是一款采用 Rust 语言开发的现代化宏内核操作系统，基于 NPUcore-BLOSSOM[1] 内核架构进行设计与实现，可同时于 RISC-V64 和 LoongArch64 平台运行，集成了较为完整的中断处理机制、进程管理系统、内存管理模块、文件系统等核心组件，通过系统调用接口为用户提供高效可靠的服务支持。

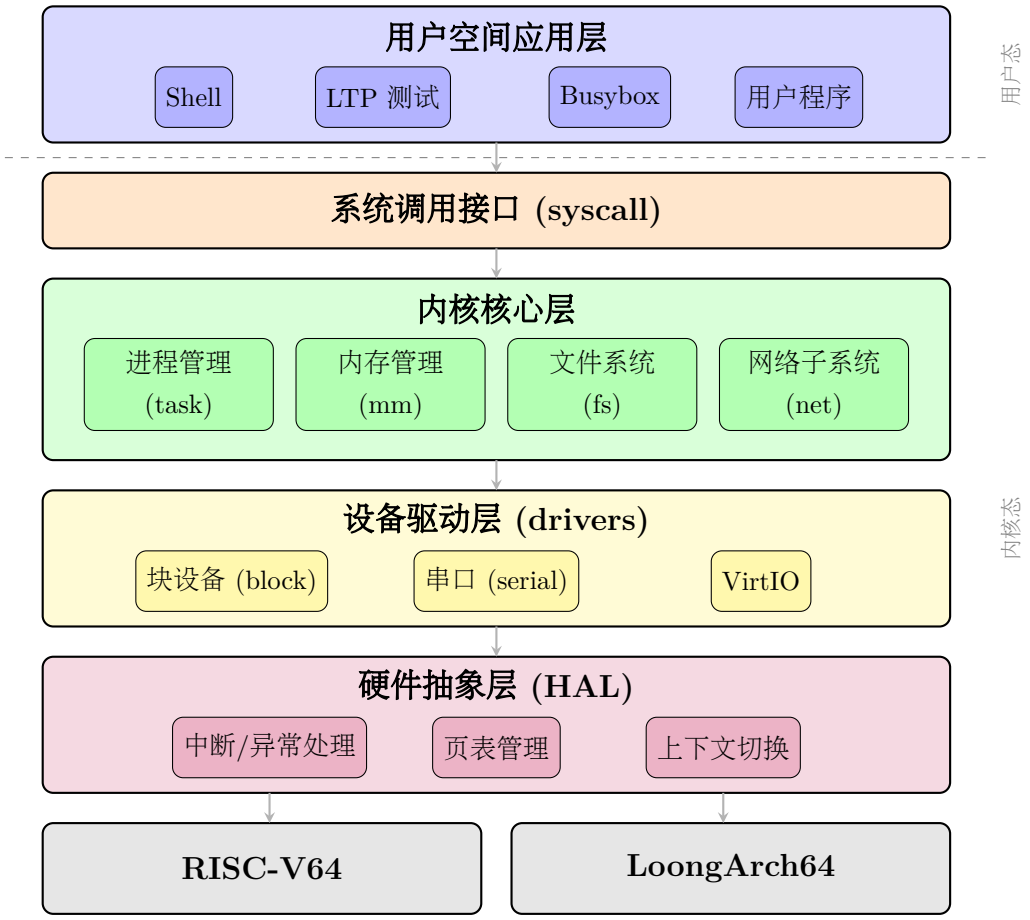

图 1: NPUcore-Canary 整体架构

1.2 NPUcore 简介

NPUcore[2] 是西北工业大学的实践型教学操作系统，使用 Rust 编程语言进行编写，旨在提升操作系统原理实践体验以及探索新型跨指令集教学型操作系统。其由西北工业大学团队在 2022 年针对全国计算机系统能力大赛内核赛道、基于 rCore 所设计实现。在 NPUcore 的最早的版本中，仅支持 RISC-V 架构，而后在 2024 年，其在原有基础上添加了对国产 LoongArch 体系结构的支持。

在 2025 年上半年的开发工作中，NPUcore-BLOSSOM 项目团队成功实现了对 EXT4 文件系统的支持，同时构建了 HAL（硬件抽象层），使得一份上层核心代码能够同时流畅运行于 LoongArch 和 RISC-V 体系结构之上。

1.3 NPUcore-Canary 当前工作

NPUcore-Canary 作为 2025 年全国大学生计算机系统能力大赛-操作系统设计赛-中西部区域赛-内核实现赛道的参赛作品，在 NPUcore-BLOSSOM 的基础上进行改进与扩展，参考借鉴了其它内核赛道优秀参赛队伍[3][4]与 Linux 内核的诸多优秀设计，主要进行了以下增量开发：

- **优先级调度**
  将 BLOSSOM 原有的简单 FIFO 队列逐步演进为支持多策略的优先级调度器，
  实现了与 Linux 兼容的调度语义。

- **LTP 测试兼容性改进**
  修复已知问题并将支持的系统调用的数量从 107 增加到 125，
  将 musl-rv 的 ltp 测试分数从 319 分提高到 649 分。

- **应用**
  将 vi 编辑器、2048 游戏、tetris 游戏成功移植到 NPUcore-Canary 上运行。

本项目严格遵循软件工程规范进行增量开发。截至撰写本文时，NPUcore-Canary 共计进行了 84 次提交（Commit），其中剔除文档维护等非功能性更新后，核心代码的实质性提交为 57 次。

目前，在 2025 年中西部区域赛-内核实现赛道-VisionFive 2 上取得了满分 102 分：

用户名	队伍	学校	提交次数(ASC)	最后提交时间(ASC)	rank
T202510699997582	堆栈溢出幸存者	西北工业大学	4	2026-01-02 15:55:33	102.0000

图 2: 2025 年中西部区域赛-内核实现赛道-VisionFive 2 分数

在 2K1000 开发板上也取得了 102 分：

用户名	队伍	学校	提交次数(ASC)	最后提交时间(ASC)	rank
T202510699997582	堆栈溢出幸存者	西北工业大学	14	2026-01-14 20:11:09	102.0000

图 3: 2025 年中西部区域赛-内核实现赛道-2K1000 分数

这里特别感谢 NPUcore-rainbowww[5] 对 2K1000 平台适配所做的贡献，NPUcore-Canary 参考该仓库进行了适配，才得以在 2K1000 平台上顺利运行。

在 2025 年全国赛测试集[6]上的评测分数如下：

表 1: NPUcore-Canary 在全国赛测试集上的评测结果

测试点	**glibc-la**	**glibc-rv**	**musl-la**	**musl-rv**	**总分**
basic	102	102	102	102	408
busybox	49	48	54	53	204
cyclictest	0	0	0	0	0
iozone	0	0	0	0	0
iperf	0	0	0	0	0
libcbench	0	0	0	0	0
libctest	-	-	216	216	432
lmbench	35.53	36.27	26.95	28.41	127.16
ltp	0	0	618	649	1267
lua	9	9	9	9	36
netperf	0	0	0	0	0
总分	**195.53**	**195.27**	**1025.95**	**1057.41**	**2474.16**

对比 NPUcore-BLOSSOM 在全国赛测试集上的总分 1780.18 分，NPUcore-Canary 提升了 693.98 分，提升幅度达 39%。

二、 NPUcore 内核架构分析

本节将对 NPUcore-Canary 内核项目的基准版本 NPUcore-BLOSSOM 进行全面分析与介绍。NPUcore-BLOSSOM 是西北工业大学 2025 年参加全国大学生计算机系统能力大赛——操作系统设计赛的参赛作品，在吸取了 NPUcore+[7] (2023 年二等奖) 和 NPUcore-IMPACT!!![8] (2024 年一等奖) 两个优秀的 NPUcore 家族作品的经验后，具备了完善的 EXT4 文件系统支持、HAL 层双架构支持（LoongArch64 与 RISC-V）以及丰富的应用程序兼容性。

本节中的插图均来自 NPUcore-BLOSSOM 项目仓库中的设计文档。

2.1 整体架构概述

NPUcore 采用分层架构设计，自底向上分为平台层、硬件抽象层（HAL）、内核层和应用层四个主要层次。这种分层设计使得内核具有良好的可移植性和可维护性。

平台层是系统的基础支撑，为上层提供硬件资源和运行环境。NPUcore 支持多种平台：

- **虚拟平台**：
  QEMU 模拟器（同时支持 LoongArch64 和 RISC-V）
- **物理平台**：
  K210 Board、fu740 Board、VisionFive2 Board（RISC-V）；
  2K1000 Board（LoongArch64）

硬件抽象层（HAL）是 NPUcore-BLOSSOM 团队按照 2025 全国赛要求新增的关键中间层，通过这层抽象，NPUcore 从单一指令集内核转变为同时支持 LoongArch64 和 RISC-V 两种指令集的内核，大大提高了系统的扩展性。HAL 层的核心由 arch 模块、config 模块和 platform 模块组成：

- **arch 模块**：实现不同处理器架构的底层操作，包括寄存器操作、trap 处理、内存管理、上下文切换等
- **config 模块**：为特定架构在 QEMU 虚拟环境下的运行提供参数配置
- **platform 模块**：适配不同硬件平台，提供底层操作接口以屏蔽硬件差异

内核层是系统的核心，负责管理进程、内存、文件系统等关键资源，并提供进程间通信、中断处理等核心功能。

应用层是系统与用户交互的接口，NPUcore 在实现完整系统调用的基础上，能够通过多项测试程序并支持额外的轻量级应用（如 kilo 文本编辑器、终端版俄罗斯方块等）。

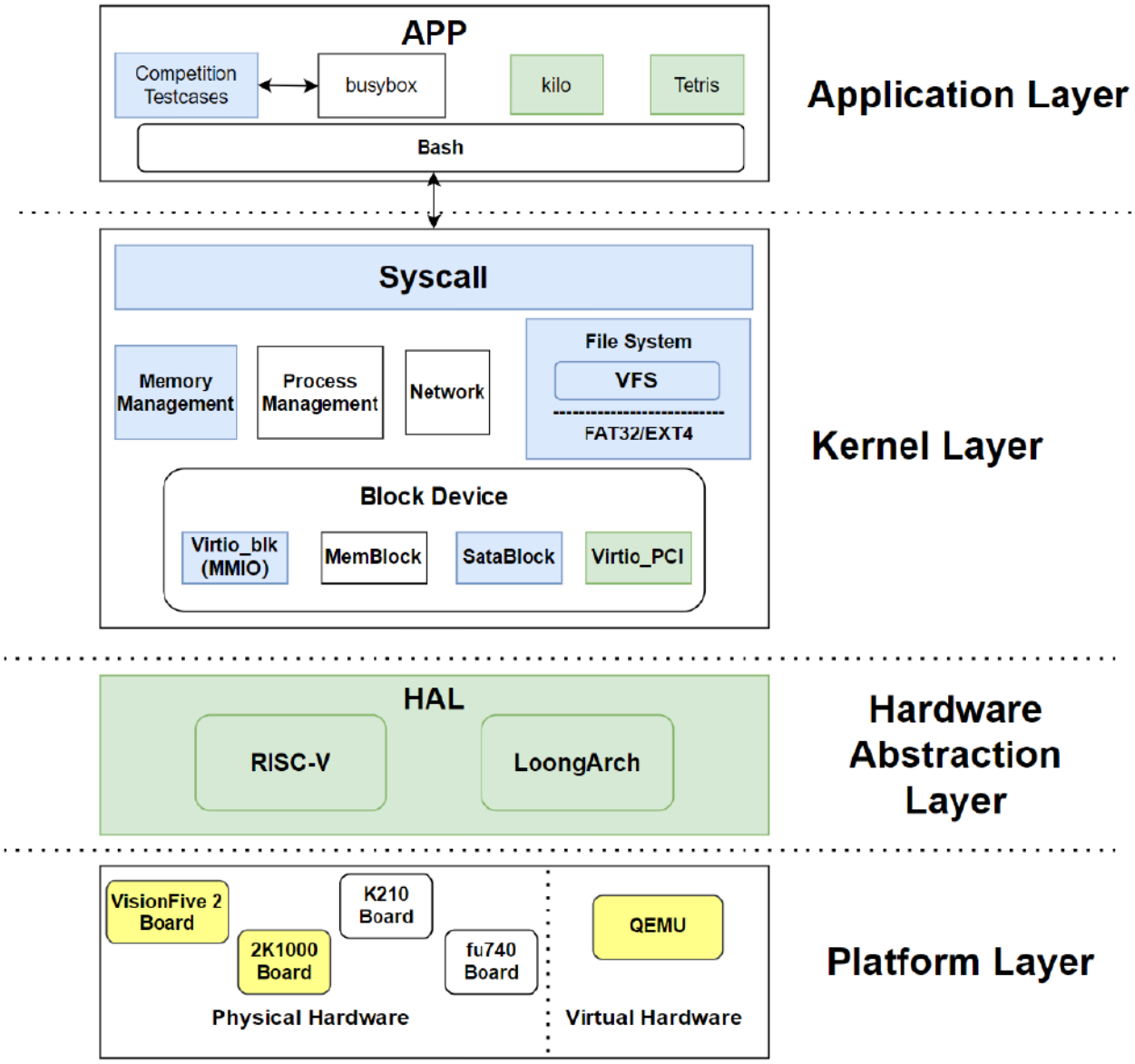

图 4: NPUcore 整体架构图

2.2 进程管理模块

进程是在系统中运行的程序实例，其生命周期包括创建、就绪、阻塞、运行和退出五个阶段。NPUcore 采用如图 5 所示的进程管理流程。

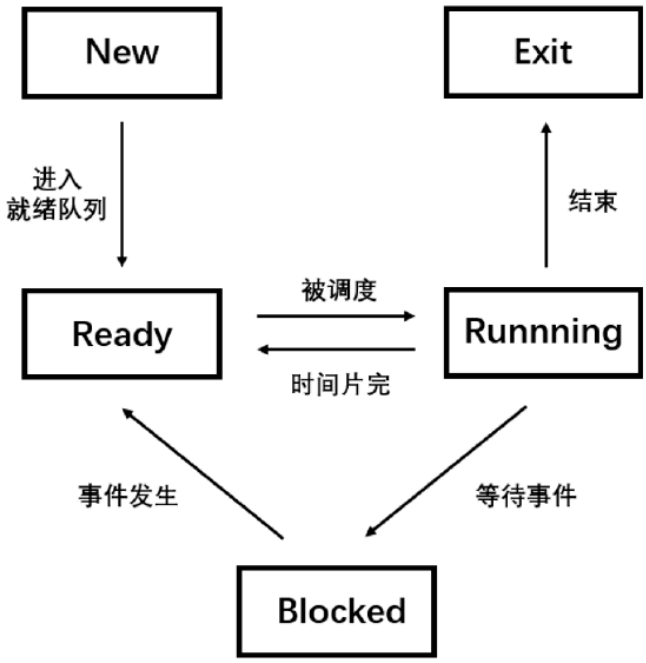

图 5: NPUcore 进程管理流程图

2.2.1 进程的创建

NPUcore 中进程的创建分为三个步骤:

1. **初始进程创建**：内核初始化完毕后，从系统文件中找到 initproc 的 ELF 文件，调用 TaskControlBlock 的 new() 方法创建第一个初始进程
2. **clone 系统调用**：其余所有进程均由初始进程通过 clone 系统调用克隆而来，initproc 是所有进程的祖先
3. **execve 系统调用**：新进程通过 execve 系统调用加载独立的程序代码和数据，完成进程的” 变身”

进程控制块（TaskControlBlock）包含进程的所有状态信息，包括 PID、TID、内核栈、用户栈、文件描述符表、内存映射、信号处理等。采用 RAII（资源获取即初始化）思想管理资源，当进程退出时自动释放所有相关资源。

2.2.2 进程的调度

NPUcore 实现了阻塞式进程调度，核心数据结构包括：

- **TaskManager**：任务管理器，包含就绪队列（ready_queue）和可中断睡眠队列（interruptible_queue）
- **WaitQueue**：等待队列，存放等待特定事件的进程弱引用
- **TimeoutWaitQueue**：超时等待池，使用二叉堆实现，支持定时唤醒

调度器采用轮询机制，在操作系统运行的任一时刻都尝试从 idle 流程切换到下一个进程。进程可通过以下方式被唤醒：

- 操作系统主动唤醒处于 interruptible 状态的进程
- 定时器中断触发，自动唤醒超时等待池中的进程

2.3 内存管理模块

内存管理是操作系统的核心功能之一，NPUcore 实现了完整的虚拟内存机制，包括地址空间抽象、多级页表、物理内存分配等功能。

2.3.1 内核虚拟地址空间

地址空间（Address Space）通过在内核中建立虚拟地址到物理地址的映射机制，为应用程序构建安全、统一的虚拟内存环境。NPUcore 采用图 6 所示的内核虚拟地址空间分配策略。

各区域的功能如下：

- **trampoline**：跳板页面，位于地址空间最高虚拟页面，用于用户态和内核态之间的安全切换
- **User Stack**：用户栈区域，由高地址向低地址增长
- **Guard Page**：保护页面，位于相邻用户栈之间，用于检测栈溢出
- **kernel program**：内核程序加载区，采用恒等映射
- **temporary storage area**：临时存储区，用于 exec 系统调用时的文件加载

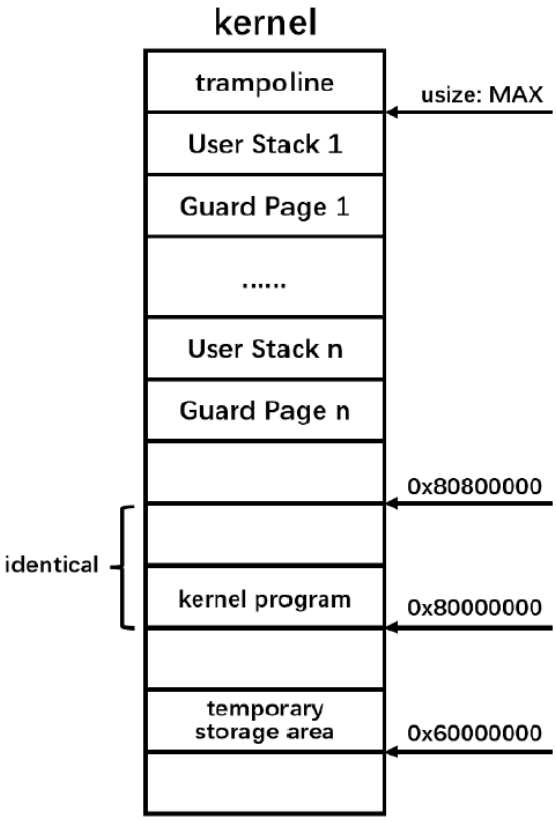

图 6: NPUcore 内核虚拟地址空间图

2.3.2 物理地址空间分布

在 RISC-V 架构下，QEMU 平台的地址空间分布如图所示：

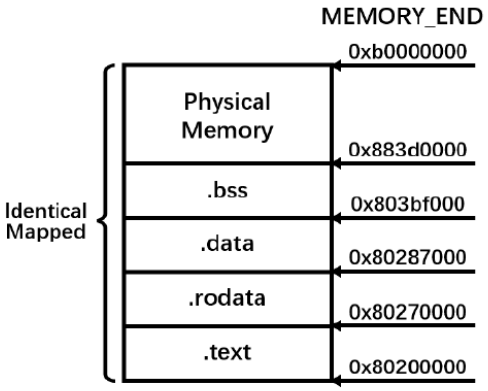

图 7: RISC-V 架构下 QEMU 平台地址分布图

内核采用恒等映射方式映射自身的代码段（.text）、只读数据段（.rodata）、数据段（.data）和 BSS 段（.bss），确保在启用页表机制后仍能直接访问自身各个段。

2.3.3 地址空间数据结构

NPUcore 中地址空间的核心数据结构如图所示：

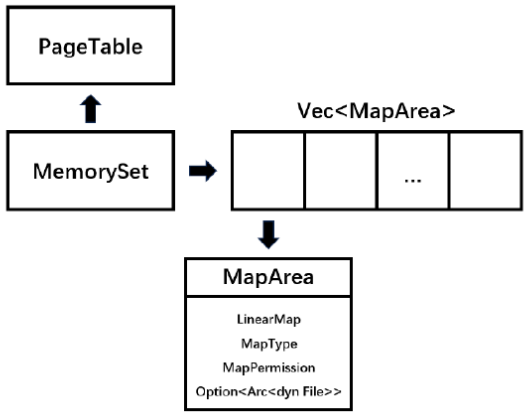

图 8: NPUcore 地址空间数据结构图

地址空间由页表和一系列内存映射区域构成：

- **MemorySet**：地址空间的顶层抽象，包含页表和 MapArea 向量
- **MapArea**：逻辑段，表示一段连续的虚拟内存区域

- **LinearMap**：线性映射，包含虚拟页号范围和物理页帧追踪器
- **FrameTracker**：物理页帧追踪器，实现 RAII 自动资源管理

2.3.4 多级页表机制

在 RISC-V 架构下，NPUcore 实现了 SV39 三级页表机制。SV39 支持 39 位虚拟内存空间，每页 4KB，虚拟地址的高 27 位划分为三级页号，每级 512 个页表项。地址转换过程如图所示：

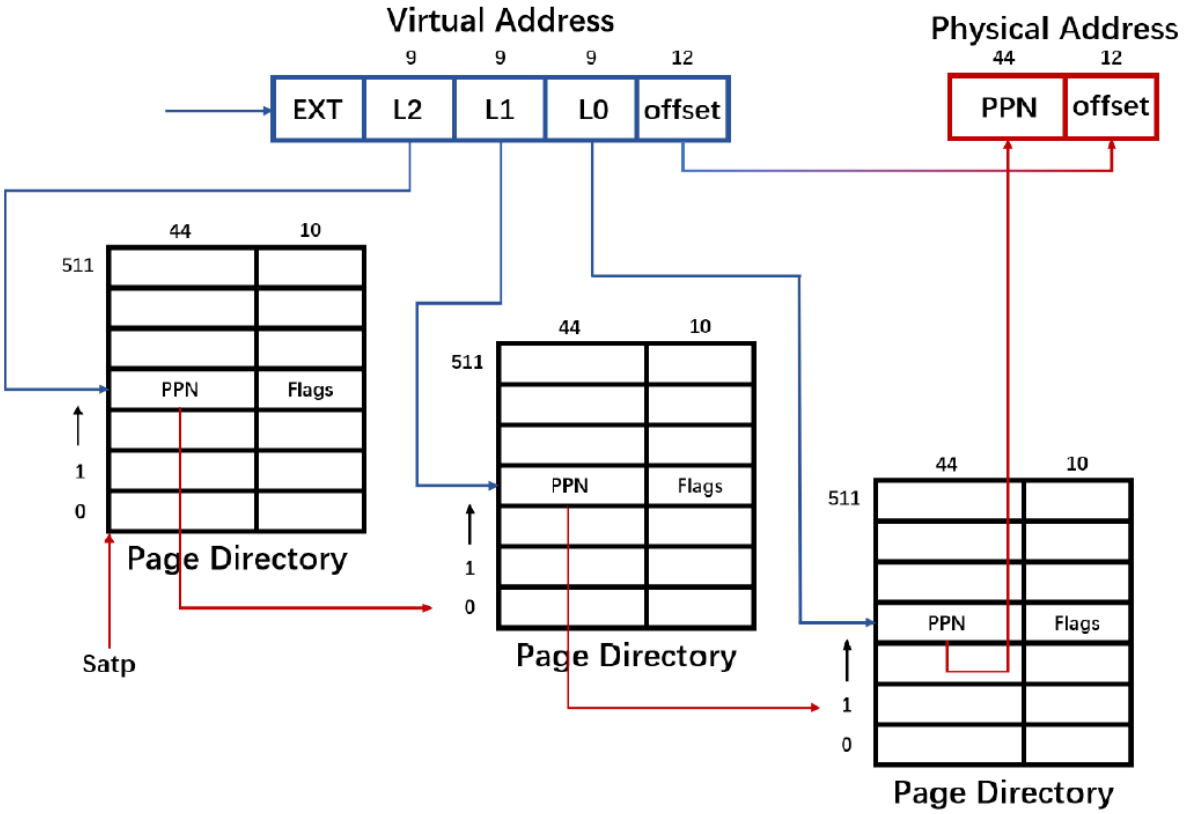

图 9: RISC-V sv39 三级索引地址转换示意图

页表项（PTE）包含物理页号和标志位，标志位包括：

- V（Valid）：有效位，仅当为 1 时页表项合法
- R/W/X（Read/Write/Execute）：控制页面的读/写/执行权限
- U（User）：控制页面是否允许用户态访问
- G（Global）：全局标志位，任务切换时 TLB 不失效

- A（Accessed）：访问位，记录页面是否被访问过
- D（Dirty）：脏位，记录页面是否被修改过

在 LoongArch64 架构下，NPUcore 实现了 LAMex 分页机制，同样支持 52 位虚拟内存空间和多级页表索引。

2.3.5 页面错误处理

NPUcore 实现了完整的页面错误（Page Fault）处理机制，支持懒分配和写时复制（Copy-on-Write）：

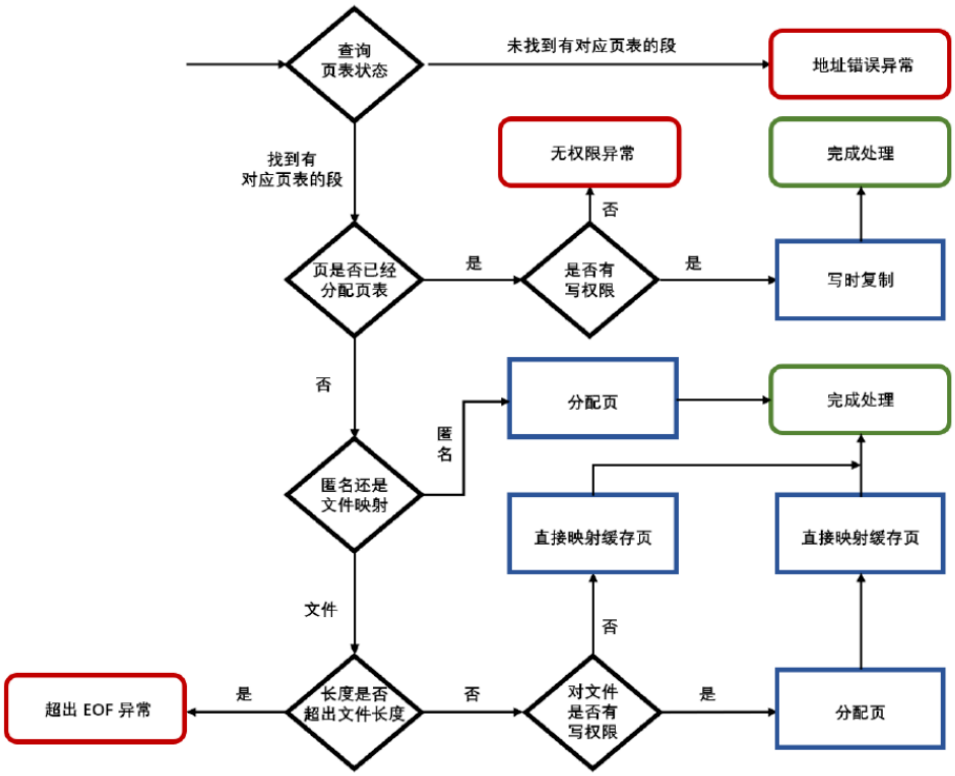

图 10: NPUcore 页面错误处理流程图

页面错误的处理逻辑：

1. 检查发生错误的虚拟地址是否在合法的地址空间范围内
2. 若页面未映射：执行懒分配，分配物理页帧并建立映射
3. 若页面已映射但触发写错误：执行写时复制，复制页面内容到新的物理页帧
4. 若地址非法或权限不足：发送 SIGSEGV 或 SIGBUS 信号

2.3.6 物理内存分配

NPUcore 采用栈式物理内存分配器（StackFrameAllocator），具有以下特点：

- 使用全局物理内存分配器确保内存分配的一致性和线程安全
- 采用” 后进先出” 策略管理空闲物理页帧
- 通过 FrameTracker 实现 RAII，页帧在追踪器析构时自动回收
- 支持不初始化分配（alloc_uninit）以提高性能

2.4 文件系统模块

NPUcore 实现了完整的虚拟文件系统（VFS）层，支持 FAT32 和 EXT4 两种文件系统格式。

2.4.1 文件系统架构

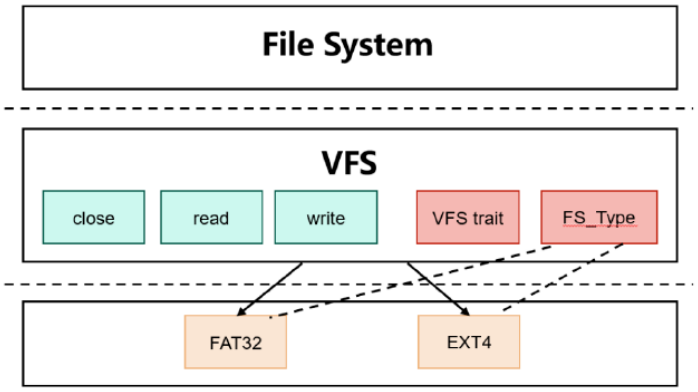

图 11: NPUcore 文件系统架构图

文件系统采用分层设计：

- **VFS 层**：提供统一的文件操作接口，屏蔽底层文件系统差异
- **具体文件系统层**：实现 FAT32 和 EXT4 的具体操作
- **块缓存层**：提供磁盘块的缓存管理
- **块设备层**：与底层存储设备交互

2.4.2 VFS 模块组成

VFS 层包含以下核心模块：

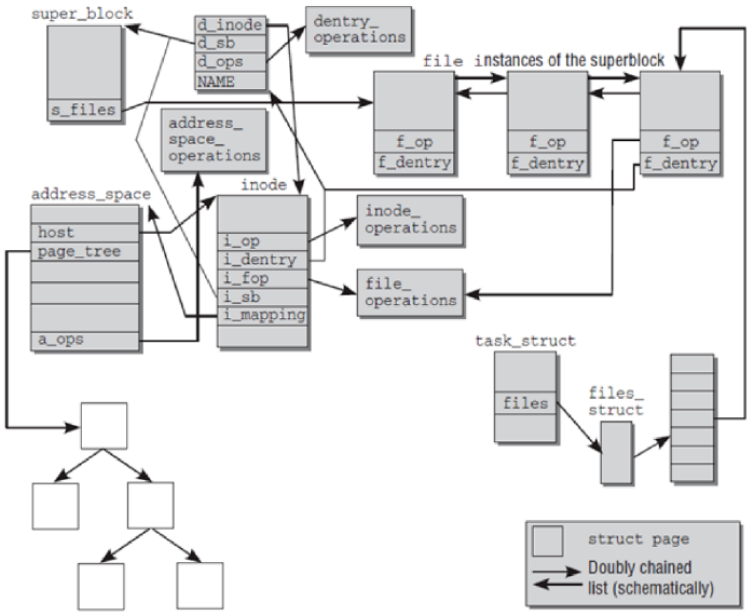

图 12: NPUcore 文件系统模块图

- **超级块（super block）**：保存文件系统的所有元数据
- **目录项模块**：管理路径的目录项，采用树形结构组织
- **inode 模块**：管理具体文件，是文件的唯一标识
- **打开文件列表模块**：维护所有已打开的文件句柄
- **file_operations 模块**：包含所有可用的文件操作函数指针
- **address_space 模块**：管理文件在页缓存中的物理页

2.4.3 文件抽象

NPUcore 定义了统一的 File trait，提供以下核心操作：

- 基本操作：read、write、readable、writable
- 元数据操作：get_size、get_stat、get_file_type
- 目录操作：open、create、unlink、get_dirent

- 偏移管理：lseek、get_offset
- 大小管理：modify_size、truncate_size

支持的文件类型包括：普通文件、目录、管道（FIFO）、字符设备、块设备、套接字和符号链接。

2.5 系统调用模块

系统调用是应用程序与操作系统内核交互的标准接口。NPUcore-BLOSSOM 实现了 POSIX 标准的系统调用接口，支持 90 余个系统调用。

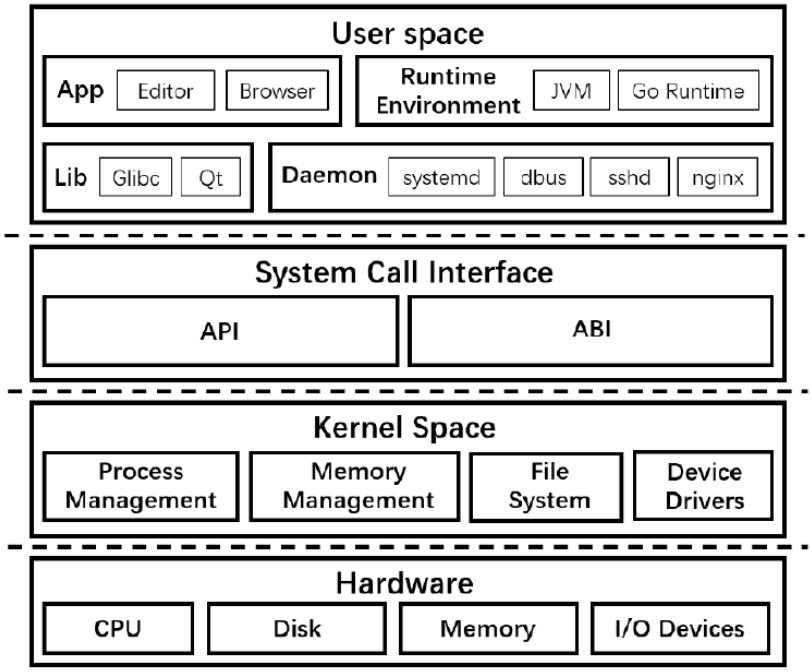

图 13: NPUcore 系统调用整体架构图

2.5.1 系统调用机制

在 RISC-V 架构中，系统调用通过 ecall 指令触发。当用户态程序执行 ecall 指令时，CPU 跳转到由 STVEC 寄存器指定的中断处理程序入口地址。系统调用的具体流程：

1. 用户态程序调用库函数（如 write）
2. 库函数调用 syscall() 函数，将参数放入寄存器
3. 执行 ecall 指令，触发 trap 进入内核态

4. 内核的 trap_handler 根据系统调用号分发处理

5. 执行完成后通过 trap_return 返回用户态

在 LoongArch 架构中，系统调用通过 syscall 指令触发，CPU 自动切换到内核态并跳转到 CSR_EBASE 寄存器指定的处理程序入口。

2.5.2 核心系统调用

NPUcore-BLOSSOM 已实现的核心系统调用包括：

进程管理类：

- fork/clone：创建子进程，支持多种克隆标志
- execve：加载并执行新程序，支持 ELF 可执行文件和脚本文件
- wait4/waitpid：等待子进程状态变化
- exit/exit_group：进程退出

内存管理类：

- brk/sbrk：调整堆空间大小
- mmap/munmap：内存映射和解除映射
- mprotect：修改内存区域保护属性

文件操作类：

- open/close：打开和关闭文件
- read/write：读写文件
- lseek：移动文件读写位置
- dup/dup2/dup3：复制文件描述符
- pipe/pipe2：创建管道

信号处理类：

- sigaction：设置信号处理程序
- sigprocmask：设置信号屏蔽字
- kill：发送信号

NPUcore-Canary 项目在 NPUcore-BLOSSOM 的基础上进行进一步的优化和扩展，后续章节将详细介绍我们所做的改进工作。

三、 优先级调度系统的设计与实现

本节将详细介绍 NPUcore-Canary 对进程调度系统的重构与优化。NPUcore-Canary 将 BLOSSOM 原有的简单 FIFO 队列逐步演进为支持多策略的优先级调度器，实现了与 Linux 兼容的调度语义。

3.1 调度系统演进

NPUcore-BLOSSOM 原有的调度器采用简单的 FIFO（先进先出）队列，所有就绪进程被放入同一个队列中，按照入队顺序依次调度执行。这种设计虽然简单，但无法满足实时任务和交互式任务的需求。

为了提升系统的调度能力和 POSIX 兼容性，我们对调度系统进行了三个阶段的重构：

3.1.1 第一阶段：8 级优先级队列

首先，我们将单一的 FIFO 队列扩展为 8 级优先级队列（优先级 0-7）。每个优先级对应一个独立的就绪队列，调度器总是从最高优先级的非空队列中取出进程执行。

```
pub fn fetch(&self) -> Option<Arc<TaskControlBlock>> {
    for i in (0..8).rev() { // 优先扫描高优先级队列
        if let Some(task) = self.ready_queues[i].pop_front() {
            return Some(task);
        }
    }
    None
}
```

3.1.2 第二阶段：Linux nice 值标准

为了与 Linux 标准兼容，我们将优先级系统从 0-7 的范围迁移到 Linux 的 nice 值标准（-20 到 19），如图 14 所示。nice 值越小，优先级越高。这一改动使得 NPUcore-Canary 能够正确响应用户空间的优先级调整请求。

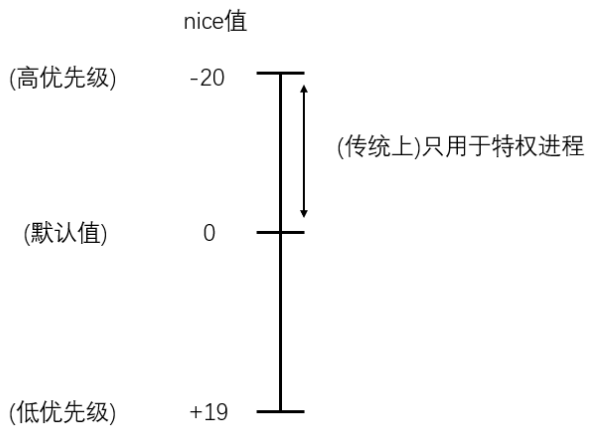

图 14: 进程 nice 值的范围和解释

3.1.3 第三阶段：多策略调度器

最终，我们实现了完整的多策略调度器，支持三类调度队列：

- **实时队列（RT Queue）**：
  包含 99 个优先级级别（1-99），用于 SCHED_FIFO 和 SCHED_RR 策略
- **普通队列（Normal Queue）**：
  包含 40 个优先级级别，用于 SCHED_OTHER/SCHED_NORMAL 策略
- **空闲队列（Idle Queue）**：
  用于 SCHED_IDLE 策略，仅在系统完全空闲时调度

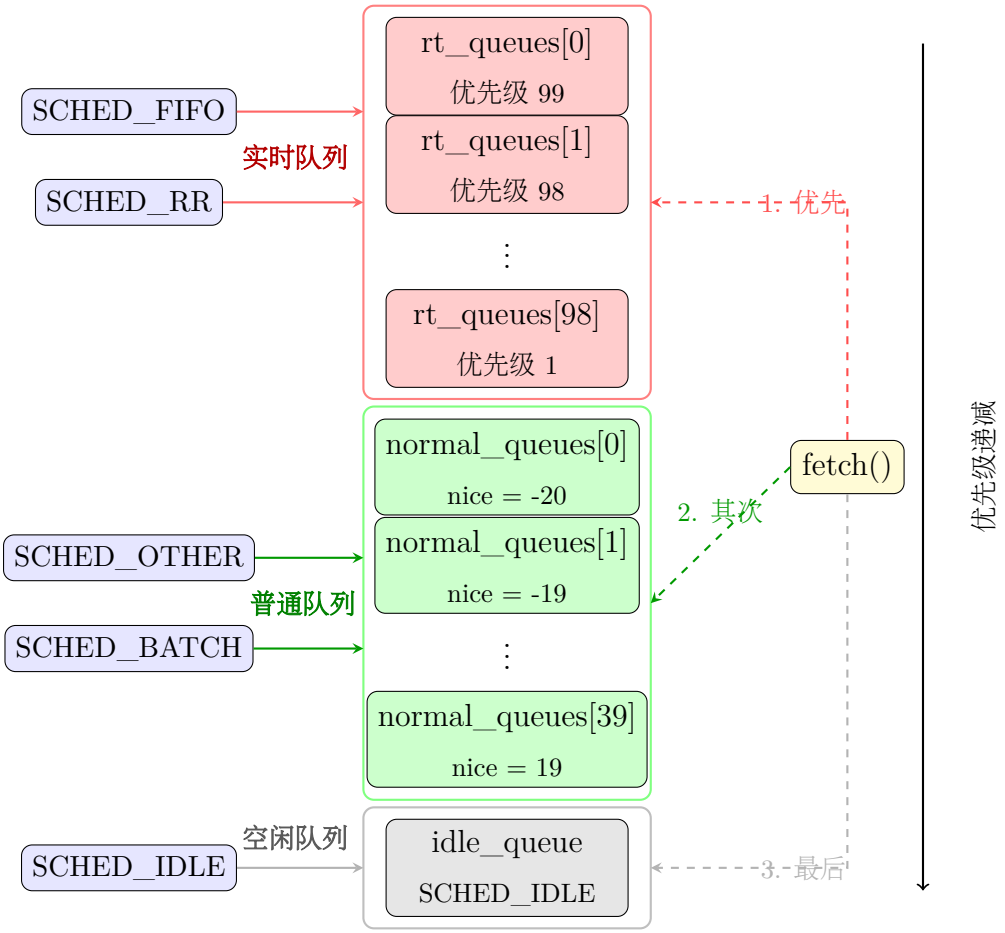

图 15: 多策略调度器架构

3.2 核心数据结构

3.2.1 任务控制块扩展

为支持多策略调度，我们在 TaskControlBlock 中新增了以下字段：

```rust
pub struct TaskControlBlock {
    // ... 原有字段 ...
    /// 调度策略: SCHED_NORMAL, SCHED_FIFO, SCHED_RR, SCHED_IDLE
    pub sched_policy: SchedPolicy,
    /// nice 值，范围 -20 到 19，仅用于 SCHED_NORMAL
    pub nice: i8,
    /// 实时优先级，范围 1-99，用于 SCHED_FIFO/SCHED_RR
    pub rt_priority: u8,
}
```

3.2.2 多级队列结构

```rust
pub struct TaskManager {
    pub rt_queues: [VecDeque<Arc<TaskControlBlock>>;
       RT_PRIORITY_LEVELS],
    pub normal_queues: [VecDeque<Arc<TaskControlBlock>>;
       PRIORITY_LEVELS],
    pub idle_queue: VecDeque<Arc<TaskControlBlock>>,
    pub interruptible_queue: VecDeque<Arc<TaskControlBlock>>,

    pub active_tracker: ActiveTracker,
}
```

TaskManager 是调度器的核心，管理着所有处于就绪状态和等待状态的任务。其中：

- rt_queues: 实时优先级队列数组，包含 99 个双端队列，分别对应实时优先级 1-99。索引 0 对应最高优先级 99。
- normal_queues: 普通优先级队列数组，包含 40 个双端队列，对应 Linux nice 值 -20 到 19。
- idle_queue: 空闲任务队列，存放 SCHED_IDLE 策略的任务，优先级最低。
- interruptible_queue: 可中断等待队列，存放处于睡眠状态等待事件的任务。
- active_tracker: 活跃状态追踪器，用于在内存不足（OOM）时识别近期活跃的进程，避免将其错误回收。

3.3 调度算法实现

3.3.1 进程入队逻辑

```rust
pub fn add(&mut self, task: Arc<TaskControlBlock>) {
    let inner = task.acquire_inner_lock();
    let sched_policy = inner.sched_policy;
    let nice_val = inner.nice;
    let rt_prio = inner.rt_priority;
    drop(inner);

    match SchedClass::from_policy(sched_policy) {
        SchedClass::Realtime => {
```

```
                // SCHED_FIFO 和 SCHED_RR 都使用实时优先级队列
                let index = rt_priority_to_index(rt_prio);
                self.rt_queues[index].push_back(task);
            }
            SchedClass::Normal => {
                // SCHED_OTHER 和 SCHED_BATCH 都使用 nice 值
                let index = priority_to_index(nice_val);
                self.normal_queues[index].push_back(task);
            }
            SchedClass::Idle => {
                // SCHED_IDLE 进入空闲队列
                self.idle_queue.push_back(task);
            }
            SchedClass::Deadline => {
                // TODO: DEADLINE 调度支持
                log::warn!(
                    "[TaskManager::add] DEADLINE scheduling not
                       implemented, pid={} falls back to normal queue",
                    task.pid.0
                );
                self.normal_queues[0].push_back(task);
            }
        }
    }
```

add 方法负责将任务加入特定的就绪队列。它首先获取任务内部锁以读取调度策略、Nice 值和实时优先级，然后根据策略类型（SchedClass）进行分发：

1. **实时任务**：对于 SCHED_FIFO 或 SCHED_RR，根据 rt_priority 计算队列索引，加入 rt_queues。
2. **普通任务**：对于 SCHED_NORMAL，根据 nice 值计算索引，加入 normal_queues。
3. **空闲任务**：对于 SCHED_IDLE，直接加入 idle_queue。

这种基于策略的分发机制确保了高优先级任务和低优先级任务在物理结构上的隔离，使调度器能够以 $O(1)$ 的时间复杂度定位到特定优先级的队列。

3.3.2 进程出队逻辑

```rust
pub fn fetch(&mut self) -> Option<Arc<TaskControlBlock>> {
    // 首先检查实时队列 (SCHED_FIFO/SCHED_RR)
    // rt_queues[0] = rt_priority 99 (最高), rt_queues[98] =
       rt_priority 1 (最低实时)
    for i in 0..RT_PRIORITY_LEVELS {
        if let Some(task) = self.rt_queues[i].pop_front() {
            self.active_tracker.mark_active(task.pid.0);
            return Some(task);
        }
    }

    // 然后检查普通队列 (SCHED_OTHER/SCHED_BATCH)
    for i in 0..PRIORITY_LEVELS {
        if let Some(task) = self.normal_queues[i].pop_front() {
            self.active_tracker.mark_active(task.pid.0);
            return Some(task);
        }
    }

    // 最后检查空闲队列 (SCHED_IDLE)
    if let Some(task) = self.idle_queue.pop_front() {
        self.active_tracker.mark_active(task.pid.0);
        return Some(task);
    }

    None
}
```

`fetch` 方法实现了**严格优先级调度**逻辑，其执行顺序如下：

1. **扫描实时队列**：优先遍历 `rt_queues`。由于实时优先级 99 对应索引 0，因此从前往后的遍历顺序自然保证了最高优先级的实时任务最先被调度。
2. **扫描普通队列**：若无实时任务，则遍历 `normal_queues`。同样遵循 Nice 值越小（优先级越高）越先调度的原则。
3. **扫描空闲队列**：仅当上述所有队列均为空时，才从 `idle_queue` 中获取任务。

一旦成功获取任务，调度器会立即调用 `active_tracker.mark_active` 标记该进程为活跃状态，这一机制有效防止了关键进程在系统内存紧张时被 OOM Killer 误杀。

3.4 相关系统调用

为了让用户空间程序能够控制调度行为，我们实现了以下系统调用：

系统调用号	名称	功能
118	sched_setscheduler	设置进程的调度策略和参数
119	sched_getscheduler	获取进程的调度策略
120	sched_setparam	设置进程的调度参数
121	sched_getparam	获取进程的调度参数
125	sched_get_priority_max	获取指定策略的最大优先级
126	sched_get_priority_min	获取指定策略的最小优先级
140	setpriority	设置进程/进程组/用户的优先级
141	getpriority	获取进程/进程组/用户的优先级

表 2: 调度相关系统调用

3.4.1 调度策略设置 (sched_setscheduler)

`sched_setscheduler` 系统调用用于设置进程的调度策略和实时优先级。其核心逻辑如下：

```
pub fn sys_sched_setscheduler(pid: usize, policy: i32, param: *const
    SchedParam) -> isize {
    // 验证调度策略合法性
    if !matches!(policy, SCHED_OTHER | SCHED_FIFO | SCHED_RR |
        SCHED_BATCH | SCHED_IDLE) {
        return EINVAL;
    }

    // 验证实时优先级范围
    let rt_prio = unsafe { (*param).sched_priority };
    match policy {
        SCHED_FIFO | SCHED_RR => {
            // 实时策略要求优先级在 1-99 之间
            if rt_prio < 1 || rt_prio > 99 { return EINVAL; }
        }
        _ => {
            // 普通策略要求实时优先级为 0
            if rt_prio != 0 { return EINVAL; }
        }
```

```
    }

    // 查找进程并更新 TCB
    if let Some(task) = find_task_by_tgid(pid) {
        let mut inner = task.acquire_inner_lock();
        inner.sched_policy = policy as u8;
        inner.rt_priority = rt_prio as u8;
        return SUCCESS;
    }
    ESRCH
}
```

该实现确保了实时调度策略必须配置有效的实时优先级（1-99），而普通调度策略（如 SCHED_OTHER）则必须将实时优先级置为 0，严格遵循 POSIX 标准。

3.4.2 优先级设置 (setpriority)

setpriority 系统调用用于调整进程、进程组或特定用户的 Nice 值（静态优先级）。系统会对用户传入的优先级值进行边界检查，确保其限制在 valid 范围内（-20 到 19）：

```
pub fn sys_setpriority(which: i32, who: isize, prio: i32) -> isize {
    // 将优先级截断至 [-20, 19] 范围
    let prio = prio.clamp(PRIORITY_MIN as i32, PRIORITY_MAX as i32)
        as i8;

    match which {
        PRIO_PROCESS => { // 设置单个进程
            // 查找并更新指定进程的 nice 值
        }
        PRIO_PGRP => { // 设置进程组
            // 遍历进程组 update 每一项
        }
        PRIO_USER => { // 设置用户所有进程
             // 遍历该用户所有任务
        }
        _ => EINVAL,
    }
}
```

四、 LTP 测试兼容性改进

本节将介绍 NPUcore-Canary 为提升 LTP（Linux Test Project）测试通过率所做的一系列改进。通过系统性地分析测试失败原因并逐一修复，我们将 musl-rv 下的 LTP 测试分数从 319 分提升至 649 分。

4.1 LTP 测试概述

LTP 是 Linux 内核的标准测试套件，包含数千个测试用例，覆盖系统调用、文件系统、内存管理、进程管理等各个方面。通过 LTP 测试是验证操作系统 POSIX 兼容性的重要手段。

4.2 目前最大的问题

目前跑 LTP 测试到一半的时候会出现类似这样的错误：

```
[kernel] panicked at 'Heap allocation error, layout = Layout { size:
    4096, align: 1 (1 << 0) }', src/mm/heap_allocator.rs:12:5
```

经过分析，我们发现这是由于内存泄漏导致的内存耗尽问题。该问题导致无法一次性测试所有能通过的 ltp 测试样例，只能选择性地跑一部分测例。

目前 NPUcore-Canary 的临时方案是把 KERNEL_HEAP_SIZE 从 128MB 提升到 256MB 以规避该问题，但这并非根本解决方案。未来如果需要进一步提升 LTP 测试分数，仍需定位并修复内存泄漏问题。

4.3 /proc

Linux 的 /proc 文件系统提供了访问内核数据结构的接口，许多用户空间程序依赖它获取进程信息。NPUcore-BLOSSOM 对 /proc 的支持较为有限，导致多个 LTP 测试失败。

4.3.1 /proc/self/maps

`/proc/self/maps` 文件显示当前进程的内存映射信息，格式如下：

```
起始地址-结束地址 权限 偏移 设备号 inode 路径名
# 示例：
00400000-00452000 r-xp 00000000 08:02 173521 /usr/bin/dbus-daemon
```

我们实现了 `ProcMaps` 结构体，通过遍历进程的内存区域动态生成映射信息：

```rust
pub struct ProcMaps {
    pub pid: Option<usize>,  // None 表示当前进程
    pub offset: Mutex<usize>,
}

impl ProcMaps {
    fn get_maps_string(&self) -> String {
        let task = if let Some(pid) = self.pid {
            find_task_by_pid(pid)
        } else {
            current_task()
        };
        let vm = task.vm.lock();
        let mut result = String::new();

        for area in vm.get_areas() {
            let start_addr = area.inner.get_start().0 * PAGE_SIZE;
            let end_addr = area.inner.get_end().0 * PAGE_SIZE;

            // 构建权限字符串 (r/w/x + s/p)
            let mut perms = String::with_capacity(4);
            let perm = area.map_perm;
            perms.push(if perm.contains(MapPermission::R) { 'r' }
                else { '-' });
            perms.push(if perm.contains(MapPermission::W) { 'w' }
                else { '-' });
            perms.push(if perm.contains(MapPermission::X) { 'x' }
                else { '-' });
            perms.push(if area.is_shared { 's' } else { 'p' });

            result.push_str(&format!(
                "{:08x}-{:08x} {} {:08x} {:02x}:{:02x} {} {}\n",
                start_addr, end_addr, perms, offset,
                dev_major, dev_minor, inode, pathname
            ));
        }
        result
```

```
    }
}
```

4.3.2 /proc/[pid]/stat

/proc/[pid]/stat 文件以单行形式提供进程状态信息，包括 PID、进程名、状态、父进程 PID、进程组 ID 等字段。这对于 getpgid01、wait401 等测试至关重要。

```rust
pub struct ProcPidStat {
    pub pid: usize,
    pub offset: Mutex<usize>,
}

impl ProcPidStat {
    fn get_stat_string(&self) -> String {
        if let Some(task) = find_task_by_pid(self.pid) {
            let inner = task.acquire_inner_lock();
            let pid = task.pid.0;
            let ppid = inner.parent.as_ref()
                .and_then(|p| p.upgrade())
                .map(|p| p.pid.0).unwrap_or(0);
            let pgid = inner.pgid;

            // 进程状态映射
            let state = match inner.task_status {
                TaskStatus::Ready | TaskStatus::Running => 'R',
                TaskStatus::Interruptible => 'S',
                TaskStatus::Zombie => 'Z',
            };

            // Linux stat 格式: pid (comm) state ppid pgrp session ...
            format!("{} (process) {} {} {} {} 0 0 ...\n",
                pid, state, ppid, pgid, task.tgid)
        } else {
            String::new()
        }
    }
}
```

4.3.3 /proc/[pid]/oom_score_adj

OOM（Out of Memory）killer 使用 oom_score_adj 来调整进程被杀死的优先级。我们在 TCB 中添加了 oom_score_adj 字段（范围 -1000 到 1000），并实现了对应的读写接口。

```rust
/// 范围: -1000 (永不被杀) 到 1000 (优先被杀)
pub struct OomScoreAdj {
    pub pid: Option<usize>,
    pub offset: Mutex<usize>,
}

impl OomScoreAdj {
    fn get_oom_score_string(&self) -> String {
        if let Some(task) = self.get_target_task() {
            let inner = task.acquire_inner_lock();
            format!("{}\n", inner.oom_score_adj)
        } else {
            String::from("0\n")
        }
    }

    fn set_oom_score_from_string(&self, input: &str) -> Result<(), ()
        > {
        let value = input.trim().parse::<i16>().map_err(|_| ())?;
        if value >= -1000 && value <= 1000 {
            if let Some(task) = self.get_target_task() {
                task.acquire_inner_lock().oom_score_adj = value;
                return Ok(());
            }
        }
        Err(())
    }
}

// TCB 中的字段定义
pub struct TaskControlBlockInner {
    // ...
```

```
32      pub oom_score_adj: i16,  // 默认为 0，fork 时继承父进程的值
33  }
```

4.4 时间子系统完善

时间相关功能是许多 LTP 测试的基础，我们进行了以下改进:

4.4.1 Goldfish RTC 驱动

QEMU 平台提供了 Goldfish RTC 设备用于获取真实世界时间。我们实现了该驱动，使内核能够在启动时同步 Unix 时间戳，而不是从 0 开始计时。

```
1  /// 初始化Unix时间（从Goldfish RTC读取）
2  pub fn init_unix_time() {
3      // Goldfish RTC寄存器: 0x00=TIME_LOW, 0x04=TIME_HIGH
4      const GOLDFISH_RTC_BASE: usize = 0x10_1000;
5
6      unsafe {
7          let time_low = read_volatile(GOLDFISH_RTC_BASE as *const u32)
               as u64;
8          let time_high = read_volatile((GOLDFISH_RTC_BASE + 4) as *
              const u32) as u64;
9          let rtc_ns = (time_high << 32) | time_low;
10         let rtc_sec = (rtc_ns / 1_000_000_000) as usize;
11
12         let uptime_sec = get_time() / get_clock_freq();
13         let boot_unix_time = rtc_sec.saturating_sub(uptime_sec);
14         UNIX_TIME_OFFSET.store(boot_unix_time, AtomicOrdering::
              Relaxed);
15     }
16 }
```

4.4.2 64 位时间戳支持

ext4 文件系统的 inode 结构中，时间戳字段（atime、mtime、ctime）使用 32 位存储秒数，额外的 `i_*_extra` 字段存储纳秒和扩展纪元。我们完善了这部分的解析逻辑，支持完整的 64 位时间戳，避免 2038 年问题。

4.4.3 clock_nanosleep 改进

修复了 clock_nanosleep 系统调用的参数验证：

- 验证 tv_nsec 范围（0 到 999999999）
- 对不支持的时钟类型返回 ENOTSUP

4.5 文件系统改进

4.5.1 ext4 ftruncate 零填充

POSIX 标准要求：当使用 ftruncate 扩展文件大小时，扩展部分必须填充为零。原实现仅更新 inode 大小，未对扩展区域进行零填充。我们添加了 clear_at_block_cache() 方法实现这一功能：

```
// truncate 时，若文件扩展则清零扩展区域
if new_size > old_size {
    self.clear_at_block_cache(old_size, new_size - old_size);
}

fn clear_at_block_cache(&self, offset: usize, length: usize) -> usize
    {
    let mut start = offset;
    let end = offset + length;
    loop {
        let end_current_block = ((start / CACHE_SZ + 1) * CACHE_SZ).
            min(end);
        let block_write_size = end_current_block - start;
        // 将块缓存中对应区域填充为 0
        self.file_cache_manager.get_cache(...).lock()
            .modify(0, |data: &mut [u8; PAGE_SIZE]| {
                data[start % CACHE_SZ..][..block_write_size].fill(0);
            });
        if end_current_block == end { break; }
        start = end_current_block;
    }
}
```

4.5.2 chown 实现

实现了完整的 fchownat 系统调用：

```rust
pub fn sys_fchownat(dirfd: i32, pathname: *const u8,
                    owner: u32, group: u32, _flags: i32) -> isize {
    // ... 路径解析 ...
    let stat = target_file.file.get_stat();
    // -1 (0xFFFFFFFF) 表示保持原值不变
    let new_uid = if owner == u32::MAX { stat.get_uid() } else {
        owner };
    let new_gid = if group == u32::MAX { stat.get_gid() } else {
        group };
    target_file.file.set_owner(new_uid, new_gid)
}
```

4.5.3 symlinkat 系统调用

实现了 symlinkat 系统调用（syscall 36），支持创建符号链接。

4.5.4 路径验证增强

添加了完整的路径验证逻辑：

- PATH_MAX（4096）：路径总长度限制
- NAME_MAX（255）：单个路径组件长度限制
- 符号链接循环检测，超过限制返回 ELOOP

4.6 内存管理增强

4.6.1 MAP_SHARED 支持

实现了 mmap 的 MAP_SHARED 标志支持。在 MapArea 中添加标志位，fork 时对共享映射区域共享物理页面而非写时复制：

```rust
pub struct MapArea {
    pub is_shared: bool,      // MAP_SHARED 标志
    pub wipe_on_fork: bool,   // MADV_WIPEONFORK 标志
    // ...
}
```

```

// fork 时的处理逻辑
if src_area.is_shared {
    // 共享映射：直接共享物理页面
    new_area.map_shared(src_area);
} else if src_area.wipe_on_fork {
    // WIPEONFORK：子进程获得零填充页面
    new_area.map_zeroed();
} else {
    // 私有映射：写时复制（COW）
    new_area.map_cow(src_area);
}
```

4.6.2 madvise 改进

实现了 MADV_WIPEONFORK 和 MADV_KEEPONFORK 标志：

```
pub fn madvise(&mut self, addr: usize, length: usize, advice: u32) ->
    Result<(), isize> {
    const MADV_WIPEONFORK: u32 = 18;
    const MADV_KEEPONFORK: u32 = 19;

    match advice {
        MADV_WIPEONFORK | MADV_KEEPONFORK => {
            let wipe = advice == MADV_WIPEONFORK;
            for area in self.areas.iter_mut() {
                // 仅对私有匿名页生效，跳过共享/文件映射
                if area.is_shared || area.map_file.is_some() {
                   continue; }
                area.wipe_on_fork = wipe;
            }
            Ok(())
        }
        // ...
    }
}
```

4.6.3 brk 系统调用修复

修复了 brk 系统调用的若干问题：

- 正确处理非页对齐的堆增长
- 添加溢出保护

4.7 系统调用修复

4.7.1 dup2/dup3 修复

- 修正 SYSCALL_DUP3 的系统调用号（从 20 改为 24）
- 修复 newfd 无效时应返回 EBADF 而非 EMFILE
- 在新架构上，glibc/musl 将 dup2 转换为 dup3 调用

4.7.2 statx 改进

- 添加 STATX__RESERVED 掩码验证
- 支持 AT_EMPTY_PATH 标志
- 支持 AT_SYMLINK_NOFOLLOW 标志

4.7.3 fcntl 文件锁

实现了 F_GETLK、F_SETLK、F_SETLKW 命令，定义了与 Linux ABI 兼容的 Flock 结构：

```
pub const F_RDLCK: i16 = 0;  // 读锁（共享）
pub const F_WRLCK: i16 = 1;  // 写锁（独占）
pub const F_UNLCK: i16 = 2;  // 解锁

#[repr(C)]
pub struct Flock {
    pub l_type: i16,    // 锁类型
    pub l_whence: i16,  // 起始位置基准
    pub l_start: i64,   // 锁起始偏移
    pub l_len: i64,     // 锁定字节数，0 表示到文件末尾
    pub l_pid: i32,     // 持有锁的进程 PID（F_GETLK 返回）
}
```

4.7.4 新增系统调用

系统调用号	名称	功能
33	mknodat	创建设备文件或 FIFO
36	symlinkat	创建符号链接
44	fstatfs	获取文件系统统计信息
47	fallocate	预分配文件空间
81	sync	同步文件系统缓存
92	personality	设置进程执行域
114	clock_getres	获取时钟分辨率
144	setgid	设置进程组 ID
146	setuid	设置进程用户 ID

表 3: 新增系统调用

4.8 错误处理规范化

为了符合 POSIX 标准，我们对错误处理进行了全面的规范化：

4.8.1 标准错误码

- `pread/pwrite`：负偏移返回 `EINVAL`
- `splice`：文件到文件的 splice 返回 `EINVAL`
- `getcwd`：按 POSIX 标准调整错误检查顺序

4.8.2 边界检查与溢出保护

- `translated_byte_buffer`：添加地址溢出保护
- `contains_valid_buffer`：对溢出地址返回 false

五、 Vi 编辑器适配

为了提升 NPUcore 的实用性，我们将轻量级 vi 编辑器移植到了系统中。这使得用户可以直接在 NPUcore 上进行文本编辑，极大地增强了系统的交互能力。

5.1 移植背景

在操作系统开发和测试过程中,文本编辑器是不可或缺的工具。虽然 NPUcore-BLOSSOM 已经提供了 kilo 编辑器，但 vi 作为 Unix/Linux 世界中最经典的编辑器，具有以下优势：

- 功能更加完善，支持撤销/重做、搜索替换、标记跳转等高级功能
- 符合 POSIX 标准，是许多脚本和工具的默认编辑器
- 用户基础广泛，学习成本低

5.2 代码来源

我们选择从 pshell 项目[9] 移植 vi 编辑器。该项目已经将 BusyBox 中的 vi 实现提取为独立可编译的版本，原始代码约 3800 行 C 代码。

BusyBox vi 的原始版权信息：

- 作者：Sterling Huxley
- 许可证：GPLv2 or later

5.3 适配工作

pshell 项目的 vi 原本是为 Raspberry Pi Pico 设计的，依赖 Pico SDK 和 littlefs 文件系统。为了在 NPUcore 上运行，我们进行了以下适配工作：

5.3.1 移除硬件依赖

原代码依赖以下 Pico SDK 头文件，需要全部移除：

```
// 移除的依赖
#include "hardware/timer.h"
#include "pico/stdio.h"
#include "pico/time.h"
#include "io.h"
```

替换为标准 POSIX 头文件：

```c
// 添加的 POSIX 头文件
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <termios.h>
#include <poll.h>
```

5.3.2 文件系统 API 替换

原代码使用 littlefs 文件系统 API，需要替换为 POSIX 标准调用：

原 API (littlefs)	新 API (POSIX)
`fs_file_open()`	`open()`
`fs_file_read()`	`read()`
`fs_file_write()`	`write()`
`fs_file_close()`	`close()`
`fs_stat()`	`stat()`
`lfs_file_t`	`int` (文件描述符)
`struct lfs_info`	`struct stat`

表 4: 文件系统 API 映射表

5.3.3 终端处理

终端是 vi 编辑器的核心交互界面，需要进行以下适配：

获取终端尺寸

原代码使用 Pico 特定的 `get_screen_xy()` 函数，我们使用 `ioctl` 系统调用替代：

```c
static void get_screen_xy(uint32_t* cols, uint32_t* row) {
    struct winsize ws;
    if (ioctl(STDOUT_FILENO, TIOCGWINSZ, &ws) == 0
        && ws.ws_col > 0 && ws.ws_row > 0) {
        *cols = ws.ws_col;
        *row = ws.ws_row;
    } else {
        *cols = 80;  // 默认值
        *row = 24;
```

```
    }
}
```

键盘输入处理

原代码使用 `getchar_timeout_us()` 进行带超时的输入读取。我们使用 `poll()` + `read()` 组合实现相同功能:

```
static int safe_poll(uint8_t* buffer, int ms) {
    if (ms < 0) {
        // 阻塞读取
        ssize_t n = read(STDIN_FILENO, buffer, 1);
        return (n > 0) ? 1 : 0;
    } else {
        // 带超时的非阻塞读取
        struct pollfd pfd;
        pfd.fd = STDIN_FILENO;
        pfd.events = POLLIN;
        int ret = poll(&pfd, 1, ms);
        if (ret > 0 && (pfd.revents & POLLIN)) {
            ssize_t n = read(STDIN_FILENO, buffer, 1);
            return (n > 0) ? 1 : 0;
        }
        return 0;
    }
}
```

终端原始模式

vi 需要逐字符读取用户输入，必须将终端设置为原始模式（raw mode）:

```
static void enable_raw_mode(void) {
    struct termios raw;

    if (tcgetattr(STDIN_FILENO, &orig_termios) == 0) {
        termios_saved = 1;
        atexit(disable_raw_mode);

        raw = orig_termios;
        // 禁用输入处理
        raw.c_iflag &= ~(BRKINT | ICRNL | INPCK | ISTRIP | IXON);
```

```
        // 禁用输出处理
        raw.c_oflag &= ~(OPOST);
        // 8位字符
        raw.c_cflag |= (CS8);
        // 禁用回显和规范模式
        raw.c_lflag &= ~(ECHO | ICANON | IEXTEN | ISIG);

        tcsetattr(STDIN_FILENO, TCSAFLUSH, &raw);
    }
}
```

5.3.4 其他修改

- 移除 UDATA 节属性宏，使用普通静态变量
- 重命名 argc/optind 变量避免与标准库冲突
- 实现简化版 full_path() 函数
- 添加 main() 入口函数

5.4 功能验证

我们通过实际运行测试验证了 vi 编辑器的核心功能。

首先，在命令行启动 vi 并指定文件名 test.txt：

```
    vi-riscv64 test.txt
```

如图 16 所示，进入编辑器后可以正常进行文本输入（插入模式）。随后使用 :wq 命令保存并退出，如图 17 所示。最后，通过 cat 命令查看文件内容，确认数据已正确写入磁盘，如图 18 所示。

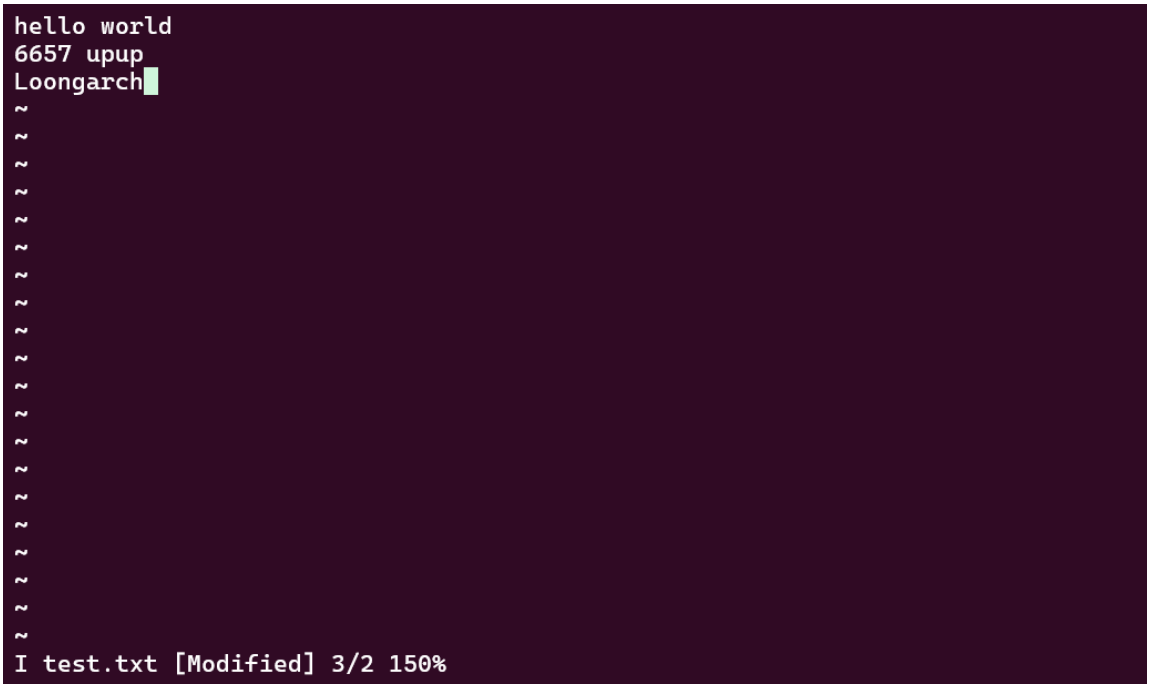

图 16: 在 vi 中编辑文件内容

图 17: 使用:wq 命令保存并退出

```
NPUCore:/# vi-riscv64 test.txt
NPUCore:/# cat test.txt
hello world
6657 upup
Loongarch
NPUCore:/#
```

图 18: 验证文件内容正确写入

5.5 其他应用

除了 vi 编辑器，我们还成功将 2048 游戏移植到了 NPUcore:

图 19: 2048 游戏运行截图

成功将 tetris 俄罗斯方块游戏移植到了 NPUcore:

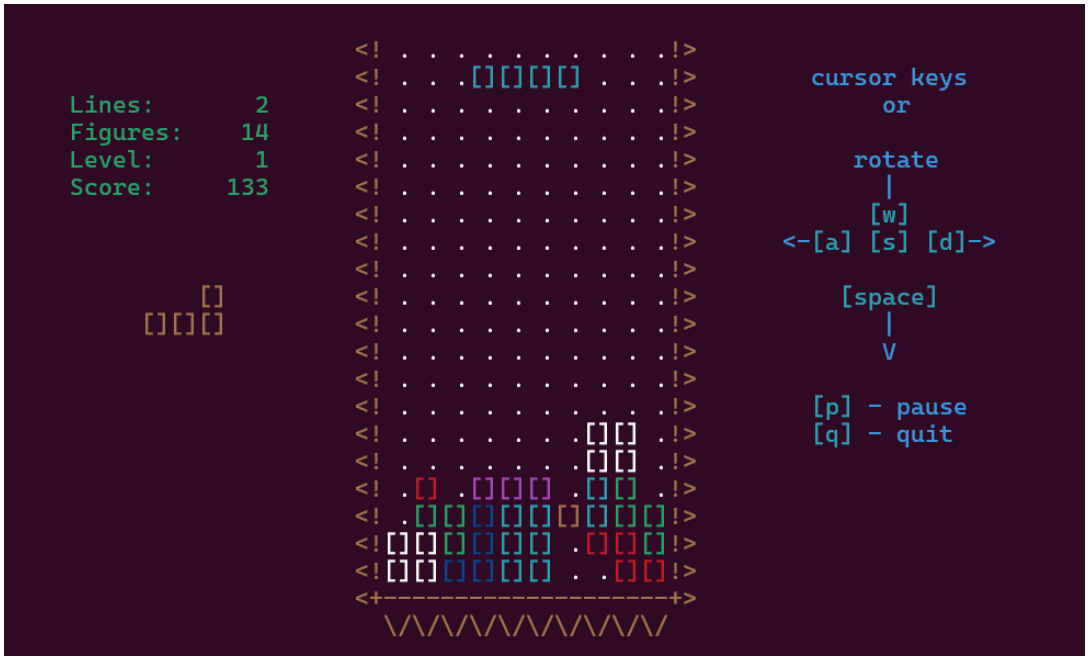

图 20: tetris 俄罗斯方块游戏运行截图

六、 总结与展望

6.1 工作总结

NPUcore-Canary 在 NPUcore-BLOSSOM 的基础上进行了系统性的改进与优化，主要工作包括：

6.1.1 优先级调度系统

重构了进程调度子系统，从简单的 FIFO 队列演进为支持多策略的优先级调度器：

- 实现了 99 级实时优先级队列和 40 级普通优先级队列
- 支持 SCHED_FIFO、SCHED_RR、SCHED_NORMAL、SCHED_IDLE 等调度策略
- 兼容 Linux nice 值标准（-20 到 19）
- 实现了完整的调度相关系统调用

6.1.2 POSIX 兼容性提升

通过 LTP 测试驱动的开发方式，系统性地提升了 POSIX 兼容性：

- LTP 测试分数从 319 分提升至 649 分
- 实现了 /proc 虚拟文件系统的关键接口
- 完善了时间子系统，支持 RTC 和 64 位时间戳
- 修复了文件系统、内存管理、系统调用的多个问题
- 规范化了错误处理逻辑

6.1.3 代码质量

- 保持了 Rust 语言的安全特性和 RAII 设计模式
- 完善了日志和调试信息
- 改进了代码结构和可维护性

6.2 未来展望

NPUcore-Canary 仍有许多可以改进的方向：

6.2.1 功能完善

- 完善网络协议栈，支持更多网络应用
- 实现更多的 /proc 和 /sys 虚拟文件
- 支持更多的文件系统类型
- 完善信号机制和进程间通信

6.2.2 性能优化

- 优化调度器的时间复杂度
- 改进文件系统缓存策略
- 优化内存分配器
- 支持多核调度

6.2.3 测试与文档

- 继续提升 LTP 测试通过率
- 添加单元测试和集成测试
- 完善技术文档和用户文档

6.3 致谢

在 NPUcore-Canary 的设计与实现过程中，我得到了指导老师张羽的悉心指导与帮助。老师深厚的学术造诣和丰富的工程经验为项目的顺利推进提供了坚实保障，在关键技术难点的攻关上给予了极具价值的建议。

同时，衷心感谢 NPUcore 团队的前辈们——林祥霖、郭晥、栾承澈、张家文……，他们在项目开发过程中提供的技术支持与宝贵建议，极大地拓宽了我的视野并帮助解决了诸多难题。

此外，特别感谢 NPUcore-BLOSSOM 团队提供的优秀开源基础代码，这为本项目的进一步扩展与优化奠定了良好的基石；以及 NPUcore-rainbowww 在 2K1000 平台适配方面的贡献，使得 NPUcore-Canary 能够顺利运行于该平台。

最后，感谢全国大学生计算机系统能力大赛组委会提供的学习平台与竞技机会，让我们能够在实践中深入理解操作系统原理，提升系统编程能力。

参考文献

[1] NPUcore-BLOSSOM 项目仓库[EB/OL]. 2025. https://gitlab.eduxiji.net/T202510699995278/oskernel2025-npucore-blossom.

[2] 张羽, 郭晥. 基于 Rust 语言的 NPUcore 操作系统内核构建实践[M]. 西安: 西北工业大学出版社, 2024.

[3] RocketOS 项目仓库[EB/OL]. 2025. https://gitlab.eduxiji.net/educg-group-32146-2710490/T202510213995926-3349.

[4] NPUcore-Aspera 项目仓库[EB/OL]. 2025. https://gitlab.eduxiji.net/T202510701995284/oskernel2025-npucore-aspera.

[5] NPUcore-rainbowww 项目仓库[EB/OL]. 2025. https://gitlab.eduxiji.net/T202510699997612/OSKernel2025-rainbowww.

[6] 2025 年系统能力培养赛操作系统赛题目[EB/OL]. 2025. https://github.com/oscomp/testsuits-for-oskernel/tree/pre-2025/.

[7] NPUcore-plus 项目仓库[EB/OL]. 2023. https://gitlab.eduxiji.net/202310699101073/oskernel2023-npucore-plus.

[8] NPUcore-IMPACT 项目仓库[EB/OL]. 2024. https://github.com/Fediory/NPUcore-IMPACT.

[9] pshell 项目仓库[EB/OL]. 2023. https://github.com/lurk101/pshell.