武漢大學

WUHAN UNIVERSITY

SC7 操作系统

决赛文档

参赛队名: 智核速启队

队伍成员: 李岩 陈震雄 陆冕

指导老师: 李文海 蔡朝晖

全国大学生计算机系统能力大赛

操作系统赛

内核实现赛道

2025 年 08 月

摘　　要

SC7 (SmartCore7) 是基于 **MIT XV6** 操作系统开发的教学用操作系统，支持 RISC-V 和 LoongArch 双架构，采用 **C 语言**实现以提供底层硬件控制能力和高效性能。系统已通过全国大学生计算机系统能力大赛初赛的 Basic、Busybox、Libctest、Libcbench、Iozone、Lmbench 和 Ltp 测例。，功能完整且架构清晰。核心创新点包括：

- **进程-线程分离模型**：进程作为资源管理单元（含页表/文件描述符/VMA），线程作为调度单元（独立内核栈/陷阱帧/信号处理函数及掩码）
- **双架构支持**：通过硬件抽象层（HAL）和硬件服务抽象接口（HSAI）统一 RISC-V 与 LoongArch 的架构差异，可以上板运行。
- **高效内存管理**：伙伴系统减少外部碎片，Slab 分配器优化小内存，懒分配机制提升 mmap 性能

关键词：双架构支持；进程线程分离；伙伴系统；虚拟文件系统

目　录

1 概述

1.1 SC7 介绍

SC7(SmartCore7)是一个基于 MIT XV6 操作系统开发的教学用操作系统，支持 RISC-V 和 LoongArch 两种架构，该操作系统目前已经通过初赛的 Basic、Busybox、Libctest 和 Libcbench 测例，是一个功能完整、架构清晰的教学用操作系统。

SC7 采用 C 语言开发。相较于新兴的 Rust 语言，C 语言在操作系统开发领域展现出以下优势：

- **底层控制能力强**：可以直接操作硬件寄存器、内存地址和汇编指令，实现对硬件的精确控制。
- **性能高效**：编译后代码紧凑，运行时开销最小，能够提供接近汇编语言的执行效率。
- **跨平台兼容性好**：拥有成熟的条件编译支持和丰富的工具链生态，便于在不同架构间移植。
- **学习价值高**：代码直观易懂，有助于深入理解计算机底层原理，是学习系统编程的理想语言。

SC7 使用 C 语言的核心原因在于其简单直接的设计理念，语法简洁明了，没有复杂的所有权系统和借用检查，这一特性使其特别契合教学场景——学生能将精力集中于操作系统核心概念的学习，而不会被语言的次要复杂性分散注意力。

1.2 我们的特点

- **从零开始**：最初只支持 uart 串口输出，到现在支持完整的内核功能，能通过 Basic、Busybox、Libctest、Libcbench、Iozone、Lmbench 和 Ltp 测例。
- **上板运行**：可以在 VisionFive 和 2K1000 板上运行。
- **高级功能**：信号机制，线程管理，伙伴系统和 Slab 分配器，懒分配。
- **多核运行**：两种架构都支持在 qemu 多核启动和运行。
- **原子化渐进开发**：逐步添加异常处理、物理内存、虚拟内存、中断处理、文件系统的模块，每一次提交都是可以运行的版本。
- **可追踪的协作提交**：超过 590 次 commit，三人协作开发，有完整规范的提交记录。

- **架构体系**：采用硬件抽象层（HAL）、硬件服务抽象接口（HSAI），内核层（KERNEL）的三层结构，结构清晰。

1.3 借鉴和复用的代码

HAL 的 Loongarch 部分复用了 **XN6(俺争取不掉队)** 的代码，HAL 的 RISC-V 部分复用了 **XV6** 的代码；HAL,HSAI 的设计思想来自 **XN6(俺争取不掉队)**, 但是 HSAI 层的实现与之不同。HSAI 的中断异常处理参考了 **XV6-riscv** 和 **XV6-loongarch**，在此基础上 SC7 对二者提取出共同行为并整合。KERNEL 的虚拟内存模块参考了 **XV6**。用户程序 initcode 的构建思想受启发于 **XV6** 和 **XN6(俺争取不掉队)**, 实现为独立创新。Slab 分配器参考了 **Linux** 和 **RT-Thread**，但是是独立实现。

部分系统调用和 VMA(虚拟内存区域) 实现参考了 **AVX（刀片超车队）**

文件系统移植了 lwext4，在此基础上有改进

1.4 系统架构

1.4.1 整体架构图

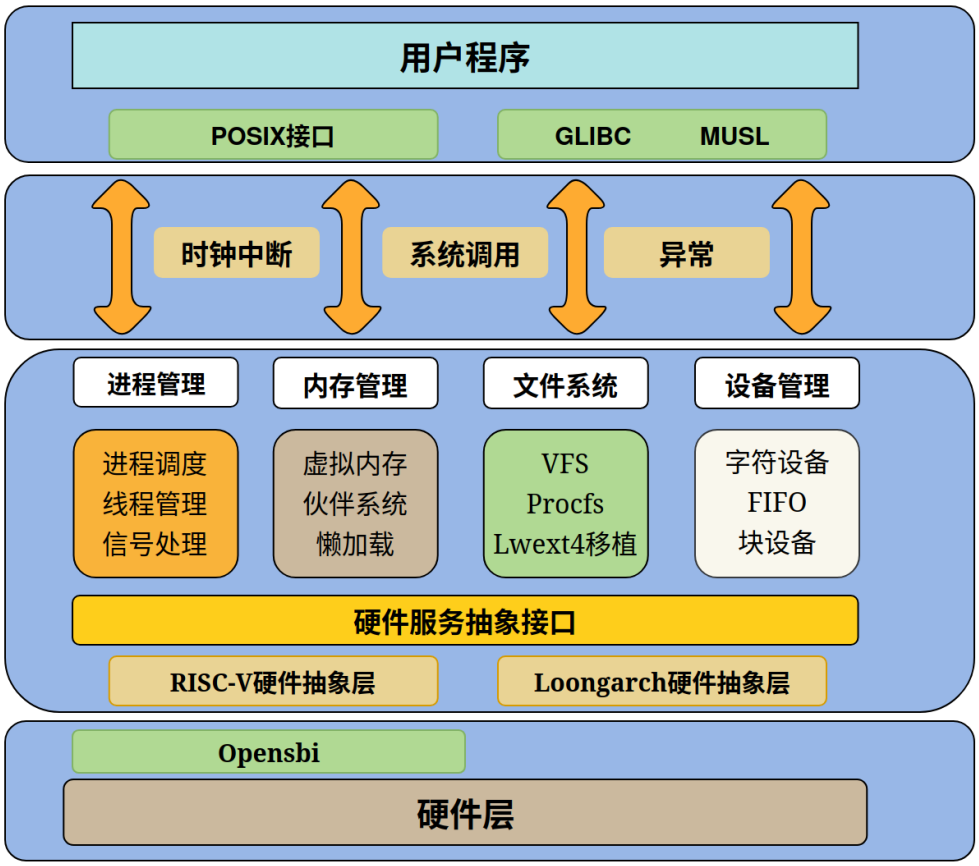


图 1.1　SC7 操作系统架构图

1.4.2 功能模块

SC7 操作系统包含以下主要功能模块：

表 1.1　SC7 操作系统功能模块详细说明

模块名称	实现功能详细说明
进程管理	多进程调度算法、上下文切换机制、进程创建与销毁、线程支持、进程间通信
内存管理	伙伴系统物理内存分配、Slab 分配器、虚拟内存区域 (VMA) 管理、多级页表管理、内存映射、缺页异常处理、共享内存
文件系统	虚拟文件系统 (VFS) 抽象层、ext4 文件系统移植、文件/目录操作
系统调用	POSIX 兼容系统调用接口、用户态/内核态切换
设备驱动	UART 串口驱动、VirtIO 磁盘驱动、PLIC 中断控制器驱动、sd 卡驱动、ahci 驱动
同步机制	自旋锁 (spinlock)、睡眠锁 (sleeplock)、Futex、读写锁
信号系统	信号集 (sigset)、信号动作、信号检查、sigtrampoline
硬件抽象功能层	RISC-V/LoongArch 双架构支持、架构无关接口设计

1.4.3　开发流程

整个开发过程使用规范的的 git 提交规范，使用 feat、fix、docs、refactor、等标准前缀来标识不同类型的提交，每个功能模块都有清晰的提交记录，便于团队协作和代码维护。

项目从物理内存管理起步，逐步构建起一个功能完善的操作系统核心。逐步实现了虚拟内存管理、进程管理、磁盘驱动等核心功能，并支持用户程序运行。我们实现了大量系统调用，同时支持静态链接和动态链接，通过了 basic、busybox、lua、libctest 等标准测例。文件系统方面，我们移植并适配了 lwext4，构建 vfs 层定义统一的文件操作接口。在高级功能领域，我们实现了 futex 和线程管理，并设计了**线程-进程分离模型**，开发了伙伴系统和懒加载机制，还做了简单的 slab 模块来优化内存管理, 最后成功通过 libcbench 测例。在 7 月份之后的决赛阶段，我们适配了 VisionFive 星光版和 Ls2k 星云板，完善了信号系统和共享内存机制，通过了 lmbench 和部分 ltp 测例。

图 1.2　开发流程

1.4.4　项目结构

```
| Makefile                  // 构建脚本，支持RISC-V和LoongArch双架构编译
| README.md                 // 项目说明文档
|---hal                     // 硬件抽象层
|   |---loongarch           // LoongArch架构相关
|   |---riscv               // RISC-V架构相关
|---hsai                    // 硬件抽象接口层
|   | hsai_mem.c            // 内存管理抽象接口
|   | hsai_trap.c           // 中断处理抽象接口
|   | plic.c                // PLIC中断控制器驱动
|---include                 // 头文件目录
|   |---hal                 // HAL头文件
|   |---hsai                // HSAI头文件
|   |   | hsai_mem.h        // 内存管理抽象接口头文件
|   |   | hsai_trap.h       // 中断处理抽象接口头文件
|   |---kernel              // 内核头文件
|   |   |---fs              // 文件系统头文件
|   |   |   | ext4.h        // EXT4文件系统头文件
|   |   |   | file.h        // 文件操作头文件
|   |   |   | fs.h          // 文件系统抽象头文件
|   |   | process.h         // 进程管理头文件
|   |   | thread.h          // 线程管理头文件
```

```
|   |   | vmem.h          // 虚拟内存管理头文件
|---kernel                // 内核源码目录
|   | SC7_start_kernel.c // 系统启动入口
|   | process.c          // 进程管理核心
|   | pmem.c             // 物理内存管理
|   | vmem.c             // 虚拟内存管理
|   | syscall.c          // 系统调用处理
|   | timer.c            // 定时器实现
|   | thread.c           // 线程管理
|   |---driver           // 设备驱动
|   |   |---loongarch    // LoongArch驱动
|   |   |---riscv        // RISC-V驱动
|   |---fs               // 文件系统实现
|   |   | ext4.c         // EXT4文件系统实现
|   |   | file.c         // 文件操作实现
|   |   | fs.c           // 文件系统核心
|   |   | vfs_ext4.c     // VFS EXT4接口实现
|---user                  // 用户程序目录
|   |---include          // 用户程序头文件
|   |   | usercall.h     // 用户系统调用接口
|   |---loongarch        // LoongArch用户程序
|   |---riscv            // RISC-V用户程序
```

2　进程与线程管理

2.1　概述

SC7 采用**进程-线程分离**设计：进程是**资源管理**的基本单位，线程是**调度**的基本单位。每个进程包含一个线程队列 (`thread_queue`)，管理其所有线程。关键设计特点包括：

- **进程资源**：页表 (`pagetable`)、文件描述符表 (`ofile`)、虚拟内存区域 (`vma`)
- **线程资源**：独立的内核栈 (`kstack`)、陷阱帧 (`trapframe`)、上下文 (`context`)
- **调度单元**：调度器以线程为单位进行调度，同一进程的线程共享进程资源
- **Futex 支持**：实现快速用户态互斥锁，支持线程同步

2.2　进程生命周期

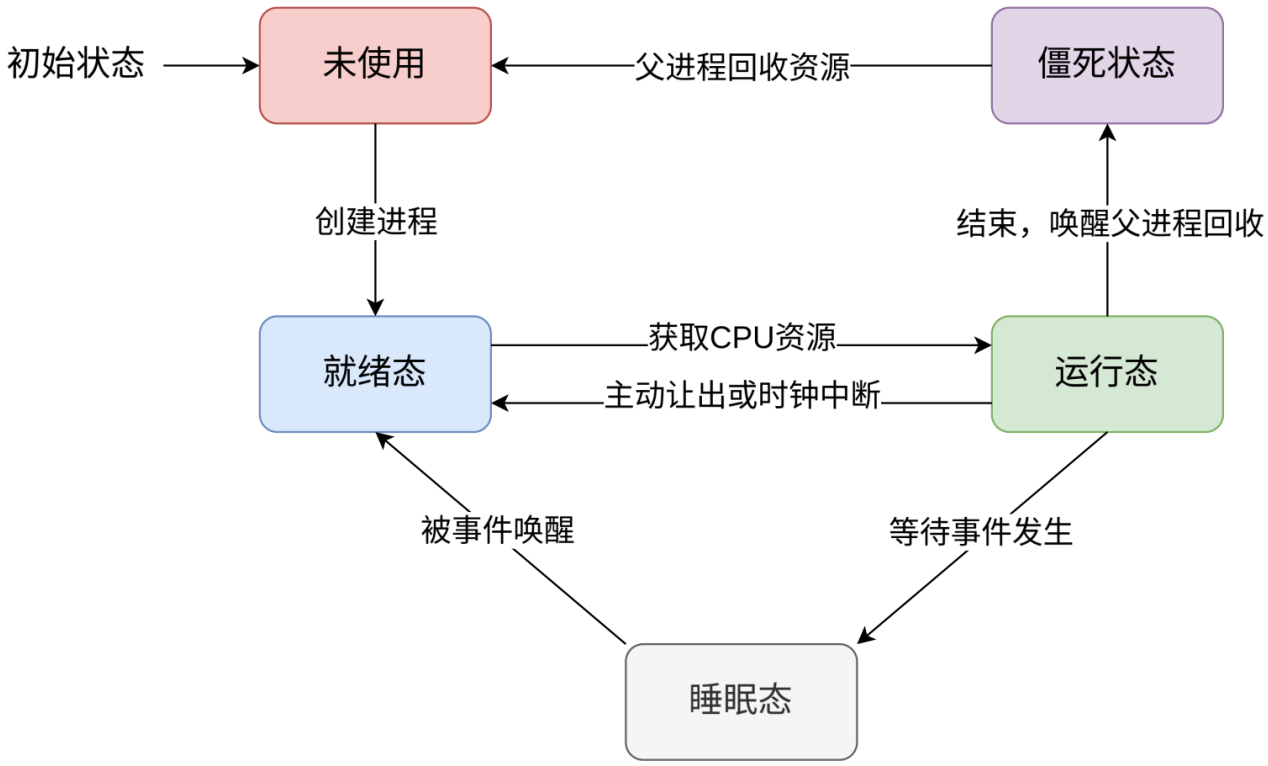


图 2.1　进程状态转移图

内核初始化时所有 struct proc 都是未使用状态，创建进程后变为就绪态。就绪态下被调度到则获取 CPU 资源，进入运行态。运行态下可以主动让出 CPU 或者被时钟中断剥夺 CPU

使用权，进入就绪态；运行态也可以等待事件发生，进入睡眠态。睡眠态被事件唤醒后进入就绪态。运行态运行结束时唤醒父进程回收，进入僵死态。僵死态下被父进程回收资源后，进入未使用状态。

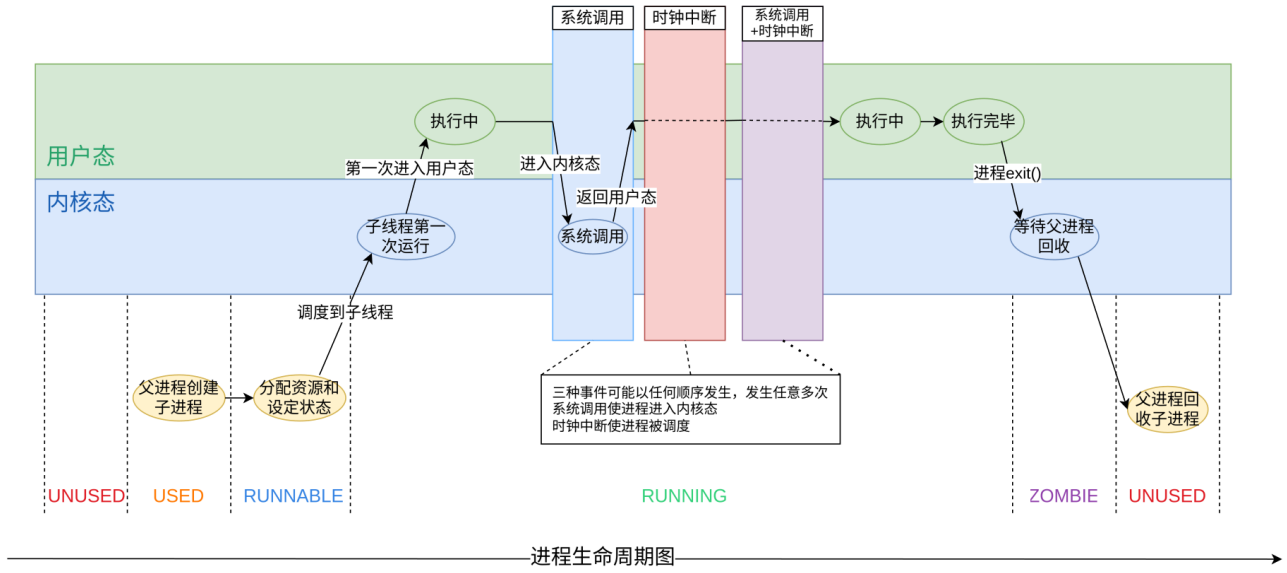


图 2.2　进程全生命周期图

上图描述了进程从创建到运行结束、被回收的全过程。先由父进程创建子进程并分配资源和设定状态。然后第一次调度到子进程，子进程首先运行在内核态，返回用户态后正式开始执行用户程序。中途可能发生系统调用和时钟中断的事件。时钟中断会导致进程被调度进入就绪态，系统调用可使进程进入睡眠态。执行完毕后，子进程调用 exit 等系统调用退出，进入僵死态。僵死态下被父进程回收后进入未使用状态。

系统调用的切换过程较简单，已经在上图给出。时钟中断和系统调用 + 时钟中断的切换较复杂，在下图给出：

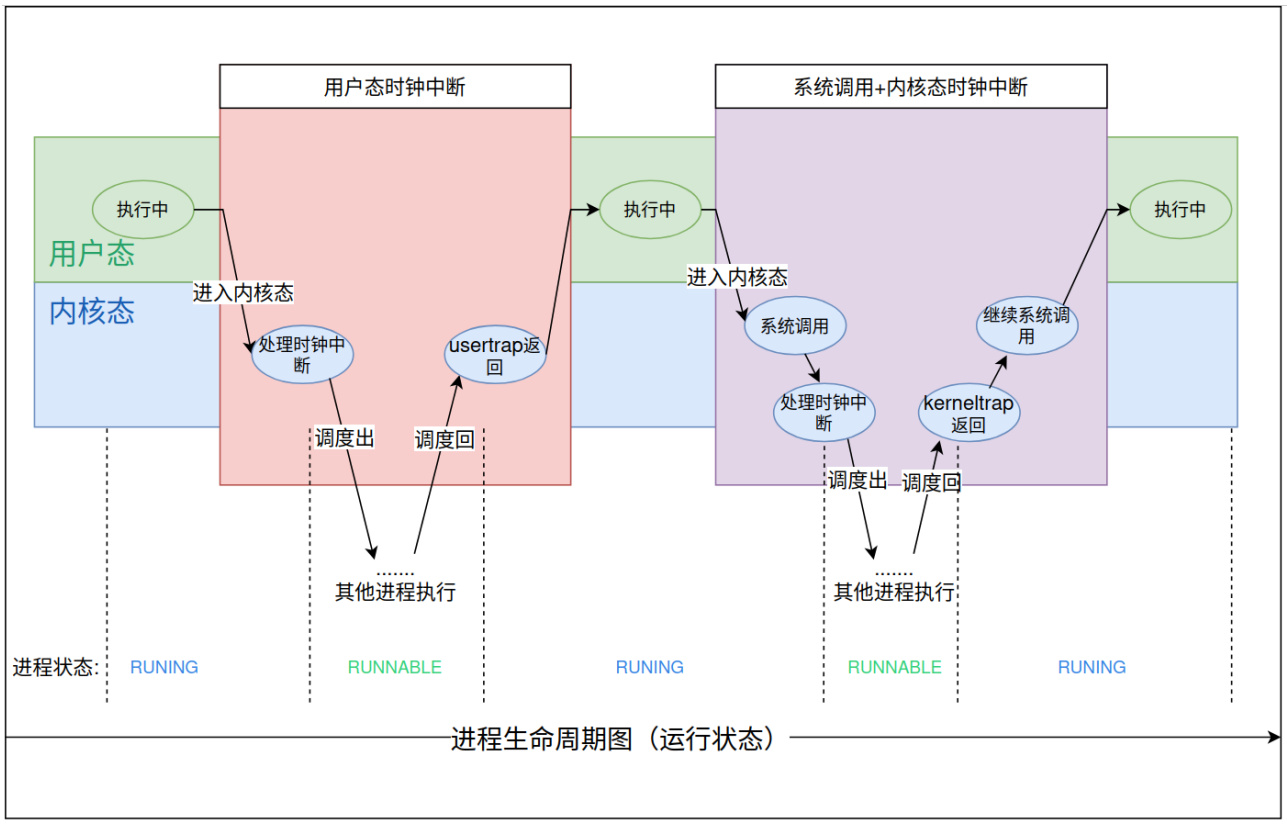


图 2.3　进程运行状态子图

首先是单独的时钟中断，因为中断是进程在用户态运行时产生的，所以称作用户态时钟中断。进程先进入内核态，在 usertrap 的 devintr 处理时钟中断，然后调用 yield 导致进程被调度出去，一段时间后进程又被调度回来，获取 CPU，再从 yield 的下一行代码恢复执行，然后返回用户态继续中断前的执行。

系统调用 + 时钟中断更复杂一点。系统调用在内核态处理，因此这里的时钟中断是内核态中断，由 kerneltrap 的 devintr 处理，然后调用 yield 导致进程被调度出去，一段时间后进程又被调度回来，从 yield(kerneltrap 中) 的下一行代码恢复执行，然后从 kernelvec 恢复中断前的状态，继续未完成的系统调用。最后系统调用完成后，返回用户态执行。

2.3　进程控制块与线程控制块设计

在 SC7 中，进程和线程的核心信息分别通过进程控制块（struct proc）和线程控制块（struct thread）进行管理。

进程控制块（struct proc）定义了进程共享的资源和状态：

进程控制块 (kernel/process.h)

```
struct proc {
    spinlock_t lock;       // 进程锁
    enum procstate state; // 进程状态(UNUSED/USED/RUNNABLE等)
    struct list thread_queue; // 线程队列头
    thread_t *main_thread; // 当前运行线程

    // 资源共享部分
    pgtbl_t pagetable;     // 页表
    struct file *ofile[NOFILE]; // 打开文件表
    struct vma *vma;       // 虚拟内存区域

    // 标识信息
    int pid;               // 进程ID
    int thread_num;        // 线程计数
    uint64 sz;             // 进程内存大小
    // ... 其他字段省略 ...
};
```

线程控制块（struct thread）则管理线程私有的信息和资源。它包含线程的锁（lock）、当前状态（state，如 t_UNUSED/t_RUNNING 等）、指向所属进程的指针（p）。线程私有资源包括独立的内核栈虚拟地址（kstack）及其对应的物理地址（kstack_pa），用于保存用户态和内核态切换信息的陷阱帧（trapframe），保存线程执行上下文的上下文结构体（context）以及信号处理相关变量（sig_set）。此外，线程控制块还包含线程 ID（tid）、定时唤醒时间（awakeTime）以及用于在线程队列中链接的列表元素（elem）。这些私有资源确保了线程能够独立地进行调度和执行。

kernel/include/thread.h

```
typedef struct thread
{
    struct spinlock lock;

    /* 当使用下面这些变量的时候，thread的锁必须持有 */
    enum thread_state state; //< 线程的状态
    proc_t *p;           //< 这个线程属于哪一个进程
```

```
    void *chan;           //< 如果不为NULL,则在chan地址上睡眠
    int tid;              //< 线程ID
    uint64 awakeTime;     //< futex 睡眠时间
    int timeout_occurred; //< 标记是否因为超时而唤醒
    pid_t ppid;           //< 父进程ID,用于线程退出时的父进程回收资源

    /* 使用下面这些变量的时候，thread的锁不需要持有 */
    uint64 kstack; //< 线程内核栈的地址,一个进程的不同线程所用的内核栈的地址应该不
        ↪ 同，地址最小处
    uint64 vtf;   //< 线程的trapframe的虚拟地址
    uint64 sz;    //< 复制自进程的sz
    int thread_idx; //< 线程列表中的第几个
    trapframe_t *trapframe;
    context_t context;  //< 每个进程应该有自己的context
    uint64 kstack_pa;   //< 当线程的栈和进程的栈不是一个的时候，用它保存物理地址，地
        ↪ 址最小处
    struct list_elem elem; //< 用于进程的线程链表

    uint64 clear_child_tid; //< 子线程ID清除标志
    vma_t *stack_vma;    //< 线程栈VMA的引用，用于退出时减少引用计数

    __sigset_t sig_set;  // 信号掩码
    __sigset_t sig_pending; // pending signal

    /* 信号处理函数数组 - 每个线程独立的信号处理 */
    sigaction sigaction[SIGRTMAX + 1]; // signal action 信号处理函数

    ...
} thread_t;
```

2.4 进程初始化

2.4.1 第一个 init 进程的创建

系统启动时，由 sc7_start_kernel 函数调用 init_process 完成第一个进程的创建。此过程首先通过 allocproc 分配并初始化一个进程控制块，将其状态设置为 USED。然后，该进程控制块被指定为 initproc，其状态进一步设为 RUNNABLE。接着，系统将预编译好的二进制程序映像加载到 initproc 的地址空间中，并根据映像长度计算进程内存大小。同时，设置进程的当前工作目录。最后，通过设置陷阱帧的返回地址（epc）和用户栈指针（sp）来准备进

程的首次执行环境，并将主线程的状态设为 t_RUNNABLE，释放进程锁，等待调度。

2.4.2 其他进程的创建

除了第一个 init 进程,其他进程通常通过 fork 系统调用创建。fork 函数首先调用 allocproc 为新进程分配一个进程控制块。接着，它将父进程的页表、虚拟内存区域、陷阱帧和文件描述符等资源复制给子进程。一个关键的设置是：子进程的陷阱帧中的返回值（通常是 a0 寄存器）被设置为 0，这样子进程在从 fork 返回时能够通过返回值识别自己是子进程，而父进程则会收到子进程的 PID。此外，fork 还会建立父子进程之间的关系，以便父进程可以通过 wait 系统调用来等待子进程的结束。最后，子进程的状态被设置为 RUNNABLE，其主线程的状态也被设置为 t_RUNNABLE，然后释放进程锁并返回子进程的 PID 给父进程。

2.5 线程管理机制

2.5.1 线程初始化

系统启动时，线程池会进行初始化。thread_init 函数会初始化一个名为 free_thread 的链表，用于存放所有空闲的线程控制块。随后，它会遍历预设的 THREAD_NUM 个线程控制块数组 thread_pools，为每个线程控制块初始化自旋锁，将其状态设置为 t_UNUSED，并将其添加到 free_thread 空闲链表的末尾。这样，在需要创建新线程时，可以直接从该空闲链表中获取可用的线程控制块。

2.5.2 线程创建

clone_thread 函数负责创建新的线程。它首先从 free_thread 空闲链表中获取一个可用的线程控制块。接着，为新线程分配独立的内核栈（包括物理内存 kstack_pa 和通过 mappages 映射的虚拟地址 kstack)。然后，复制父进程（或调用线程）的陷阱帧到新线程的陷阱帧中，并根据新线程的执行逻辑（如用户栈指针 sp、函数参数 a0 和入口地址 epc/era）进行调整。最后，新线程被加入到其所属进程的线程队列 p->thread_queue 中，线程计数器 p->thread_num 增加，并将新线程的状态设置为 t_RUNNABLE，使其可以被调度器选中执行。

2.6 线程与进程分离模型

SC7 采用**进程-线程分离**设计模型，其核心思想是**将资源管理与执行调度解耦**。进程作为**资源管理**的基本单位，负责维护共享资源，包括页表、文件描述符表、虚拟内存区域（VMA）

以及信号处理设置等。线程则作为**调度执行**的基本单位，拥有独立的执行上下文，包括内核栈、陷阱帧（trapframe）和上下文结构（context）。这种分离设计允许同一进程内的多个线程并发执行，同时高效地共享进程资源。

在实现上，每个进程控制块（struct proc）包含一个线程队列（thread_queue），通过链表管理其所有线程。线程创建时（如通过 clone_thread 系统调用），内核仅为新线程分配私有执行资源（内核栈、陷阱帧等），而**无需复制整个进程的地址空间或文件表**。这种轻量级线程创建机制显著降低了线程创建开销。调度器以线程为基本调度单元，在进程的线程队列中选择可运行线程执行。当线程阻塞时（如等待 Futex），仅该线程进入睡眠状态，进程内其他线程仍可继续运行。

图 2-3 展示了**线程与进程分离模型**的架构：

- 进程作为**资源容器**，持有页表、文件列表和 VMA 链表
- 线程作为**执行实体**，通过进程的 thread_queue 链表组织
- 调度器直接操作线程队列，选择下一个执行的线程

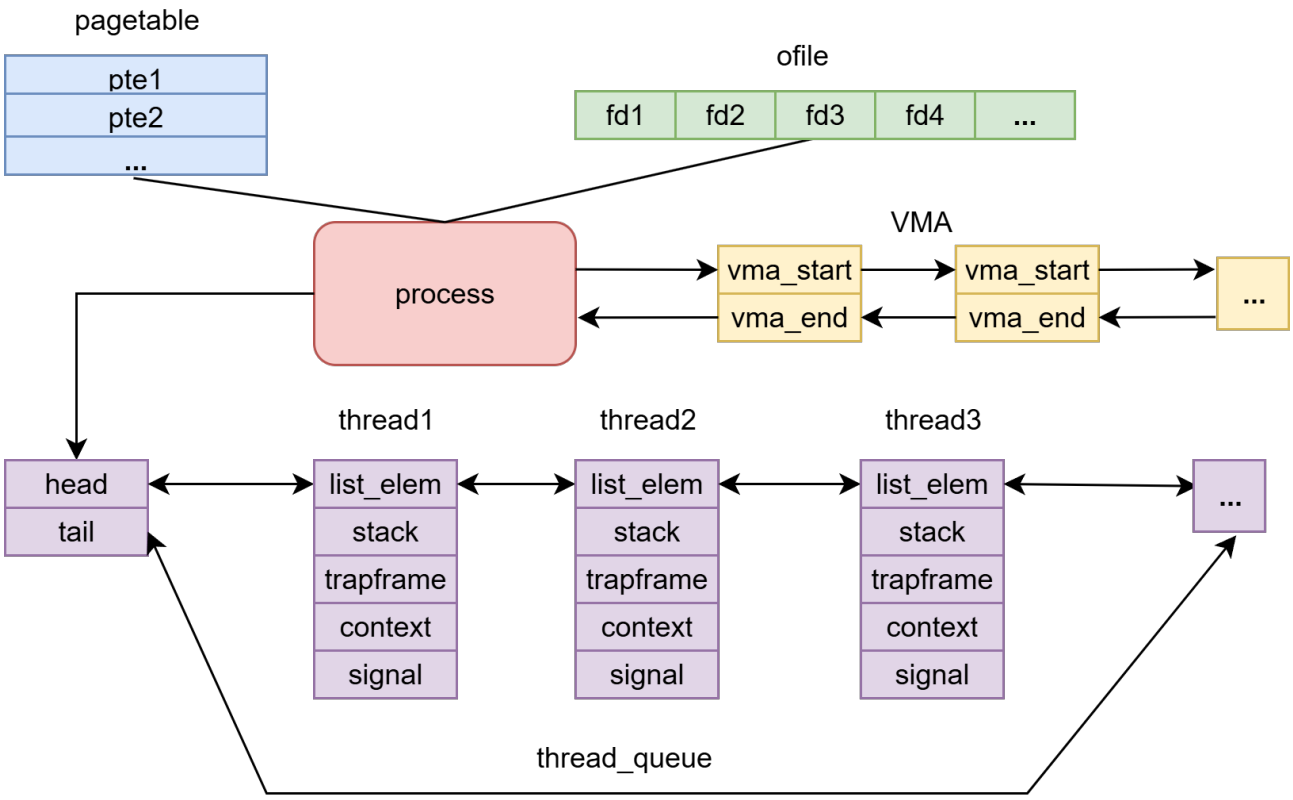


图 2.4　线程与进程分离模型

该模型的优势在于：

资源共享高效性：线程共享进程资源，避免资源重复分配

创建开销低：线程创建仅需分配执行上下文，无需复制资源

调度灵活性：支持线程级调度，提高 CPU 利用率

同步粒度细：Futex 等同步机制可直接作用于线程级

2.7 线程进程调度机制

2.7.1 调度器实现

调度器（scheduler）是一个无限循环，负责在就绪队列中选择下一个要执行的线程。它遍历系统中的所有进程，获取每个进程的锁。如果一个进程的状态是 RUNNABLE，调度器便会遍历该进程的线程队列。在线程队列中，调度器查找状态为 t_RUNNABLE 的线程，或者状态为 t_TIMING 但其 awakeTime 已到期的线程。一旦找到一个可运行的线程，调度器会将其设置为当前进程的 main_thread，并进行上下文切换：将当前 CPU 的上下文保存到旧进程的上下文，并将目标线程的上下文加载到 CPU 中。在切换回来后（即当前线程主动放弃 CPU 或被中断），调度器会将 CPU 的上下文保存回当前线程的上下文。随后，该线程的状态被设置为 t_RUNNING，并释放进程锁，继续下一次调度循环。

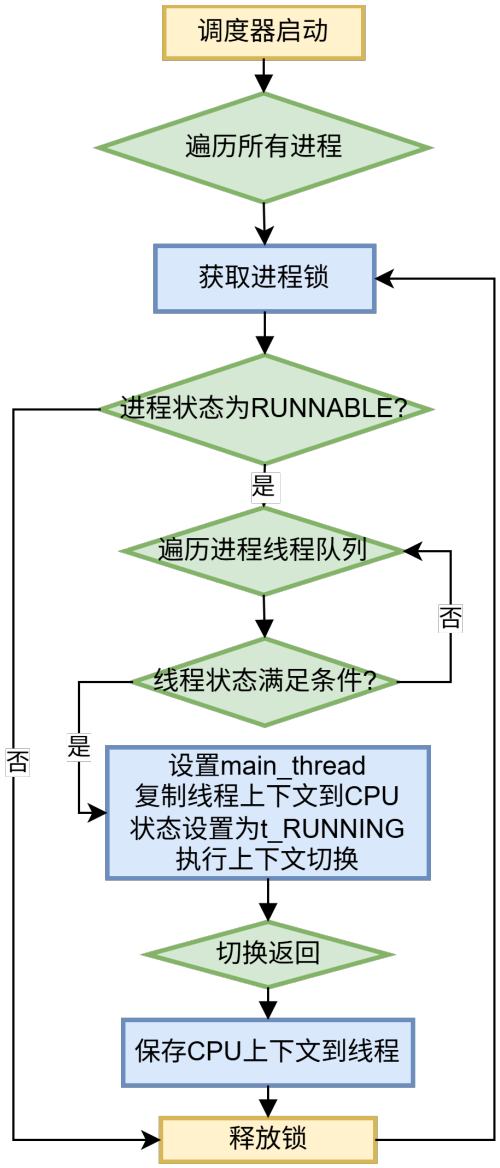


图 2.5　调度器工作流程图

2.7.2　主动放弃 CPU

线程可以通过调用 sched() 函数主动放弃 CPU 的控制权。当一个线程调用 sched() 时，它首先保存当前线程的陷阱帧，以记录其用户态的执行状态。然后，它将当前进程的上下文（即当前线程的内核态上下文）保存起来，并通过 hsai_swtch 函数切换到 CPU 的调度器上下文。这使得调度器能够重新选择其他可运行的线程来执行。当调度器再次选择该线程执行时，它会从之前保存的上下文和陷阱帧恢复执行。

2.8 Futex 同步机制

Futex（Fast Userspace Mutex，快速用户空间互斥体）是一种高效的、内核辅助的进程间/线程间同步机制。其核心思想是：在无竞争的情况下，同步操作（如加锁和解锁）完全在用户空间完成，避免了昂贵的系统调用和上下文切换；只有当出现竞争（例如，尝试加锁失败，需要等待锁释放）时，才通过系统调用进入内核，由内核管理等待队列和唤醒操作。

与传统的信号量或互斥量（它们通常每次操作都涉及内核）不同，Futex 显著提升了同步效率，尤其适用于高并发场景。它允许用户态程序在共享内存上进行原子操作，如果操作未能成功（表示存在竞争），线程才会“坠入”内核空间，请求内核将其挂起。当另一个线程完成其操作并释放资源时，它可以“唤起”等待的线程。

在内核层面，Futex 通过维护一个或多个等待队列来实现线程的挂起和唤醒。这些队列通常与用户空间共享的内存地址相关联。我们在 SC7 中做了一个简化的实现供用户态调用，目前支持操作 futex_wait，futex_wake，futex_requeue，futex_cmp_requeue，futex_wait_bitset 和 futex_wake_bitset。

2.8.1 线程 futex_wait 核心流程

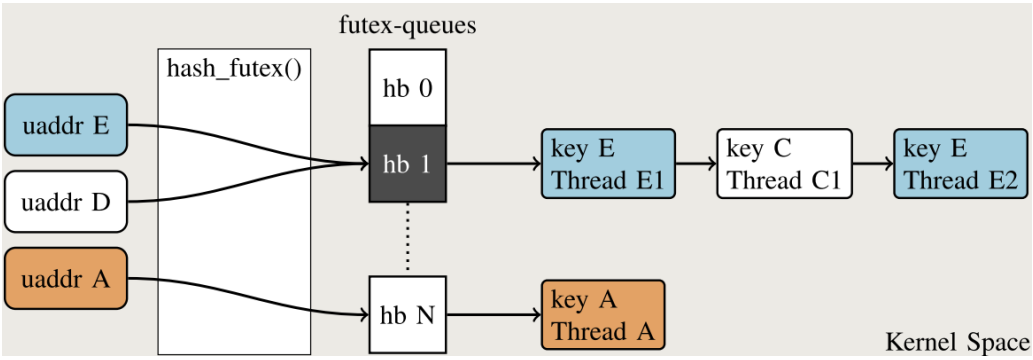


图 2.6　futex 队列

1. **地址映射 (Key Generation)**: 将用户空间地址 `uaddr` 转换为内核空间中唯一的 `futex_key`。这一步至关重要，因为它决定了不同进程是否能共享同一个 Futex 等待点。
   - **进程内同步 (Intra-process)**: 当 Futex 用于同一进程内的线程同步时，`futex_key` 通常直接基于虚拟地址 `uaddr` 生成。因为同一进程的线程共享相同的虚拟地址空间，所以虚拟地址本身就能唯一标识一个 Futex。
   - **进程间同步 (Inter-process)**: 当 Futex 用于不同进程间的同步（例如通过共享内存 `mmap`），`futex_key` 必须能够跨进程唯一标识 Futex。此时，`futex_key` 通常由底层

物理页面地址（或更准确地说，共享文件/匿名内存区域的 `inode` 和相对于该区域的偏移量）与页内偏移量组合而成。这确保了即使不同进程将同一物理内存映射到不同的虚拟地址，它们也能引用同一个 Futex。

2. **哈希计算 (Hashing)**：通过 `hash_futex()` 函数，将上一步生成的 `futex_key` 映射到 `futex_queues` 哈希表中的一个特定哈希桶。`hash_futex()` 函数封装了内核中高效的哈希算法（`jhash`），旨在将不同的 `futex_key` 均匀地分布到哈希表的各个桶中，以减少冲突。

$$\text{bucket\_index} = \texttt{hash\_futex}(\texttt{futex\_key})$$

其中 `bucket_index` 是 `futex_queues` 数组中的索引。

3. **队列插入 (Queue Insertion)**：当一个线程(或进程)调用 Futex 系统调用(如 `FUTEX_WAIT`)并进入等待状态时，它会被封装为一个 `futex_q` 结构体。该 `futex_q` 结构体随后被插入到由哈希计算确定的 `futex_queues` 哈希桶对应的链表尾部，遵循 FIFO (First-In, First-Out) 原则，即先等待的线程先被唤醒。

2.8.2 线程一次 futex 过程

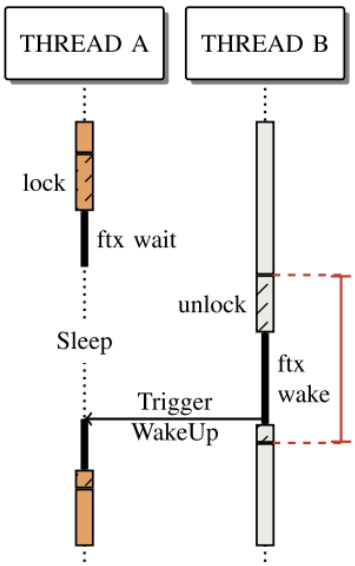


图 2.7　一次 futex 流程

无竞争场景：

- 加锁：检查原子变量 `state`，若为 0 则通过 CAS 置为 1。
- 解锁：若 `state=1` 则直接置 0，无需内核介入。

竞争场景：

- 加锁失败：当检测到 `state=1` 时，线程通过 CAS 将其置为 2 并调用 `futex_wait`。
- 内核挂起：验证 `*uaddr` 值后，将线程加入等待队列。
- 解锁唤醒：若解锁时发现 `state=2`，则调用 `futex_wake` 唤醒等待线程

2.9 资源回收

2.9.1 线程资源回收

当一个进程退出时，其内部所有线程的私有资源都需要被回收。这通常通过一个内部函数（例如 freeproc 的一部分逻辑）来完成。该函数会遍历进程控制块中 thread_queue 链表上的每一个线程。对于每个线程，它会释放线程私有的陷阱帧（trapframe）所占用的内存。如果线程拥有独立的内核栈（即该内核栈不是主线程的默认栈），则会通过 vmunmap 解除其内核栈的虚拟内存映射，并释放其物理内存（kstack_pa）。完成资源释放后，该线程的状态会被设置为 t_UNUSED，并将其添加到全局的 free_thread 空闲链表头部，以便后续创建新线程时可以复用。

同时，注意到 glibc 的 pthread_join 实现要求线程退出的时候，若设置了 clear_child_tid，除了将该地址清零外，还需要 futex_wake 等待于该地址的所有线程。

2.9.2 进程资源回收

进程退出时，通过调用 exit 系统调用来释放其所有共享资源。exit 函数首先获取当前进程的控制块，然后遍历并关闭该进程所有打开的文件描述符。接着，它会释放进程的页表和所有相关的虚拟内存区域（vma 列表），包括解除虚拟地址映射并释放底层物理页帧。完成所有资源释放后，进程的状态会被设置为 ZOMBIE，并保存其退出状态 exit_state。最后，当前进程会调用 sched() 主动放弃 CPU，因为处于 ZOMBIE 状态的进程不再可运行，它将等待其父进程通过 wait 系统调用来最终回收其进程控制块。

3 内存管理

3.1 物理内存管理

在操作系统的内存管理系统中，物理内存是内核初始化和运行的基础。

3.1.1 地址空间

SC7 操作系统的地址空间采用经典的 64 位虚拟内存布局，其中用户空间占据低地址区域（0x0 到 0x80000000），包含从 0x0 开始的用户程序映像（text、data、bss 段）、向上增长的用户堆、256MB 的 MMAP 区域、以及向下增长的用户栈；内核空间占据高地址区域，从 0x80000000 开始向上延伸，包含内核代码和数据段、每个进程的内核栈区域（32KB 栈空间加 64KB 保护页）、trapframe 和 trampoline，最终到达最高地址 0x4000_0000_0000（MAXVA），这种布局确保了用户态和内核态的完全隔离，同时为进程提供了充足的虚拟内存空间进行程序执行和动态内存管理。

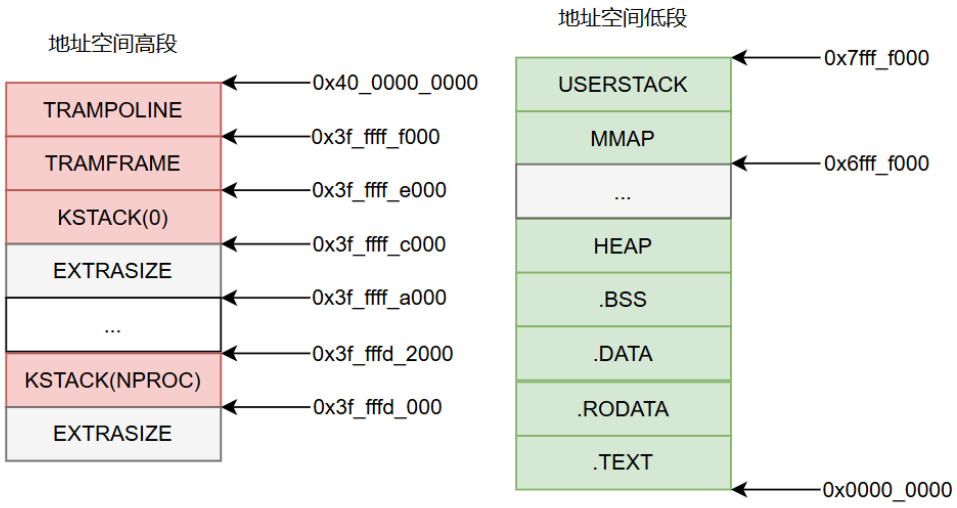


图 3.1 SC7 操作系统地址空间布局

3.2 伙伴系统

3.2.1 设计原理

伙伴系统是一种高效的内存分配算法，通过将内存划分为不同大小的块来减少内存碎片。核心思想是：

- 内存按 2 的幂次方大小进行划分（1 页、2 页、4 页、8 页等）
- 每个块可以分裂为两个相等的伙伴块
- 相邻的空闲伙伴块可以合并为更大的块
- 使用位图和链表管理空闲块

3.2.2 数据结构设计

伙伴系统核心数据结构 (kernel/pmem.h)

```
#define BUDDY_MAX_ORDER 10 // 最大阶数（1024页）
#define PGSIZE 4096      // 页面大小

// 伙伴块节点
typedef struct buddy_node {
    uint64 addr;         // 块起始地址
    int order;           // 块阶数
    struct buddy_node *next, *prev; // 链表指针
} buddy_node_t;

// 伙伴系统全局结构
typedef struct buddy_system {
    uint64 mem_start;    // 内存起始地址
    uint64 mem_end;      // 内存结束地址
    uint64 total_pages; // 总页面数

    uint64 *bitmap;      // 位图（标记页面使用状态）
    buddy_node_t *nodes; // 页面元数据数组
    buddy_node_t free_lists[BUDDY_MAX_ORDER + 1]; // 空闲链表数组
} buddy_system_t;
```

3.2.3 核心算法

伙伴系统的核心算法思想是通过将物理内存划分为一系列大小按 2 的幂次方对齐的块（称为阶），在分配时从最接近需求大小的空闲链表开始查找，若当前阶无空闲块则向上查找更大阶的块并递归分割成两个伙伴块（每次分割大小减半），直到获得合适大小的块；在释放时立即检查释放块的伙伴块是否同阶且空闲，若满足条件则递归合并成更大的空闲块，从而在常数时间内完成伙伴查找与合并，有效减少外部碎片，实现高效的内存分配与回收。

3.3 Slab 分配器

仿照 Linux 命名风格的 Slab 分配器，用于分配小内存。基本思想是每次从页分配器请求一个页作为缓存，每次分配小内存时从缓存中分配。

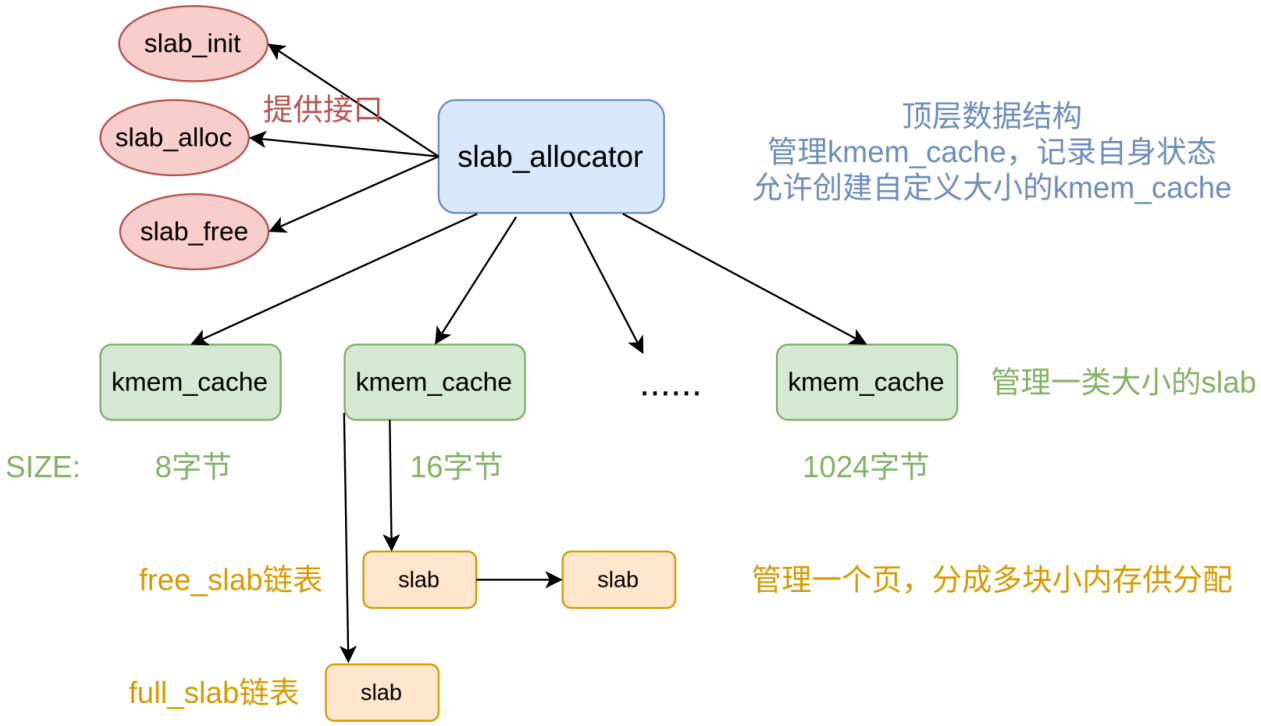


图 3.2　Slab 结构图

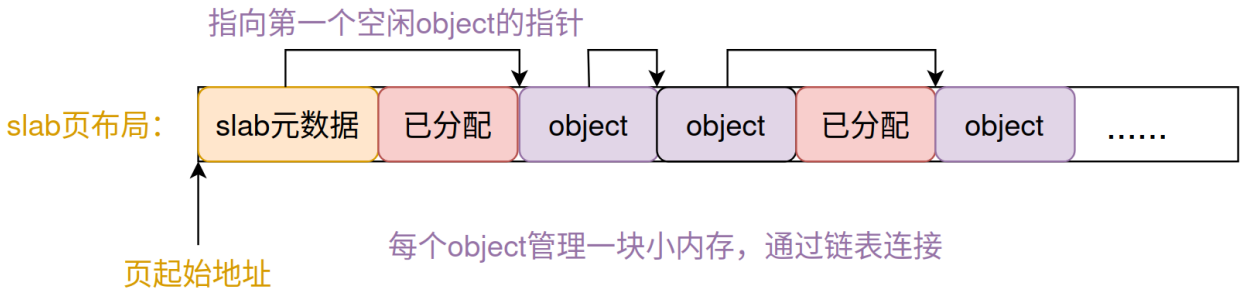


图 3.3　Slab 页布局图

SC7 的 Slab 分配器分为以下几个层级：顶层的 slab_allocator 管理着所有的 kmem_cache，并且管理 slab 分配器的全局状态。下一级的每一个 kmem_cache 表示一类相同大小的 slab。再下一级的 slab 管理一个页，并把这个页分成多个 size 大小的 object。最底层的 object 把要分配的小内存块以链表的形式连接起来，挂载对应的 slab 上。

后续开发中会增加创建自定义大小的 kmem_cache 的功能，增加 CPU 缓存等高级功能

kernel/slab.h

```
struct object{
    struct object* next;
};

struct slab{
    uint64 magic;
    struct slab *next;
    struct object *object; // 顺序链表，指向第一个空闲object
    uint32 size;
    uint32 free;
};

struct kmem_cache{
    struct slab *free_slab; // 有空闲object的slab块
    struct slab *full_slab; // object分配完了的slab块
    uint64 size; // 要分配对象的大小
};

struct slab_allocator{
    struct kmem_cache *fixed_cache_list[FIXED_CACHE_LEVEL_NUM]; // 固定大小的
        ↪ kmem_cache
```

```
    struct kmem_cache *custom_cache_list[CUSTOM_CACHE_LEVEL_NUM]; // 自定义大小的
        ↪ kmem_cache
    enum slab_state state;
};
```

3.4 虚拟内存管理

3.4.1 页表管理

页表是操作系统内存管理的核心组件，负责虚拟地址到物理地址的转换。RISC-V 使用 sv39 分页模式，具有三级页表结构，最大虚拟地址为 1ULL«38, 而 LoongArch 使用四级页表结构，支持更大的虚拟地址空间, 最大虚拟地址为 1ULL«46。

在页表遍历过程中，两种架构使用相同的 PX 宏来提取虚拟地址中的页表索引，但二者遍历的层级数量不同，RISC-V 为三级页表，而 Loongarch 为四级页表

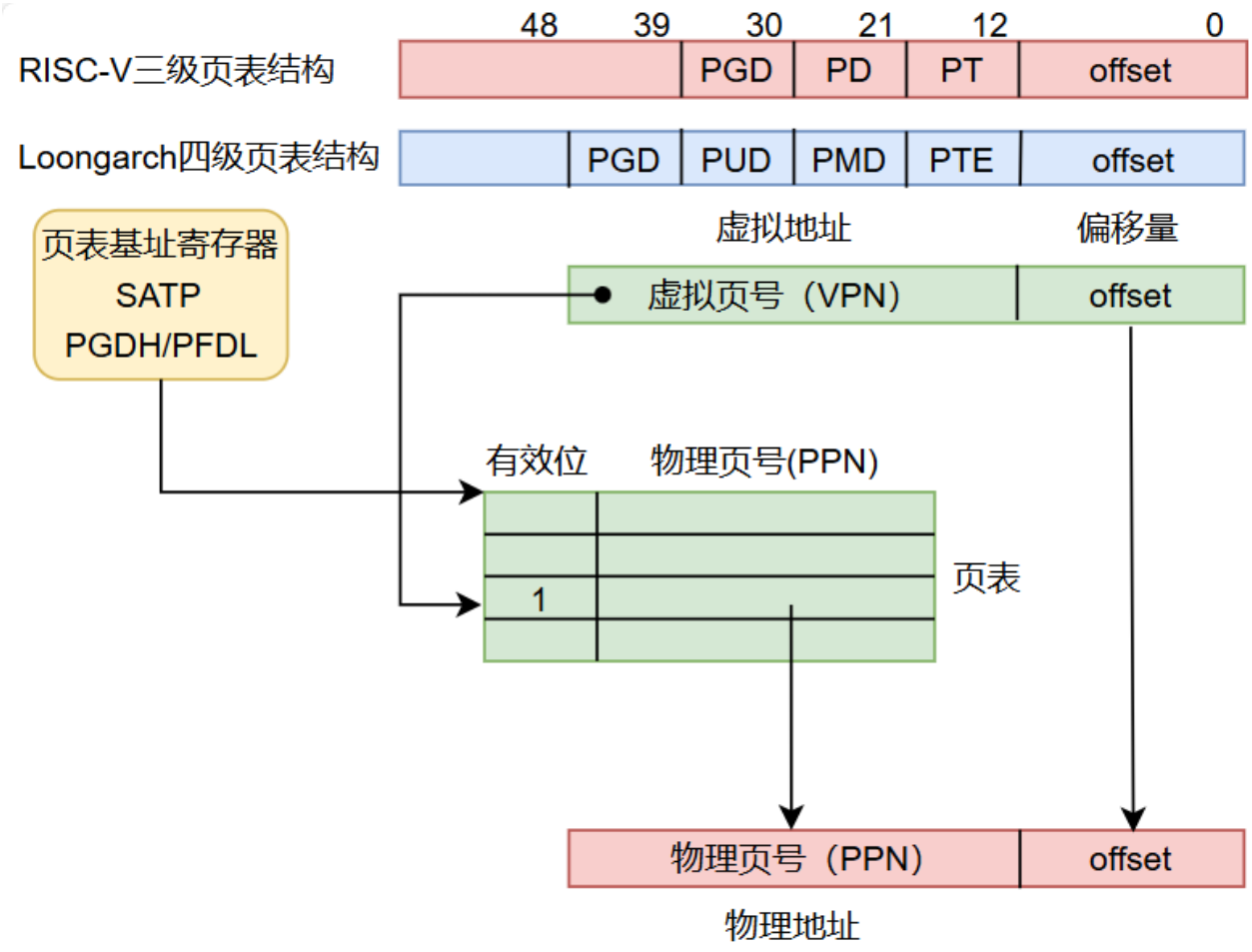


图 3.4　使用页表的地址翻译过程

两种架构在页表项的格式设计上存在着根本性的差异，RISC-V 使用相对简单的权限位设计，页表项包含 PTE_V(有效位)、PTE_R(可读位)，PTE_W(可写位)、PTE_X(可执行位)、PTE_U(用户态访问位)，设计直观简介。Loongarch 除以上基本权限位外，还引入了 PTE_D(脏位)、PTE_PLV3(特权级 3)、PTE_MAT（内存访问类型）、PTE_P(物理页存在)、PTE_NR（不可读）等特殊权限位，其中 PTE_NX 和 PTE_NR 的设计体现了 LoongArch 的安全理念，默认禁止执行和读取。

LoongArch 架构的直接映射窗口（Direct Mapping Window，简称 DMWIN）是其内存管理系统的一项创新特性，这种机制通过硬件支持实现了高效的地址转换，显著简化了操作系统的内存管理复杂度。与传统的页表机制相比，直接映射窗口提供了一种更为直接和高效的内存访问方式。虚拟地址的最高 4 位（第 60-63 位）被用作窗口标识符，不同的标识符代表不同的映射窗口。当处理器访问带有特定窗口标识的虚拟地址时，硬件会自动将其转换为对应的物理地址，这个转换过程完全由硬件完成，无需软件干预。LoongArch 架构提供了四个直接映射窗口寄存器（DMWIN0-DMWIN3），每个窗口都可以独立配置。在系统启动时，这些窗口需要在 entry.S 中进行初始化配置。

3.4.2 用户地址空间

用户虚拟地址空间的设计遵循现代操作系统的分层内存管理理念，通过虚拟内存机制为每个进程提供独立的地址空间。用户虚拟地址空间被划分为多个不同功能的区域，每个区域都有特定的用途和访问权限。这种分区设计不仅提高了内存管理的效率，还增强了系统的安全性。程序的代码段通常从虚拟地址 0 开始加载。

栈区域是用户虚拟地址空间中一个特殊的区域，它承担着函数调用、局部变量存储和参数传递等重要功能。在这个系统中，栈的设计体现了精细化的内存管理理念。栈的分配通过 alloc_vma_stack 函数实现，该函数不仅分配物理内存，还建立相应的 VMA（Virtual Memory Area）结构来管理这个内存区域。

SC 使用 VMA 结构管理用户地址空间，VMA 结构记录了内存区域的类型、权限、起始地址、大小等信息，为后续的内存管理操作提供了重要的元数据。VMA 管理采用双向链表结构，每个进程维护一个 VMA 链表来记录其所有的内存区域。这种设计支持高效的区域查找、插入和删除操作。vma_init 函数负责初始化进程的 VMA 管理结构，创建头节点并分配初始的内存映射区域。内存区域的分配通过 alloc_vma 函数实现，该函数不仅分配 VMA 结构体，还根据需要分配实际的物理内存页面。函数支持延迟分配机制，即只分配虚拟地址空间而不

立即分配物理内存，直到实际访问时才通过页面错误处理机制分配物理页面。VMA 的权限管理支持动态修改，experm 函数可以在运行时修改页表项的权限位，支持内存保护等高级功能。这种设计为实现写时复制、内存保护、调试支持等功能提供了基础。

kernel/vma.h

```
struct vma
{
    enum segtype type; // 内存区域类型（MMAP、STACK等）
    int perm;          // 权限
    uint64 addr;       // 起始地址
    uint64 size;       // 大小
    uint64 end;        // 结束地址
    int flags;         // 标志
    int fd;            // 关联的文件描述符
    uint64 f_off;      // 文件偏移
    struct vma *prev; // 前驱节点
    struct vma *next; // 后继节点
};
```

3.4.3 文件映射

文件映射是一种将文件内容映射到进程虚拟地址空间的机制，它允许进程像访问内存一样直接访问文件数据，从而避免了传统的文件 I/O 操作。在 SC7 中，mmap 系统调用通过 VMA（Virtual Memory Area）结构来管理这些映射区域，为进程提供了一个统一的虚拟内存接口。

当进程调用 mmap 时，操作系统首先会分配一个 VMA 结构体来记录映射区域的信息，包括起始地址、大小、权限、关联的文件描述符和偏移量等。这个 VMA 结构会被插入到进程的 VMA 链表中，建立起虚拟地址空间到文件内容的映射关系。

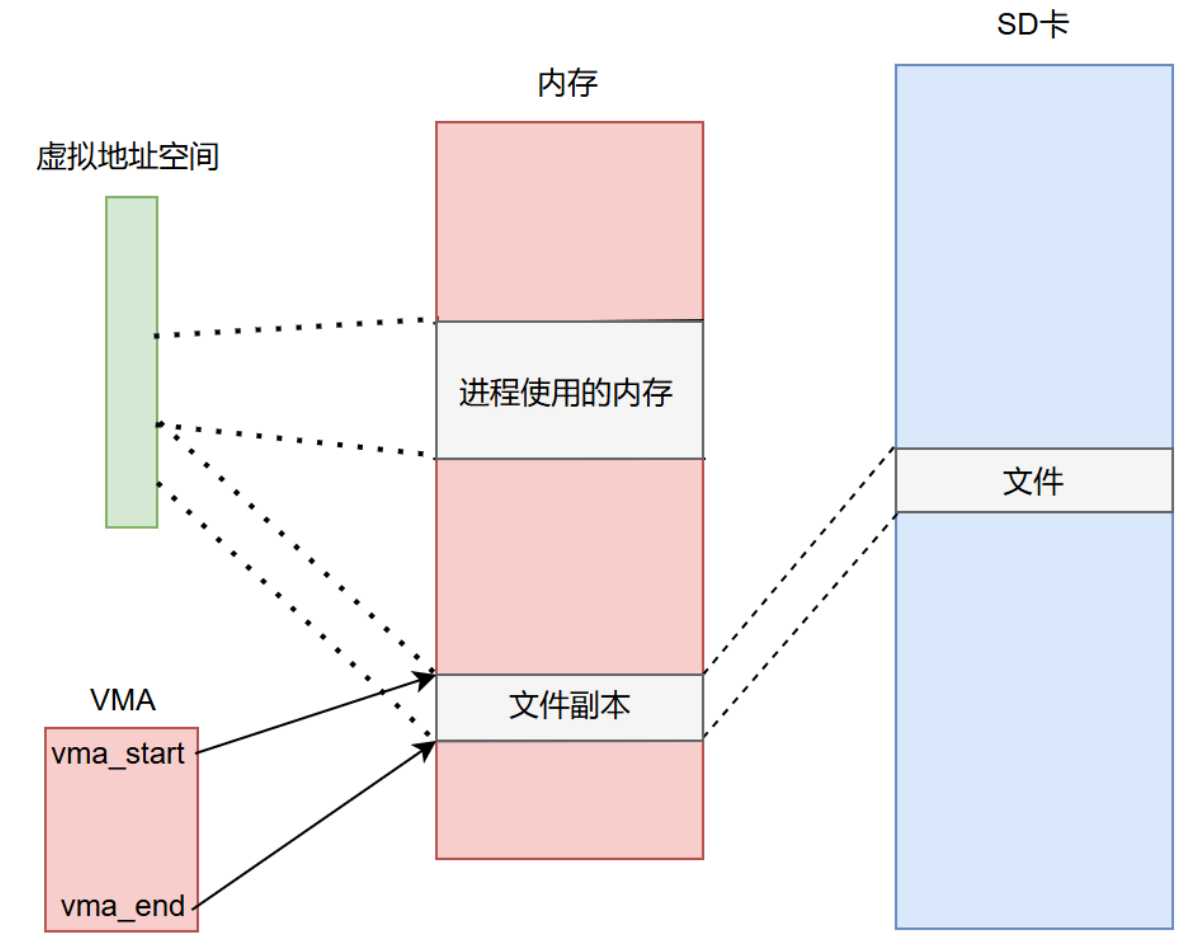


图 3.5　文件映射机制

在内存管理子系统中，mmap 系统调用提供了三种不同类型的映射机制以满足不同场景的内存需求。匿名映射机制通过 **MAP_ANONYMOUS** 标志创建与文件无关的纯净内存区域，内核将物理页面初始化为全零状态，适用于进程内部的大容量内存分配需求，例如替代传统堆内存管理或存储临时数据结构。这种映射完全独立于文件系统，不涉及任何持久化存储操作。

文件私有映射采用 **MAP_PRIVATE** 标志建立进程专属的文件数据副本，内核将文件内容加载到私有内存空间但保持原始文件不变。当进程尝试修改映射区域时，写时复制机制触发物理页面的复制操作，确保修改仅存在于当前进程的私有副本中。这种机制特别适用于加载可执行程序代码或配置文件等只读场景，既提供文件数据访问能力，又隔离进程修改影响。

文件共享映射通过 **MAP_SHARED** 标志构建多进程协作通道，将文件内容直接映射

到进程地址空间。内核保证所有映射同一文件的进程共享相同的物理页面，任何进程对映射区域的修改不仅实时可见于其他映射进程，还会通过底层文件系统同步持久化到存储设备。这种机制为进程间大数据交换提供高效通道，同时支持将内存数据持久化存储，常用于数据库缓存和进程间通信场景。

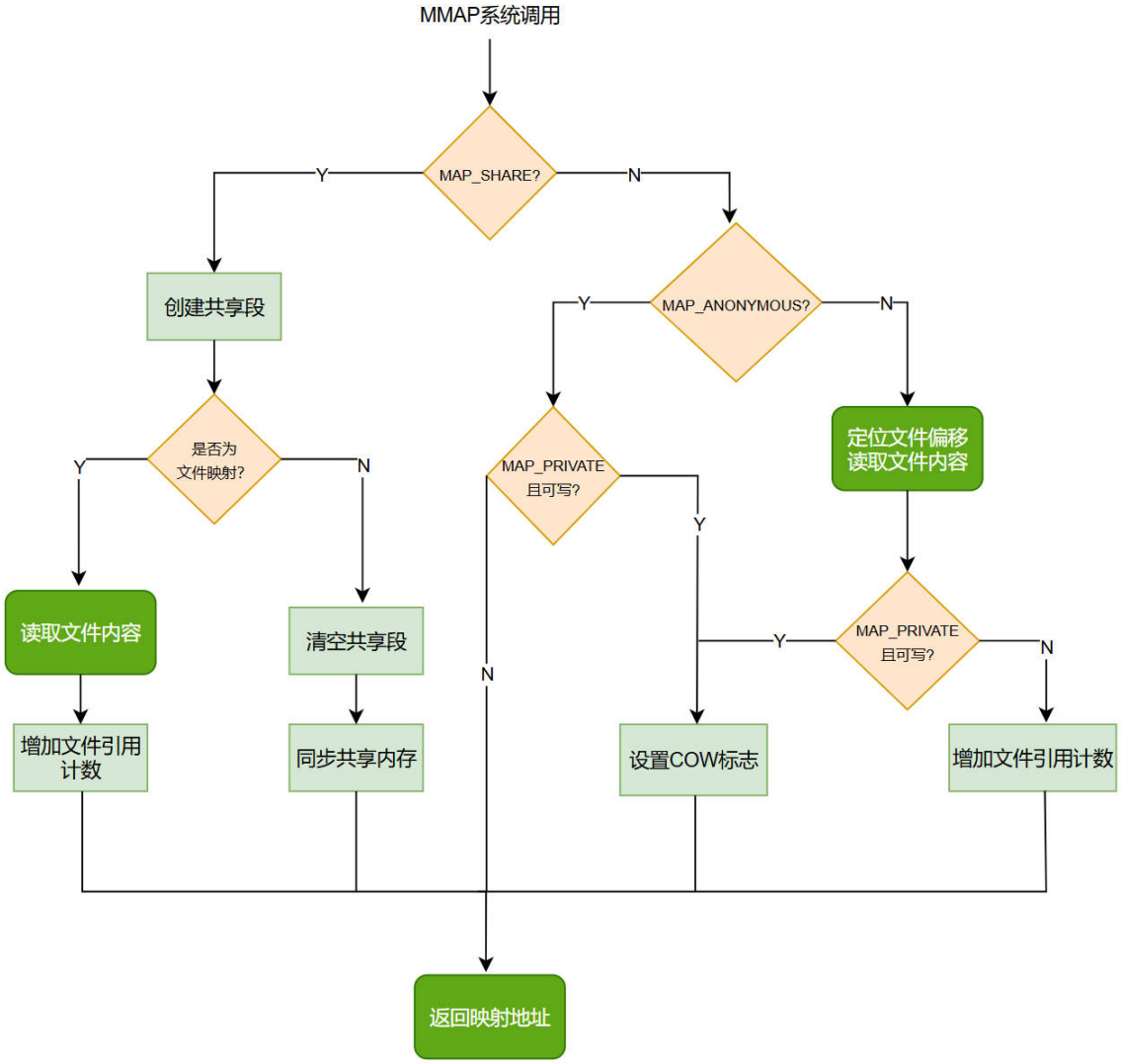


图 3.6　mmap 系统调用流程

3.5　懒分配

SC7 的内存映射机制采用**懒分配**策略，核心思想是**按需分配资源**以提升效率。在 mmap 调用时，系统仅建立虚拟地址空间映射（通过 VMA 结构记录范围、权限和文件信息），暂不分配物理内存或加载文件内容；当进程首次访问该内存区域触发缺页异常时，缺页处理程序 pagefault_handler 才动态分配物理页、建立页表映射，并根据 VMA 信息从关联文件读取数

据到内存。

当发生页面错误时，系统首先会记录错误地址，然后获取当前进程的 VMA 链表。页面错误处理器会遍历 VMA 链表，查找包含错误地址的 VMA 结构。这个过程通过比较错误地址与每个 VMA 的起始地址和结束地址来完成，如果找到匹配的 VMA，就说明这是一个合法的内存访问请求。找到对应的 VMA 后，系统会分配一个物理页面，并将这个物理页面映射到发生错误的虚拟地址。这个映射过程通过调用 mappages 函数来完成，该函数会在页表中建立虚拟地址到物理地址的映射关系，并设置相应的权限位。如果 VMA 关联了文件，系统还会从文件中读取相应的内容到新分配的物理页面中。

页面错误处理的核心在于其**按需分配**的特性。系统不会在进程创建时就分配所有的物理内存，而是等到进程实际访问某个虚拟地址时才分配对应的物理页面。这种设计大大提高了内存利用效率，特别是对于使用 mmap 映射大文件的场景，因为只有被实际访问的页面才会占用物理内存。

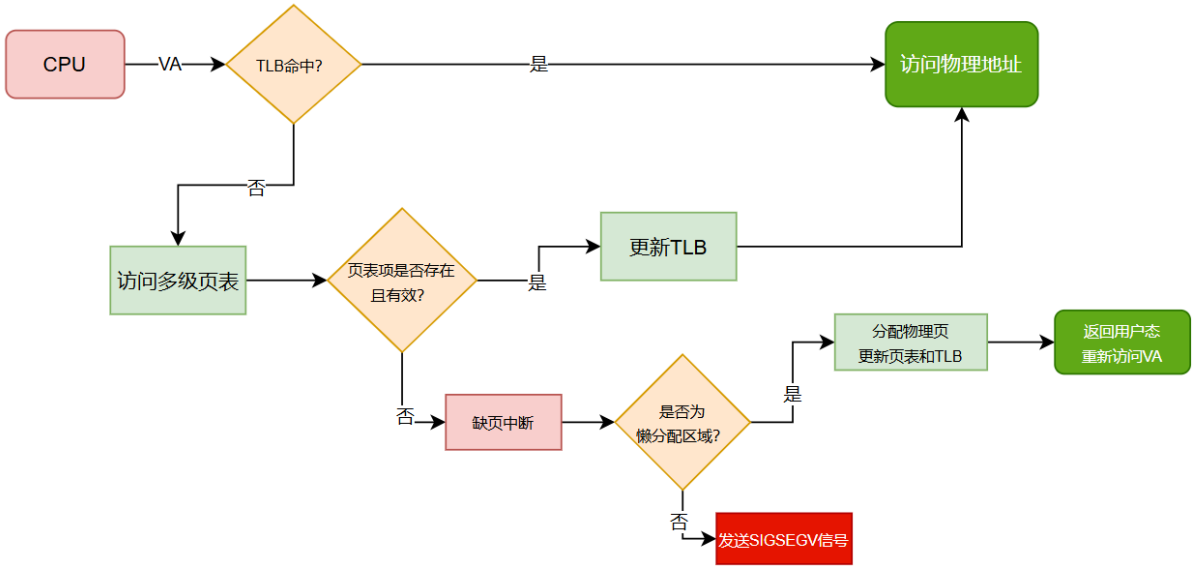


图 3.7　用户空间访存全流程

4 硬件抽象层

4.1 概述

SC7 操作系统的硬件抽象层采用了两层架构设计：**硬件服务抽象接口（Hardware Service Abstract Interface）**和**硬件抽象层（Hardware Abstraction Layer）**。这种设计实现了内核与硬件之间的完全解耦，提供了更加灵活和可靠的系统架构。

这样设计有以下优点：

- **架构无关性**: 通过条件编译和统一接口，支持多种硬件架构
- **代码复用**: 最大化共享代码，减少重复实现
- **易于扩展**: 便于添加新的硬件架构支持

4.2 整体架构

4.3 HAL 层设计

4.3.1 整体架构设计

HAL 层的整体架构采用分层设计，按照硬件架构进行组织。每个架构目录下包含该架构特有的硬件初始化代码、设备驱动、汇编代码和链接脚本。这种设计使得不同架构的实现相互独立，便于维护和扩展。HAL 层与 HSAI 层通过统一的接口进行交互，HSAI 层调用 HAL 层的具体实现，HAL 层则直接操作硬件寄存器。

HAL 层用来处理汇编级别的差异和架构特有功能。HAL 层两种架构共有的功能是: 启动 entry.S, 内核态中断与异常 kernelvec.S, 用户态中断与异常 uservec.S, 进程上下文切换 swtch.S, 还有一个串口输出 uart.c。HAL 层架构特有功能包括：RISC-V 的 sbi 和 start.c;Loongarch 的 tlb 重填和 merrvec.S

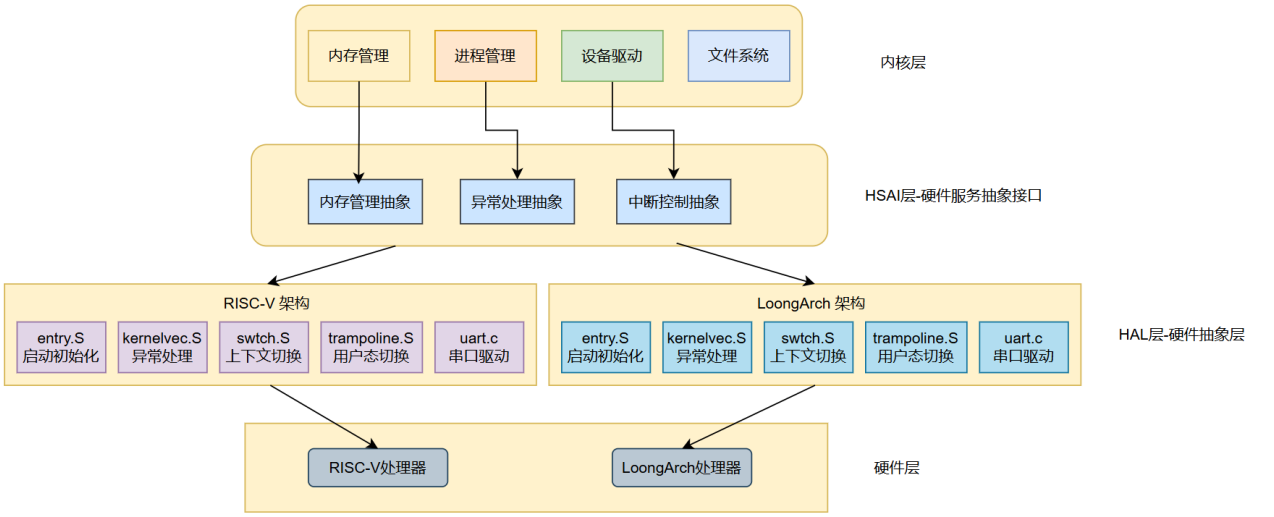


图 4.1　整体架构设计

4.3.2　启动初始化设计

在系统启动阶段，LoongArch 架构的初始化过程相对复杂，其首要任务是建立直接映射窗口（DMWIN），为后续的内存访问提供基础。紧接着，系统会为每个 CPU 核心精心配置独立的启动栈，确保它们拥有各自安全的运行环境。完成这些底层硬件环境的设置后，引导流程最终会跳转至内核的核心入口函数 sc7_start_kernel，标志着内核正式接管控制权。相比之下，当 RISC-V 架构选择借助 SBI（Supervisor Binary Interface）固件服务进行引导时，其启动过程则显得更为简洁：核心的操作集中在为启动的 CPU 设置好专属的启动栈，随后便直接调用内核的 start 函数进入内核世界，跳过了 LoongArch 中所需的 DMWIN 配置等步骤。

4.3.3　中断与异常处理入口

系统对中断与异常的处理路径被清晰地划分，分别由 **trampoline.S** 和 **kernelvec.S** 两个关键的汇编文件实现。trampoline.S 专门负责处理源自用户态（User Mode）的陷入（Trap），而 kernelvec.S 则接管发生在内核态（Kernel Mode）下的中断与异常。硬件抽象层（HAL）负责**保存和后续恢复陷入发生时处理器的完整上下文信息**（即 trapframe），确保执行流能正确回溯。

4.3.4 进程上下文切换

进程（或线程）间的上下文切换是实现多任务的核心，其底层实现在 **swtch.c**（及相关汇编）中完成。由于 LoongArch 和 RISC-V 这两种处理器架构在定义哪些寄存器由被调用者（callee）负责保存（callee-saved registers）方面存在差异，这种差异无法在纯 C 语言层面完全统一处理。因此，上下文切换的核心保存与恢复操作必须在汇编语言级别进行精细控制。swtch.c 提供的函数封装了这部分汇编逻辑，确保在切换进程时能够准确、高效地保存当前执行流的寄存器状态（上下文）并加载新进程的上下文，同时严格遵守不同架构对 callee-saved 寄存器的约定。

4.3.5 设备驱动设计

在硬件抽象层(HAL)中,设备驱动扮演着连接硬件与操作系统的桥梁角色,其核心职责聚焦于硬件设备的初始化和提供基础的操作接口。目前，该层已实现的主要驱动是针对 **UART**（通用异步收发传输器）串口设备。该驱动为系统提供了至关重要的调试信息输出能力，是早期启动和诊断的核心工具。在实现方式上，UART 驱动采用了典型的寄存器映射（Memory-Mapped I/O, **MMIO**）技术来访问硬件。通过精确定义描述 UART 控制寄存器的结构体布局及其在内存地址空间中的偏移量，驱动能够通过读写这些映射到内存地址的寄存器，直接而高效地控制串口设备的数据发送、接收以及状态查询等基本操作。

4.4 HSAI 层设计

4.4.1 异常与中断抽象

HSAI 层通过 **hsai_trap.c** 文件实现了异常与中断处理的架构抽象。该模块使用条件编译宏来区分不同架构的实现，同时提供统一的函数接口。核心数据结构是 Trapframe，它包含了通用寄存器、架构相关寄存器和内核相关信息。对于 RISC-V 架构，使用 epc 寄存器保存异常程序计数器；对于 LoongArch 架构，使用 era 寄存器保存异常返回地址。

异常处理流程分为用户态和内核态两种情况。用户态异常处理首先根据架构设置相应的 trap 入口，然后保存程序计数器到 trapframe，接着根据异常类型进行不同处理：系统调用会将程序计数器加 4 并调用 syscall 函数，缺页异常会调用 pagefault_handler 分配页面，时钟中断会调用 devintr 处理中断，其他异常则进行 panic 或特殊处理。内核态异常处理类似，但会进行特权级检查，并且对于断点异常会直接将程序计数器加 4 跳过断点指令。

中断处理通过 devintr 函数实现抽象。RISC-V 架构通过读取 scause 寄存器判断中断类型，外部中断通过 PLIC 控制器处理，时钟中断直接调用 timer_tick 函数。LoongArch 架构通过读取 estat 和 ecfg 寄存器判断中断类型，硬件中断和定时器中断分别处理。异常类型映射表显示了两种架构的异常类型对应关系，如 RISC-V 的 UserEnvCall 对应 LoongArch 的 0xb，都表示系统调用。

4.4.2 内存管理抽象

内存管理抽象通过 **hsai_mem.c** 文件实现，主要提供内存初始化和页表配置两个核心功能。hsai_get_mem_start 函数获取内核内存起始地址，虽然两种架构的实现相同，但通过条件编译保持接口一致性。hsai_config_pagetable 函数负责配置页表并开启分页模式，这是架构差异最大的地方。

RISC-V 架构使用 sv39 分页模式，通过设置 **satp** 寄存器开启分页，然后使用 sfence_vma 指令刷新 TLB。LoongArch 架构需要设置 **pgdl** 和 **pgdh** 寄存器配置页表基地址，使用 invtlb 指令刷新 TLB，然后配置 TLB 页大小、ASID、虚页号等参数，最后设置页表宽度和基地址配置寄存器。

内存布局方面存在显著差异。RISC-V 的内核基地址为 0x80000000L，物理内存结束地址为 0xb0000000L。LoongArch 的内核基地址为 0x9000000090000000L，物理内存结束地址为 0x90000000b0000000L。这种差异反映了两种架构的地址空间设计理念不同。

4.4.3 中断控制抽象

中断控制器抽象主要体现在 **plic.c** 文件中，该文件专门为 RISC-V 架构的 PLIC（Platform-Level Interrupt Controller）提供支持。PLIC 初始化包括设置 IRQ 优先级和配置 hart 的中断使能位。plic_claim 函数用于获取当前中断请求，plic_complete 函数用于完成中断处理。

LoongArch 架构使用 **PCIE** 中断机制，与 RISC-V 的 **MMIO** 方式不同，因此没有对应的 PLIC 实现。这种差异反映了两种架构在中断处理设计上的不同思路。RISC-V 采用统一的平台级中断控制器，而 LoongArch 则依赖 PCIE 标准的中断机制。

5 系统调用

5.1 系统调用

SC7 通过了 basic，busybox，libctest、libcbench、lmbencht、部分 ltp 测例，实现了 130 余个系统调用

5.2 用户态异常与中断处理

异常与中断处理的逻辑是架构相关的。为了支持两个架构，SC7 在 hsai 层对异常和中断进行处理，并向上提供函数接口，使得内核可以不必关心架构的差异。hsai_trap.c 集中了异常与中断处理函数。为了支持两种架构，使用条件宏来包含对应架构头文件和编译对应架构代码。编译 riscv 架构的内核时，传入参数-DRISCV=1; 编译 loongarch 架构的内核时，不传入。

hsai/hsai_trap.c

```
#if defined RISCV
#include "riscv.h"
#include "riscv_memlayout.h"
#else
#include "loongarch.h"
#endif

void hsai_trap_init(void)
{
#if defined RISCV
    // w_sie(r_sie() | SIE_SEIE | SIE_STIE | SIE_SSIE); //使用sbi不需要自行设置
    w_stvec((uint64)kernelvec); ///< 设置内核trap入口
#else
    uint32 ecfg = (0U << CSR_ECFG_VS_SHIFT) | HWI_VEC | TI_VEC; ///< 例外配置
    w_csr_ecfg(ecfg);                                          ///< 设置例外配置
    w_csr_eentry((uint64)kernelvec);                           ///< 设置内核trap入口
    w_csr_tlbrentry((uint64)handle_tlbr);                      ///< TLB重填exception
    w_csr_merrentry((uint64)handle_merr);                      ///< 机器exception
    timer_init();                                              ///< 启动时钟中断
```

```
    /*busybox需要开启浮点扩展*/
    w_csr_euen(FPE_ENABLE);
#endif
}
```

用户态异常与中断通过 trampoline 进入内核态，在 usertrap 处理异常和中断。如果 usertrap 判断是系统调用，就调用 syscall，然后调用 usertrapret 返回 epc/era 加四的用户地址继续执行。如果是中断，则处理对应中断后，返回当前 epc 的地址。如果是异常，就进入对应的异常处理程序，例如如果是 Lazy Alloc 触发缺页异常，则分配对应页后返回当前 epc/era，如果是其他访存异常则 panic。

5.2.1 中断和异常处理流程

读取状态寄存器，判断异常后，进行对应的处理

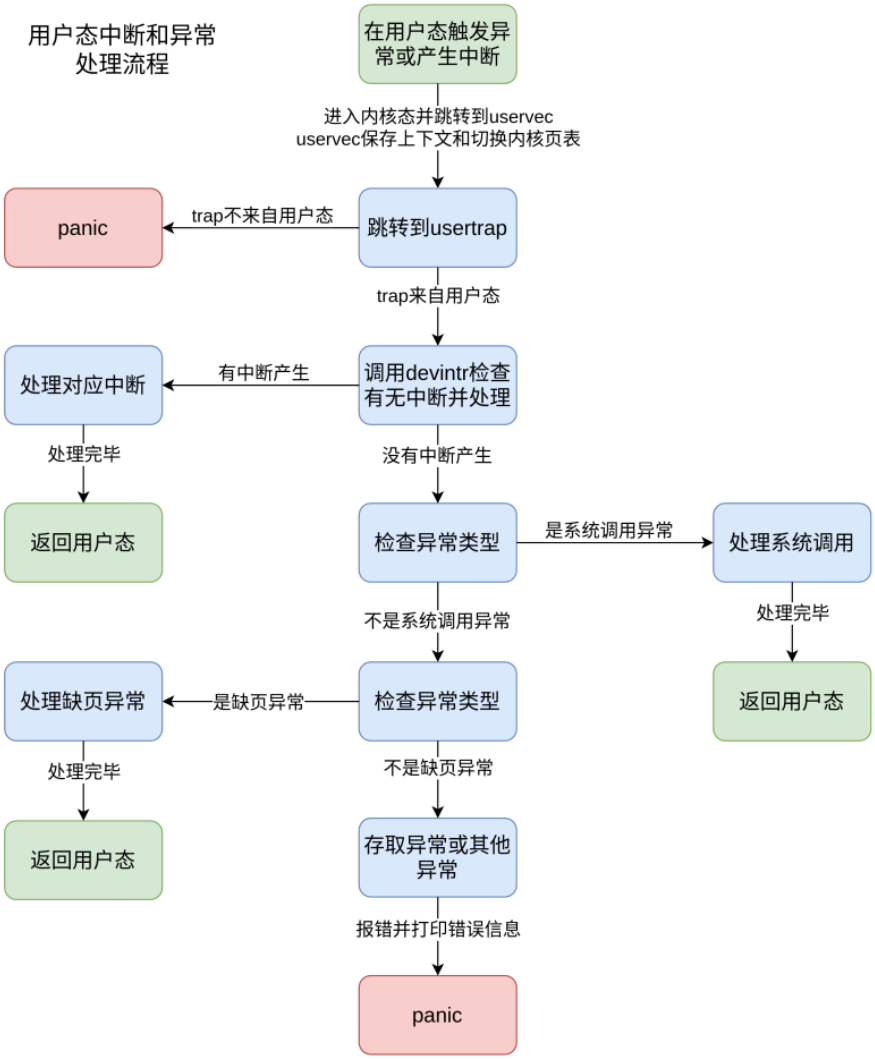


图 5.1　用户态中断和异常处理流程

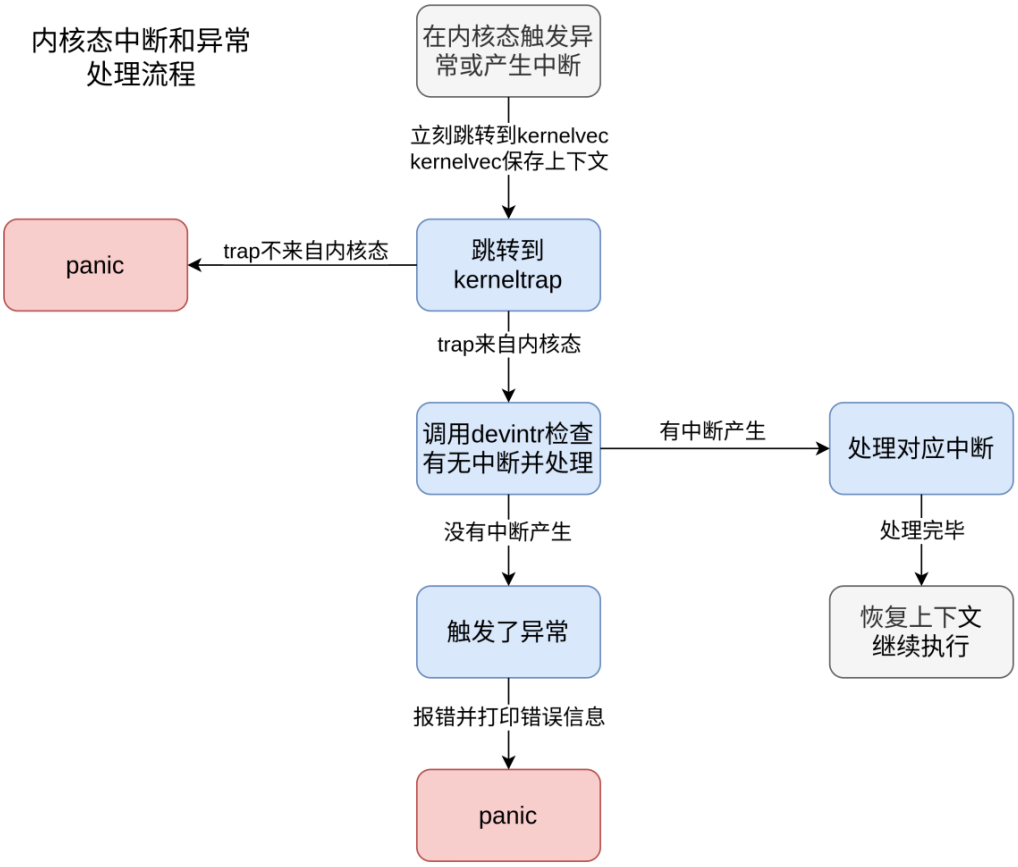


图 5.2　内核态中断和异常处理流程

5.2.2　系统调用

在 usertrap 中判断时系统调用，进入 syscall 处理后，将返回地址 +4 并返回。syscall 逻辑如下，使用 hsai_get_arg 获取参数后，根据调用号调用对应的 sys_ 函数进行处理

kernel/syscall.c

```
void syscall(struct trapframe *trapframe)
{
    for (int i = 0; i < 8; i++)
        a[i] = hsai_get_arg(trapframe, i);
    long long ret = -1;

    switch (a[7])
    {
    ......
    case SYS_execve:
        ret = sys_execve((const char *)a[0], (uint64)a[1], (uint64)a[2]);
```

```
        break;
    ......
    default:
        ret = -1;
        panic("unknown syscall with a7: %d", a[7]);
    }
    trapframe->a0 = ret;
}
```

5.3 内核态异常与中断处理

内核态异常与中断的处理逻辑类似于用户态，区别在于内核态异常与中断使用 kerneltrap 进行处理，并且保存 trapframe 在栈上而不是进程的 *trapframe 地址，且内核态不会有系统调用

值得一提的是，SC7 开发到支持 busybox 时，loongarch 架构的文件系统模块的部分位置出现了 ebreak 调试断点指令，触发了断点异常。对此的处理是：在 kerneltrap 的 loongarch 部分增加判断断点异常的逻辑，如果是，则将 era+4 后 ertn，即可跳过断点指令。

5.4 进程相关系统调用

表 5.1 SC7 操作系统进程管理相关系统调用

系统调用	功能说明
fork (300)	创建子进程，复制父进程的地址空间和资源
clone (220)	创建新进程或线程，支持共享地址空间和资源
clone3 (435)	扩展的进程创建接口，提供更多控制选项
execve (221)	执行新程序，替换当前进程的地址空间
exit (93)	终止当前进程，释放资源并通知父进程
exit_group (94)	终止进程组中的所有进程
wait (260)	等待子进程状态改变，获取子进程退出信息
getpid (172)	获取当前进程的进程 ID

续下页

表 5.1 – *(续表)*

系统调用	功能说明
getppid (173)	获取父进程的进程 ID
gettid (178)	获取当前线程的线程 ID
getuid (174)	获取当前进程的真实用户 ID
geteuid (175)	获取当前进程的有效用户 ID
getgid (176)	获取当前进程的真实组 ID
getegid (177)	获取当前进程的有效组 ID
setuid (146)	设置进程的用户 ID
setgid (144)	设置进程的组 ID
setsid (157)	创建新的会话并设置进程组 ID
kill (129)	向指定进程发送信号
tkill (130)	向指定线程发送信号
tgkill (131)	向指定进程的指定线程发送信号
sched_yield (124)	主动让出 CPU，允许其他进程运行
gettimeofday (169)	获取当前系统时间
clock_gettime (113)	获取高精度时钟时间
clock_nanosleep (115)	高精度睡眠，支持相对和绝对时间
sleep (101)	进程睡眠指定秒数
times (153)	获取进程和子进程的 CPU 时间统计
getrusage (165)	获取进程资源使用情况统计
sysinfo (179)	获取系统整体信息，如内存使用、负载等
uname (160)	获取系统信息，如操作系统名称、版本等
pipe2 (59)	创建管道，用于进程间通信
futex (98)	快速用户空间互斥锁，用于线程同步

续下页

表 5.1 – *(续表)*

系统调用	功能说明
set_robust_list (99)	设置健壮互斥锁列表，处理线程异常退出
get_robust_list (100)	获取健壮互斥锁列表
set_tid_address (96)	设置线程 ID 地址，用于线程清理
prlimit64 (261)	获取或设置进程资源限制
getrandom (278)	获取随机数，用于加密和安全应用
waitid (95)	等待特定子进程的状态变化
clock_getres (114)	获取时钟的时间分辨率
futex_waitv (449)	等待多个 futex 变量
rt_sigaction (134)	设置信号处理函数
rt_sigprocmask (135)	设置信号掩码
getrlimit (163)	获取进程资源限制
rt_sigtimedwait (137)	带超时的等待信号
getsid (156)	获取进程的会话 ID
sched_setaffinity (122)	设置进程的 CPU 亲和性
sched_getaffinity (123)	获取进程的 CPU 亲和性
getcpu (168)	获取当前 CPU 和 NUMA 节点信息
setpgid (154)	设置进程组 ID
getpgid (155)	获取进程组 ID
setresuid (147)	设置真实、有效和保存的用户 ID

续下页

表 5.1 – *(续表)*

系统调用	功能说明
sched_get_priority_max (125)	获取调度策略的最大优先级
sched_get_priority_min (126)	获取调度策略的最小优先级
setgroups (159)	设置进程的附加组 ID 列表
getgroups (158)	获取进程的附加组 ID 列表
setresgid (149)	设置真实、有效和保存的组 ID
getresgid (150)	获取真实、有效和保存的组 ID
getresuid (148)	获取真实、有效和保存的用户 ID
setreuid (145)	设置真实和有效用户 ID
setregid (143)	设置真实和有效组 ID
prctl (167)	进程控制（设置名称、能力等）
settimer (103)	设置定时器
sigreturn (715)	从信号处理函数返回

5.5 内存管理相关系统调用

表 5.2 SC7 操作系统内存管理相关系统调用

系统调用	功能说明
brk (214)	调整程序数据段边界，用于动态内存分配，改变堆空间大小
mmap (222)	内存映射，将文件或设备映射到进程地址空间，支持匿名映射
munmap (215)	取消内存映射，释放通过 mmap 分配的内存区域
mprotect (226)	修改内存页保护属性，如设置读写执行权限
mremap (216)	重新映射内存区域，调整已映射内存的大小和位置
madvise (233)	向内核提供内存使用建议，优化内存管理策略

续下页

表 5.2 – *(续表)*

系统调用	功能说明
membarrier (283)	内存屏障，确保多核处理器间的内存操作顺序
msync (227)	同步内存映射与文件内容
shmget (194)	创建或获取共享内存段
shmat (196)	附加共享内存段到进程地址空间
shmdt (197)	分离共享内存段
shmctl (195)	控制共享内存段（状态、权限等）

5.6 文件系统相关系统调用

表 5.3 SC7 操作系统文件系统相关系统调用

系统调用	功能说明
openat (56)	打开文件，支持相对路径和目录文件描述符
read (63)	从文件描述符读取数据到缓冲区
write (64)	将缓冲区数据写入文件描述符
readv (65)	向量化读取，支持多个缓冲区的分散读取
writev (66)	向量化写入，支持多个缓冲区的聚集写入
pread (67)	从指定偏移量读取文件数据，不改变文件指针
close (57)	关闭文件描述符，释放相关资源
dup (23)	复制文件描述符，创建新的文件描述符指向同一文件
dup3 (24)	复制文件描述符并指定新描述符编号
fstat (80)	获取文件状态信息，包括大小、权限、时间戳等
statx (291)	扩展的文件状态获取，提供更详细的文件属性信息
fstatat (79)	相对于目录文件描述符获取文件状态
getcwd (17)	获取当前工作目录的绝对路径
chdir (49)	改变当前工作目录

续下页

表 5.3 – *(续表)*

系统调用	功能说明
mkdirat (34)	创建目录，支持相对路径
mknod (16)	创建特殊文件，如设备文件、命名管道等
unlinkat (35)	删除文件或目录，支持相对路径
renameat2 (276)	重命名文件或目录，支持原子操作
getdents64 (61)	读取目录项，获取目录中的文件和子目录列表
mount (40)	挂载文件系统到指定目录
umount (39)	卸载已挂载的文件系统
statfs (43)	获取文件系统统计信息，如总空间、可用空间等
llseek (62)	设置文件读写位置指针
ftruncate (46)	截断文件到指定长度
fsync (82)	将文件缓冲区数据同步到磁盘
sync (81)	将所有文件系统缓冲区同步到磁盘
faccessat (48)	检查文件访问权限，支持相对路径
utimensat (88)	设置文件访问和修改时间
readlinkat (78)	读取符号链接的目标路径
fcntl (25)	文件描述符控制，设置文件锁、非阻塞模式等
ioctl (29)	设备控制接口，用于设备特定的操作
sendfile64 (71)	在文件描述符间高效传输数据
fchdir (50)	通过文件描述符改变当前工作目录
chroot (51)	改变进程的根目录
shutdown (1000)	关闭套接字通信
faccessat2 (439)	扩展的文件访问权限检查
ppoll (73)	带超时的多路复用 I/O
socket (198)	创建通信端点（套接字）

续下页

表 5.3 – *(续表)*

系统调用	功能说明
bind (200)	绑定套接字到地址
getsockname (204)	获取套接字绑定的地址
setsockopt (208)	设置套接字选项
sendto (206)	发送数据到指定地址
recvfrom (207)	从指定地址接收数据
listen (201)	监听套接字连接请求
connect (203)	建立套接字连接
accept (202)	接受套接字连接
umask (166)	设置文件创建掩码
fchmod (52)	修改文件权限
fchmodat (53)	相对路径修改文件权限
fchmodat2 (452)	扩展的文件权限设置（带标志）
fchownat (54)	相对路径修改文件所有者和组
fallocate (47)	预分配文件空间
mknodat (33)	相对路径创建设备节点
linkat (37)	相对路径创建硬链接
pwrite64 (68)	在指定偏移写入文件数据
symlinkat (36)	相对路径创建符号链接
fchown (55)	修改文件所有者和组
fgetxattr (10)	获取文件扩展属性
copy_file_range (285)	文件间数据复制
preadv2 (286)	带标志的分散读取
pwritev2 (287)	带标志的聚集写入
splice (76)	零拷贝数据移动

续下页

表 5.3 – *(续表)*

系统调用	功能说明
preadv (69)	分散读取到多个缓冲区
pwritev (70)	聚集写入多个缓冲区
pselect6_time32 (72)	带超时的多路复用（32 位时间）

6 文件系统

6.1 SC7 文件系统简介

文件系统采用分层架构设计，核心是**虚拟文件系统（VFS）抽象层**，底层集成 lwext4 作为主要文件系统实现。VFS 层定义统一的文件操作接口（file_operations）、索引节点操作（inode_operations）和文件系统操作（filesystem_op），通过**函数指针**实现多态行为，使系统能透明支持 EXT4、VFAT 等多种文件系统。物理设备层通过**块设备抽象结构**（devsw）对接不同硬件驱动，当前支持 VirtIO 磁盘和模拟设备，块设备接口包含 bread/bwrite/brelse 等标准操作。缓存机制采用 **LRU 算法**管理的缓冲区（bcache），包含 NBUF 个缓冲块，每个缓冲块通过自旋锁（spinlock）和睡眠锁（sleeplock）实现读写并发控制，显著减少磁盘访问频率。

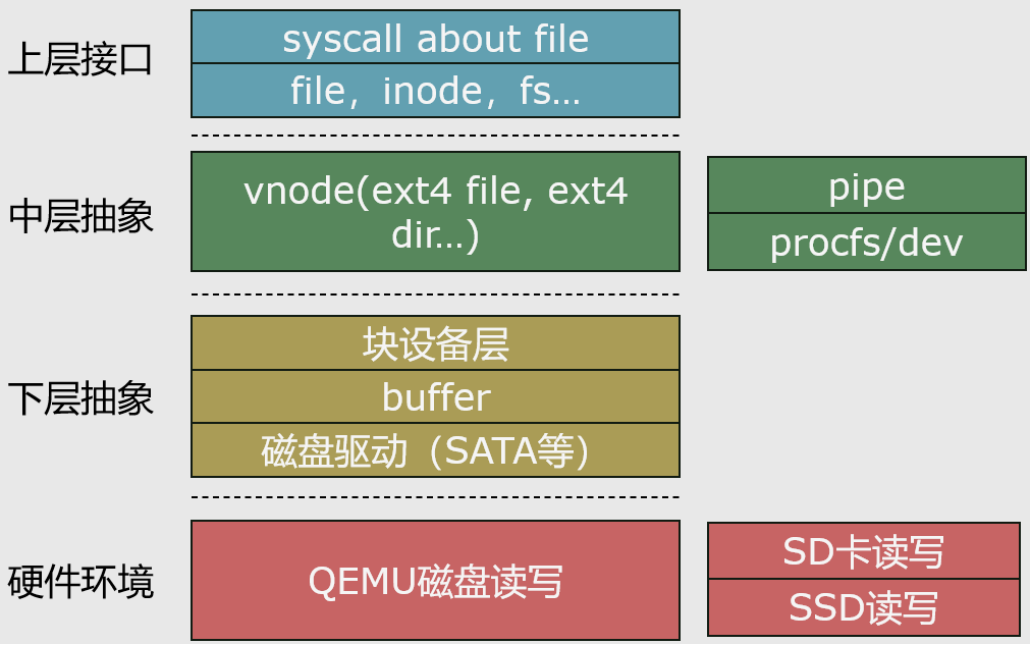


图 6.1 虚拟文件系统架构

6.2 上层接口

6.2.1 VFS 文件对象

文件对象（struct file）作为 VFS 核心实体，通过引用计数（f_count）管理生命周期，支持管道（FD_PIPE）、设备（FD_DEVICE）、常规文件（FD_REG）、套接字（FD_SOCKET）和目录（FD_DIR）等类型。全局文件描述符表（ftable）管理 NFILE 个文件实例，进程通过 ofile 数组持有打开文件，fdalloc 函数实现文件描述符动态分配。vnode 结构（file_vnode_t）抽象文件系统节点，通过 vfs_alloc_dir/vfs_alloc_file 函数动态创建目录和文件对象，由独立锁（file_vnode_table.lock）保护分配过程。设备驱动层实现控制台（console）的完整输入输出处理，包含行编辑功能（退格/整行删除），/dev/null 和/dev/zero 虚拟设备提供数据吞噬和零流生成功能。

include/fs/file.h

```
typedef struct file_vnode
{
    char path[MAXPATH]; /* full path */
    struct filesystem *fs; /* filesystem */
    void *data;         /* file-specific data
                         * e.g. ext4 file/dir */
} file_vnode_t;

union file_data
{
    struct pipe *f_pipe; //< FD_PIPE
    struct fifo *f_fifo; //< FD_FIFO
    struct socket *sock; //< FD_SOCKET
    prth_info_t *pti;  //< FD_PROCFS
    file_vnode_t f_vnode; //< FD_REG
};

struct file
{
    enum { FD_NONE, FD_PIPE, FD_REG, FD_DEVICE,FD_SOCKET, FD_BUSYBOX, FD_PROCFS }
        ↪ f_type;
    uint8 f_mode;      ///< 访问模式
    uint f_flags;      ///< 打开文件时的标志（如O_APPEND等）
```

```
    uint64 f_pos;       ///< 偏移量
    uint16 f_count;     ///< 引用计数，表示有多少用户或进程持有此文件结构
    short f_major;      ///< 设备号（如果是设备文件）

    // void *private_data; ///< 文件私有数据，一般由对应子系统维护
    // int f_owner;        ///< 拥有这个文件的进程ID或进程标识
    char f_path[MAXPATH]; ///< 文件完整路径，便于调试或日志，也可能有管理作用

    uint32 removed;
    union file_data f_data; ///< 文件数据
};
```

6.2.2 VFS 的 Inode 对象

索引节点层（struct inode）封装文件元数据，itable 结构全局管理 NINODE 个 inode 实例。路径解析通过 get_absolute_path 函数处理相对路径转换，自动处理"./"、"../" 等相对路径符号，消除冗余斜杠并规范化路径格式。EXT4 文件系统实现包含完整 POSIX 接口：vfs_ext4_mount 函数初始化块设备并注册文件系统；vfs_ext4_openat 使用**惰性加载策略**打开文件，支持 O_RDONLY/O_WRONLY 等标准标志；读写操作通过分页机制处理用户/内核空间缓冲，vfs_ext4_read 函数实现 PGSIZE 粒度的分块传输；文件元数据（stat）从磁盘 inode 直接映射，时间戳支持 30 位纳秒精度；目录操作通过 ext4_dir 结构实现，vfs_ext4_getdents 函数转换 ext4 目录项为 linux_dirent64 结构。

include/fs/inode.h

```
    struct inode
{
    uint8 i_dev;
    uint16 i_mode; // 类型 & 访问权限
    uint16 i_type;
    uint32 i_ino; // 编号
    uint32 i_valid; // 是否可用 0 -> 未使用

    uint16 i_count; // 引用计数
    uint16 i_nlink; // 硬链接数
    uint i_uid;   // 拥有者编号
    uint i_gid;   // 拥有者组编号
```

```
    uint64 i_rdev; // 当文件代表设备的时候，rdev代表这个设备的实际编号
    uint64 i_size; // 文件大小

    long i_atime; // 文件最后一次访问时间
    long i_mtime; // 文件最后一次修改时间
    long i_ctime; // inode最后一次修改时间

    uint64 i_blocks; // 文件有多少块
    uint64 i_blksize; // 块大小 bytes
    struct spinlock lock;      // inode锁
    struct inode_operations *i_op; // inode操作函数

    struct superblock *i_sb;

    /*
     * 指向的底层文件系统inode，实际使用的时候采用了
     * 内存复用，临时存储ext4_inode，拿到对应的inode号后
     * 存储对应路径，后续使用路径来访问对应文件
     */
    struct inode_data i_data;
};
```

6.3 中层接口

6.3.1 lwext4 移植

lwext4 作为轻量级嵌入式文件系统实现，通过深度定制化移植方案集成至系统内核。移植核心在于实现 lwext4 所需的块设备驱动接口（ext4_blockdev_iface），该接口包含五个关键操作函数：blockdev_lock/unlock 处理并发控制，blockdev_open/close 管理设备状态，blockdev_read/write 实现扇区级数据传输。块设备抽象层（vfs_ext4_blockdev）通过 vbdev 结构封装物理设备属性，其中 ph_bbuf 作为 DMA 缓冲区，part_size 固定为 4GB 以适应嵌入式场景，part_offset 支持未来分区扩展。

6.3.2 pipe 与字符设备

管道系统（pipe）作为进程间通信核心机制，通过 pipe 结构体实现环形缓冲区管理。每个管道包含独立自旋锁（lock）、数据缓冲区（data）和读写位置（nread/nwrite）。创建时

pipealloc 分配两个文件对象（读端/写端），类型标记为 FD_PIPE。写入操作 pipewrite 在缓冲区满时休眠等待，通过 wakeup 唤醒读取者；读取操作 piperead 在空管道时休眠，支持部分读取满足即时通信需求。引用计数机制确保双端关闭时自动回收资源，pipeclose 函数验证 writable 标志后释放缓冲区内存。

6.4 底层抽象

6.4.1 VFS 的块设备

块设备抽象层(blockdev.c)提供统一接口,vfs_ext4_blockdev 结构包含物理块大小(ph_bsize)、块数(ph_bcnt)等属性。EXT4 文件操作实现高级特性：硬链接(vfs_ext4_link)通过 inode 引用计数实现；权限控制(ext4_mode_set)支持 0777 等标准模式；时间戳更新(vfs_ext4_utimens)处理 UTIME_NOW/UTIME_OMIT 特殊值；崩溃恢复通过 ext4_journal 机制保障数据一致性。VFAT 文件系统作为实验性模块提供基本框架，通过 vfat_blockdev_iface 结构定义块设备接口，其读写操作当前复用 EXT4 的缓冲区管理机制。

6.4.2 块设备读写

块设备读写函数通过缓冲层（bio.c）桥接：blockdev_read 循环调用 bread 获取每个扇区的缓冲块，memmove 复制 BSIZE 字节数据到目标缓冲区，brelse 及时释放缓冲块避免资源泄漏；blockdev_write 采用 bget 直接获取缓冲块避免无效数据读取，memmove 写入后立即 bwrite 提交修改。这种设计充分利用现有缓冲机制，减少零拷贝优化带来的复杂度。设备注册通过 ext4_device_register 将 vbdev 绑定至设备名”ext4disk”，使 lwext4 识别底层存储实体。

6.5 VFS 的故障处理

故障处理机制包含多层防御：路径解析严格限制 MAXPATH 长度防止溢出；文件操作验证 vnode 有效性（EXT4/VFAT 类型检查）；块缓存使用自旋锁（bcache.lock）保证并发安全；inode 操作通过睡眠锁（sleeplock）实现读写同步。异常路径处理包含卸载时脏缓冲区回写（vfs_ext4_flush）、无效 inode 访问（panic 触发）等场景，控制台输入实现 Ctrl+U 整行删除、Ctrl+D 文件结束符等特殊编辑功能。性能优化策略包括：缓冲区缓存减少磁盘 I/O，目录项缓存加速路径解析，文件读写采用 seek 位置缓存避免重复定位。

6.6 虚拟文件

经过拓展，目前我们的虚拟文件系统已经支持虚拟文件。

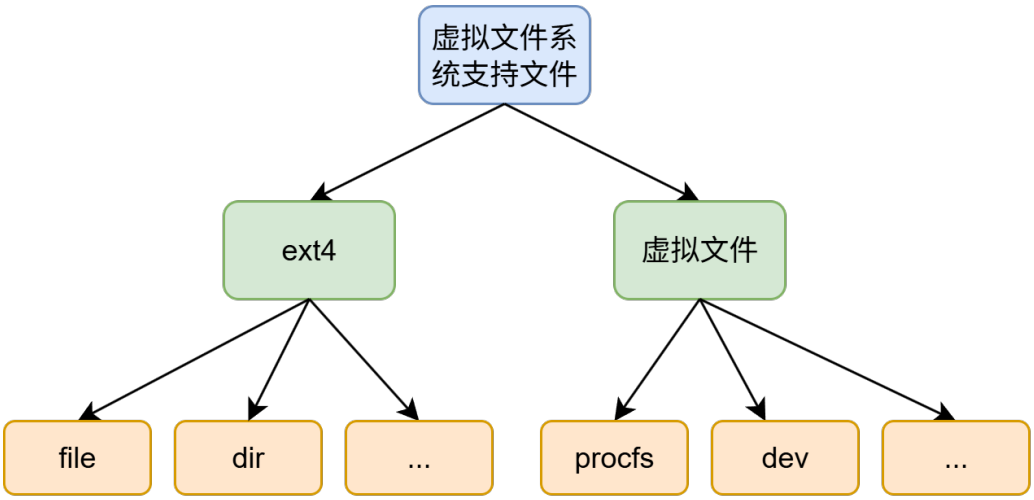


图 6.2 虚拟文件系统支持的文件

6.6.1 PROCFS 实现细节

PROCFS 通过动态内容生成机制实现内核信息透明访问，核心特性包括：

- **中断统计**：interrupt_counts 数组记录 256 个 IRQ 触发次数，通过 interrupt_lock 自旋锁保证原子性，generate_proc_interrupts_content 生成格式化输出
- **进程信息**：is_proc_pid_stat 等路径解析函数支持"self" 符号链接，generate_proc_stat_content 生成 52 字段标准格式，包含进程状态、CPU 时间等关键指标
- **线程模型**：task 目录结构反映进程线程关系，get_thread_by_pid_tid 实现线程查找，状态字符（R/S/Z）与 Linux 标准保持一致

6.6.2 虚拟设备子系统设计

虚拟设备通过统一文件接口提供特殊功能：

- **设备类型标识**：file 结构体 f_type 字段区分 FD_DEVICE 与其他类型，f_major/f_minor 标识主次设备号
- **零成本抽象**：/dev/null 实现即时写入丢弃，read 返回 EOF；/dev/zero 通过预分配 zeros 数组提供高效零流
- **控制台管理**：行编辑支持退格（0x08）和整行删除（0x15）控制字符，输入缓冲区分读写指针与编辑指针

6.7 对于 lwext4 的拓展

6.7.1 原有架构限制

原生 lwext4 在路径解析过程中存在关键缺陷：当路径中间组件为符号链接时，系统会直接返回 ENOENT 错误，而非遵循符号链接进行递归解析。这种设计不符合 POSIX 标准，导致无法处理如"/usr/bin/ls"（其中"/usr" 为符号链接）等常见路径场景。问题根源在于 ext4_generic_open2 函数的路径遍历逻辑仅支持纯目录路径，遇到非目录节点即终止解析。

6.7.2 改进方案实现

通过重构路径解析流程，新增符号链接即时处理机制：

- **符号链接检测**：在路径组件解析阶段(imode 检查),识别到 EXT4_INODE_MODE_SOFTLINK 类型时触发特殊处理
- **递归解析控制**：通过 O_NOFOLLOW 标志检查实现安全控制，当且仅当符号链接为路径末尾组件时遵循标志要求
- **绝对路径处理**：若符号链接目标以'/' 开头，释放当前 inode 引用并重新从根目录开始解析
- **路径重组**：动态拼接原始路径前缀、符号链接目标及剩余路径组件，构建完整解析路径

6.7.3 关键技术创新

- **路径状态保存**：使用 ori_path 保留原始路径前缀，确保符号链接处理后能正确拼接剩余路径
- **资源安全释放**：在切换绝对路径解析前显式释放当前 inode 引用(ext4_fs_put_inode_ref),避免资源泄漏
- **边界条件处理**：严格校验 linkpath 缓冲区长度，防止拼接时溢出；对相对符号链接路径显式 panic 保证系统安全

该增强使 lwext4 完全符合 POSIX 路径解析语义，支持多级符号链接嵌套（需防范循环链接)。

7 信号处理

7.1 信号结构体

信号相关结构体 (kernel/signal.h)

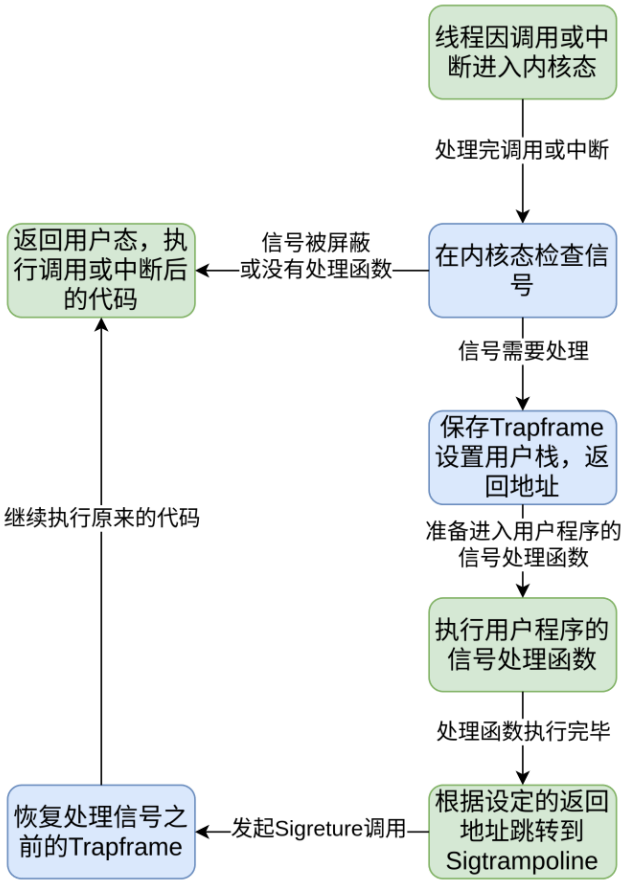
```
// 信号集结构体
#define SIGSET_LEN 1
typedef struct {
    unsigned long __val[SIGSET_LEN];
} __sigset_t;

// 信号动作结构体
typedef struct sigaction {
    union {
        __sighandler_t sa_handler; // 标准处理函数
        // void (*sa_sigaction)(int, siginfo_t *, void *); // 扩展处理函数
    } __sigaction_handler;
    __sigset_t sa_mask;  // 信号处理期间要阻塞的信号集
    int sa_flags;        // 控制信号行为的标志位
} sigaction;
```

7.2 信号处理流程

当线程在系统调用或处理中断结束后，内核调用 check_and_handle_signals() 函数进行信号检查，通过扫描线程的信号位图，跳过被阻塞或显式忽略的信号，找到需要立即处理的最高优先级信号。

当确定要处理的信号后，对于自定义信号，内核保存当前寄存器状态，根据信号标志在用户栈上布置参数信息，最后将程序返回地址 epc/era 设置为用户注册的处理函数地址，将返回地址设置为 sigtrapoline。

当 CPU 返回到用户空间执行信号处理函数完成后，通过 sigtrapoline 触发 sigreturn 系统调用，内核从 sigtrapframe 中取出保存的执行上下文，复原寄存器状态、栈指针，清除处理状态标记，最后进程回到被信号中断的代码位置继续执行。



信号处理机制

图 7.1　信号处理流程

8 总结与展望

8.1 决赛工作总结

从 7 月份开始，决赛阶段我们主要做的工作为：

1. RISC-V、Loongarch 支持多核启动

2. 实现进程组管理，新增权限管理，新增 UID/GID 权限体系，支持 umask 权限掩码，实现了包括 setpgid/getpgid

3. 线程模型重构，完善 futex 和线程信号量机制

4. 完善内存管理，实现共享内存机制，优化 mmap 支持 MAP_PRIVATE 写时复制

5. 虚拟文件系统实现 procfs（/proc/self/stat、/proc/cpuinfo 等）

6. 扩展 lwext4 文件操作, 支持目录项为符号链接，支持 ftruncate 文件扩容

7. 新增信号与中断处理，实现线程级信号处理

8. 新增系统调用，通过 lmbench 和部分 ltp 测试

8.2 未来计划

1. 重构操作系统，增强可读性和可维护性

2. 增加网卡驱动，支持网络功能

3. 支持更多应用程序