Lrix: 一个草稿 (scratch) RISC-V 操作系统

项目: Lrix

GitHub: https://github.com/lrisguan/Lrix

作者: 官稚淇

2026 年 1 月 1 日

目录

摘要

Lrix 是一个针对 RISC-V 架构和 QEMU `virt` 机器平台，从零开始编写的运行在机器模式下的教学操作系统。这也是我为什么称它为 scratch 的原因（也许在将来我会解决这个问题）。本报告对内核与用户空间的设计进行了简要的技术概览，重点介绍了内存管理、进程管理、文件系统、Trap 与中断处理、系统调用接口以及内置 Shell。

1 引言

Lrix 是一个使用 C 语言和 RISC-V 汇编实现的小型教学导向操作系统。它在 QEMU `riscv64 virt` 平台上启动，使用简单的 UART 驱动进行控制台 I/O，并提供了极简的类 UNIX 进程与文件抽象。顶层的 `Makefile` 构建内核镜像和用户程序，并使用 VirtIO 块设备作为后端存储在 QEMU 中运行。注：在开发 Lrix 时，我参考了 [4], [6], [5], [1], [1], [2], [3]。

内核入口点 `kmain` 执行以下高层级步骤：

1. 初始化 UART 以进行串行输出。
2. 初始化 Trap/中断子系统和 PLIC。
3. 初始化物理页分配器和虚拟内存管理器。
4. 初始化调度器并创建 Idle 进程。
5. 初始化 VirtIO 块设备驱动和基于 inode 的简单文件系统。
6. 创建运行 Shell 的初始用户进程，然后开启中断，并通过 `wfi` 进入空闲状态，由定时器中断驱动调度。

2 内核子系统

本节总结了 `kernel/` 下的主要内核模块，重点介绍内存管理、进程管理、文件系统和 Trap 处理。

2.1 引导与入口 (boot/)

`boot/start.S` 中的引导代码负责底层的 CPU 初始化：

- 将初始栈指针设置为链接器提供的 `_stack_top`。
- 通过从 `_bss_start` 迭代到 `_bss_end` 来清零 BSS 段。
- 调用 C 入口点 `kmain`，如果该函数返回，则进入死循环。

这段代码在 Machine 模式下运行，为内核准备一个干净的 C 环境。

2.2 UART 与日志 (uart/, include/log.h)

uart/uart.c 中的 UART 驱动实现了针对 QEMU virt 机器上映射在 0x10000000 的兼容 NS16550 的 UART 的中断 I/O。(对于中断 I/O，你可以开启编译选项 **TRAP_DEBUG** 来查看)。它提供:

- uart_init: 将 UART 配置为 8N1 模式。
- uart_putc, uart_getc, 和 uart_getc_blocking: 用于基本的字符 I/O。
- uart_getline 和一个小型的类似 scanf 的 scank: 用于解析用户输入。
- printk: 一个极简的格式化打印例程，支持标准的整数、字符串和指针格式。

头文件 include/log.h 在 printk 和 ANSI 转义码的基础上构建, 提供了诸如 INFO, WARNING, 和 ERROR 等内核通用的彩色日志宏。

2.3 字符串库 (string/)

string/string.c 文件实现了内核和用户空间使用的一小部分 libc 字符串和内存函数，包括 memset, memcpy, memmove, memcmp, strlen, strcmp, 和 strncmp。这避免了对外部 C 库的依赖。

2.4 底层 RISC-V 辅助函数 (include/riscv.h, include/types.h)

头文件 include/riscv.h 包含常用 RISC-V CSR 的内联汇编辅助函数，特别是 intr_on 和 intr_off，它们用于设置或清除 mstatus 中的机器中断使能 (MIE) 位。这在调度器和 Trap 代码中被广泛用于划定临界区。

头文件 include/types.h 定义了基本类型和 RegState 结构体，该结构体保存调度器和上下文切换机制使用的 RISC-V 寄存器上下文（通用寄存器、sepc、sp 和 mstatus）。

表 1: 寄存器状态结构体字段描述

Field Name	Register Alias	Field Type	Description
x1	ra	uint64_t	返回地址（Return Address）
x5	t0	uint64_t	临时寄存器 0
x6	t1	uint64_t	临时寄存器 1
x7	t2	uint64_t	临时寄存器 2
x8	s0/fp	uint64_t	保存寄存器 0/帧指针
x9	s1	uint64_t	保存寄存器 1
x10	a0	uint64_t	参数/返回值寄存器 0
x11	a1	uint64_t	参数/返回值寄存器 1
x12	a2	uint64_t	参数寄存器 2
x13	a3	uint64_t	参数寄存器 3
x14	a4	uint64_t	参数寄存器 4
x15	a5	uint64_t	参数寄存器 5
x16	a6	uint64_t	参数寄存器 6
x17	a7	uint64_t	参数寄存器 7
x18	s2	uint64_t	保存寄存器 2
x19	s3	uint64_t	保存寄存器 3
x20	s4	uint64_t	保存寄存器 4
x21	s5	uint64_t	保存寄存器 5
x22	s6	uint64_t	保存寄存器 6
x23	s7	uint64_t	保存寄存器 7
x24	s8	uint64_t	保存寄存器 8
x25	s9	uint64_t	保存寄存器 9
x26	s10	uint64_t	保存寄存器 10
x27	s11	uint64_t	保存寄存器 11
x28	t3	uint64_t	临时寄存器 3
x29	t4	uint64_t	临时寄存器 4
x30	t5	uint64_t	临时寄存器 5
x31	t6	uint64_t	临时寄存器 6
sepc	-	uint64_t	异常返回地址
sp	-	uint64_t	栈指针（Stack Pointer）
mstatus	-	uint64_t	机器态状态寄存器

2.5 物理内存管理 (mem/kmem.c)

物理内存分配由 `mem/kmem.c` 中的简单页式分配器执行。分配器由 `kinit` 初始化，后者接收链接描述文件中定义的堆范围。

该实现维护：

- 一个位于堆起始位置的 `Page` 描述符数组。
- 一个 `Page` 对象的单向空闲链表。
- 总页数、空闲页数和已用页数的计数器。

关键操作包括：

- **初始化**: `Page` 描述符数组占用堆起始处的整数个页面；这些页面被标记为已用，其余页面插入空闲链表。
- **分配 (kalloc)**: 从空闲链表中弹出一个 `Page`，将其标记为已用，根据其索引计算物理地址，并在返回给调用者之前将该页面清零。
- **释放 (kfree)**: 验证地址和对齐方式，将其映射回 `Page` 描述符，防止双重释放，并将其推回空闲链表。

诊断辅助函数暴露统计信息（总/空闲/已用页数）并可打印分配器状态摘要。

2.6 虚拟内存管理实现（mem/vmm.c）

虚拟内存管理器（Virtual Memory Manager, VMM）实现了标准的 RISC-V Sv39 页表层级结构，负责内核基础映射（内存 RAM 和内存映射 I/O（MMIO））以及用户堆页面的按需分配。

Sv39 页表布局 Lrix 采用标准的三级 Sv39 页表方案：

- 虚拟地址分为 VPN2（38-30 位）、VPN1（29-21 位）、VPN0（20-12 位）和 4KB 页偏移量（11-0 位）。
- 每个页表页大小为 4KB，包含 512 个 64 位页表项（PTE）。
- 叶子页表项（Leaf PTE）编码物理页号（PPN）以及 Sv39 标志位（V/R/W/X/U/A/D）。
- 非叶子页表项（中间层级）标记为有效（V=1），但清除所有 R/W/X/A/D 位，这符合 Sv39 规范要求。

内核维护一个单独的二级根页表(level-2),名为 `kernel_pd`,其物理地址存储在 `kernel_pd_phys` 中。所有页表内存均通过物理内存分配器（`kalloc`）分配，并在使用前显式初始化为零。

初始化与恒等映射 入口函数 `vmm_init` 执行以下步骤:

- 分配并清零二级根页表,将其虚拟地址和物理地址分别存储到 `kernel_pd` 和 `kernel_pd_phys`。
- 执行最小化自测试（`vmm_self_test`）：将测试虚拟地址映射到新分配的物理页，通过 `vmm_translate` 验证地址转换正确性，然后解除映射并确认转换失效。
- 为 `virt` 虚拟机的关键内存区域建立恒等映射（虚拟地址 = 物理地址）:
  - 主内存（Main RAM）：`0x80000000–0x80000000 + 128MB`，映射为可读可写且用户可访问（`VMM_P_RW | VMM_P_USER`），确保启用分页后内核和用户的代码/数据/栈仍能正常工作。
  - UART MMIO（地址 `0x10000000`）和 VirtIO MMIO 窗口 `[VIRTIO_MMIO_START, VIRTIO_MMIO_END)`，映射为内核独占的可读可写区域。
  - CLINT 定时器（`0x02000000–0x02010000`），映射为内核可读可写。
  - PLIC MMIO 区域（起始地址 `PLIC_BASE`），为中断控制器寄存器分配小型恒等映射窗口。

这些恒等映射确保启用 Sv39 地址转换后，内核仍可使用相同的虚拟地址访问代码、数据、堆和设备寄存器。

启用 Sv39 分页机制 初始化完成后，`kmain` 调用 `vmm_activate` 启用硬件分页:

- 计算页表物理页号（PPN）：`kernel_pd_phys >> 12`。
- 构造 Sv39 格式的 `satp` 寄存器值：MODE=8（表示 Sv39 模式）、ASID=0，低 34 位存储页表 PPN。
- 将该值写入 `satp` 控制状态寄存器（CSR），并执行 `sfence.vma` 指令刷新过时的 TLB 条目。

为保证可移植性，保留了两个占位钩子函数 `arch_set_cr3` 和 `arch_enable_paging`，但在 RISC-V 架构上不执行任何操作；所有实际的分页启用逻辑均通过 `satp` 寄存器实现。

映射、解除映射与地址转换 核心操作基于一个遍历三级页表的辅助函数实现:

- **`get_next_level`**：接收页表指针、索引和分配标志（`alloc`）。若对应页表项无效且分配标志已启用，则分配新的页表页，安装非叶子页表项（有效但无 R/W/X/A/D 位），并返回下一级页表指针。若页表项已有效，则将其物理页号（PPN）转换为（恒等映射的）虚拟地址，作为下一级页表指针返回。

- **vmm_map**（建立映射）：
  - 要求虚拟地址和物理地址均按页对齐。
  - 遍历 L2→L1→L0 页表，按需分配中间层级页表。
  - 将抽象的 VMM 标志位（VMM_P_*）转换为 Sv39 页表项标志位（V/R/W/X/U/A/D）。目前 VMM_P_RW 隐含 R/W/X 权限（以便代码页和数据页无需单独设置执行权限），且主动设置 A/D 位（访问位和脏位）。
  - 在 L0 层级安装叶子页表项，包含目标物理地址和权限标志。
- **vmm_map_page**（映射新物理页）：
  - 通过 kalloc 分配新的物理页并清零，调用 vmm_map 将其映射到指定虚拟地址。
- **vmm_unmap**（解除映射）：
  - 遍历页表（不分配中间层级）。
  - 若找到有效叶子页表项，则清除该页表项；当 free_phys 为非零时，将底层物理页归还给分配器。
  - 目前不回收已空的中间页表（以简化实现，代价是少量内存开销）。
- **vmm_translate**（地址转换）：
  - 对指定虚拟地址执行软件页表遍历，若找到有效页表项，则返回对应的物理地址指针（物理页号 PPN 与原页偏移量组合）。
  - 主要用于调试和内置自测试。

额外辅助函数 vmm_get_page_directory、vmm_set_page_directory 和 vmm_get_pd_phys 用于暴露当前内核根页表，为未来的进程独立地址空间奠定基础（当前系统仍使用单一全局 Sv39 页表）。

Sv39 架构下的用户堆分配 用户堆基于 Sv39 映射机制实现，布局设计简洁：

- 每个进程分配独立的堆区域，起始地址为 HEAP_USER_BASE + pid × PER_PROC_HEAP，其中 HEAP_USER_BASE 为 0x80400000，PER_PROC_HEAP 为 8KB（每个进程堆大小）。
- syscall.c 中的系统调用实现 SYS_SBRK（sys_sbrk）通过调用 vmm_map_page 为每个新增 4KB 页扩展堆区域，始终使用 VMM_P_RW | VMM_P_USER 标志，确保用户代码可读写新映射的内存。
- 全局 Sv39 页表与进程独立堆窗口的组合，实现了进程级连续虚拟堆空间，同时保持整体设计简洁（暂未支持进程专属 satp 寄存器切换）。

2.7 进程管理与调度 (proc/)

进程管理在 `proc/proc.c` 中实现。核心数据结构是进程控制块 (PCB)，包含：

- 进程标识符 (PID) 和父进程 PID (PPID)。
- 调度状态 (READY, RUNNING, BLOCKED, TERMINATED) 和优先级。
- 内核栈顶地址和保存的寄存器上下文 (`RegState`)。
- 每进程堆基址 `brk_base` 和大小 `brk_size`。
- 用于调试和 Shell 输出的进程名称。

表 2: 进程控制块（ProcessControlBlock, PCB）结构体定义

成员名	数据类型	说明
pid	int	进程唯一标识符
pstat	ProcState	进程运行状态（如就绪、运行、阻塞等）
name[20]	char	进程名称（最大支持 20 个字符）
prior	int	进程优先级（值越小，优先级越高）
entrypoint	uint64_t	进程入口点（起始指令地址）
stacktop	uint64_t	进程栈顶的虚拟地址
ppid	int	父进程 ID（内核/初始化进程的父进程 ID 为 0）
brk_base	void *	程序中断点基地址（堆起始地址）
brk_size	uint64_t	已分配堆空间大小（单位：字节）
cpu_time	uint64_t	进程消耗的总 CPU 时间
remain_time	uint64_t	进程调度剩余时间片
arriv_time	uint64_t	进程进入系统的到达时间
regstat	RegState	用于上下文切换的保存寄存器状态
next	PCB *	用于进程队列管理的链表指针

全局变量跟踪当前进程、Idle 进程、就绪队列以及阻塞和僵尸进程的链表。

Idle 进程与调度器初始化 `scheduler_init` 初始化就绪队列并创建一个 PID 为 0 的 *idle* 进程，拥有自己的内核栈和入口函数 `idle_entry`。Idle 进程只是简单地开启中断并在循环中执行 `wfi`，出让 CPU 直到定时器中断到达。

进程创建 存在两条创建路径：

- **`proc_create`** 由内核用于创建初始 Shell 进程。它分配一个 PCB 和内核栈，初始化寄存器状态，以便从第一次上下文切换返回时跳转到给定的入口点，并将进程加入就绪队列。
- **`proc_fork`** 实现 `fork` 系统调用。它克隆父进程的 PCB 和内核栈，调整子进程的栈指针以镜像父进程的栈帧布局，将子进程的 `a0` 寄存器设为 0（使子进程看到 `fork()` 返回 0），并使用 textttvmm_map_page 和 `memcpy` 深度复制用户堆。随后子进程被作为 READY 加入队列。

退出、等待与僵尸进程 当进程调用 `exit` 时，`proc_exit` 将其标记为 TERMINATED，链接到 `zombie_list`，如果父进程阻塞在 `wait` 中，则唤醒父进程。`proc_wait_and_reap` 在 `zombie_list` 中搜索调用进程的子进程，释放其内核栈，通过 `vmm_unmap` 取消映射其堆页面，释放 PCB，并返回子进程的 PID。`ppid == 0` 的孤儿僵尸进程最终会被 `zombies_free` 回收，该函数由调度器择机调用。

调度算法 `schedule` 函数提供了一个简单的轮转 (Round-robin) 风格调度器：

- 它从 `ready_queue` 中取出下一个 READY 进程。如果没有可用的进程，且当前进程仍为 RUNNING 且非 Idle 进程，则当前进程继续运行。否则，调度器选择 Idle 进程。
- 正在运行的进程（如果有）被降级为 READY 并重新入队（Idle 进程除外）。TERMINATED 进程保留在僵尸链表中。
- 通过汇编例程 `switch_context` 执行上下文切换，保存旧的 `RegState` 并恢复新的。
- 从上下文切换返回后，调度器调用 `zombies_free` 清理任何孤儿僵尸进程。

调度器由周期性的机器定时器中断驱动，中断处理程序会重新编程 CLINT 定时器并从 Trap 处理程序调用 `schedule`。

其他进程操作 进程子系统还提供：

- `proc_dump`: 打印所有进程的快照（由 `ps` 系统调用使用）。
- `proc_suspend_current`: 将当前进程移入阻塞链表并 `schedule` 另一个进程；由 Shell 的 `bg` 命令使用。
- `proc_kill`: 按 PID 杀死进程，搜索就绪、阻塞和僵尸链表，立即释放资源。
- `proc_shutdown_all`: 用于系统关机期间，释放就绪、阻塞和僵尸链表中的所有非 Idle 进程。

2.8 文件系统与块设备 (fs/, fs/blk.c)

文件系统支持构建在 VirtIO 块驱动程序和极简的基于 inode 的文件系统之上。

VirtIO 块驱动 `fs/blk.c` 中的驱动程序探测 MMIO 范围内的 VirtIO 设备，寻找 `DEVICE_ID == 2`（块设备）的设备。它通过配置适当的队列寄存器支持 VirtIO MMIO 版本 1 和版本 2。

每个 I/O 请求使用一个包含三个描述符的小型 virtqueue：

1. 读写操作和目标扇区的请求头。
2. 一个 512 字节扇区的数据缓冲区。
3. 由设备写入的状态字节。

通过更新可用环并通过 MMIO `QUEUE_NOTIFY` 寄存器通知设备来提交请求。完成检测通过轮询已用环索引实现；设备中断由 `blk_intr` 通过 Trap 和 PLIC 代码处理。

公共接口导出 `blk_read_sector` 和 `blk_write_sector`，专供文件系统层使用。

磁盘布局与 Inode `fs/fs.c` 和 `fs/fs.h` 中的文件系统在 64 KB 磁盘镜像上使用非常小的固定大小布局：

- 块 0: 超级块 (`struct superblock`)。
- 块 1–4: Inode 表 (`struct dinode`)。
- 块 5: 空闲块位图。
- 块 6–127: 数据块。

每个 Inode (`struct dinode`) 包含类型（空闲、普通文件、目录）、链接计数、文件大小、一个包含 10 个直接块地址的数组和一个间接块指针。间接块保存一个数据块地址数组，因此每个文件的最大块数为 MAXFILE = NDIRECT + NINDIRECT。

块与 Inode 管理 内部辅助函数包括：

- `b_read` / `b_write`: 针对 512 字节扇区的 VirtIO 驱动包装器。
- `balloc` / `b_free`: 通过操作位图块中的位图来分配和释放数据块。
- `read_dinode` / `write_dinode`: 根据 Inode 编号加载和更新单个 Inode。
- `bmap`: 将文件块索引映射到物理块号，按需分配直接或间接块。

高层操作 `inode_read` 和 `inode_write` 实现了跨越可能不连续的磁盘块的稀疏块读写。

目录与命名空间 当前文件系统实现了一个单一的扁平根目录，由 `sb.root_inum` 标识。目录项 (`struct dirent`) 将文件名映射到 Inode 编号。辅助函数包括：

- `dir_lookup`: 在根目录中按名称查找 Inode 编号。
- `dir_add`: 追加新的目录项。
- `dir_remove_inum`: 按 Inode 编号移除目录项。

`fs_format` 初始化一个新的文件系统：它清除 Inode 和位图块，写入新的超级块，创建根目录 Inode，并可选择在根目录中创建一个包含编译进内核的顶级项目 README 内容的 `README.md` 文件。

文件操作与 FD 表 在运行时，文件系统层维护一个内存表，最多包含 `FS_MAX_FILES` 个打开的文件 (`FSFileDesc`)。返回给用户空间的文件描述符从 `FS_FD_BASE` 开始，以便描述符 0, 1, 2 可以保留用于标准输入/输出/错误。

公共 API 包括：

- `fs_init`: 初始化超级块和 FD 表，如果魔数无效则格式化新文件系统。
- `fs_create`: 在根目录中创建一个新的普通文件并打开它。
- `fs_open`: 按名称打开现有文件。
- `fs_read` / `fs_write`: 对打开的文件执行基于偏移量的读写，更新文件偏移量。
- `fs_close`: 释放文件描述符槽位。
- `fs_unlink`: 按名称移除文件，释放其数据和间接块，并清除其 Inode 和目录项。
- `fs_trunc`: 将文件大小设置为零而不释放其块（简单的截断语义）。
- `fs_list_root`: 将根目录项枚举到调用者提供的缓冲区中；用于实现 `ls`。

2.9 Trap 与中断处理 (trap/)

Trap 子系统由汇编向量入口和 C 语言处理程序组成。

Trap 入口 `trap/trapentry.S` 中的入口点 `trap_vector_entry` 由 `trap_init` 安装到 `mtvec` 中。它：

1. 在栈上分配 128 字节作为 Trap 帧。
2. 将部分通用寄存器 (`ra`, `t0--t2`, `a0--a7`) 保存到该帧中。

3. 将 Trap 帧指针作为 `a0` 传递给 C 处理程序 `trap_handler_c`。

4. 恢复保存的寄存器并执行 `mret` 返回到 `mepc` 中存储的地址。

C 级 Trap 处理程序 核心逻辑在 `trap/trap.c` 中。它读取 `mcause`, `mepc`, `mtval`, 和 `mstatus`，解码事件是异常还是中断，并进行相应分发。

对于 **环境调用 (Environment call)** 异常（来自用户或 Machine 模式的 ecall），处理程序：

- 从保存的 `a7` (Trap 帧中的 `tf[11]`) 提取系统调用号，从 `a0--a5` 提取最多六个参数。
- 从 Trap 帧更新当前进程的 `RegState` 快照，以便随后的 `fork` 能看到最新的用户上下文。
- 对于 `SYS_EXEC`，执行特殊处理：它查找程序名称以获取入口点，更新 `mepc` 到新入口，并为 `argc` 和 `argv` 配置 `a0` 和 `a1`。此路径绕过通用分发器。
- 对于所有其他系统调用，调用 `syscall_dispatch`，将返回值写回 Trap 帧中的 `a0`，并将 `mepc` 增加 4 以跳过 `ecall` 指令。

对于 **ecall 以外的异常**，处理程序记录诊断信息（启用 Trap 调试时），并将该错误视为当前进程的致命错误，调用 `proc_exit`。

对于 **中断**，主要情况有：

- 机器定时器中断：重新编程 CLINT 定时器以在固定间隔后再次触发，并调用 `schedule` 执行上下文切换。
- 机器外部中断：通过 `plic_claim` 从 PLIC 获取中断号，将 VirtIO 相关中断分发给 `blk_intr`，然后通过 `plic_complete` 完成中断。

任何意外的 Trap 都会落入无限 `wfi` 循环，以便在调试时安全停机。

PLIC 设置 `trap/plic.c` 模块通过以下方式配置 QEMU 的平台级中断控制器：

- 为 IRQ 1–8 (VirtIO 设备) 设置非零优先级。
- 为 hart 0 启用这些 IRQ。
- 将中断优先级阈值设置为零。

辅助函数 `plic_claim` 和 `plic_complete` 与 PLIC claim/complete 寄存器交互。

2.10 系统调用实现 (syscall/)

系统调用在 `kernel/syscall/syscall.h` 中定义 ID，并在 `kernel/syscall/syscall.c` 中实现。分发器 `syscall_dispatch` 使用 switch 语句调用正确的内核函数。

2.10.1 系统调用表

表 3 总结了当前实现的系统调用。

表 3: Lrix 系统调用接口。

ID	名称	描述
1	SYS_EXIT	终止当前进程并将其转为僵尸状态；如果父进程正在等待，则唤醒它。
2	SYS_GETPID	返回调用进程的 PID。
3	SYS_FORK	通过复制调用者的 PCB、内核栈和用户堆来创建子进程。
4	SYS_WAIT	阻塞直到一个子进程终止；回收其资源并返回其 PID。
5	SYS_SBRK	使用 VMM 按请求的增量增长调用进程的堆。
6	SYS_SLEEP	基于 CLINT `mtime` 滴答的忙等待睡眠，使用 `wfi`。
7	SYS_KILL	按 PID 杀死进程；如果找到，立即释放其资源。
8	SYS_UPTIME	返回当前 CLINT `mtime` 值作为单调递增的滴答计数器。
9	SYS_WRITE	通过 `printk` 向 stdout/stderr (fd 1/2) 写字节，或向文件系统支持的描述符写字节。
10	SYS_OPEN	按名称打开文件，可选创建标志；返回文件系统文件描述符。
11	SYS_READ	从文件系统支持的描述符读取字节到用户缓冲区。
12	SYS_CLOSE	关闭文件系统支持的描述符。
13	SYS_LS	将根目录条目列出到用户提供的 `dirent` 数组中。
14	SYS_GETC	从 UART 控制台阻塞读取单个字符。
15	SYS_UNLINK	移除根目录中的文件，释放其块并清除其 Inode。
16	SYS_EXEC	用命名的用户程序替换当前进程镜像；在 Trap 处理程序中特殊处理。
17	SYS_TRUNC	按名称截断文件，将其大小设为零但保留其块分配。
18	SYS_PS	通过 `proc_dump` 转储内核进程列表。
19	SYS_SHUTDOWN	关闭系统：禁用中断，释放所有进程，并通过 `wfi` 停机。
20	SYS_SUSPEND	将当前进程移至阻塞链表并调度另一个进程；由后台 worker 使用。

2.10.2 Exec 支持

Exec 风格的程序替换以简单的方式实现。一个小型的静态 `exec_table` 将程序名称映射到链接进内核镜像的函数入口点，例如：

- `"sh"` → `user_shell`.

辅助函数 `sys_exec_lookup` 在处理 `SYS_EXEC` 时由 Trap 处理程序调用。成功时，处理程序更新 `mepc` 到新的用户入口点并重新初始化 `a0` 和 `a1` (`argc` 和 `argv`)。

3 用户空间与 Shell (usr/)

用户空间代码位于 `usr/` 下。所有程序使用公共头文件 `user.h`，它导入内核可见的定义（文件系统类型、字符串辅助函数、系统调用号）并声明建立在 `sys_call.c` 实现的通用 `sys_call3` 函数之上的瘦系统调用封装。

3.1 系统调用封装

每个用户可见的函数如 `sys_write`, `sys_open`, `sys_fork`, 或 `sys_shutdown` 都是一个小型的封装，它在寄存器 `a0--a2` 中设置参数，在 `a7` 中设置系统调用号，并执行 `ecall`。内核的 Trap 处理程序随后分发到相应的系统调用实现。

3.2 Shell 概览

主要的用户程序是在 `usr/shell.c` 中实现的交互式 Shell。它作为一个常规进程运行，并仅通过系统调用与内核交互。

Shell 的架构包括：

- 一个简单的行编辑器，使用 `sys_getc` 读取字符，本地处理退格键，并通过 `sys_write` 回显输入。
- 一个标记化例程，将命令行分割成最多八个以空格分隔的参数 (`argv`)。
- 一组直接用 C 实现的内置命令，无需衍生外部程序。
- 通过临时文件名 (`__pipe.tmp`) 对 “管道” 的极简支持，用于在简单场景下将一个命令的输出传递给另一个命令。

3.3 支持的 Shell 命令

Shell 目前支持以下内置命令：

- **`ls`**: 通过调用 `sys_ls` 并打印每个返回的 `dirent` 名称来列出根目录中的文件。
- **`cat FILE`**: 使用 `sys_open` 打开文件，使用 `sys_read` 读取内容，并写入标准输出。当在管道模式下不带参数调用时，它从临时管道文件读取。
- **`read FILE`**: `cat FILE` 的别名。
- **`echo ARGS...`**: 打印以空格分隔并以换行符结尾的参数。
- **`touch F`**: 如果文件不存在则创建空文件，尝试 `sys_open(F, 0)`，失败则 `sys_open(F, 1)`。
- **`rm F`**: 使用 `sys_unlink` 移除文件；失败时打印错误消息。
- **`mv A B`**: 移动（重命名）文件，通过将数据从 A 复制到 B 然后 unlink A。如果 B 存在，则先 unlink 并重新创建。
- **`pwd`**: 打印当前目录，由于文件系统目前是扁平的，始终为 `/`。
- **`mkdir D`**, **`rmdir D`**: 打印信息性消息，表明不支持分层目录。
- **`write F TEXT...`**: (重) 写文件。该命令将文件名后的所有参数连接成一个空格分隔的字符串并写入文件，先通过 `sys_trunc` 截断文件。当与管道一起使用时，也可以将临时管道文件的内容写入目标文件。
- **`fork`**: `fork` 系统调用的演示。父进程打印子进程的 PID 然后调用 `sys_wait` 进行同步。子进程打印消息并退出。
- **`bg`**: 创建一个简单的后台 worker 进程，通过 fork 并在子进程中调用 `sys_suspend`，将其移入阻塞链表而不消耗 CPU。
- **`kill PID`**: 通过 `sys_kill` 按 PID 杀死进程。
- **`ps`**: 通过调用 `SYS_PS` 系统调用列出进程，该调用在内核中调用 `proc_dump`。
- **`help`**: 打印所有内置命令的摘要及每个命令的简短描述。
- **`exit`**: 通过调用 `sys_shutdown` 关闭系统。Shell 打印有关退出 QEMU 模拟器的提示。
- **`halt`**: `exit` 的同义词，直接调用 `sys_shutdown` 而不显示额外消息。

通过这些命令，Shell 演练了大部分内核子系统：文件系统操作 (`ls`, `cat`, `write`, `rm`, `mv`, `touch`)、进程控制 (`fork`, `bg`, `kill`, `ps`) 以及关机路径 (`exit`, `halt`)。

4 总结

Lrix 展示了从裸机 RISC-V 启动到交互式用户 Shell 的紧凑而完整的路径。尽管体积小巧，它涵盖了许多经典的操作系统概念：基于页的物理分配器、极简的虚拟内存抽象、带有阻塞和僵尸状态的轮转进程调度、基于 VirtIO 块存储的简单 inode 文件系统、绑定到 RISC-V CSR 和 PLIC 的 Trap 与中断子系统，以及供用户空间程序使用的系统调用接口。

由于设计特意保持直观且代码库完全自包含，Lrix 可以作为实验和扩展的良好基础。但是我还是得强调 Lrix 只运行在 Machine Mode 下。

References

[1] Russ Cox, Frans Kaashoek, and Robert Morris. *xv6: a simple, Unix-like teaching operating system for RISC-V*. https://github.com/mit-pdos/xv6-riscv. Accessed: 2025-12-29. 2022.

[2] OASIS Open. *Virtual I/O Device (VIRTIO) Version 1.0*. http://docs.oasis-open.org/virtio/virtio/v1.0/virtio-v1.0.html. Mar. 2016.

[3] OASIS Open. *Virtual I/O Device (VIRTIO) Version 1.1*. https://docs.oasis-open.org/virtio/virtio/v1.1/virtio-v1.1.html. Apr. 2019.

[4] David Patterson and Andrew Waterman. *The RISC-V Reader: an open architecture Atlas*. Strawberry Canyon, 2017.

[5] Andrew Waterman and Krste Asanovic, eds. *The RISC-V instruction set manual Volume I: unprivileged specification ISA*. https://drive.google.com/file/d/17GeetSnT5wW3xNuAHI95-SI1gPGd5sJ_/view?usp=drive_link. 2024.

[6] Andrew Waterman, Krste Asanovic, and John Hauser, eds. *The RISC-V instruction set manual Volume II: privileged specification*. https://drive.google.com/file/d/1uviu1nH-tScFfgrovvFCrj7Omv8tFtkp/view?usp=drive_link. 2024.