Lrix: A scratch RISC-V Operating System

Project: Lrix
GitHub: https://github.com/lrisguan/Lrix
Author: Lris Guan

January 1, 2026

Contents

Abstract

Lrix is a scratch educational operating system targeting the RISC-V architecture and QEMU's `virt` machine running under **Machine Mode**. That's why I call it scratch(In the future maybe I will fix this issue). This report gives a concise technical overview of the kernel and user space design, focusing on memory management, process management, filesystem, trap and interrupt handling, system call interface, and the built-in shell.

1 Introduction

Lrix is a small teaching-oriented OS implemented in C and RISC-V assembly. It boots on the QEMU `riscv64 virt` platform, uses a simple UART driver for console I/O, and provides a minimal UNIX-like process and file abstraction. The top-level `Makefile` builds a kernel image and user programs, and runs them inside QEMU using a VirtIO block device as backing storage. Note: When developing Lrix, I referred to [4], [6], [5][1], [1], [2], [3]

The kernel entry point `kmain` performs the following high-level steps:

1. Initialize UART for serial output.
2. Initialize the trap/interrupt subsystem and the PLIC.
3. Initialize the physical page allocator and virtual memory manager.
4. Initialize the scheduler and create an idle process.
5. Initialize the VirtIO block driver and the simple inode-based filesystem.
6. Create the initial user process running the shell, then enable interrupts and idle with `wfi` while timer interrupts drive scheduling.

2 Kernel Subsystems

This section summarizes each major kernel module under `kernel/`, with emphasis on memory management, process management, filesystem, and trap handling.

2.1 Boot and Entry (boot/)

The bootstrap code in `boot/start.S` is responsible for low-level CPU initialization:

- Set the initial stack pointer to the linker-provided `_stack_top`.
- Zero the BSS segment by iterating from `_bss_start` to `_bss_end`.
- Call the C entry point `kmain`, and if it ever returns, loop forever.

This code runs in machine mode and prepares a clean C environment for the kernel.

2.2 UART and Logging (uart/, include/log.h)

The UART driver in `uart/uart.c` implements interrupt I/O for the NS16550-compatible UART mapped at `0x10000000` on QEMU's `virt` machine. (For the interrupt I/O, you can enable compile flag **TRAP_DEBUG** to see). It provides:

- `uart_init`: configure the UART in 8N1 mode.
- `uart_putc`, `uart_getc`, and `uart_getc_blocking` for basic character I/O.
- `uart_getline` and a small scanf-like `scank` to parse user input.

- `printk`: a minimal formatted printing routine supporting standard integer, string, and pointer formats.

The header `include/log.h` then builds on top of `printk` and ANSI escape codes to offer colored logging macros such as `INFO`, `WARNING`, and `ERROR` used throughout the kernel.

2.3 String Library (string/)

The `string/string.c` file implements a small subset of libc string and memory functions used in the kernel and user space, including `memset`, `memcpy`, `memmove`, `memcmp`, `strlen`, `strcmp`, and `strncmp`. This avoids depending on an external C library.

2.4 Low-Level RISC-V Helpers (include/riscv.h, include/types.h)

The header `include/riscv.h` contains inline assembly helpers for common RISC-V CSRs, in particular `intr_on` and `intr_off`, which set or clear the machine interrupt enable (MIE) bit in `mstatus`. This is used pervasively in the scheduler and trap code to delimit critical sections.

The header `include/types.h` defines basic types and the `RegState` structure, which holds the RISC-V register context (general-purpose registers, `sepc`, `sp`, and `mstatus`) used by the scheduler and context switch mechanism.

Table 1: Description of RegisterState Structure Fields

Field Name	Register Alias	Field Type	Description
x1	ra	uint64_t	Return Address
x5	t0	uint64_t	Temporary Register 0
x6	t1	uint64_t	Temporary Register 1
x7	t2	uint64_t	Temporary Register 2
x8	s0/fp	uint64_t	Saved Register 0 / Frame Pointer
x9	s1	uint64_t	Saved Register 1
x10	a0	uint64_t	Argument/Return Value Register 0
x11	a1	uint64_t	Argument/Return Value Register 1
x12	a2	uint64_t	Argument Register 2
x13	a3	uint64_t	Argument Register 3
x14	a4	uint64_t	Argument Register 4
x15	a5	uint64_t	Argument Register 5
x16	a6	uint64_t	Argument Register 6
x17	a7	uint64_t	Argument Register 7
x18	s2	uint64_t	Saved Register 2
x19	s3	uint64_t	Saved Register 3
x20	s4	uint64_t	Saved Register 4
x21	s5	uint64_t	Saved Register 5
x22	s6	uint64_t	Saved Register 6
x23	s7	uint64_t	Saved Register 7
x24	s8	uint64_t	Saved Register 8
x25	s9	uint64_t	Saved Register 9
x26	s10	uint64_t	Saved Register 10
x27	s11	uint64_t	Saved Register 11
x28	t3	uint64_t	Temporary Register 3
x29	t4	uint64_t	Temporary Register 4
x30	t5	uint64_t	Temporary Register 5
x31	t6	uint64_t	Temporary Register 6
sepc	-	uint64_t	Exception Return Address
sp	-	uint64_t	Stack Pointer
mstatus	-	uint64_t	Machine Status Register

2.5 Physical Memory Management (mem/kmem.c)

Physical memory allocation is performed by a simple page-based allocator in `mem/kmem.c`. The allocator is initialized by `kinit`, which is passed a heap range defined in the linker script.

The implementation maintains:

- A `Page` descriptor array, placed at the beginning of the heap.
- A singly-linked free list of `Page` objects.
- Counters for total, free, and used pages.

Key operations are:

- **Initialization**: the page descriptor array occupies an integral number of pages at the start of the heap; those pages are marked used, and the remaining pages are inserted into the free list.
- **Allocation (kalloc)**: pop a `Page` from the free list, mark it used, compute its physical address from its index, and zero the page before returning it to callers.
- **Free (kfree)**: validate the address and alignment, map it back to a `Page` descriptor, guard against double free, and push it back onto the free list.

Diagnostic helpers expose statistics (total/free/used pages) and can print a summary of the allocator state.

2.6 Virtual Memory Management (mem/vmm.c)

The virtual memory manager implements a real RISC-V Sv39 page-table hierarchy and is responsible both for basic kernel mappings (RAM and MMIO) and for on-demand allocation of user heap pages.

Sv39 Page-Table Layout Lrix uses the standard three-level Sv39 scheme:

- Virtual addresses are split into VPN2 (bits 38–30), VPN1 (29–21), VPN0 (20–12), and a 4 KB page offset (11–0).
- Each page table page is 4 KB and holds 512 64-bit entries.
- Leaf PTEs encode a physical page number plus Sv39 flags (V/R/W/X/U/A/D).
- Non-leaf PTEs (for intermediate levels) are marked valid but have all R/W/X/A/D bits cleared, as required by the Sv39 specification.

The kernel maintains a single root page table (level-2) called `kernel_pd`, along with its physical address `kernel_pd_phys`. All page-table memory is allocated from the physical allocator (`kalloc`) and explicitly zeroed before use.

Initialization and Identity Mappings The entry point `vmm_init` performs the following steps:

- Allocate and zero the root level-2 page table, store its virtual and physical addresses in `kernel_pd` and `kernel_pd_phys`.
- Run a minimal self-test (`vmm_self_test`) that maps a test virtual address to a freshly allocated page, verifies translation via `vmm_translate`, then unmaps it and confirms the translation disappears.
- Build identity mappings for the key regions of the `virt` machine:
  - Main RAM: `0x80000000`–`0x80000000 + 128MB`, mapped as readable/writable and user-accessible (`VMM_P_RW | VMM_P_USER`) so that both kernel and user code/data/stacks continue to work when paging is enabled.
  - UART MMIO at `0x10000000` and the VirtIO MMIO window `[VIRTIO_MMIO_START, VIRTIO_MMIO_END)` mapped as kernel-only read/write.
  - CLINT timer at `0x02000000`–`0x02010000`, mapped for kernel read/write.

  - PLIC MMIO region starting at `PLIC_BASE`, with a small identity-mapped window for interrupt controller registers.

These identity mappings ensure that when Sv39 translation is turned on, the kernel can continue to use the same virtual addresses for code, data, heap, and device registers.

Activating Sv39 Paging After initialization, `kmain` calls `vmm_activate` to enable hardware paging:

- Compute the page-table physical page number (PPN) as `kernel_pd_phys >> 12`.
- Construct an Sv39 `satp` value with MODE=8 (Sv39), ASID=0, and the PPN in the low bits.
- Write this value into the `satp` CSR and execute `sfence.vma` to flush stale TLB entries.

For portability, two stub hooks `arch_set_cr3` and `arch_enable_paging` are still present but do nothing on RISC-V; all real activation is performed via `satp`.

Mapping, Unmapping, and Translation Core operations are implemented on top of a helper that walks down the three-level page tables:

- **`get_next_level`** takes a page-table pointer, an index, and an `alloc` flag. If the entry is invalid and allocation is enabled, it allocates a new page-table page, installs a non-leaf PTE (valid, no R/W/X/A/D), and returns a pointer to the next-level table. For an already valid entry, it interprets the PTE as a page-table pointer by converting the PPN back to a (identity-mapped) virtual address.
- **`vmm_map`**:
  - Requires page-aligned virtual and physical addresses.
  - Walks L2→L1→L0, allocating intermediate tables as needed.
  - Translates abstract VMM flags (`VMM_P_*`) into Sv39 PTE bits (V/R/W/X/U/A/D). For now, `VMM_P_RW` implies R/W/X so that both code and data pages work without a separate EXEC flag, and A/D are set eagerly.
  - Installs a leaf PTE at L0 with the chosen physical address and flags.
- **`vmm_map_page`**:
  - Allocates a fresh physical page via `kalloc`, zeroes it, and then calls `vmm_map` to map it at the requested virtual address.
- **`vmm_unmap`**:
  - Walks the page tables without allocating intermediate levels.
  - If a valid leaf PTE is found, clears it and, optionally, frees the underlying physical page back to the allocator when `free_phys` is non-zero.
  - Does not currently reclaim now-empty intermediate page tables; this keeps the implementation simple at the cost of a small memory overhead.
- **`vmm_translate`**:

- Performs a software page-table walk for a given virtual address and, if a valid PTE is found, returns a pointer to the corresponding physical address (PPN combined with the original page offset).
- Used primarily for debugging and the built-in self-test.

Additional helpers `vmm_get_page_directory`, `vmm_set_page_directory`, and `vmm_get_pd_phys` expose the current kernel root page table, laying the groundwork for future per-process address spaces even though the current system still uses a single global Sv39 page table.

User Heap Allocation on Sv39 User heaps are implemented on top of the Sv39 mapper but keep a very simple layout:

- Each process is assigned a per-PID heap region starting at HEAP_USER_BASE+pid×PER_PROC_HEAP, where `HEAP_USER_BASE` is `0x80400000` and `PER_PROC_HEAP` is 8 KB.
- The `SYS_SBRK` implementation (`sys_sbrk`) in `syscall.c` grows this region by calling `vmm_map_page` for each additional 4 KB page, always with `VMM_P_RW | VMM_P_USER` flags so that user code can read/write the newly mapped memory.
- The combination of a global Sv39 page table and per-PID heap windows provides contiguous virtual heaps per process while still keeping the overall design simple (no process-specific `satp` switching yet).

2.7 Process Management and Scheduling (proc/)

Process management is implemented in `proc/proc.c`. The core data structure is the process control block (PCB), which contains:

- Process identifier (PID) and parent PID (PPID).
- Scheduling state (READY, RUNNING, BLOCKED, TERMINATED) and priority.
- Kernel stack top address and saved register context (`RegState`).
- Per-process heap base `brk_base` and size `brk_size`.
- A process name for debugging and shell output.

Table 2: Definition of ProcessControlBlock (PCB) Structure

Member Name	Data Type	Description
pid	int	Process unique identifier
pstat	ProcState	Process running state (e.g., ready, running, blocked)
name[20]	char	Process name (maximum 20 characters)
prior	int	Process priority (lower value indicates higher priority)
entrypoint	uint64_t	Process entry point (starting instruction address)
stacktop	uint64_t	Virtual address of the process stack top
ppid	int	Parent process ID (0 for kernel or init process)

Continued on next page

Table 2: Definition of ProcessControlBlock (PCB) Structure (Continued)

Member Name	Data Type	Description
brk_base	void *	Base address of the program break (heap starting address)
brk_size	uint64_t	Allocated heap size in bytes
cpu_time	uint64_t	Total CPU time consumed by the process
remain_time	uint64_t	Remaining time slice for the process scheduling
arriv_time	uint64_t	Process arrival time in the system
regstat	RegState	Saved register state for context switching
next	PCB *	Linked list pointer for process queue management

Global variables track the current process, an idle process, a ready queue, and linked lists for blocked and zombie processes.

Idle Process and Scheduler Initialization `scheduler_init` initializes the ready queue and creates an *idle* process with PID 0, its own kernel stack, and entry function `idle_entry`. The idle process simply enables interrupts and executes `wfi` in a loop, relinquishing CPU until a timer interrupt arrives.

Process Creation Two creation paths exist:

- **`proc_create`** is used by the kernel to create the initial shell process. It allocates a PCB and a kernel stack, initializes the register state so that returning from the first context switch jumps into the given entry point, and enqueues the process onto the ready queue.
- **`proc_fork`** implements the `fork` system call. It clones the parent's PCB and kernel stack, adjusts the child's stack pointer to mirror the parent's frame layout, sets the child's `a0` register to 0 (so the child sees `fork()` return 0), and deep-copies the user heap using `vmm_map_page` and `memcpy`. The child is then enqueued as READY.

Exit, Wait, and Zombies When a process calls `exit`, `proc_exit` marks it as TERMINATED, links it into `zombie_list`, and wakes its parent if the parent is blocked in `wait`. `proc_wait_and_reap` searches `zombie_list` for a child of the calling process, frees its kernel stack, unmaps its heap pages via `vmm_unmap`, frees the PCB, and returns the child's PID. Orphaned zombies with `ppid == 0` are eventually reclaimed by `zombies_free`, which is called opportunistically by the scheduler.

Scheduling Algorithm The `schedule` function provides a simple round-robin style scheduler:

- It dequeues the next READY process from `ready_queue`. If none is available and the current process is still RUNNING and not the idle process, the current process continues to run. Otherwise, the scheduler selects the idle process.
- The outgoing RUNNING process (if any) is downgraded to READY and re-enqueued (except for the idle process). TERMINATED processes remain on the zombie list.
- A context switch is performed via the assembly routine `switch_context`, saving the old `RegState` and restoring the new one.

- After returning from a context switch, the scheduler invokes `zombies_free` to clean up any orphaned zombies.

The scheduler is driven by periodic machine timer interrupts, which reprogram the CLINT timer and call `schedule` from the trap handler.

Additional Process Operations The process subsystem also provides:

- `proc_dump`: print a snapshot of all processes (used by the `ps` system call).
- `proc_suspend_current`: move the current process into the blocked list and `schedule` another process; used by the shell's `bg` command.
- `proc_kill`: kill a process by PID, searching ready, blocked, and zombie lists, freeing resources immediately.
- `proc_shutdown_all`: used during system shutdown to free all non-idle processes from ready, blocked, and zombie lists.

2.8 Filesystem and Block Device (fs/, fs/blk.c)

Filesystem support is built on top of a VirtIO block driver and a minimal inode-based filesystem.

VirtIO Block Driver The driver in `fs/blk.c` probes the MMIO range for VirtIO devices, looking for a device with `DEVICE_ID == 2` (block). It supports both VirtIO MMIO version 1 and version 2 by configuring the appropriate queue registers.

Each I/O request uses a small virtqueue with three descriptors:

1. A request header describing the operation (read/write) and target sector.
2. A data buffer for one 512-byte sector.
3. A status byte written by the device.

Requests are submitted by updating the available ring and notifying the device through the MMIO `QUEUE_NOTIFY` register. Completion is detected by polling the used ring index; interrupts from the device are handled by `blk_intr` via the trap and PLIC code.

The public interface exports `blk_read_sector` and `blk_write_sector`, which are used exclusively by the filesystem layer.

On-Disk Layout and Inodes The filesystem in `fs/fs.c` and `fs/fs.h` uses a very small fixed-size layout over a 64 KB disk image:

- Block 0: superblock (`struct superblock`).
- Blocks 1–4: inode table (`struct dinode`).
- Block 5: free block bitmap.
- Blocks 6–127: data blocks.

Each inode (`struct dinode`) contains a type (free, regular file, directory), link count, file size, an array of 10 direct block addresses, and a single indirect block pointer. The indirect block holds an array of data block addresses, so the maximum number of blocks per file is MAXFILE = NDIRECT + NINDIRECT.

Block and Inode Management Internal helpers include:

- `b_read` / `b_write`: wrappers around the VirtIO driver for 512-byte sectors.
- `balloc` / `b_free`: allocate and free data blocks by manipulating the bitmap in the bitmap block.
- `read_dinode` / `write_dinode`: load and update individual inodes given their inode number.
- `bmap`: map a file block index to a physical block number, allocating direct or indirect blocks as needed.

Higher-level operations `inode_read` and `inode_write` implement sparse block-wise reads and writes across possibly non-contiguous disk blocks.

Directory and Namespace The current filesystem implements a single flat root directory, identified by `sb.root_inum`. Directory entries (`struct dirent`) map file names to inode numbers. Helper functions include:

- `dir_lookup`: find an inode number by name in the root directory.
- `dir_add`: append a new directory entry.
- `dir_remove_inum`: remove a directory entry by inode number.

`fs_format` initializes a fresh filesystem: it clears inode and bitmap blocks, writes a new superblock, creates the root directory inode, and optionally creates a `README.md` file in the root directory containing the top-level project README, which is compiled into the kernel as static data.

File Operations and FD Table At runtime, the filesystem layer maintains an in-memory table of at most `FS_MAX_FILES` open files (`FSFileDesc`). File descriptors returned to user space start at `FS_FD_BASE` so that descriptors 0, 1, 2 can be reserved for standard input/output/error.

The public API includes:

- `fs_init`: initialize the superblock and FD table, formatting a new filesystem if the magic number is invalid.
- `fs_create`: create a new regular file in the root directory and open it.
- `fs_open`: open an existing file by name.
- `fs_read` / `fs_write`: perform offset-based reads and writes on an open file, updating the file offset.
- `fs_close`: release a file descriptor slot.
- `fs_unlink`: remove a file by name, freeing its data and indirect blocks and clearing its inode and directory entry.
- `fs_trunc`: set a file's size to zero without freeing its blocks (simple truncate semantics).
- `fs_list_root`: enumerate root directory entries into a caller-provided buffer; this is used to implement `ls`.

2.9 Trap and Interrupt Handling (trap/)

The trap subsystem consists of an assembly vector entry and a C-level handler.

Trap Entry The entry point `trap_vector_entry` in `trap/trapentry.S` is installed into `mtvec` by `trap_init`. It:

1. Allocates 128 bytes on the stack as a trap frame.
2. Saves a subset of general-purpose registers (`ra`, `t0--t2`, `a0--a7`) into this frame.
3. Passes the trap frame pointer in `a0` to the C handler `trap_handler_c`.
4. Restores the saved registers and executes `mret` to return to the address stored in `mepc`.

C-Level Trap Handler The core logic is in `trap/trap.c`. It reads `mcause`, `mepc`, `mtval`, and `mstatus`, decodes whether the event is an exception or an interrupt, and dispatches accordingly.

For **environment call** exceptions (ecall from user or machine mode), the handler:

- Extracts the system call number from the saved `a7` (`tf[11]` in the trap frame) and up to six arguments from `a0--a5`.
- Updates the current process's `RegState` snapshot from the trap frame so that a subsequent `fork` sees the latest user context.
- For `SYS_EXEC`, performs special handling: it looks up the program name to get an entry point, updates `mepc` to the new entry, and configures `a0` and `a1` for `argc` and `argv`. This path bypasses the generic dispatcher.
- For all other syscalls, calls `syscall_dispatch`, writes the return value back into `a0` in the trap frame, and advances `mepc` by 4 to skip the `ecall` instruction.

For **exceptions other than ecall**, the handler logs diagnostic information (when trap debugging is enabled) and treats the fault as fatal to the current process by calling `proc_exit`.

For **interrupts**, the main cases are:

- Machine timer interrupt: reprogram the CLINT timer to fire again after a fixed interval and invoke `schedule` to perform a context switch.
- Machine external interrupt: claim the interrupt number from the PLIC via `plic_claim`, dispatch VirtIO-related interrupts to `blk_intr`, and then complete the interrupt via `plic_complete`.

Any unexpected traps fall into an infinite `wfi` loop to halt the CPU for safety when debugging.

PLIC Setup The `trap/plic.c` module configures QEMU's platform-level interrupt controller by:

- Setting non-zero priority for IRQ 1–8 (VirtIO devices).
- Enabling these IRQs for hart 0.
- Setting the interrupt priority threshold to zero.

Helper functions `plic_claim` and `plic_complete` interact with the PLIC claim/complete registers.

2.10 System Call Implementation (syscall/)

System calls are defined by numeric IDs in `kernel/syscall/syscall.h` and implemented in `kernel/sysca`
The dispatcher `syscall_dispatch` uses a switch statement to invoke the correct kernel function.

2.10.1 System Call Table

Table 3 summarizes the currently implemented system calls.

Table 3: Lrix system call interface.

ID	Name	Description
1	SYS_EXIT	Terminate the current process and turn it into a zombie; parent is woken if waiting.
2	SYS_GETPID	Return the PID of the calling process.
3	SYS_FORK	Create a child process by duplicating the caller's PCB, kernel stack, and user heap.
4	SYS_WAIT	Block until a child terminates; reap its resources and return its PID.
5	SYS_SBRK	Grow the calling process's heap by a requested increment using the VMM.
6	SYS_SLEEP	Busy-wait sleep based on CLINT `mtime` ticks, using `wfi`.
7	SYS_KILL	Kill a process by PID; immediately free its resources if found.
8	SYS_UPTIME	Return the current CLINT `mtime` value as a monotonically increasing tick counter.
9	SYS_WRITE	Write bytes either to stdout/stderr (fd 1/2) via `printk`, or to a filesystem-backed descriptor.
10	SYS_OPEN	Open a file by name with optional create flag; returns a filesystem file descriptor.
11	SYS_READ	Read bytes from a filesystem-backed descriptor into a user buffer.
12	SYS_CLOSE	Close a filesystem-backed descriptor.
13	SYS_LS	List entries in the root directory into a user-provided array of `dirent`.
14	SYS_GETC	Blocking read of a single character from the UART console.
15	SYS_UNLINK	Remove a file in the root directory, freeing its blocks and clearing its inode.
16	SYS_EXEC	Replace the current process image with a named user program; handled specially in the trap handler.
17	SYS_TRUNC	Truncate a file by name, setting its size to zero but keeping its block allocations.
18	SYS_PS	Dump the kernel process list via `proc_dump`.
19	SYS_SHUTDOWN	Shut down the system: disable interrupts, free all processes, and halt the CPU with `wfi`.
20	SYS_SUSPEND	Move the current process to the blocked list and schedule another process; used by background workers.

2.10.2 Exec Support

Exec-style program replacement is implemented in a simple way. A small static `exec_table` maps program names to function entry points that are linked into the kernel image, for example:

- `"sh"` → `user_shell`.

The helper `sys_exec_lookup` is called by the trap handler when processing `SYS_EXEC`. On success, the handler updates `mepc` to the new user entry point and reinitializes `a0` and `a1` (`argc` and `argv`).

3 User Space and Shell (usr/)

User-space code resides under `usr/`. All programs use a common header `user.h`, which imports kernel-visible definitions (filesystem types, string helpers, syscall numbers) and declares thin syscall wrappers built on top of the generic `sys_call3` function implemented in `sys_call.c`.

3.1 Syscall Wrappers

Each user-visible function like `sys_write`, `sys_open`, `sys_fork`, or `sys_shutdown` is a small wrapper that sets up arguments in registers `a0--a2`, the system call number in `a7`, and executes an `ecall`. The kernel's trap handler then dispatches to the appropriate system call implementation.

3.2 Shell Overview

The main user program is an interactive shell implemented in `usr/shell.c`. It runs as a regular process and interacts with the kernel exclusively via syscalls.

The shell's architecture includes:

- A simple line editor that reads characters using `sys_getc`, handles backspace locally, and echoes input via `sys_write`.
- A tokenization routine that splits a command line into at most eight whitespace-separated arguments (`argv`).
- A set of built-in commands implemented directly in C, without spawning external programs.
- Minimal support for "pipelines" through a temporary file name (`__pipe.tmp`), which is used to pass the output of one command to another in simple scenarios.

3.3 Supported Shell Commands

The shell currently supports the following built-in commands:

- **`ls`**: list files in the root directory by calling `sys_ls` and printing each returned `dirent` name.
- **`cat FILE`**: open a file with `sys_open`, read its contents with `sys_read`, and write them to standard output. When invoked without an argument in a pipeline mode, it reads from the temporary pipe file.
- **`read FILE`**: an alias of `cat FILE`.

- **`echo ARGS...`**: print its arguments separated by spaces and terminated by a newline.
- **`touch F`**: create an empty file if it does not exist, by attempting `sys_open(F, 0)` and, on failure, `sys_open(F, 1)`.
- **`rm F`**: remove a file using `sys_unlink`; prints an error message on failure.
- **`mv A B`**: move (rename) a file by copying data from A to B and then unlinking A. If B exists, it is first unlinked and recreated.
- **`pwd`**: print the current directory, which is always `/` because the filesystem is currently flat.
- **`mkdir D`**, **`rmdir D`**: print informative messages indicating that hierarchical directories are not supported.
- **`write F TEXT...`**: (re)write a file. The command concatenates all arguments after the filename into a single space-separated string and writes it into the file, truncating it first via `sys_trunc`. When used with a pipeline, it can also write the contents of the temporary pipe file into the target file.
- **`fork`**: demonstration of the `fork` system call. The parent prints the child's PID and then calls `sys_wait` to synchronize. The child prints a message and exits.
- **`bg`**: create a simple background worker process by forking and calling `sys_suspend` in the child, which moves it into the blocked list without consuming CPU.
- **`kill PID`**: kill a process by PID via `sys_kill`.
- **`ps`**: list processes by invoking the `SYS_PS` system call, which calls `proc_dump` in the kernel.
- **`help`**: print a summary of all built-in commands and a short description of each.
- **`exit`**: shutdown the system by calling `sys_shutdown`. The shell prints a hint about exiting the QEMU emulator.
- **`halt`**: synonym for `exit`, directly calling `sys_shutdown` without additional messages.

Through these commands, the shell exercises most of the kernel subsystems: filesystem operations (`ls`, `cat`, `write`, `rm`, `mv`, `touch`), process control (`fork`, `bg`, `kill`, `ps`), and shutdown paths (`exit`, `halt`).

4 Summary

Lrix demonstrates a compact yet complete path from bare-metal RISC-V boot to an interactive user shell. Despite its small size, it covers many classic OS concepts: a page-based physical allocator, a minimal virtual memory abstraction, round-robin process scheduling with blocking and zombies, a simple inode-based filesystem over VirtIO block storage, a trap and interrupt subsystem tied into RISC-V CSRs and the PLIC, and a syscall interface consumed by user-space programs.

Because the design is intentionally straightforward and the codebase is fully self-contained, Lrix serves as a good foundation for experimentation and extension, such as adding per-process address spaces, hierarchical directories, more sophisticated scheduling policies, or richer user-level utilities. Here I still want to emphasize that Lrix only runs under Machine Mode.

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.