Writing a RISC-V OS in Lateralus

Practical guide: startup, trap handling, memory, and process management

Lateralus Language

bad-antics · March 2026 · Lateralus Language Research

ABSTRACT This paper is a practical guide to writing an operating system kernel for RISC-V in Lateralus. We walk through the four core implementation tasks: the startup sequence (reset vector to first user process), trap handling (exception and interrupt dispatch), memory management (physical allocator and page tables), and process management (context switching and scheduling). Code excerpts are drawn from the FRISC-OS implementation, which serves as a reference implementation. By the end of this guide, the reader should have a working kernel that can run a single user process.

1. Prerequisites and Toolchain

To follow this guide, you need: the Lateralus compiler (v1.0+), a RISC-V cross-compilation target (riscv64-linux-gnu), QEMU for RISC-V (qemu-system-riscv64), and the riscv64-unknown-elf GNU binutils for linking.

```
# Install the RISC-V target
ltl toolchain add riscv64-linux-gnu

# Verify
ltl build --target riscv64-linux-gnu hello.lt
qemu-system-riscv64 -machine virt -nographic -bios none \
    -kernel hello.elf
# Output: Hello, RISC-V!
```

The -bios none flag skips the OpenSBI firmware and boots directly to the kernel entry point. This matches what we want for a bare-metal OS.

2. The Startup Sequence

At reset, the RISC-V core executes from address 0x80000000 (QEMU virt machine). Our startup code is a small assembly stub that sets up the stack and calls kernel_main:

```
// src/startup.s (linked first)
.section .text.start
.global _start
_start:
    la   sp, _stack_top      # set up stack pointer
    call kernel_main          # call Lateralus entry point
    j    .                    # halt if kernel_main returns
```

The linker script places the startup code at 0x80000000 and the stack at the top of the first 64 KiB of RAM:

```
// kernel.ld (linker script excerpt)
SECTIONS {
    . = 0x80000000;
    .text.start : { *(.text.start) }
    .text       : { *(.text*) }
    .data       : { *(.data*) }
    .bss        : { *(.bss*) }
    _stack_top  = 0x80010000;  /* 64 KiB for initial stack */
}
```

3. Trap Handling

Exceptions and interrupts are vectored through the machine trap handler, set up by writing the handler address to the mtvec CSR. We set it in Lateralus using inline assembly:

```
// Set the trap vector
#[inline(never)]
fn setup_trap_vector() {
    unsafe {
        asm!("la t0, trap_entry; csrw mtvec, t0")
    }
}

// Trap entry: saves registers, calls trap_dispatch
#[no_mangle]
#[naked]
fn trap_entry() {
    unsafe {
        asm!(
            // Save all registers to the current thread's trap frame
            "addi sp, sp, -256",
            "sd   ra,  0(sp)",
            // ... (all 32 registers)
            "call trap_dispatch",
            "ld   ra,  0(sp)",
            // ... restore
            "mret"
        )
    }
}
```

The #[naked] attribute tells the compiler to emit no prologue or epilogue for this function; the assembly is executed as-is.

4. Physical Memory Allocator

We implement a page-frame allocator using a free list embedded in the free pages themselves. Each free 4 KiB page stores a pointer to the next free page at its base address:

```
struct PageAllocator {
    free_list: *mut u8,  // head of free list
    total:     usize,
    used:      usize,
}

fn alloc_page(a: &mut PageAllocator) -> Option<*mut u8> {
    if a.free_list.is_null() { return None; }
    let page = a.free_list;
    a.free_list = unsafe { *(page as *const *mut u8) };
    a.used += 1;
    Some(page)
}
```

5. Virtual Memory: The Sv39 Page Table

RISC-V Sv39 uses a three-level radix page table. Each level indexes 9 bits of the 39-bit virtual address. A page table entry (PTE) is 8 bytes:

```
// 64-bit page table entry
struct Pte(u64);

impl Pte {
    fn valid(&self)    -> bool { self.0 & 1 != 0 }
    fn readable(&self) -> bool { self.0 & 2 != 0 }
    fn writable(&self) -> bool { self.0 & 4 != 0 }
    fn executable(&self) -> bool { self.0 & 8 != 0 }
    fn user(&self)     -> bool { self.0 & 16 != 0 }
    fn ppn(&self)      -> u64  { (self.0 >> 10) & 0xFFF_FFFF_FFFF }
}

fn map_page(root: &mut PageTable, va: usize, pa: usize, flags: u64) {
    let vpn = [va >> 30, (va >> 21) & 0x1FF, (va >> 12) & 0x1FF];
    // Walk three levels, allocating intermediate page tables as needed
    // ...
}
```

6. Context Switching

Context switching saves the current thread's registers to its kernel stack and restores the next thread's registers. We use the callee-saved RISC-V registers (s0-s11, ra, sp):

```
// Context switch: save current, restore next
#[naked]
fn context_switch(current: *mut Context, next: *const Context) {
    unsafe {
        asm!(
            // Save callee-saved registers to current context
            "sd ra,  0(a0)",
            "sd sp,  8(a0)",
            "sd s0, 16(a0)",
            // ... s1-s11
            // Restore next context
            "ld ra,  0(a1)",
            "ld sp,  8(a1)",
            "ld s0, 16(a1)",
            // ... s1-s11
            "ret"
        )
    }
}
```

7. A Simple Round-Robin Scheduler

For a first working kernel, a round-robin scheduler is sufficient. We maintain a run queue of ready processes and switch to the next process on each timer interrupt:

```
let mut run_queue: VecDeque<Process> = VecDeque::new();

fn schedule() {
    let current = run_queue.pop_front().unwrap();
    if current.state == ProcessState::Running {
        run_queue.push_back(current);
    }
    let next = run_queue.front_mut().unwrap();
    next.state = ProcessState::Running;
    context_switch(&mut current.context, &next.context);
```

```
}
```

8. Running a User Process

To run a user process, we load its binary from the initrd, create a page table mapping its text and stack, and set up a trap frame that returns to user mode at the entry point:

```
fn spawn_user_process(elf: &[u8]) -> Process {
    let mut page_table = PageTable::new();
    let entry = load_elf_into(&mut page_table, elf);
    let stack = alloc_page(&mut ALLOCATOR).unwrap();
    map_page(&mut page_table, USER_STACK_VA, stack as usize,
             PTE_READ | PTE_WRITE | PTE_USER);
    Process {
        page_table,
        trap_frame: TrapFrame {
            pc:   entry,
            sp:   USER_STACK_VA + PAGE_SIZE,
            mode: MachineMode::User,
            ..default()
        },
        state: ProcessState::Ready,
    }
}
```

With this in place, the kernel can run a simple user-mode program that makes system calls via ecall. The ecall trap handler reads a7 (system call number) and dispatches to the appropriate handler.