Building a Bare-Metal OS

Step-by-step construction of Lateralus OS: from linker script to process scheduler

Lateralus Language

bad-antics · April 2026 · Lateralus Language Research

ABSTRACT This paper is a construction guide for Lateralus OS: a minimal RISC-V operating system written entirely in Lateralus. It covers the full build from the linker script through to preemptive multitasking. Unlike tutorial-style introductions, this paper is structured around the decisions made during construction — why each component was designed the way it was, and what the alternatives were.

1. What We Are Building

Lateralus OS is a monolithic kernel for RISC-V (RV64GC) that provides:

- Interrupt-driven I/O on the UART, PLIC, and CLINT peripherals.
- Physical memory management via a buddy allocator.
- Virtual memory using Sv39 page tables.
- A preemptive round-robin scheduler.
- A minimal system call interface (read, write, fork, exec, exit, wait).

The kernel is approximately 8,000 lines of Lateralus. It boots on QEMU's virt machine and on the SiFive HiFive Unmatched development board.

2. The Linker Script

The linker script defines the memory layout of the kernel binary. RISC-V kernels typically load at a high physical address (0x80200000 on QEMU's virt machine after OpenSBI loads):

```
/* lateralus-os.ld */
ENTRY(_start)
SECTIONS {
    . = 0x80200000;
    .text   : { *(.text.boot) *(.text .text.*) }
    .rodata : { *(.rodata .rodata.*) }
    .data   : { *(.data .data.*) }
    .bss    : { __bss_start = .; *(.bss .bss.*); __bss_end = . }
    . = ALIGN(4096);
    __kernel_end = .;
}
```

The .text.boot section is the assembly entry point placed first; it initializes the stack and calls the Lateralus _start function.

3. Boot Sequence

The kernel boot sequence has five steps:

```
# Assembly entry (_start.S)
1. Zero BSS segment (from __bss_start to __bss_end)
2. Set stack pointer (sp = __stack_top)
3. Set up trap vector (csrw mtvec, trap_vec)
4. Call kmain (j kmain)

# Lateralus kmain
5. uart::init()          → enable UART interrupts
6. mem::phys_init()      → initialize buddy allocator
```

```
7. mem::virt_init()      → set up kernel page tables
8. plic::init()          → configure interrupt controller
9. sched::init_idle()    → create idle task
10. sched::run()         → start scheduler (never returns)
```

4. Physical Memory Management

Physical memory is managed by a buddy allocator. The buddy system splits and merges power-of-two blocks, providing O(log n) allocation and deallocation:

```
// Allocate 2^order contiguous pages
fn phys_alloc(order: u8) -> Option<PhysAddr> {
    for o in order..=MAX_ORDER {
        if let Some(block) = free_lists[o].pop() {
            split_down_to(block, o, order);
            return Some(block);
        }
    }
    None
}
```

The allocator manages memory from __kernel_end to the top of physical RAM (read from the RISC-V Device Tree). Kernel data structures are allocated from the first 16 MB; everything above is available for user processes.

5. Virtual Memory: Sv39 Page Tables

Lateralus OS uses RISC-V Sv39: 39-bit virtual addresses mapped via three-level page tables (each 4 KB, 512 entries). The kernel is mapped in the upper half of the address space (VA >= 0xFFFFFFC000000000):

```
// Map kernel at high virtual address
fn map_kernel() {
    let root = page_alloc_zeroed();
    // Gigapage identity map: VA == PA for kernel range
    for gb in (0x80000000..kernel_end).step_by(GB) {
        root.entries[high_vpn2(gb)] = PTE::leaf(
            PhysAddr(gb), Perm::RWX | Perm::GLOBAL
        );
    }
    csrw!(satp, SATP::sv39(root));
    sfence_vma!();
}
```

6. Trap Handling and the Timer

All RISC-V exceptions and interrupts go through the trap vector. The trap handler saves the register file into a TrapFrame and dispatches by cause code:

```
fn handle_trap(frame: &mut TrapFrame) {
    frame
        |>  read_mcause
        |>  classify_cause
        |?> dispatch_interrupt_or_exception
        |>  restore_frame
}

fn dispatch_interrupt_or_exception(cause: Cause) -> Result<(), TrapError> {
```

```
    match cause {
        Cause::TimerInterrupt => sched::tick(),
        Cause::ExternalInterrupt => plic::handle(),
        Cause::Syscall => syscall::dispatch(frame),
        other => panic!("unhandled trap: {:?}", other),
    }
}
```

7. The Preemptive Scheduler

The scheduler is a round-robin preemptive scheduler. Each timer tick (1ms) triggers a context switch if the current task's quantum has expired. Context saving/restoring is done in assembly:

```
// Scheduler state
static TASKS: SpinLock<VecDeque<Task>> = SpinLock::new(VecDeque::new());

fn tick() {
    let mut tasks = TASKS.lock();
    let current = tasks.pop_front().unwrap();
    tasks.push_back(current.save_context());
    let next = tasks.front_mut().unwrap();
    next.restore_context();   // resumes in assembly, never returns here
}
```

8. Lessons and Next Steps

Building Lateralus OS taught us several things about the language itself:

- The unsafe block requirement for MMIO and CSR access is valuable — it makes hardware-touching code easy to grep and audit.
- The pipeline model is a natural fit for trap dispatch: each stage is a pure transformation of the trap frame.
- The absence of a GC eliminates an entire class of kernel bugs (GC running at interrupt time) that are common in managed-language kernels.
- Formal verification of the page table mapping code is the most important next step — an incorrect mapping can lead to privilege escalation that is invisible to testing.

Lateralus OS v1.0 is available as open source and serves as the reference platform for Lateralus kernel development documentation.