Lateralus OS: Architecture Reference

Extended reference: boot sequence, trap handling, driver model, and system call table

Lateralus Language

bad-antics · July 2025 · Lateralus Language Research

ABSTRACT This document is an extended reference for Lateralus OS internals. It covers topics not addressed in the architecture overview: the boot sequence from reset vector to user shell, trap and interrupt handling, the userspace driver model, the complete system call table, and the kernel build system. It is intended as a reference for kernel contributors and driver authors.

1. Boot Sequence

On reset, the RISC-V core begins execution at address 0x1000 (SiFive boot ROM) or at the DRAM base depending on the board. The Lateralus OS boot sequence has five stages:

- **Stage 0 — ROM bootstrap**: the board ROM initializes DDR and loads the first-stage bootloader from flash.
- **Stage 1 — First-stage bootloader (fsbl)**: initializes the UART, sets up a minimal stack, and loads the kernel ELF from the storage medium.
- **Stage 2 — Kernel entry**: sets up the machine trap vector, switches from M-mode to S-mode, and calls kernel_main().
- **Stage 3 — Kernel initialization**: initializes the physical memory allocator, the virtual address space, the capability table, the scheduler, and the IPC subsystem.
- **Stage 4 — Init process**: the kernel spawns the init process from the initrd, which in turn starts the device driver servers and the shell.

```
// Kernel entry point (Lateralus source)
#[no_mangle]
pub fn kernel_main(dtb_ptr: *const u8) {
    let dt = DeviceTree::parse(dtb_ptr)?;
    mem::init(&dt);
    cap::init();
    sched::init(&dt);
    ipc::init();
    drivers::probe(&dt);
    init::spawn();
    sched::run_forever();
}
```

2. Trap and Interrupt Handling

RISC-V traps fall into two categories: synchronous exceptions (page faults, illegal instructions, system calls via the ecall instruction) and asynchronous interrupts (timer, external, software).

The kernel sets the machine trap vector to a single entry point trap_entry written in assembly. It saves all registers to the thread's trap frame and calls trap_dispatch():

```
// Trap dispatch pipeline
fn trap_dispatch(frame: &mut TrapFrame) {
    let cause = frame.mcause;
    match cause.interrupt {
        true  => handle_interrupt(cause.code, frame),
        false => handle_exception(cause.code, frame),
    }
}
```

```
fn handle_exception(code: u64, frame: &mut TrapFrame) {
    let result = frame
        |> classify_exception(code)
        |?> check_user_permission
        |?> dispatch_to_handler;
    match result {
        Ok(()) => return,
        Err(e) => signal_process(frame.pid, e),
    }
}
```

3. Timer and Clock

The RISC-V time CSR (mtime) is a monotonically increasing 64-bit counter driven by the platform clock. The kernel programs the timer comparator (mtimecmp) to fire at the next scheduling quantum boundary.

The scheduler quantum is 1 ms for interactive threads and 10 ms for batch threads. Real-time threads specify their own deadline; the scheduler programs the timer comparator to the earliest deadline among runnable RT threads.

```
// Timer interrupt handler
fn handle_timer_interrupt() {
    let now = time::rdtime();
    sched::tick(now);
    let next_quantum = sched::next_deadline();
    clint::set_mtimecmp(next_quantum);
}
```

4. Userspace Driver Model

Lateralus OS follows the L4 tradition of userspace drivers: device drivers run in userspace processes with capability tokens for their device MMIO regions and interrupt lines. The kernel routes interrupts to the registered driver process via an IPC message.

```
// Driver registration
fn register_driver(irq: IrqCap, mmio: MemMapCap) -> Result<DriverHandle, SysError>

// Interrupt notification to driver (kernel calls this)
fn irq_notify(handle: DriverHandle, irq_number: u32)
```

A driver is a Lateralus userspace program that registers for one or more interrupt lines and memory-mapped IO regions, then processes interrupt notifications in a pipeline:

```
// Example: network driver main loop
loop {
    let irq = ipc::recv_irq()?
    irq
        |>  read_dma_ring
        |>  process_received_packets
        |>  forward_to_network_stack
        |>  acknowledge_irq
}
```

5. System Call Table

The complete Lateralus OS system call table (v1.0):

```
syscall_nr  Name              Signature
```

```
----------------------------------------
0           sys_exit           (code: i32) -> !
1           sys_read           (cap: FileReadCap, buf: *mut u8, len: usize) -> isize
2           sys_write          (cap: FileWriteCap, buf: *const u8, len: usize) -> isize
3           sys_mem_map        (cap: MemMapCap, sz: usize, fl: u32) -> *mut u8
4           sys_mem_unmap      (addr: *mut u8, sz: usize) -> i32
5           sys_thread_create  (fn: usize, stack: usize) -> ThreadId
6           sys_thread_exit    (code: i32) -> !
7           sys_ipc_send       (cap: IpcCap, msg: *const u8, len: usize) -> i32
8           sys_ipc_recv       (cap: IpcCap, buf: *mut u8, len: usize) -> isize
9           sys_cap_delegate   (cap: AnyCap, to: ThreadId) -> CapId
10          sys_cap_revoke     (cap_id: CapId) -> i32
11          sys_irq_register   (irq: u32, cap: IpcCap) -> i32
12          sys_rdtime         () -> u64
```

6. Virtual Memory Layout

Each process's 64-bit virtual address space is divided into fixed regions:

```
0x0000_0000_0000_0000 - 0x0000_FFFF_FFFF_FFFF  User space (128 TiB)
  0x0000_0000_1000_0000  Executable and data (text, data, bss)
  0x0000_7FFF_E000_0000  Stack (grows down, 8 MiB by default)
  0x0000_7FFF_F000_0000  Heap (grows up, mmap region)
0xFFFF_0000_0000_0000 - 0xFFFF_FFFF_FFFF_FFFF  Kernel space (64 TiB)
  0xFFFF_C000_0000_0000  Kernel text and data
  0xFFFF_D000_0000_0000  Direct physical memory map
  0xFFFF_E000_0000_0000  Kernel heap (vmalloc region)
```

The kernel address space is mapped in every process's page table but is not accessible from user mode (RISC-V page table PMP entries enforce this).

7. Build System

Lateralus OS is built with the Lateralus build tool (ltl build) using a workspace manifest. The kernel, each driver, and the init process are separate packages in the workspace.

```
# Build the OS image for RISC-V QEMU
ltl build --target riscv64-linux-gnu --release kernel
ltl package os-image --kernel kernel.elf --initrd drivers/

# Run in QEMU
qemu-system-riscv64 -machine virt -bios none \
    -kernel os-image.bin -nographic -m 512M
```

The build produces a flat binary image with the kernel ELF and the initrd (containing driver binaries) concatenated. QEMU loads the combined image at the DRAM base and begins execution.

8. Testing Infrastructure

The kernel test suite runs in three environments: QEMU (for full integration tests), a Lateralus user-mode simulation (for unit tests of individual subsystems without hardware), and hardware (nightly on a SiFive HiFive Unmatched board).

```
# Run kernel unit tests in user-mode simulation
ltl test --features=user-mode-sim kernel

# Run integration tests in QEMU
ltl test --target riscv64-linux-gnu --qemu kernel
```

```
# Check for memory safety violations (address sanitizer)
ltl build --sanitize=address --target riscv64-linux-gnu kernel
```

The user-mode simulation compiles the kernel as a Lateralus program that replaces hardware operations with in-process simulations. This allows the full kernel test suite to run on any platform that supports the Lateralus compiler, including the CI server.