Lateralus OS Architecture

A pipeline-native RISC-V operating system: kernel, scheduler, and userspace ABI

Lateralus Language

bad-antics · June 2025 · Lateralus Language Research

ABSTRACT Lateralus OS is a RISC-V operating system written in Lateralus. It is structured as a microkernel with capability-based access control, a pipeline-native IPC mechanism, and a userspace that exposes the four pipeline operators as first-class system calls. The kernel is approximately 12,000 lines of Lateralus, excluding device drivers. This paper describes the kernel architecture, the scheduling model, the capability system, and the userspace ABI, with emphasis on how the pipeline-native language features improve the reliability and auditability of OS code.

1. Design Philosophy

Operating system code is notoriously difficult to get right. The combination of concurrency, hardware interaction, and safety-critical invariants creates a class of bugs (race conditions, use-after-free, capability leaks) that are rare in userspace code but catastrophic in the kernel.

Lateralus OS addresses these problems through three design choices: a microkernel architecture (minimize trusted code), capability-based security (every operation requires an explicit, unforgeable token), and Lateralus's ownership system (eliminate use-after-free and data races at the language level). The pipeline model additionally makes the control flow of kernel request processing explicit and auditable.

2. Microkernel Structure

The kernel provides five primitive services and nothing else:

- **Memory management**: physical frame allocation, virtual address space mapping.
- **Scheduling**: thread creation, destruction, and preemptive scheduling.
- **IPC**: synchronous message passing and asynchronous channel passing.
- **Capability management**: creation, delegation, and revocation of capabilities.
- **Interrupt routing**: mapping hardware interrupts to userspace capability invocations.

All device drivers, the file system, and the network stack run in userspace. The kernel boundary is enforced by RISC-V privilege levels: kernel code runs in machine mode (M-mode), trusted services run in supervisor mode (S-mode), and user applications run in user mode (U-mode).

3. Capability-Based Access Control

A capability is an unforgeable token that grants access to a kernel resource. All system calls take a capability as the first argument; a call without the correct capability is rejected immediately.

```
// System call: read from a file descriptor (capability-based)
fn read(cap: FileReadCap, buf: &mut [u8]) -> Result<usize, IoError>

// Capability table per process
struct CapabilityTable {
    entries: Vec<Capability>,
}

enum Capability {
    FileRead  { inode: u64, offset_limit: u64 },
    FileWrite { inode: u64, offset_limit: u64 },
    NetConnect { allowed_addr: IpNet },
```

```
    MemMap    { phys_base: PhysAddr, size: usize, flags: MapFlags },
    Irq       { irq_number: u32 },
}
```

Capabilities can be delegated: a process with a FileRead capability can create a restricted sub-capability (e.g., read-only access to a file with a lower offset_limit) and pass it to a child process. Revocation removes the capability from the delegating process's table and all derived capabilities.

4. The Pipeline IPC Mechanism

IPC in Lateralus OS is modeled as a pipeline: a request passes through a sequence of service handlers, each of which may transform it before passing it to the next. This is a natural fit for the Lateralus pipeline operators.

```
// Kernel IPC: sending a request through a service pipeline
let response = request
    |?> auth_service::check_capability
    |?> rate_limiter::check
    |>  file_server::dispatch
    |>  compress_if_requested
```

The kernel enforces that each stage in an IPC pipeline is a capability-checked operation. A malicious userspace process cannot skip a stage by manipulating the pipeline value; the capability check is in the kernel and is invisible to userspace.

5. Scheduler Design

The Lateralus OS scheduler is a multi-level feedback queue (MLFQ) with priority levels for real-time, interactive, and batch workloads. Threads are scheduled per-CPU with work-stealing for load balancing.

```
// Scheduler priority levels
enum Priority {
    RealTime    { deadline_ns: u64 },   // EDF within RT class
    Interactive { boost_factor: f32 },  // MLFQ boost on I/O
    Batch,                              // round-robin, no boost
    Idle,                               // only when CPU is idle
}
```

The pipeline-native language enables the scheduler to be expressed as a pipeline of scheduling decisions:

```
let next_thread = cpu_run_queue
    |> select_real_time_if_deadline_approaching
    |> select_interactive_if_recently_blocked
    |> select_highest_priority_batch
    |> idle_if_no_runnable
```

This pipeline formulation makes the scheduling policy explicit and auditable: each stage is a separate function that can be unit-tested independently.

6. Memory Management

Physical memory is managed by a buddy allocator. Virtual memory is managed per-process in a red-black tree of virtual memory areas (VMAs). The ownership system prevents double-free of physical frames: a frame is represented as an owned value; freeing it moves it back to the allocator pool.

```
// Physical frame allocation — returns owned frame
```

```
fn alloc_frame() -> Result<OwnedFrame, MemError>

// Freeing a frame requires ownership (cannot double-free)
fn free_frame(frame: OwnedFrame)  // frame is consumed
```

The Lateralus compiler enforces the ownership invariant: if the programmer writes code that would free a frame twice, the compiler rejects it at compile time with a use-after-move error.

7. Userspace ABI

The Lateralus OS userspace ABI is a set of typed system call wrappers that expose the kernel's five services as Lateralus functions. The ABI is versioned and stable across kernel versions within a major version series.

```
// Userspace ABI stubs (in the standard library)
pub fn mem_map(cap: MemMapCap, size: usize, flags: MapFlags) -> Result<*mut u8, SysError>
pub fn thread_create(f: fn(), stack_size: usize) -> Result<ThreadHandle, SysError>
pub fn ipc_send(cap: IpcCap, msg: &[u8]) -> Result<(), SysError>
pub fn ipc_recv(cap: IpcCap, buf: &mut [u8]) -> Result<usize, SysError>
```

The system call wrappers handle the RISC-V syscall convention (argument registers a0-a5, system call number in a7) and convert the integer return codes to Lateralus Result values.

8. Current Status and Roadmap

Lateralus OS is in active development. Current status: the kernel boots to a shell on RISC-V QEMU and on the SiFive HiFive Unmatched board. The file system (a journaling filesystem adapted from ext2 semantics) is functional for reads; writes are stable but not yet crash-safe. The network stack handles Ethernet + IPv4 + TCP; UDP is in progress.

Roadmap: crash-safe filesystem (6 months), UDP network stack (3 months), GPU framebuffer driver for HiFive (6 months), and a native Lateralus package manager running on the OS itself (12 months).