Bare-Metal OS in a High-Level Language

Why Lateralus is viable for kernel development: ownership, no GC, and zero-cost abstractions

Lateralus Language

bad-antics · April 2026 · Lateralus Language Research

ABSTRACT Conventional wisdom holds that operating system kernels must be written in C or assembly: high-level languages impose garbage collection pauses, abstraction overhead, or runtime dependencies that are incompatible with bare-metal execution. Lateralus challenges this assumption. Its ownership-based memory model provides C-level control without GC; its zero-cost abstractions compile to code equivalent to hand-written C; and its inline assembly support covers the 5-10% of kernel code that requires direct hardware access. This paper explains why Lateralus is viable for kernel development and compares it directly against C, Rust, and Ada/SPARK.

1. The Kernel Development Requirements

A language suitable for kernel development must satisfy five requirements:

- **No garbage collector**: GC pauses are incompatible with real-time interrupt handling.
- **No runtime**: the language runtime must be absent or optional; kernels are their own runtimes.
- **Direct hardware access**: memory-mapped I/O, inline assembly for CSR manipulation, pointer arithmetic.
- **Predictable performance**: no hidden allocations, no unexpected indirections, no nondeterministic behavior.
- **Safety guarantees**: the language should prevent common kernel bugs (use-after-free, data races) at compile time.

C satisfies 1-4 but not 5. Rust satisfies all five but its ergonomics for systems programming are sometimes laborious. Lateralus satisfies all five with higher-level abstractions than Rust and a syntax designed for pipeline-first code.

2. No Garbage Collector

Lateralus uses ownership-based memory management: every value has exactly one owner, and memory is freed when the owner goes out of scope. There is no GC, no reference counting (unless the programmer explicitly uses Rc<T>), and no finalization pauses.

```
// Memory is freed deterministically at scope exit
fn process_packet(data: Vec<u8>) -> Result<(), Error> {
    let frame = parse_ethernet(&data)?;  // data owned here
    let ip_pkt = parse_ip(&frame)?;
    route(ip_pkt)?;
    // 'data' and 'frame' freed here, deterministically
    Ok(())
}
```

3. No Required Runtime

Lateralus programs can be compiled with --no-std to exclude the standard library and produce a bare-metal binary with no runtime dependencies. Only the core language (arithmetic, ownership, pattern matching, inline assembly) is available.

```
// Bare-metal Lateralus kernel entry
#![no_std]
#![no_main]
```

```
#[no_mangle]
pub fn _start() -> ! {
    uart::init();
    mem::init();
    loop { sched::run_once() }
}
```

The #![no_std] attribute disables the standard library import. The #![no_main] attribute disables the default entry point; _start is the ELF entry defined in the linker script.

4. Hardware Access: Inline Assembly and MMIO

Approximately 5-10% of kernel code requires direct hardware access. Lateralus provides two mechanisms: inline assembly for CPU instructions and typed wrappers for memory-mapped I/O.

```
// Inline assembly: read the RISC-V time CSR
fn rdtime() -> u64 {
    let t: u64;
    unsafe { asm!("csrr {0}, time", out(reg) t) }
    t
}

// MMIO: typed wrapper for a UART register block
struct Uart16550 { base: *mut u32 }
impl Uart16550 {
    fn write_byte(&self, b: u8) {
        unsafe { self.base.add(THR_OFFSET).write_volatile(b as u32) }
    }
}
```

5. Zero-Cost Pipeline Abstractions

Kernel code often has complex control flow: interrupt routing, syscall dispatch, network stack processing. Lateralus pipelines express this clearly without overhead:

```
// Syscall dispatch as a pipeline — compiles to a jump table
fn handle_syscall(frame: &TrapFrame) -> SyscallResult {
    frame
        |>  extract_syscall_number
        |>  validate_arguments
        |?> dispatch_to_handler
        |>  return_to_user
}
```

The compiler emits a jump table for the dispatch stage when it can enumerate the cases statically. There is no function call overhead for the pipeline: all stages are inlined into the dispatch function.

6. Comparison with C, Rust, and Ada/SPARK

Property	Lateralus	C	Rust	Ada/SPARK
No GC	Yes	Yes	Yes	Yes
No required runtime	Yes	Yes	Yes	Yes
MMIO support	Yes	Yes	Yes	Yes
Inline assembly	Yes	Yes	Yes	Limited
Memory safety (CT)	Yes	No	Yes	Partial
Data race freedom	Yes	No	Yes	No
Formal verification	Planned	No	Partial	Yes
Pipeline model	Yes	No	No	No

Lateralus's advantages over C: memory and data-race safety. Over Rust: the pipeline model reduces boilerplate for data-flow-heavy kernel code. Over Ada/SPARK: full formal verification is planned but not yet available; SPARK leads here.

7. Limitations

Current limitations of Lateralus for kernel development:

- No Lateralus-native AArch64 backend (x86-64 and RISC-V only). AArch64 support is on the roadmap for v2.0.
- No built-in support for ACPI parsing or PCIe enumeration; these are available via C library bindings through the polyglot bridge.
- The async pipeline model requires a runtime scheduler; for kernel interrupt handlers (which are not async), only the total and error variants can be used.

8. Conclusion

Lateralus is viable for kernel development today for RISC-V and x86-64 targets. The combination of ownership-based memory management, zero-cost pipeline abstractions, and inline assembly support covers the full range of kernel programming requirements. Lateralus OS is proof by construction: a working RISC-V kernel written entirely in Lateralus.