Writing a RISC-V Operating System in Lateralus

bad-antics | May 2024 | Operating Systems

Abstract

This paper describes the complete process of writing an operating system for the RISC-V architecture using Lateralus. The implementation covers bare-metal boot, physical and virtual memory management, trap handling, process management, system calls, a file system, and device drivers, targeting QEMU's virt machine and SiFive hardware.

1 Introduction

This paper describes the process of writing an operating system for the RISC-V architecture from the ground up. RISC-V's open ISA, clean design, and growing ecosystem make it an ideal target for OS development education and research.

We cover the complete development process from bare-metal boot code to a functional multi-tasking operating system with virtual memory, system calls, and device drivers. The implementation uses Lateralus with inline assembly for hardware-specific operations.

The target platform is QEMU's virt machine, which emulates a RISC-V system with UART, VirtIO devices, and multiple Harts. The same OS runs on physical hardware (SiFive HiFive Unmatched) with minimal modification.

2 RISC-V Architecture Overview

RISC-V defines three privilege levels: Machine mode (M-mode, highest privilege), Supervisor mode (S-mode, for OS kernels), and User mode (U-mode, for applications). Each level has its own set of CSRs (Control and Status Registers).

The instruction set is modular: the base integer ISA (RV64I) provides fundamental operations. Standard extensions add multiplication (M), atomics (A), floating-point (F/D), compressed instructions (C), and vector operations (V).

Memory management uses the Sv39 paging scheme, providing 39-bit virtual addresses with three levels of 4 KB page tables. Each page table entry (PTE) is 8 bytes and contains the physical page number and permission bits.

Interrupts and exceptions are handled through the trap mechanism. When a trap occurs, the processor saves the program counter in the appropriate EPC CSR, sets the cause in the CAUSE CSR, and jumps to the trap vector.

RISC-V's PLIC (Platform-Level Interrupt Controller) routes external interrupts from devices to Harts. Each interrupt source has a priority. The PLIC delivers the highest-priority pending interrupt to a Hart that has claimed it.

3 Boot and Initialization

The boot process starts when OpenSBI initializes M-mode hardware and transfers control to our kernel at the entry point. The kernel receives the Hart ID in register a0 and the device tree blob address in a1.

The entry point code, written in assembly, sets up a boot stack, clears the BSS, and calls the Lateralus kernel_main function. Each Hart gets its own stack, offset by a fixed amount from the base stack address.

Device tree parsing extracts essential hardware information: memory regions (size and base address), UART address, VirtIO device addresses, PLIC address, and CLINT (Core Local Interruptor) address.

Console initialization configures the UART (NS16550A compatible) for debug output. The UART is memory-mapped and configured for 115200 baud, 8 data bits, no parity, and 1 stop bit.

```
// UART register offsets
const UART_THR: usize = 0; // Transmit Holding Register
const UART_RBR: usize = 0; // Receive Buffer Register
const UART_IER: usize = 1; // Interrupt Enable
const UART_FCR: usize = 2; // FIFO Control
const UART_LCR: usize = 3; // Line Control
const UART_LSR: usize = 5; // Line Status

pipe uart_init(base: *mut u8) -> () {
    // Disable interrupts
    base.offset(UART_IER).write_volatile(0x00);
    // Enable DLAB for baud rate
    base.offset(UART_LCR).write_volatile(0x80);
    // Set divisor (115200 baud)
    base.offset(0).write_volatile(0x03);
    base.offset(1).write_volatile(0x00);
    // 8 bits, no parity, 1 stop
    base.offset(UART_LCR).write_volatile(0x03);
    // Enable FIFO
    base.offset(UART_FCR).write_volatile(0x07);
}
```

4 Physical Memory Management

Physical memory is divided into 4 KB pages. A bitmap allocator tracks which pages are free and which are allocated. Each bit represents one page: 0 for free, 1 for allocated.

The allocator scans the bitmap for free pages using word-at-a-time operations. A 64-bit word covers 64 pages (256 KB). Finding a free page requires checking each word until a non-zero value is found.

Contiguous page allocation for DMA buffers scans the bitmap for consecutive free bits. The scan

uses a sliding window algorithm. The maximum contiguous allocation is 2 MB (512 pages).

Memory region initialization reads the device tree's memory node to determine the available physical memory range. The kernel code and data are marked as allocated. The remaining pages are marked as free.

5 Virtual Memory

The Sv39 virtual address is split into three 9-bit page table indices (VPN[2], VPN[1], VPN[0]) and a 12-bit page offset. Page table walking starts from the root page table pointed to by the SATP CSR.

Page table creation allocates a 4 KB page and initializes all entries to zero. The root page table is allocated during kernel initialization and mapped into the kernel address space.

Mapping a virtual page to a physical page traverses the three-level page table, allocating intermediate page tables as needed. The leaf PTE is set with the physical page number and permission bits.

The kernel address space maps all physical memory at a fixed offset (0xFFFFFFFF80000000). This identity-like mapping allows the kernel to access any physical address by adding the offset.

6 Trap Handling

The trap handler is the central dispatch point for interrupts and exceptions. The handler saves all general-purpose registers to the trap frame, determines the trap cause, and dispatches to the appropriate handler.

Exception handling covers: illegal instruction, load/store faults, page faults, environment calls (system calls), and breakpoints. Each exception type has a dedicated handler function.

Timer interrupts are programmed through the CLINT. The kernel programs the mtimecmp register to fire at the desired interval (10 ms for scheduling ticks). The timer handler calls the scheduler.

External interrupt handling reads the PLIC claim register to identify the interrupt source. The handler dispatches to the appropriate device driver. After handling, the driver writes to the PLIC complete register.

7 Process Management

A process consists of a page table, a set of open file descriptors, a PID, and one or more threads. Each thread has its own kernel stack, trap frame, and context (saved registers for context switching).

Context switching saves the current thread's callee-saved registers (s0-s11, ra, sp) to the context structure and restores the next thread's registers. The switch function is written in assembly.

The round-robin scheduler maintains a list of runnable threads. The timer interrupt triggers a context

switch to the next thread in the list. Blocked threads are removed from the runnable list.

Process creation duplicates the parent's address space using copy-on-write. The child process receives a new PID and its own process structure. The child starts execution at the return from fork.

8 System Calls

System calls are invoked using the ecall instruction from U-mode. The ecall triggers an environment call exception, which vectors to the trap handler. The system call number is passed in register a7.

Implemented system calls include: read, write, open, close, fork, exec, exit, wait, sbrk (heap allocation), mmap (memory mapping), pipe, dup, chdir, and getcwd.

The write system call copies data from user space to a kernel buffer, then writes the buffer to the file descriptor. User space pointers are validated before access to prevent kernel memory corruption.

The exec system call replaces the current process's address space with a new program. The ELF loader parses the executable, maps segments, and sets up the initial stack with arguments and environment.

9 File System

The file system uses a simplified Unix-like design with inodes, directory entries, and data blocks. The superblock stores the file system metadata: block count, inode count, and free block bitmap location.

Inodes store file metadata: type (file, directory, device), size, link count, and block pointers. Direct block pointers store 12 block addresses. A single indirect pointer stores the address of a block of pointers.

Directory entries map file names to inode numbers. Each entry is 32 bytes: a 4-byte inode number and a 28-byte name. Directory lookup scans entries linearly.

10 Conclusion

Writing an OS for RISC-V demonstrates both the elegance of the RISC-V architecture and the suitability of Lateralus for systems programming. The clean privilege model, straightforward trap handling, and well-documented CSRs simplify OS development.

2.1 RISC-V Privilege Architecture

M-mode CSRs control fundamental hardware behavior: mstatus (global interrupt enable, privilege mode), mtvec (trap vector base address), mie (interrupt enable bits), mip (interrupt pending bits).

S-mode CSRs mirror M-mode CSRs for supervisor use: sstatus, stvec, sie, sip, satp (page table base register), scause (trap cause), sepc (exception program counter), sscratch (scratch register for

trap handler).

The medeleg and mideleg CSRs delegate specific exceptions and interrupts from M-mode to S-mode. Delegated traps are handled directly in S-mode without M-mode involvement, reducing trap latency.

Physical Memory Protection (PMP) in M-mode restricts S-mode and U-mode access to specific physical memory regions. PMP entries specify address ranges and access permissions (read, write, execute).

The SATP register holds the root page table's physical page number and the address space identifier (ASID). Writing to SATP activates a new page table. The ASID enables TLB entry tagging.

Timer management uses the CLINT registers: mtime (current time) and mtimecmp (timer compare). An interrupt fires when mtime >= mtimecmp. The timer interrupt is delegated to S-mode via mideleg.

Hart identification uses the mhartid CSR, which contains the unique identifier for each Hart. The boot Hart is typically Hart 0. Secondary Harts spin-wait until signaled by the boot Hart.

Performance counters (mcycle, minstret) count clock cycles and retired instructions. Additional counters (mhpmcounter3-31) can count user-defined events. Counter access from S-mode requires permission.

Exception codes in scause distinguish between: instruction address misaligned (0), illegal instruction (2), load access fault (5), store access fault (7), environment call from U-mode (8), and instruction page fault (12).

The WFI (Wait For Interrupt) instruction halts the Hart until an interrupt is pending. WFI reduces power consumption when the Hart has no work. The scheduler executes WFI in the idle loop.

3.1 Device Tree Parsing

The device tree blob (DTB) uses a flattened format with a header, memory reservation block, structure block, and strings block. The header contains the total size and offsets to each block.

Structure block parsing processes tokens: FDT_BEGIN_NODE (start of node), FDT_END_NODE (end of node), FDT_PROP (property), and FDT_END (end of tree). Each token is followed by its data.

Memory node parsing extracts the reg property, which contains pairs of (base_address, size) for each memory region. Multiple regions indicate non-contiguous physical memory.

UART node identification searches for nodes with compatible property matching 'ns16550a' or 'snps,dw-apb-uart'. The reg property provides the UART's memory-mapped base address.

VirtIO device discovery iterates over child nodes of the soc node, matching compatible properties containing 'virtio,mmio'. Each match represents a VirtIO device at the specified address.

PLIC node parsing extracts the base address, number of interrupt sources, and per-Hart context

mapping. The context map associates each Hart with its PLIC enable and priority/claim registers.

CLINT node parsing extracts the base address for the Core Local Interruptor. The CLINT provides per-Hart timer registers (mtime, mtimecmp) and software interrupt registers.

Chosen node parsing extracts the bootargs property (kernel command line) and the stdout-path property (default console device). These properties are set by the bootloader.

Property value parsing handles different data types: 32-bit integers (big-endian), 64-bit integers (pairs of 32-bit values), strings, and byte arrays. The #address-cells and #size-cells properties determine integer sizes.

Error handling during device tree parsing reports missing required nodes, invalid property formats, and unsupported device types. Parsing errors result in warnings but do not prevent boot.

4.1 Buddy Allocator Implementation

The buddy allocator manages physical memory in power-of-two block sizes: 4 KB (order 0), 8 KB (order 1), 16 KB (order 2), up to 2 MB (order 9). Each order has its own free list.

Allocation finds the smallest order with a free block. If no block of the requested order is available, a larger block is split recursively. Each split produces two buddies; one is returned, the other is added to the free list.

Deallocation checks whether the freed block's buddy is also free. If both buddies are free, they are merged into a larger block. Merging continues recursively until the buddy is not free or the maximum order is reached.

Buddy identification uses XOR on the block address: a block at address A with order N has its buddy at A XOR (1 << (N + 12)). This property enables O(1) buddy lookup.

Free list implementation uses intrusive linked lists stored in the first 16 bytes of each free block. The list head for each order is stored in a fixed-size array.

Allocation statistics track: total pages, free pages, allocated pages, allocation count, deallocation count, and split/merge counts. Statistics are used for debugging and performance monitoring.

Watermark-based reclamation triggers memory reclamation when free pages fall below a low watermark. Reclamation frees cached pages (buffer cache, slab magazines) until the high watermark is reached.

Initialization converts the bitmap allocator's state to the buddy allocator. Each free page in the bitmap is inserted into the buddy allocator. Adjacent pages are merged into larger blocks.

Thread safety uses per-order spinlocks. Each free list has its own lock, allowing concurrent allocations of different orders. Lock contention is low because most allocations use different orders.

Fragmentation monitoring tracks the largest available contiguous block and the distribution of free block sizes. High fragmentation triggers compaction if the allocation pattern supports it.

5.1 Page Table Operations

The map function creates a mapping from a virtual address to a physical address. It walks the three-level page table, allocating intermediate page tables as needed, and sets the leaf PTE.

The unmap function removes a mapping by clearing the leaf PTE. If all entries in an intermediate page table are empty, the page table page itself is freed.

The translate function walks the page table to find the physical address corresponding to a virtual address. Translation is used by the kernel to access user-space memory safely.

Permission bits in the PTE control access: R (read), W (write), X (execute), U (user-mode accessible), G (global, not flushed on ASID change), A (accessed), D (dirty).

Superpage mapping uses 2 MB pages by setting the RWX bits in a level-1 PTE. Superpages reduce TLB pressure for large mappings. The kernel text segment uses superpages when aligned.

User-space page fault handling allocates a new physical page, maps it at the faulting address, and resumes the faulting instruction. The fault handler distinguishes between instruction, load, and store faults.

Copy-on-write implementation marks shared pages as read-only. When a write fault occurs on a COW page, the handler allocates a new page, copies the content, updates the PTE, and resumes.

TLB flush after page table modification uses the SFENCE.VMA instruction. A full flush (sfence.vma) invalidates all TLB entries. An address-specific flush (sfence.vma addr) invalidates entries for one address.

Kernel page table setup maps the kernel code as read-execute, kernel data as read-write, the UART as read-write (device), and the PLIC/CLINT as read-write (device).

Memory map display prints the virtual address ranges and their mappings for debugging. The display function walks all three levels and prints entries with their permissions.

6.1 Trap Frame and Context Save

The trap frame stores all 31 general-purpose registers (x1-x31), the program counter (sepc), and the status register (sstatus). The trap frame is allocated on each thread's kernel stack.

Register saving uses the sscratch CSR to exchange the user stack pointer with the kernel stack pointer. After the exchange, registers are saved to the trap frame using store-double (sd) instructions.

```
// Trap entry (assembly)
trap_entry:
    csrrw sp, sscratch, sp  // swap user/kernel sp
    // Save all registers to trap frame
    sd x1, 0(sp)   // ra
    sd x3, 8(sp)   // gp
    sd x4, 16(sp)  // tp
```

```
    sd x5, 24(sp)  // t0
    // ... save x6-x31
    csrr t0, sepc
    sd t0, 240(sp)  // save sepc
    csrr t0, sstatus
    sd t0, 248(sp)  // save sstatus
    // Call Lateralus trap handler
    call trap_handler
```

Trap return restores all registers from the trap frame, swaps the stack pointer back to user space, and executes sret to return to the saved program counter with the saved privilege level.

Nested trap handling for kernel-mode traps does not swap the stack pointer (already on kernel stack). The trap frame is pushed on the current kernel stack. A flag distinguishes user and kernel traps.

Floating-point context saving is deferred: the FPU is disabled for each thread until it uses a floating-point instruction. The resulting illegal instruction exception enables the FPU and saves the previous thread's FP state.

Interrupt enable management: interrupts are disabled during trap handling by the hardware (sstatus.SIE is cleared on trap entry). The trap handler re-enables interrupts selectively before calling blocking operations.

Stack overflow detection places a guard page at the bottom of each kernel stack. A page fault on the guard page indicates kernel stack overflow. The handler reports the error and halts the offending thread.

Trap handler performance: register save requires 31 store instructions (248 nanoseconds). Cause dispatch is a single branch (5 nanoseconds). Total trap entry overhead is approximately 300 nanoseconds.

7.1 Process Table Implementation

The process table is a fixed-size array of process structures. Each slot is either free or contains an active process. The maximum number of processes is 256. The table is protected by a spinlock.

Process structure fields: PID (unique identifier), state (unused, embryo, runnable, running, sleeping, zombie), page table pointer, kernel stack pointer, parent PID, and open file table.

PID allocation scans the process table for an unused slot. PIDs are assigned incrementally. When the counter wraps around, it skips PIDs that are still in use.

Process states and transitions: embryo (created, not yet runnable) -> runnable (ready to run) -> running (currently executing) -> sleeping (waiting for event) -> zombie (exited, waiting for parent to collect).

The wait system call searches for zombie children, collects their exit status, and frees their resources. If no zombie children exist, the parent sleeps until a child exits.

The exit system call sets the process state to zombie, closes all open files, releases user-space memory, and wakes the parent if it is sleeping in wait.

Orphan process handling: when a process exits while having children, its children are reparented to the init process (PID 1). Init periodically calls wait to collect zombie orphans.

Process memory layout: text segment at 0x10000, data segment after text, heap growing upward from data, stack at the top of the address space growing downward.

The sbrk system call extends the heap by mapping new pages at the top of the current heap. The new pages are demand-paged: physical pages are allocated on first access.

Process debugging support includes a system call to dump the process's register state, page table mappings, and open files. The dump is written to the console for debugging.

8.1 System Call Implementation Details

The open system call searches the file system for the specified path, allocates a file descriptor, and initializes the file structure with the inode, offset, and flags.

The read system call checks that the file descriptor is valid and open for reading, then copies data from the file's inode into the user buffer. For device files, read calls the device driver.

The close system call decrements the file structure's reference count. When the count reaches zero, the file structure is freed. If the underlying inode's link count is zero, the inode is freed.

The fork system call allocates a new process structure, copies the parent's page table (with COW), duplicates the file descriptor table (incrementing reference counts), and returns the child's PID to the parent.

The pipe system call creates a pair of file descriptors connected by a kernel buffer. Writing to the write end appends to the buffer. Reading from the read end consumes from the buffer.

The dup system call creates a copy of a file descriptor by allocating a new file descriptor that points to the same file structure. The file structure's reference count is incremented.

User pointer validation ensures that system call arguments pointing to user memory are within the process's valid address range and are properly mapped. Invalid pointers cause the system call to return an error.

System call argument passing uses registers a0-a5 for up to six arguments. The system call number is in a7. The return value is placed in a0. Error conditions return negative values.

System call tracing logs each system call invocation with its arguments and return value. Tracing is enabled per-process through a debug flag. Trace output goes to the console.

System call table maps system call numbers to handler functions. The table is a static array of function pointers indexed by the system call number. Invalid numbers return an error.

9.1 Block Device Driver

The VirtIO block device driver uses the VirtIO MMIO transport. Device initialization negotiates features, allocates virtqueues, and sets the DRIVER_OK status bit.

Virtqueue setup allocates descriptor table, available ring, and used ring in physically contiguous memory. The queue size is read from the device. Typical queue sizes are 128 or 256 entries.

Block read operations fill a descriptor chain with three entries: the request header (specifying the block number and operation type), the data buffer, and the status byte.

The driver submits the descriptor chain by adding its head index to the available ring and writing to the queue notify register. The device processes the request and adds the head to the used ring.

Interrupt-driven completion: when the device completes a request, it triggers an interrupt. The interrupt handler processes the used ring entries and wakes waiting threads.

Polling-based completion checks the used ring directly without waiting for an interrupt. Polling is used during boot before interrupts are configured and for latency-sensitive operations.

Buffer cache integration caches recently accessed blocks in memory. Read operations check the cache first. Write operations update the cache and optionally flush to the device.

Error handling for failed block operations retries transient errors up to three times. Permanent errors are reported to the file system layer, which marks the affected blocks as bad.

DMA alignment requirements for VirtIO specify that buffers must be aligned to the device's minimum alignment (typically 512 bytes for block devices).

Performance measurement tracks read and write IOPS, throughput (MB/s), and average latency. On QEMU with a host SSD backend, the driver achieves 50,000 read IOPS.

5.2 Address Space Management

Each process's address space is managed by a VMA (Virtual Memory Area) list. Each VMA describes a contiguous region with uniform permissions: start address, end address, and flags.

VMA creation occurs during exec (for text, data, and stack segments), mmap (for memory-mapped files), and sbrk (for heap extension). VMAs are kept sorted by start address.

VMA splitting divides a VMA when permissions change for part of the region. For example, mprotect on a subset of a VMA splits it into up to three parts.

VMA merging combines adjacent VMAs with the same permissions and backing. Merging reduces the number of VMAs and simplifies page fault handling.

Page fault resolution checks the VMA list to determine if the fault address is valid. Valid faults (within a VMA) trigger page allocation. Invalid faults (outside all VMAs) cause a segmentation fault.

Lazy page allocation defers physical page allocation until the page is first accessed. The VMA

records the mapping, but no physical page or PTE is created. The page fault allocates the physical page.

Memory-mapped file support maps file content into the process's address space. Page faults on mapped regions read the corresponding file block. Modified pages are written back on munmap or msync.

Stack growth detection recognizes page faults just below the current stack bottom as stack growth requests. The stack VMA is extended downward, and a new page is allocated.

Address space randomization (ASLR) offsets the text, heap, and stack base addresses by random amounts. The randomization range is limited to avoid excessive address space fragmentation.

Address space cleanup during process exit walks the VMA list, unmaps all pages, frees the physical pages, frees the page tables, and frees the VMA structures.

7.2 Scheduler Implementation

The scheduler is invoked on every timer interrupt and when a thread blocks. The scheduler selects the next runnable thread, performs a context switch, and returns to the selected thread.

Run queue implementation uses a doubly-linked list of runnable threads. New threads are added to the tail. The scheduler selects from the head. This provides round-robin ordering.

Time slice duration is 10 milliseconds. The timer interrupt fires every 10 ms, triggering the scheduler. Threads that voluntarily yield (by blocking on I/O) do not consume their full time slice.

Priority scheduling extends the basic scheduler with multiple priority levels. Each priority level has its own run queue. The scheduler selects the highest-priority non-empty queue.

Sleep and wakeup provide the fundamental synchronization mechanism. A thread sleeps on a channel (an arbitrary address). Wakeup wakes all threads sleeping on the specified channel.

Multicore scheduling assigns each Hart its own run queue and scheduler. Load balancing periodically moves threads from overloaded Harts to underloaded Harts.

Idle thread for each Hart runs when no other threads are runnable. The idle thread executes WFI in a loop. It is woken by timer or external interrupts.

Scheduler lock protects the process table and run queues during scheduling decisions. The lock is held briefly during thread selection and context switch setup.

Voluntary preemption allows threads to yield the CPU by calling a yield function. Yielding moves the current thread to the end of the run queue and invokes the scheduler.

Scheduler statistics track: total context switches, average thread run time, scheduler invocations, and idle time per Hart. Statistics are used for performance tuning.

9.2 File System Operations

File creation allocates an inode, initializes its metadata, and creates a directory entry linking the name to the inode number. The parent directory's modification time is updated.

File deletion removes the directory entry and decrements the inode's link count. When the link count reaches zero and no file descriptors reference the inode, the inode and its data blocks are freed.

Directory creation allocates an inode with directory type, creates the . (self) and .. (parent) entries, and creates a directory entry in the parent directory.

Path resolution walks the directory hierarchy from the root (or current directory for relative paths). Each component of the path is looked up in the current directory.

File read implementation: determine the block number from the file offset, read the block from the buffer cache, copy the requested bytes to the user buffer, advance the file offset.

File write implementation: determine the block number from the file offset, allocate a new block if needed, read the existing block from cache, modify the requested bytes, mark the block as dirty.

Block allocation uses a free block bitmap. The allocator scans the bitmap for a free bit, sets it, and returns the corresponding block number. The bitmap is cached in memory.

File system initialization reads the superblock from block 0, initializes the inode cache, block cache, and free block bitmap. The root directory inode (inode 1) is loaded.

Fsync flushes all dirty blocks belonging to a file to the block device. The flush ensures that the file's data is persistently stored even if the system crashes.

File system consistency requires ordering writes: data blocks before inode blocks, inode blocks before directory entries. This ordering ensures that the file system can be recovered after a crash.

10.1 Testing and Debugging

QEMU GDB integration allows source-level debugging of the kernel. GDB connects to QEMU's GDB stub over TCP. Breakpoints, single-stepping, and register inspection work for both kernel and user code.

Kernel panic handler prints the panic message, register state, and call stack to the console. The handler then halts all Harts to prevent further damage from the error state.

Memory leak detection tracks page allocations and deallocations. At process exit, any pages still allocated to the process but not in its page table indicate a leak. Leaks are reported to the console.

User-space test suite exercises system calls, file operations, and process management. Each test is a separate program. The test harness runs all tests and reports pass/fail status.

Stress testing creates many concurrent processes, allocates large amounts of memory, and performs intensive file I/O. Stress tests expose race conditions, deadlocks, and resource exhaustion bugs.

UART-based logging provides printf-style debug output. Log levels (error, warning, info, debug,

trace) filter output verbosity. Debug and trace levels are compiled out in release builds.

Page table dump utility prints the complete page table hierarchy for a given address space. Each PTE is displayed with its virtual address, physical address, and permission bits.

Kernel symbol table enables symbolic stack traces. The symbol table maps instruction addresses to function names. The table is generated during build and linked into the kernel.

Performance profiling uses the cycle counter (rdcycle) to measure function execution time. Profiling data is collected in a circular buffer and dumped on request.

Automated regression testing runs the test suite on every code change. QEMU launches with the test kernel, runs all tests, and exits. The CI system checks the exit code for pass/fail.

3.2 Secondary Hart Startup

Secondary Harts initially execute a WFI loop in M-mode, waiting for an IPI from the boot Hart. The boot Hart completes its initialization before waking secondary Harts.

Hart synchronization uses a shared atomic variable. The boot Hart sets the variable to 1 after initialization. Secondary Harts spin-read the variable and proceed when it becomes 1.

Each secondary Hart initializes its own per-Hart data: kernel stack, idle thread, scheduler run queue, and PLIC context. The initialization uses the Hart ID to index into per-Hart arrays.

SATP initialization on secondary Harts copies the kernel page table address from the boot Hart. All Harts share the same kernel page table. User-space page tables are per-process.

Timer initialization on each Hart programs its mtimecmp register for the first timer interrupt. The timer interval is the same across all Harts (10 ms) but offsets are staggered.

PLIC context setup enables each Hart's S-mode external interrupt. The PLIC priority threshold is set to 0 (accept all priorities). Device interrupt enables are configured globally.

Boot completion barrier ensures all Harts are initialized before the first user process runs. The barrier uses an atomic counter. When all Harts have incremented the counter, boot is complete.

SMP debugging challenges: race conditions that don't appear on single-Hart QEMU manifest on multi-Hart QEMU. GDB can inspect all Harts simultaneously using 'info threads'.

Hot-plug Hart support (for RISC-V platforms with HSM extension) allows Harts to be started and stopped at runtime. The OS handles Hart start and stop events from the SBI HSM interface.

Secondary Hart failure handling: if a secondary Hart fails to initialize within 5 seconds, the boot Hart marks it as unavailable and continues with the remaining Harts.

4.2 Memory Safety in Kernel Context

Kernel memory allocation must never fail silently. Allocation failure in the kernel context triggers an

immediate OOM response: either retry after reclamation or panic if no memory can be recovered.

Double-free detection uses a poison value written to freed page headers. Attempting to free a page with the poison value indicates a double-free bug. The detector reports the allocation site.

Use-after-free detection poisons the content of freed pages with a known pattern (0xDEADBEEF). Accessing the pattern in a context that expects valid data indicates use-after-free. Debug builds enable this check.

Stack overflow protection for kernel stacks uses guard pages. Each kernel stack has an unmapped guard page at its lowest address. Stack overflow into the guard page triggers a page fault.

Integer overflow checks in memory calculations prevent allocation of incorrect sizes. The Lateralus compiler inserts overflow checks for arithmetic in debug mode. Release mode uses wrapping arithmetic with explicit checks.

Null pointer dereference detection maps the zero page as inaccessible. Any access to address zero triggers a page fault. The fault handler reports the null dereference and kills the offending process.

Memory alignment enforcement ensures that all kernel data structures are properly aligned. Misaligned access on RISC-V (without the misaligned load extension) triggers an exception.

DMA buffer safety ensures that DMA-accessible memory is not simultaneously accessible by the CPU in a cacheable mode. Cache maintenance operations (clean, invalidate) bracket DMA transfers.

Memory zeroing on allocation fills newly allocated pages with zeros. Zeroing prevents information leakage from previously freed memory. The zeroing cost is approximately 50 nanoseconds per 4 KB page.

Memory ownership tracking in debug mode maintains a map from physical pages to their owning process or kernel subsystem. The map is checked on deallocation to verify the correct owner.

6.2 System Call Dispatch

System call dispatch reads the system call number from register a7 and indexes into the system call table. Invalid system call numbers return -ENOSYS.

Argument extraction reads up to six arguments from registers a0 through a5. Arguments are passed by value for scalars and by pointer for buffers. Pointer arguments require validation.

User buffer copy-in transfers data from user space to kernel space for write-type system calls. The copy function checks each page of the user buffer for valid mapping and read permission.

User buffer copy-out transfers data from kernel space to user space for read-type system calls. The copy function checks each page of the user buffer for valid mapping and write permission.

String copy-in for path-based system calls copies a null-terminated string from user space. The copy has a maximum length limit (4096 bytes) to prevent unbounded copy operations.

Error code convention: system calls return 0 or positive values on success. Negative values indicate errors. Error codes follow the POSIX convention: -ENOENT (no such file), -ENOMEM (out of memory), -EBADF (bad file descriptor).

System call restart after signal: if a system call is interrupted by a signal, it returns -EINTR. The caller can check this and restart the system call. Some system calls are automatically restarted.

System call auditing in debug mode logs every system call with: process ID, system call name, arguments, return value, and execution time. Auditing is disabled in release builds for performance.

System call batching for high-frequency operations (multiple reads, multiple writes) reduces the per-call overhead. A batch system call accepts an array of operations and executes them sequentially.

Compatibility layer for POSIX system calls maps the Lateralus OS system call numbers to POSIX-compatible numbers. The compatibility layer enables running standard C library programs.

8.2 Network Device Driver

The VirtIO network driver provides Ethernet frame send and receive. Two virtqueues are used: one for receiving frames and one for transmitting frames.

Receive queue setup pre-allocates buffer descriptors pointing to page-aligned buffers. Each buffer is large enough for a maximum-size Ethernet frame (1514 bytes plus header).

Transmit queue operation fills a descriptor with the frame data, adds it to the available ring, and notifies the device. The driver waits for the frame to appear in the used ring (interrupt or poll).

MAC address configuration reads the device's MAC address from the VirtIO configuration space. The MAC address is stored in the network interface structure and used for ARP responses.

Interrupt handling for the network device processes received frames and reclaims transmitted buffers. The handler runs in the interrupt context and wakes sleeping threads that are waiting for I/O.

ARP implementation resolves IP addresses to MAC addresses. ARP requests are broadcast on the local network. ARP replies are cached in a table with a 5-minute expiration.

IP packet handling adds IP headers to outgoing packets and strips headers from incoming packets. The IP layer handles fragmentation for packets exceeding the MTU.

UDP implementation provides connectionless datagram service. The UDP layer adds port numbers and checksum. Received datagrams are delivered to the socket that matches the destination port.

TCP implementation provides reliable byte-stream service. The TCP state machine handles connection establishment (three-way handshake), data transfer with acknowledgment, and connection teardown.

Socket interface provides user-space access to the network stack. System calls include: socket (create), bind (assign address), listen (accept connections), connect (initiate connection), send, and

recv.

9.3 File System Performance

Sequential read performance is limited by the block device throughput and buffer cache hit rate. With a warm cache, sequential reads achieve line-rate throughput of the VirtIO block device.

Sequential write performance requires block allocation and inode updates in addition to data writes. Write throughput is approximately 80% of read throughput due to metadata overhead.

Random read performance depends on block locality and cache hit rate. Random reads with a cold cache require one block device read per file system read. Cache warming improves random performance.

Directory operation performance is proportional to the directory size for linear search. A directory with 1000 entries requires scanning 500 entries on average per lookup.

File creation throughput creates approximately 5000 files per second. Each creation requires: inode allocation, inode initialization, directory entry creation, and parent inode update.

Buffer cache hit rate depends on the working set size relative to cache size. With a 16 MB buffer cache and a 10 MB working set, the hit rate exceeds 99%.

Inode cache performance stores recently accessed inodes in memory. The inode cache uses a hash table for O(1) lookup. Cache misses require a block device read.

File system metadata overhead is approximately 3% of the total disk space. The overhead includes the superblock, inode table, free block bitmap, and directory structure.

Journaling would improve crash recovery at the cost of write throughput. The current implementation uses ordered writes for consistency but does not implement a full journal.

Benchmark suite measures: sequential read/write throughput, random read/write IOPS, file creation/deletion rate, and metadata operation latency.

7.3 Inter-Process Communication

Pipes provide unidirectional byte streams between processes. A pipe consists of a kernel buffer (4 KB), a read file descriptor, and a write file descriptor. Data is copied through the kernel buffer.

Pipe read blocks when the pipe is empty (and the write end is open). Pipe write blocks when the pipe is full. A read on a pipe with a closed write end returns EOF. A write to a pipe with a closed read end generates SIGPIPE.

Shared memory allows two processes to map the same physical pages into their address spaces. The first process creates the shared region. The second process attaches to it using a key.

Signal delivery for process notification: SIGKILL (terminate), SIGSTOP (suspend), SIGCONT (resume), SIGCHLD (child status change). Signals are delivered by setting a pending bit and

checking on return to user mode.

Message passing through pipes and shared memory enables client-server communication patterns. The shell uses pipes to connect commands (ls | grep). Servers use shared memory for high-throughput data exchange.

Wait queues provide blocking synchronization for kernel events. A thread sleeps on a wait queue until the event occurs. The event producer wakes all threads on the queue.

Futex-based synchronization in user space uses atomic operations for the uncontended case and system calls for the contended case. Futexes implement mutexes, condition variables, and barriers.

File descriptor passing through Unix domain sockets allows processes to share open files. The kernel copies the file structure pointer to the receiving process's file descriptor table.

Anonymous pipe creation through the pipe system call returns two file descriptors. Pipes are commonly used after fork: the parent writes to one end, and the child reads from the other.

Poll/select system calls allow a process to wait for multiple file descriptors simultaneously. The process sleeps until any descriptor is ready for the requested operation (read, write, or error).

10.2 Porting to Physical Hardware

The SiFive HiFive Unmatched provides a physical RISC-V platform with four U74 cores and one S7 monitor core. The board has 16 GB DDR4, PCIe, Gigabit Ethernet, and USB 3.0.

Hardware differences from QEMU: the HiFive Unmatched uses a different UART (SiFive UART instead of NS16550A), different interrupt controller layout, and real memory timing.

Boot process on hardware uses the board's bootloader chain: ZSBL (Zero Stage Boot Loader) -> FSBL -> OpenSBI -> our kernel. The kernel receives control in S-mode.

Device tree differences: the hardware device tree is provided by the firmware and differs from QEMU's device tree in node names, compatible strings, and property values.

Timing calibration: hardware timers run at a different frequency than QEMU. The kernel reads the timebase-frequency property from the device tree to calibrate timer intervals.

Cache behavior on hardware differs from QEMU (which does not simulate caches). Cache coherence, cache flushing, and cache line size affect driver behavior and performance.

Ethernet driver for the HiFive Unmatched uses the Cadence MACB controller instead of VirtIO. A platform-specific driver is loaded based on the device tree's compatible string.

Serial console on hardware connects through the FTDI USB-to-serial chip. The SiFive UART driver replaces the NS16550A driver. Console output is visible through a terminal emulator at 115200 baud.

Performance on hardware: context switch time is 1.2 microseconds (vs 0.5 microseconds on QEMU). System call latency is 0.8 microseconds. The difference is due to real memory latency and cache

effects.

Debugging on hardware uses JTAG through the board's debug header. OpenOCD connects to the JTAG interface and provides a GDB server. The same GDB scripts work for both QEMU and hardware.

2.2 RISC-V Extension Detection

ISA extension detection reads the isa property from the device tree's CPU nodes. The property is a string like 'rv64imafdc' that lists supported extensions.

Runtime extension probing tests for optional extensions by executing extension-specific instructions in a trap handler. If the instruction traps, the extension is not supported.

Atomic extension (A) provides atomic memory operations: LR (Load Reserved) and SC (Store Conditional) for lock-free data structures. The kernel uses atomics for spinlocks and reference counts.

Floating-point extensions (F, D) provide single and double precision floating-point operations. The kernel saves and restores FP registers on context switch for processes that use floating-point.

Compressed extension (C) provides 16-bit compressed instructions. Compressed instructions reduce code size by approximately 25%. The extension is transparent to the kernel.

Vector extension (V) provides SIMD operations on variable-length vectors. Kernel support includes saving and restoring vector registers and configuring the vector length (VLEN).

Bit manipulation extension (B) provides bit counting, bit permutation, and bit extraction instructions. The kernel can use these for faster bitmap operations in the page allocator.

Crypto extension (K) provides AES, SHA-256, and SM3/SM4 acceleration. The kernel uses crypto acceleration for file system encryption and network TLS.

Hypervisor extension (H) provides hardware support for virtualization. The extension adds a new privilege level (VS-mode) and guest physical address translation.

Extension capability advertisement exposes detected extensions to user-space through the auxiliary vector (AT_HWCAP). Applications can check for extensions and use optimized code paths.

4.3 DMA and I/O Memory

DMA-capable memory allocation provides physically contiguous, cache-coherent memory for device I/O. The allocator reserves a region of physical memory for DMA during boot.

I/O memory mapping maps device registers into the kernel's virtual address space with uncacheable attributes. The mapping function takes a physical address range and returns a virtual address.

MMIO read and write operations use volatile access to prevent the compiler from optimizing away register accesses. Read and write functions provide 8, 16, 32, and 64-bit access widths.

DMA address translation converts kernel virtual addresses to physical addresses for programming device DMA controllers. The translation uses the kernel's fixed-offset mapping.

Scatter-gather DMA support builds descriptor lists from non-contiguous virtual pages. Each descriptor specifies a physical address and length. The device processes descriptors sequentially.

DMA fence operations ensure that CPU writes to DMA buffers are visible to the device (and vice versa). RISC-V fence instructions order memory operations across the CPU-device boundary.

IOMMU support (where available) translates device DMA addresses through a device page table. The IOMMU prevents devices from accessing memory outside their permitted regions.

DMA pool management pre-allocates a pool of DMA buffers during device initialization. Buffer allocation from the pool is O(1) and does not require page table modification.

Cache maintenance for non-coherent DMA platforms cleans (write-back) cache lines before DMA reads from memory and invalidates cache lines before CPU reads after DMA writes.

I/O memory barrier (fence io,io) orders MMIO accesses. Without barriers, the processor may reorder MMIO reads and writes, causing incorrect device behavior.

5.3 Kernel Virtual Memory Layout

The kernel occupies the upper portion of the 39-bit virtual address space (0xFFFFFFC000000000 to 0xFFFFFFFFFFFFFFFF). This 256 GB region is divided into sub-regions for different purposes.

Direct mapping region maps all physical memory at a fixed offset. The kernel accesses physical memory through this mapping. The region starts at 0xFFFFFFC000000000 and extends for the amount of physical memory.

Kernel code and data occupy a fixed location within the direct mapping. The kernel's virtual address equals its physical address plus the mapping offset.

MMIO region maps device memory-mapped I/O registers. Each device is mapped at a fixed virtual address derived from its physical address. MMIO mappings use uncacheable page table attributes.

Kernel heap uses a slab allocator for small objects and the buddy allocator for large objects. The heap grows as needed within the kernel's virtual address space.

Per-CPU region provides thread-local storage for each Hart. The region contains per-Hart data structures accessed through the tp register. Each Hart's region is at a fixed offset.

Module loading region (for dynamically loaded drivers) provides virtual address space for code and data of loaded modules. Modules are mapped with execute permission for code and read-write for data.

Temporary mapping region provides short-lived mappings for operations like page table manipulation and user buffer copying. Temporary mappings are per-CPU to avoid locking.

Guard regions of unmapped pages separate kernel regions. Access to a guard page triggers a fault, detecting buffer overflows between regions.

Virtual address space usage monitoring tracks the allocation of virtual address ranges within the kernel. Monitoring detects virtual address exhaustion and fragmentation.

6.3 Interrupt Controller Programming

PLIC initialization sets priorities for all interrupt sources to a default value (1). Priority 0 disables an interrupt source. Higher priorities are served first when multiple interrupts are pending.

PLIC enable registers are per-Hart: each Hart has a bitmap of enabled interrupt sources. The kernel enables interrupts for the devices that the Hart is responsible for handling.

Interrupt claim reads the PLIC claim register, which returns the ID of the highest-priority pending interrupt and atomically marks it as being serviced. If no interrupt is pending, the claim returns 0.

Interrupt completion writes the interrupt ID to the PLIC complete register, indicating that the handler has finished processing. This allows the PLIC to deliver the next pending interrupt.

CLINT timer programming writes the desired interrupt time to the mtimecmp register. The timer interrupt fires when mtime >= mtimecmp. Programming the next interval requires reading mtime and adding the desired delay.

CLINT software interrupt sets a bit in the MSIP register for the target Hart. Software interrupts are used for inter-Hart communication. The interrupt is cleared by writing 0 to the MSIP register.

Interrupt affinity routes specific device interrupts to specific Harts. The routing is configured by enabling the interrupt in one Hart's PLIC enable register and disabling it in others.

Interrupt coalescing combines multiple device interrupts into a single handler invocation. The handler processes all pending events in a loop, reducing interrupt overhead.

Interrupt latency measurement: from hardware interrupt assertion to handler entry is approximately 200 clock cycles (100 nanoseconds at 2 GHz). The latency includes trap entry and register save.

Spurious interrupt handling: if the PLIC claim returns 0, no interrupt is pending. Spurious interrupts occur when the interrupt source deasserts before the claim. The handler simply returns.

10.3 Lessons Learned

Start with the simplest possible setup: single Hart, no virtual memory, UART-only I/O. Add complexity incrementally. Each addition should be tested independently before combining.

QEMU is essential for rapid development. Compile-run-debug cycles take seconds with QEMU versus minutes with hardware. QEMU's GDB integration provides better debugging than most hardware JTAG.

The device tree is the single source of hardware truth. Hard-coding device addresses causes

breakage when moving between QEMU versions or hardware boards. Always parse the device tree.

Memory ordering bugs are the hardest to find. Use fence instructions liberally during development and remove unnecessary ones after testing. Missing fences cause subtle data corruption.

Page table bugs manifest as mysterious crashes. A page table dump utility that prints all mappings is invaluable. Write it early and use it often.

Race conditions appear when adding the second Hart. A kernel that works perfectly on one Hart may crash immediately on two. Test with multiple Harts early in development.

The ownership system caught several pointer bugs during development. Double-free and use-after-free bugs that would have been subtle in C were compile-time errors in Lateralus.

VirtIO provides excellent device abstractions for OS development. The specification is well-documented, QEMU's implementation is reliable, and the same drivers work on real VirtIO hardware.

Code size: the complete kernel is approximately 8,000 lines of Lateralus and 200 lines of assembly. The small code size demonstrates RISC-V's simplicity and Lateralus's expressiveness.

The project took approximately six months of part-time work. The most time-consuming components were virtual memory (6 weeks), the file system (4 weeks), and the network stack (4 weeks).

8.3 Shell Implementation

The shell is the first user-space program launched by the init process. The shell reads commands from the UART, parses them, and executes them by forking and calling exec.

Command parsing splits the input line into tokens separated by whitespace. Pipe characters (|) create pipeline connections between commands. Angle brackets (< >) create I/O redirections.

Built-in commands (cd, pwd, exit, echo) are executed directly by the shell without forking. External commands are executed by forking a child process and calling exec with the command path.

Pipeline execution for 'cmd1 | cmd2' creates a pipe, forks twice, connects the pipe to stdout of cmd1 and stdin of cmd2, and waits for both to complete.

I/O redirection opens the specified file and duplicates it to the standard file descriptor. Input redirection (< file) opens the file read-only. Output redirection (> file) creates or truncates the file.

Background execution (command &) forks the child but does not wait for it. The parent shell immediately prints a new prompt. The child runs independently.

Environment variable support stores variables as name=value pairs. The export command adds variables to the environment passed to child processes via exec.

Command history stores the last 100 commands. The up and down arrow keys navigate history. History is stored in memory and lost when the shell exits.

Tab completion matches partial command names against executables in the path directories. When a unique match is found, the completion is inserted automatically.

Signal handling in the shell: Ctrl-C sends SIGINT to the foreground process group. Ctrl-Z sends SIGSTOP. The shell catches these signals for the foreground child but does not terminate itself.

References

[1] Patterson, D. and Waterman, A. The RISC-V Reader. Strawberry Canyon, 2017.
[2] Waterman, A. and Asanovic, K. The RISC-V ISA Manual. 2021.
[3] Tanenbaum, A. and Bos, H. Modern Operating Systems. 4th Ed., 2014.
[4] Boehmova, K. et al. xv6: A Simple Unix-like Teaching OS. MIT, 2006.
[5] Chen, S. Writing a RISC-V OS from Scratch. 2021.