FRISC-OS Architecture: Design and Implementation

bad-antics | March 2024 | Technical Report

Abstract

This paper provides a detailed examination of the FRISC-OS architecture, an educational and research operating system for the RISC-V platform written in Lateralus. The architecture covers the hardware abstraction layer, memory management, trap handling, scheduling, IPC, security, and device drivers.

1 Introduction

FRISC-OS is a research and educational operating system designed around the RISC-V instruction set architecture. This paper provides a detailed examination of the FRISC-OS architecture, covering the hardware abstraction layer, kernel subsystems, memory hierarchy, scheduling framework, inter-process communication, and the security model. The architecture balances simplicity for educational use with sufficient sophistication to demonstrate real operating system design principles.

The architecture is implemented in Lateralus, leveraging the language's ownership model and pipeline-native design to prevent common kernel bugs while maintaining the low-level control necessary for operating system development. Each architectural layer is designed as a distinct module with well-defined interfaces.

This paper serves as the architectural reference for FRISC-OS, complementing the educational paper that focuses on pedagogy and laboratory exercises. Readers should have familiarity with operating system concepts and the RISC-V privilege architecture.

2 Hardware Abstraction Layer

The Hardware Abstraction Layer (HAL) provides a platform-independent interface above the RISC-V hardware. The HAL defines traits for CPU management, memory-mapped I/O, interrupt control, and timer programming. Platform-specific implementations of these traits are selected at compile time based on the target board.

The HAL currently supports three platforms: QEMU virt (primary development target), SiFive HiFive Unmatched (physical RISC-V board), and Kendryte K210 (low-cost dual-core RISC-V). Each platform provides its own device tree and peripheral drivers while sharing the common kernel code above the HAL.

```
// Hardware abstraction trait
pub trait Platform {
    fn init_console(&self);
    fn init_interrupts(&self) -> &dyn InterruptController;
    fn init_timer(&self) -> &dyn Timer;
    fn memory_regions(&self) -> &[MemoryRegion];
```

```
    fn device_tree(&self) -> &DeviceTree;
}

// Platform selection at compile time
#[cfg(platform = "qemu-virt")]
type CurrentPlatform = QemuVirtPlatform;
```

Device tree parsing extracts hardware configuration from the FDT (Flattened Device Tree) blob passed by the bootloader. The parser identifies memory regions, interrupt controllers, serial ports, and storage devices. Discovered devices are registered in the kernel's device registry for driver matching.

The HAL provides safe wrappers around RISC-V CSR (Control and Status Register) operations. Each CSR is represented as a type with methods for reading, writing, and modifying specific bit fields. The type system prevents accidental access to undefined CSR fields and ensures correct bit manipulation.

3 Boot Sequence

The boot sequence begins in machine mode with OpenSBI firmware initializing the hardware and presenting the SBI interface. OpenSBI configures the Physical Memory Protection (PMP) units, initializes the platform interrupt controller, and drops to supervisor mode where FRISC-OS takes control.

The kernel entry point in assembly configures the stack pointer, clears the BSS section, and calls the Lateralus kernel_main function. On multiprocessor systems, only the boot hart (hardware thread) executes the initialization sequence. Other harts spin in a parking loop until released by the boot hart.

Kernel initialization proceeds through a series of subsystem init calls in dependency order: console, trap vectors, physical memory allocator, virtual memory, process subsystem, device drivers, filesystem, and system call table. Each subsystem logs its initialization status to the console.

The final initialization step creates the first user process by loading the init program from the root filesystem. Init is assigned PID 1 and runs in user mode. The kernel then enters the idle loop, which runs when no other process is ready and halts the CPU with the wfi instruction until an interrupt occurs.

4 Memory Architecture

FRISC-OS implements a three-tier memory management architecture: physical page allocator, kernel virtual memory manager, and per-process address space manager. Each tier has distinct responsibilities and interfaces, allowing independent evolution and testing.

The physical page allocator manages a free list of 4 KB page frames. Allocation and deallocation are O(1) operations using a linked list threaded through the free pages themselves. The allocator is

initialized from the device tree's memory reservation map, excluding regions used by firmware and MMIO.

Kernel virtual memory uses a static mapping established during boot. Physical memory is identity-mapped in the kernel address space, and MMIO regions are mapped at fixed virtual addresses. The static mapping simplifies kernel memory access and eliminates the need for dynamic kernel page table modifications.

```
// Memory tier interfaces
pub trait PhysAllocator {
    fn alloc_page() -> Result<PhysPage, OutOfMemory>;
    fn free_page(page: PhysPage);
    fn available_pages() -> usize;
}

pub trait VirtMemManager {
    fn map_range(root: &mut PageTable, va: VirtAddr,
                 pa: PhysAddr, len: usize, perm: Perm);
    fn unmap_range(root: &mut PageTable, va: VirtAddr, len: usize);
}
```

Per-process address spaces are managed by the VirtMemManager, which creates, modifies, and destroys page table hierarchies. Each process has an independent page table root stored in its process control block. Address space operations include mapping, unmapping, permission changes, and fork-time duplication.

The Translation Lookaside Buffer (TLB) is managed explicitly by the kernel. After page table modifications, the kernel issues sfence.vma instructions to invalidate affected TLB entries. On multiprocessor systems, TLB shootdowns use inter-processor interrupts to ensure consistency across all harts.

5 Trap and Exception Architecture

RISC-V's trap architecture routes all exceptions, interrupts, and system calls through a single entry point specified by the stvec CSR. FRISC-OS uses vectored mode where the base address is stvec and the vector offset is derived from the exception code.

The trap entry code is written in assembly and saves the complete register state to a per-process trap frame. The trap frame is allocated on the kernel stack and passed as a pointer to the Lateralus trap dispatcher. Saving and restoring all 31 general-purpose registers ensures correct resume regardless of the trap type.

Exception dispatch examines the scause register to determine the trap type. Synchronous exceptions (page faults, illegal instructions, ecalls) are dispatched to dedicated handler functions. Asynchronous interrupts (timer, software, external) are dispatched to the interrupt subsystem. Unknown trap causes trigger a kernel panic with diagnostic output.

Page fault handling is the most complex exception path. The handler reads the faulting address from stval, locates the corresponding VMA (virtual memory area), and determines the appropriate action: demand-page allocation, copy-on-write duplication, or process termination for invalid accesses.

System call handling extracts the call number from a7 and arguments from a0-a5. The dispatcher indexes into the system call table and invokes the handler with the argument registers. The return value is stored in the trap frame's a0 slot for delivery to user space on trap return.

6 Scheduling Framework

The scheduler implements a pluggable framework that supports multiple scheduling policies. The default policy is round-robin with a 10 ms time quantum, which provides fair CPU distribution and good interactive response. Alternative policies include priority-based scheduling and a simplified CFS.

The run queue is maintained per-CPU on multiprocessor systems. Each CPU selects the next process from its local run queue, avoiding global lock contention. Load balancing periodically migrates processes between CPUs to maintain equitable distribution.

Context switching is performed by the switch function, which saves callee-saved registers (s0-s11, ra, sp) to the current process's context structure and restores them from the next process's context. The page table root (satp) is switched before the register restore to ensure the new process's memory is accessible.

```
// Context switch implementation
@naked fn switch_context(old: *mut Context, new: *Context) {
    asm!(
        "sd ra, 0(a0)\n sd sp, 8(a0)\n sd s0, 16(a0)",
        "sd s1, 24(a0)\n sd s2, 32(a0)\n sd s3, 40(a0)",
        // ... save remaining s-registers ...
        "ld ra, 0(a1)\n ld sp, 8(a1)\n ld s0, 16(a1)",
        "ld s1, 24(a1)\n ld s2, 32(a1)\n ld s3, 40(a1)",
        // ... restore remaining s-registers ...
        "ret"
    );
}
```

Idle processes run on each CPU when no user process is ready. The idle process executes the wfi (wait for interrupt) instruction, halting the CPU core until an interrupt arrives. This reduces power consumption and heat generation on physical hardware.

Timer interrupts trigger scheduler ticks that decrement the running process's remaining quantum. When the quantum expires, the process is moved to the back of the run queue and the scheduler selects the next process. Voluntary yields (via sleep or I/O wait) also invoke the scheduler.

7 Inter-Process Communication

FRISC-OS provides three IPC mechanisms: pipes, shared memory, and signals. Each mechanism serves different communication patterns: pipes for byte streams, shared memory for high-bandwidth data sharing, and signals for asynchronous notifications.

Pipes implement unidirectional byte streams between processes. The pipe buffer is a single kernel page (4 KB) managed as a circular buffer. Read and write operations block when the buffer is empty or full, respectively. Pipe closure is detected by reference counting on the file descriptors.

Shared memory allows processes to map the same physical pages into their address spaces. The shmget, shmat, and shmdt system calls create, attach, and detach shared memory segments. Access permissions are checked on attachment and enforced through page table permissions.

Signals provide asynchronous notification using a simplified POSIX signal model. The kernel supports signals for process termination (SIGKILL, SIGTERM), child status change (SIGCHLD), and user-defined communication (SIGUSR1, SIGUSR2). Signal handlers execute on a separate signal stack to prevent corruption of the interrupted stack.

8 Security Model

FRISC-OS implements a simplified Unix-like security model with user IDs, group IDs, and file permissions. The model demonstrates fundamental access control concepts without the complexity of capabilities or mandatory access control.

Process credentials include a user ID and a group ID inherited from the parent process. The setuid and setgid system calls change credentials when executed by privileged processes. File ownership and permission bits (read, write, execute for owner, group, and others) control access to filesystem objects.

Kernel-user isolation relies on RISC-V's supervisor and user privilege modes. User processes cannot execute privileged instructions, access kernel memory, or modify page tables. Attempts to violate these restrictions trigger exceptions that the kernel handles by terminating the offending process.

Address space isolation ensures that one process cannot access another process's memory. Each process has an independent page table hierarchy, and the kernel verifies that system call pointer arguments reference only the calling process's mapped regions.

9 Device Driver Framework

The device driver framework provides a registration and dispatch mechanism for hardware drivers. Drivers register with the framework by providing a probe function that examines device tree nodes and claims compatible devices. The framework calls probe for each discovered device until a driver claims it.

Interrupt-driven drivers register handlers with the interrupt subsystem. When a device interrupt fires, the PLIC handler identifies the interrupt source and dispatches to the registered handler. Handlers

run in interrupt context with limited kernel services available.

Block device drivers implement the BlockDevice trait with read_block and write_block methods. The filesystem layer accesses storage exclusively through this trait, allowing different storage technologies (virtio, SD card, NOR flash) to be used interchangeably.

Character device drivers implement the CharDevice trait with read, write, and ioctl methods. The console, serial ports, and input devices use this interface. Character device operations may block the calling process until data is available.

10 Conclusion

The FRISC-OS architecture provides a complete, well-structured operating system design that balances educational clarity with architectural soundness. The modular design with clean interfaces enables students and researchers to understand, modify, and extend individual subsystems without affecting the rest of the kernel.

4.1 Page Table Management

The Sv39 page table uses three levels of 512-entry tables, each entry being 8 bytes. A full page table hierarchy requires at most 1 + 512 + 262144 pages, but demand allocation ensures only populated regions consume page table memory. Most processes use fewer than 10 page table pages.

Page table entries encode physical page numbers, permission bits (read, write, execute), and status bits (valid, accessed, dirty). The kernel uses Lateralus bitfield types to manipulate these fields safely, preventing accidental corruption of reserved bits.

Megapage (2 MB) and gigapage (1 GB) mappings are used for the kernel's identity map of physical memory. Large pages reduce TLB pressure by covering more address space per TLB entry. The kernel's physical memory mapping uses gigapages when the physical memory size is a multiple of 1 GB.

Page table deallocation during process exit recursively walks the page table hierarchy, freeing leaf pages first and then intermediate page table pages. The deallocation must avoid freeing shared pages (from copy-on-write) by checking reference counts before releasing physical frames.

The page table walker is a reusable function that traverses the hierarchy and returns the leaf entry for a given virtual address. The walker supports both lookup (read-only traversal) and allocation (creating intermediate entries if absent). This dual-mode design reduces code duplication.

Address space duplication for fork creates a new page table hierarchy that shares all leaf pages with the parent through copy-on-write. Both parent and child page tables mark shared pages as read-only. The first write triggers a fault that creates a private copy of the modified page.

Kernel page table entries are marked with the Global bit, which tells the TLB to retain these entries across address space switches. Global entries avoid unnecessary TLB misses for kernel code and

data that are mapped identically in every process.

Page table isolation between kernel and user address spaces prevents speculative execution attacks. User-mode page table entries do not include kernel mappings, requiring an explicit page table switch on every trap entry and return. This isolation adds overhead but provides hardware-enforced kernel memory protection.

The kernel maintains a free list of page table pages to accelerate page table allocation during fork and mmap operations. Pre-allocated page table pages avoid blocking on the physical allocator during performance-sensitive operations.

Page table debugging tools dump the entire page table hierarchy in human-readable format, showing virtual-to-physical mappings, permissions, and page sizes. The dump tool is invaluable for debugging virtual memory issues and verifying correct page table construction.

6.1 Multiprocessor Scheduling

Symmetric multiprocessing support allows FRISC-OS to run on systems with up to 8 harts. Each hart runs an independent scheduler instance with its own run queue, idle process, and timer configuration. The boot hart initializes shared data structures before releasing secondary harts from their parking loops.

Hart startup uses SBI calls to wake parked harts and direct them to a secondary entry point. The secondary entry point initializes per-hart data structures including the trap vector, kernel stack, and timer. Once initialized, the secondary hart enters its scheduler loop.

Per-hart run queues eliminate contention for the scheduler lock in the common case. Processes are assigned to the run queue of the hart where they were created or last ran. This CPU affinity improves cache utilization by keeping processes on the same hart when possible.

Load balancing runs periodically on each hart, comparing its run queue length against the average across all harts. If a hart is significantly overloaded, it migrates processes to the least-loaded hart. Migration transfers the process descriptor and updates the affinity hint.

Spinlock implementation uses RISC-V atomic instructions (amoswap) with a test-and-set pattern. Spinlocks protect shared kernel data structures that are accessed from multiple harts. Each spinlock records the holding hart and acquisition site for debugging purposes.

Inter-processor interrupts use SBI sbi_send_ipi to signal specific harts. IPIs are used for TLB shootdowns, scheduler wake-ups, and kernel debugging. The IPI handler examines a per-hart bitmap to determine the requested action and dispatches accordingly.

Memory ordering on RISC-V uses the FENCE instruction for hardware memory barriers. The kernel inserts fences at critical points in synchronization primitives to ensure that memory operations are visible across harts in the correct order. The weak memory model requires explicit barriers.

Per-CPU data structures are allocated in a per-hart array indexed by the hart ID. The hart ID is read

from the tp (thread pointer) register, which is initialized during boot and maintained across context switches. Per-hart data avoids false sharing and contention.

Hart hot-plug support allows harts to be brought online and offline at runtime. Offlining a hart migrates all its processes to other harts, flushes its caches, and parks it in a WFI loop. Hot-plug is primarily useful for power management on physical hardware.

Debugging multiprocessor issues uses per-hart log buffers that are merged post-mortem. Each hart writes log entries to its own buffer without synchronization, eliminating the Heisenbug-inducing serialization that shared logging would require. The merged log provides a global view of events.

7.1 Advanced IPC Mechanisms

Message queues provide typed message passing between processes. Each queue has a configurable maximum message count and message size. Send operations block when the queue is full, and receive operations block when empty. Priority ordering delivers higher-priority messages first.

Unix domain sockets implement connection-oriented and connectionless communication within the same machine. Stream sockets provide reliable byte streams similar to TCP, while datagram sockets provide message-preserving unreliable delivery. Domain sockets are significantly faster than network sockets.

Event notification uses an eventfd mechanism that provides a file-descriptor-based signaling interface. Processes write to the event descriptor to signal and read from it to wait. Event descriptors integrate with the poll system call for multiplexed waiting.

The poll system call monitors multiple file descriptors for readiness. The kernel checks each descriptor against its readiness criteria and returns the set of ready descriptors. If no descriptor is ready, the process blocks until an event occurs or a timeout expires.

File locking provides advisory locking for coordinating file access between processes. Lock operations specify byte ranges and lock types (shared or exclusive). The locking implementation uses a per-inode lock list with deadlock detection for exclusive locks.

Memory-mapped files allow processes to access file contents through virtual memory operations instead of read/write system calls. Modifications to the mapped region are written back to the file when the mapping is flushed or unmapped. Shared mappings enable efficient inter-process data exchange.

The select system call provides an alternative to poll for monitoring file descriptor readiness using bitmap-based descriptor sets. Select is included for compatibility with traditional Unix programming patterns, though poll is preferred for its cleaner interface.

Process tracing via ptrace allows a parent process to observe and control the execution of a child process. Ptrace supports single-stepping, register inspection, memory reading, and breakpoint insertion. The debugger implementation in FRISC-OS uses ptrace for all debugging operations.

Named pipes (FIFOs) extend the pipe mechanism to the filesystem namespace. Any process can open a named pipe by path, enabling communication between unrelated processes. Named pipes have the same semantics as anonymous pipes but persist in the filesystem across process lifetimes.

Signal queuing extends the basic signal mechanism with queued delivery of real-time signals. Unlike standard signals which are merged when pending, real-time signals are queued and delivered in order. Each queued signal carries an integer or pointer payload for lightweight message passing.

8.1 Filesystem Security

File permissions use the traditional Unix rwx model with three permission classes: owner, group, and others. Each class independently grants read, write, and execute permissions. The kernel checks permissions on every file operation using the process's effective UID and GID.

The setuid bit on executable files temporarily elevates the process's effective UID to the file owner's UID during execution. This mechanism allows ordinary users to perform privileged operations through trusted programs. FRISC-OS implements setuid with careful attention to the security implications.

Directory permissions control the ability to list contents (read), create or delete entries (write), and traverse (execute). The sticky bit on directories prevents users from deleting other users' files, even with directory write permission. The /tmp directory uses the sticky bit.

Root user bypass allows UID 0 to bypass permission checks for administrative operations. The root bypass is intentionally simple, providing a clear demonstration of privilege escalation risks. Students learn why modern systems prefer capability-based access control.

Chroot changes a process's view of the filesystem root, providing basic filesystem isolation. The chroot system call is restricted to root and demonstrates the concept of filesystem namespaces. Students learn both the utility and the limitations of chroot-based isolation.

Symbolic link resolution follows a configurable recursion limit to prevent infinite loops from circular links. The kernel counts link resolution depth and returns ELOOP when the limit is exceeded. This defense prevents denial-of-service through malicious symbolic link chains.

File creation inherits the parent directory's group ID when the setgid bit is set on the directory. This behavior supports shared directories where all files should belong to a common group regardless of the creating user's primary group.

Inode flags provide additional file attributes including immutable (cannot be modified or deleted), append-only (can only be appended to), and no-dump (excluded from backups). These flags are stored in the inode and checked by the kernel during file operations.

Access time updates are batched to reduce write I/O. The atime field is updated in memory on each access but written to disk only when the inode is flushed for other reasons. This optimization avoids the performance penalty of updating disk metadata on every file read.

Filesystem quotas limit per-user and per-group disk usage. The quota system tracks block and inode counts against configurable limits. When a hard limit is reached, allocation fails. Soft limits allow temporary overuse within a grace period.

9.1 Driver Development Guide

New drivers are implemented by creating a module that implements the appropriate device trait (BlockDevice, CharDevice, or NetDevice). The module's init function registers the driver with the device framework by providing a compatibility string and a probe function.

The probe function receives a device tree node and determines whether the driver supports the described hardware. If compatible, probe initializes the device hardware, registers interrupt handlers, and creates device file entries. Probe returns a result indicating success or incompatibility.

Memory-mapped I/O access uses volatile read and write operations through the HAL's MMIO wrapper. The wrapper ensures that the compiler does not optimize away or reorder device register accesses. Each register is typed with its expected layout for safe field access.

DMA buffer management allocates physically contiguous memory for device data transfers. The DMA allocator returns both the virtual address (for kernel access) and the physical address (for device programming). DMA buffers are allocated from a reserved memory pool configured during boot.

Interrupt handler registration associates a handler function with a PLIC interrupt source number. The handler is called in interrupt context with interrupts disabled on the current hart. Handlers must be brief, deferring complex processing to a kernel thread.

Power management callbacks notify drivers of system power state transitions. Drivers implement suspend and resume methods that save and restore device state. The power manager calls these methods in dependency order, ensuring devices are suspended after their dependents and resumed before.

Hot-plug notification informs drivers when devices are connected or disconnected at runtime. The notification includes the device tree node for the new device. Drivers that support hot-plug implement connect and disconnect callbacks for dynamic device management.

Driver testing uses mock hardware implementations that simulate device behavior in software. The mock implementations respond to register reads and writes with programmed values, allowing driver code to be tested without physical hardware or QEMU.

Error recovery procedures handle device malfunctions including command timeouts, DMA errors, and unexpected device state. The recovery sequence resets the device to a known state and retries the failed operation. Persistent failures are reported to the user and the device is marked as offline.

Driver documentation follows a template that includes the supported hardware, register map, initialization sequence, normal operation, error handling, and known limitations. The documentation template ensures consistent quality and completeness across all drivers.

3.1 Early Boot Memory Setup

Before the memory allocator is available, the kernel uses a simple bump allocator for early boot allocations. The bump allocator advances a pointer through a statically allocated boot memory region. Early allocations are permanent and never freed.

The boot memory region is sized to accommodate initial page tables, the kernel stack, the device tree copy, and the interrupt descriptor structures. The region size is calculated at compile time based on the maximum supported memory and CPU count.

Device tree relocation copies the FDT blob from its firmware-provided location to kernel memory. The relocation ensures the device tree remains accessible after the kernel remaps memory. The copied device tree is parsed to discover hardware configuration.

Initial page table construction creates the kernel's identity map and higher-half mapping using the boot allocator. The page table is built before enabling virtual memory, so all addresses in the construction code are physical. After enabling the MMU via the satp CSR, execution continues at the virtual address.

Stack setup assigns each hart a kernel stack allocated from the boot memory region. Stack addresses are calculated from the hart ID and stack size to ensure non-overlapping regions. Guard gaps between stacks provide overflow detection.

BSS clearing zeroes the uninitialized data section before any Lateralus code accesses global variables. The clearing is performed by the assembly boot stub using the linker-provided section boundaries. Clearing BSS ensures deterministic initial values for all static variables.

Console initialization configures the UART for output before any other subsystem. Early console output uses direct register writes without interrupts or buffering. This minimal console provides diagnostic output during the critical early boot phase where most initialization failures occur.

Firmware interface discovery identifies the SBI version and available extensions. The kernel queries SBI capabilities during boot and adjusts its behavior based on available functionality. Older SBI versions may lack features like hardware-assisted timer programming.

The memory map from the device tree identifies usable RAM regions, reserved regions, and MMIO regions. The kernel parses this information to initialize the physical page allocator with the correct set of free pages. Overlapping or invalid regions are detected and reported.

Multiprocessor rendezvous synchronizes hart startup using a shared flag variable. Secondary harts spin on the flag until the boot hart completes initialization and sets it. A memory fence ensures the flag write is visible to all harts before they proceed.

5.1 Trap Frame Design Decisions

The trap frame stores all register state needed to resume a trapped process. On RISC-V, this includes 31 general-purpose registers (x1-x31), the supervisor status register (sstatus), the exception program counter (sepc), and kernel metadata fields for stack and page table switching.

The trap frame is stored at a known offset from the process's kernel stack base. This placement allows the trap entry code to locate the frame without any general-purpose register being available for pointer computation. The sscratch CSR provides the only register-free reference point.

The sscratch register swapping technique exchanges the user stack pointer with a pointer to the trap frame. On trap entry, csrrw swaps sp and sscratch atomically, giving the trap handler a pointer to the trap frame while preserving the user sp for later saving. This technique avoids clobbering any user register.

Floating-point register saving is deferred using lazy context switching. The kernel sets the FS field in sstatus to off for each new time slice. The first floating-point instruction triggers an illegal instruction trap, which saves the previous process's FP registers and enables FP for the current process.

The trap frame layout is carefully ordered to match the register save/restore sequence in the assembly trap handler. Registers are stored in numerical order (x1 through x31) to allow the use of indexed load/store instructions with a simple stride calculation.

Kernel trap frames differ from user trap frames in that they do not require page table or stack switching. When a trap occurs while already in supervisor mode, the kernel reuses the current stack and page table. This distinction simplifies the kernel trap path and reduces overhead.

Trap frame debugging includes integrity checks that verify the frame's canary values on trap return. Corrupted trap frames indicate stack overflow or memory corruption and trigger an immediate panic with diagnostic output showing the corrupted fields.

The trap return sequence restores all registers from the trap frame in reverse order of saving. The final instruction, sret, atomically restores the privilege mode and program counter, completing the return to user space. The sret instruction also restores the interrupt enable status from sstatus.

Nested traps within the kernel use the same trap frame mechanism but with the kernel's stack and page table. Interrupt handlers that trigger page faults or other exceptions are handled through nested trap frames. A maximum nesting depth prevents unbounded stack growth.

Performance optimization of the trap path minimizes the number of instructions between trap entry and handler dispatch. Each instruction in the critical path adds latency to every system call, interrupt, and exception. The FRISC-OS trap path uses approximately 40 instructions for entry and 35 for return.

2.1 Device Tree Integration

The Flattened Device Tree (FDT) is a binary data structure that describes the hardware platform. The bootloader passes the FDT base address to the kernel, which parses it to discover available memory, interrupt controllers, serial ports, timers, and storage devices.

FDT parsing extracts structured data from the binary format, building an in-memory device tree representation. Each node in the tree corresponds to a hardware device or bus, with properties describing the device's characteristics including compatible strings, register addresses, and interrupt

assignments.

Compatible string matching connects device tree nodes to kernel drivers. Each node's compatible property lists one or more strings identifying the hardware. Drivers register the compatible strings they support, and the kernel's device probe mechanism matches nodes to drivers based on these strings.

Memory reservation entries in the FDT mark regions that the kernel must not use. Reservations typically cover the FDT itself, firmware data structures, and hardware-specific memory ranges. The physical page allocator excludes reserved regions when initializing the free page list.

Interrupt tree parsing follows the interrupt-parent property chain to build a complete picture of the interrupt routing topology. Each device node specifies its interrupt number relative to its parent controller. The kernel resolves these relative numbers to global interrupt IDs for handler registration.

Clock and frequency information from the device tree configures timer programming and baud rate calculation. The timebase-frequency property provides the timer tick rate used to program deadline-based timer interrupts at the desired scheduling frequency.

Bus address translation uses the ranges property to convert device addresses to CPU-visible addresses. Different buses (PCIe, platform) may use different address spaces that need translation before the kernel can map device registers into its virtual address space.

The device tree also describes CPU topology including the number of harts, their ISA extensions, and cache hierarchy. This information configures the multiprocessor startup sequence and enables topology-aware scheduling and memory allocation.

Runtime device tree modification allows the kernel to update device status as drivers claim devices. Nodes for successfully probed devices are marked as active. Failed or unclaimed devices are marked with their status, providing diagnostic information through a virtual filesystem.

Device tree overlay support enables board-specific customization without modifying the base device tree. Overlays add, modify, or delete nodes and properties. This mechanism supports hardware accessories and expansion boards that change the platform's device configuration.

10.1 Performance Characteristics

System call latency is approximately 1.2 microseconds on QEMU, measured from the ecall instruction to handler entry. This latency includes the trap entry assembly, register saving, and system call dispatch. The measured latency validates the efficiency of the trap handling design.

Context switch time is approximately 2.5 microseconds including the page table switch and TLB flush. The bulk of this time is spent saving and restoring registers (1.0 us) and performing the TLB flush (1.2 us). The sfence.vma instruction dominates the switch cost.

Page fault handling takes approximately 5 microseconds for a demand-page allocation that requires allocating a physical frame, zeroing it, and installing the page table entry. Copy-on-write faults take

approximately 7 microseconds due to the additional page copy.

Pipe throughput achieves approximately 200 MB/s for large messages on QEMU. The throughput is limited by the single-page pipe buffer and the overhead of system calls for each buffer-full transfer. Larger pipe buffers would improve throughput at the cost of memory.

Filesystem sequential read performance reaches approximately 80 MB/s when reading from the buffer cache. Uncached reads are limited by the virtio-blk device throughput, typically 40-60 MB/s on QEMU depending on the host storage performance.

Process creation via fork takes approximately 50 microseconds, dominated by page table duplication. The copy-on-write optimization makes fork cost proportional to the number of page table pages rather than the process's total memory usage.

Scheduler overhead is negligible at less than 0.1% of CPU time for typical workloads. The round-robin scheduler's O(1) enqueue and dequeue operations contribute to this low overhead. The periodic load balancer adds minimal additional cost.

Memory allocation from the buddy allocator takes approximately 200 nanoseconds for a single page. The constant-time free list operation avoids the variability of more complex allocators. Slab allocation is even faster at approximately 100 nanoseconds.

Boot time from kernel entry to init process launch is approximately 50 milliseconds on QEMU with 256 MB of RAM. The boot time is dominated by physical memory initialization (scanning and building the free list) and device probing. Larger memory configurations increase boot time linearly.

Interrupt latency from device assertion to handler entry is approximately 3 microseconds. This latency includes the PLIC claim, trap entry, and handler dispatch. The latency is sufficient for all supported devices but would need reduction for real-time applications.

7.2 System Call Interface Design

The system call interface follows POSIX conventions where practical, providing familiar semantics for students with Unix experience. Deviations from POSIX are documented and motivated by simplification for educational purposes. Approximately 30 system calls cover process, file, memory, and IPC operations.

Error handling uses negative errno values returned in the a0 register. The user-space library translates negative returns to -1 with errno set to the absolute value. This convention matches the standard C library behavior and teaches students the Unix error handling pattern.

File descriptor management follows the Unix model where descriptors are small integers indexing into a per-process table. The table has a configurable maximum size (default 64). File descriptors are inherited across fork and preserved across exec unless marked close-on-exec.

Process hierarchy system calls implement the fork-exec-wait model. Fork creates a child process that is a copy of the parent. Exec replaces the calling process's program. Wait blocks until a child

exits and retrieves its exit status. Exit terminates the calling process.

Memory management system calls include brk for heap adjustment and mmap for arbitrary memory mapping. The brk system call provides the traditional Unix heap interface, while mmap supports file-backed mappings, anonymous mappings, and shared mappings.

I/O system calls implement open, close, read, write, seek, stat, and fstat. The open system call returns the lowest available file descriptor. Read and write operate on byte arrays with offset tracking. Stat retrieves file metadata including size, type, and permissions.

Directory operations include mkdir for creation, rmdir for removal, and getdents for reading entries. The chdir system call changes the process's working directory. Path resolution operates relative to the working directory for paths not beginning with slash.

The dup2 system call redirects file descriptors, providing the mechanism for shell I/O redirection. Combined with pipe and fork, dup2 enables the construction of command pipelines. Students implement a shell that uses these system calls to build a functional command interpreter.

The ioctl system call provides device-specific operations that do not fit the read/write model. FRISC-OS uses ioctl for terminal settings (window size, echo mode) and device configuration. The ioctl interface demonstrates the flexibility and type-safety challenges of extensible interfaces.

System call versioning uses a version number in a dedicated register to support interface evolution. Old binaries continue to work with new kernels through compatibility handlers that translate old call formats to new internal interfaces. This mechanism teaches students about ABI stability.

8.2 Capability Extensions

An experimental capability mode extends the basic Unix security model with fine-grained access control. In capability mode, process credentials are replaced by a set of capabilities that grant specific permissions. Each capability authorizes a particular operation on a particular object.

Capabilities are implemented as unforgeable tokens stored in the process's capability table. The table maps capability IDs to capability descriptors containing the target object, permitted operations, and delegation rights. System calls accept capability IDs instead of file paths.

Capability delegation allows a process to grant a subset of its capabilities to a child process. The child cannot exceed the parent's permissions, enforcing the principle of least privilege. Delegation is specified at fork time through a capability mask.

File capabilities replace the traditional open-by-path model with open-by-capability. A process opens a file by presenting a capability that authorizes the requested access mode. The capability is obtained from the parent process or through a capability-granting service.

Ambient capabilities provide a compatibility layer that automatically creates capabilities from traditional Unix permissions. When capability mode is disabled, the kernel translates permission checks to the traditional model. This dual-mode design allows incremental adoption.

Capability revocation removes a previously granted capability from all processes that hold it. The revocation mechanism uses a global revocation table that is checked on every capability use. Revoked capabilities return an error when presented, informing the process that its access has been withdrawn.

Network capabilities control socket creation and connection establishment. A process needs a specific capability to create sockets, bind to ports, or connect to addresses. This fine-grained control prevents compromised processes from establishing arbitrary network connections.

Capability-based sandboxing runs untrusted code with a minimal set of capabilities. The sandbox inherits no ambient capabilities and receives only explicitly granted permissions. This model demonstrates modern application sandboxing techniques used in production systems.

The capability security model is implemented as an optional kernel module that can be enabled for specific labs. When enabled, the module intercepts permission checks and enforces capability-based access control. When disabled, the standard Unix model is used.

Students compare the Unix and capability security models through lab exercises that attempt privilege escalation under each model. The exercises demonstrate how capabilities provide stronger isolation guarantees and why modern operating systems are adopting capability-based designs.

References

[1] Patterson, D. and Waterman, A. The RISC-V Reader. Strawberry Canyon LLC, 2017.
[2] RISC-V Privileged Specification v1.12. RISC-V International, 2021.
[3] Arpaci-Dusseau, R. and Arpaci-Dusseau, A. Operating Systems: Three Easy Pieces. 2018.
[4] Cox, R. et al. xv6: A Teaching Operating System. MIT, 2023.
[5] Tanenbaum, A.S. Modern Operating Systems, 4th Edition. Pearson, 2014.
[6] Love, R. Linux Kernel Development, 3rd Edition. Addison-Wesley, 2010.
[7] SiFive FU740-C000 Manual. SiFive Inc., 2021.