FRISC-OS: A RISC-V Educational Operating System in Lateralus

bad-antics | April 2024 | Technical Report

Abstract

FRISC-OS is an educational operating system for the RISC-V architecture, written in Lateralus. Designed for undergraduate operating system courses, it provides a clear codebase, structured laboratory exercises, and automated grading. This paper describes the kernel architecture, RISC-V platform support, and pedagogical approach.

1 Introduction

FRISC-OS is an educational operating system designed for teaching operating system concepts on the RISC-V architecture. Written primarily in Lateralus with minimal assembly bootstrapping, FRISC-OS provides a clear, readable, and well-documented codebase that serves as both a teaching tool and a reference implementation.

The project targets undergraduate operating systems courses where students learn by reading, modifying, and extending a real OS kernel. Unlike production operating systems that prioritize performance and feature completeness, FRISC-OS prioritizes clarity and pedagogical value at every design decision.

This paper describes FRISC-OS's architecture, the RISC-V platform support, the kernel design, memory management, process model, device drivers, laboratory exercises, and the pedagogical approach that guides the project's development.

2 RISC-V Platform

RISC-V is an open instruction set architecture that provides a clean, modular design ideal for teaching. The base integer instruction set (RV64I) contains only 47 instructions, making it feasible for students to understand the entire ISA within a single course. Extensions add functionality for multiplication, atomics, floating-point, and compressed instructions.

FRISC-OS targets the QEMU virt machine, which emulates a RISC-V system with configurable memory, UART, virtio devices, and a PLIC interrupt controller. Using QEMU eliminates the need for physical hardware, allowing students to develop and test on any computer running Linux, macOS, or Windows.

```
# Build and run FRISC-OS in QEMU
$ lateralus build --target riscv64-frisc-none
$ qemu-system-riscv64 \
    -machine virt \
    -bios none \
    -kernel target/riscv64/frisc-os.elf \
    -serial mon:stdio \
```

```
    -nographic \
    -smp 4 \
    -m 256M
```

The RISC-V privilege model defines three levels: Machine mode (M-mode) for firmware, Supervisor mode (S-mode) for the kernel, and User mode (U-mode) for applications. FRISC-OS runs in S-mode, relying on the SBI (Supervisor Binary Interface) firmware for M-mode services including timer programming and inter-processor interrupts.

The SBI firmware, implemented using OpenSBI, provides a hardware abstraction layer that isolates the kernel from machine-specific details. SBI calls use the ecall instruction with function IDs in registers, following a calling convention similar to system calls. This layered approach teaches students about privilege separation and hardware abstraction.

3 Kernel Architecture

FRISC-OS uses a monolithic kernel architecture where all kernel services run in a single address space. The monolithic design is chosen for simplicity: function calls between subsystems are direct rather than requiring message passing. Students can trace execution paths through the entire kernel using standard debugging tools.

The kernel is organized into modules: boot, trap, memory, process, filesystem, device, syscall, and console. Each module has a well-defined interface specified by a Lateralus trait. Module dependencies form a directed acyclic graph with boot at the root and syscall at the leaves.

```
// Kernel module initialization order
pub fn kernel_main() {
    console::init();     // UART output
    println!("FRISC-OS booting...");
    trap::init();        // interrupt vectors
    memory::init();      // page allocator + MMU
    process::init();     // scheduler
    device::init();      // virtio drivers
    fs::init();          // filesystem mount
    syscall::init();     // system call table
    process::start_init(); // launch /init
}
```

Initialization follows a strict order that respects dependencies between modules. Each module's init function configures the subsystem and registers itself with dependent modules. Initialization errors are handled by printing a diagnostic message and halting, as there is no recovery path during boot.

The kernel log provides timestamped output at configurable verbosity levels. Log messages include the source module and severity (debug, info, warning, error). The log is invaluable during development and debugging, and students learn to use structured logging for kernel diagnostics.

4 Trap Handling

RISC-V uses a unified trap mechanism for exceptions, interrupts, and system calls. The stvec register points to the trap handler, which saves registers to the stack, identifies the trap cause from the scause register, and dispatches to the appropriate handler function.

The trap handler saves all 31 general-purpose registers plus the program counter and status register to a TrapFrame structure on the kernel stack. This complete context save enables the trap handler to switch between user and kernel mode and to perform context switches between processes.

```
// Trap frame layout for RISC-V
pub struct TrapFrame {
    pub regs: [u64; 32],    // x0-x31
    pub sstatus: u64,       // supervisor status
    pub sepc: u64,          // exception program counter
    pub kernel_sp: u64,     // saved kernel stack pointer
    pub kernel_trap: u64,   // address of trap handler
    pub kernel_satp: u64,   // kernel page table
}
```

Exception handling processes synchronous traps including illegal instructions, page faults, and environment calls. The page fault handler is particularly important as it implements demand paging, copy-on-write, and stack growth. Each exception type has a dedicated handler function.

Timer interrupts drive the preemptive scheduler. The SBI timer is programmed to fire at 100 Hz, providing a 10 ms time quantum for process scheduling. Each timer interrupt increments the system tick counter and invokes the scheduler to potentially switch to another process.

5 Memory Management

FRISC-OS uses RISC-V's Sv39 virtual memory scheme, which provides a 39-bit virtual address space (512 GB) with three-level page tables. Each page table entry is 8 bytes and maps a 4 KB page. The three levels are named L2, L1, and L0 corresponding to gigapage, megapage, and standard page granularity.

The physical page allocator uses a free list of 4 KB page frames. Pages are allocated by removing from the head of the free list and deallocated by prepending to the head. The free list is initialized during boot by scanning the device tree for available memory regions.

Kernel virtual memory uses an identity mapping for the first 256 MB of physical memory, providing straightforward kernel-to-physical address translation. User processes each have their own page table hierarchy with the kernel mapped in the upper half of the address space.

```
// Page table entry for Sv39
pub struct PageTableEntry(u64);

impl PageTableEntry {
    pub fn is_valid(&self) -> bool { self.0 & PTE_V != 0 }
```

```
    pub fn is_leaf(&self) -> bool {
        self.0 & (PTE_R | PTE_W | PTE_X) != 0
    }
    pub fn ppn(&self) -> u64 {
        (self.0 >> 10) & 0xFFF_FFFF_FFFF
    }
}
```

The page fault handler distinguishes between three types of faults: instruction fetch (no execute permission), load (no read permission), and store (no write permission). Each fault type is resolved differently based on the faulting virtual memory area's configuration.

6 Process Model

FRISC-OS implements a Unix-like process model with fork, exec, wait, and exit system calls. Each process has a unique PID, an address space, a set of open file descriptors, a parent process reference, and scheduling state. The process table is a fixed-size array indexed by PID.

Process creation via fork duplicates the parent's address space using copy-on-write page sharing. The child receives copies of the parent's file descriptors, with shared underlying file objects. Fork returns the child PID to the parent and zero to the child, following the Unix convention.

The exec system call replaces the current process's address space with a new program loaded from an ELF executable. The loader parses ELF program headers, maps loadable segments into the process's address space, and sets the entry point. Arguments and environment variables are copied to the new stack.

The scheduler maintains a run queue of ready processes and selects the next process using round-robin scheduling. Context switching saves the current process's registers to its TrapFrame and restores the next process's registers. The switch also updates the satp register to activate the new process's page table.

7 Device Drivers

FRISC-OS includes drivers for UART, virtio-blk (block device), and virtio-net (network device). Drivers follow a common interface defined by the Device trait, which specifies init, read, write, and interrupt handler methods. The driver model is intentionally simple to be comprehensible in a one-semester course.

The UART driver provides character-level I/O for the console. Input characters trigger interrupts that wake processes blocked on console read. Output uses polling for simplicity, with an optional interrupt-driven mode for higher performance. The UART configuration follows the 16550 register layout.

The virtio-blk driver provides block-level access to a virtual disk. Virtio devices use a standardized interface based on descriptor rings (virtqueues) shared between the driver and the device. The driver

places requests in the available ring and retrieves completions from the used ring.

```
// Virtio block read operation
pub fn virtio_blk_read(sector: u64, buf: &mut [u8]) -> Result<()> {
    let desc = alloc_descriptors(3)?;
    desc[0].set_buf(&BlkReq::read(sector));
    desc[1].set_buf(buf).set_write();
    desc[2].set_buf(&BlkStatus::new()).set_write();
    queue.push_available(desc[0].id);
    queue.notify();
    wait_for_completion(desc[0].id)?;
    Ok(())
}
```

The PLIC (Platform-Level Interrupt Controller) routes device interrupts to CPUs. The PLIC driver configures interrupt priorities and per-CPU enable masks. When a device interrupt fires, the PLIC handler claims the interrupt, dispatches to the appropriate device driver, and completes the claim.

8 Filesystem

FRISC-OS implements a simplified version of the ext2 filesystem. The filesystem uses a superblock for metadata, a block bitmap for free block tracking, an inode table for file metadata, and data blocks for file contents. The design is close enough to real ext2 that students can use standard Linux tools to create and inspect filesystem images.

Inodes store file metadata including size, type, permissions, timestamps, and block pointers. Direct block pointers store the first 12 blocks, an indirect pointer stores blocks 13 through 1035, and a double-indirect pointer extends capacity further. This structure is identical to the classic Unix inode layout.

Directory entries are stored as linked lists of variable-length records within directory blocks. Each entry contains an inode number, entry length, name length, and the file name. The root directory is always inode 2, following the ext2 convention.

The buffer cache provides a layer between the filesystem and the block device driver. Cached blocks are held in memory to avoid repeated disk reads. The cache uses a write-back policy where modifications are flushed periodically or on explicit sync operations.

9 Laboratory Exercises

FRISC-OS includes a series of laboratory exercises that guide students through kernel development. Each lab builds on the previous one, progressively extending the kernel with new functionality. Labs are accompanied by automated tests that verify correct implementation.

- - Lab 1: Boot and Console - set up development environment, boot kernel, implement printf
- - Lab 2: Memory Allocation - implement physical page allocator and page table mapping

- - Lab 3: Trap Handling - implement trap handler, timer interrupts, system calls
- - Lab 4: Processes - implement fork, exec, wait, exit, and round-robin scheduling
- - Lab 5: Virtual Memory - implement demand paging, copy-on-write, and mmap
- - Lab 6: File System - implement inode operations, directory operations, and buffer cache
- - Lab 7: Device Drivers - implement UART input interrupt and virtio-blk driver
- - Lab 8: Networking - implement virtio-net driver and basic TCP/IP stack

Each lab provides a skeleton implementation with TODO markers where students must write code. The skeleton compiles and boots but the unimplemented features return error codes. Students replace the TODO markers with working implementations, guided by comments and specification documents.

Automated grading uses QEMU-based test harnesses that boot the student's kernel and exercise the implemented functionality. Tests check both correct behavior and error handling. Grade reports identify which tests passed and provide hints for failing tests.

10 Pedagogical Approach

The pedagogical philosophy emphasizes learning by doing. Students understand operating system concepts deeply by implementing them, not merely reading about them. Each concept is first presented in a lecture, then explored through code reading, and finally internalized through implementation.

Code clarity takes precedence over optimization. Variable names are descriptive, functions are short and focused, and complex operations are broken into clearly named helper functions. Comments explain the rationale for design decisions, not just what the code does.

Incremental complexity ensures students are never overwhelmed. Early labs work with a single process and no virtual memory, allowing students to focus on basic kernel mechanisms. Later labs add complexity gradually, each building on previously tested foundations.

11 Conclusion

FRISC-OS provides an effective platform for teaching operating system concepts on RISC-V. The combination of a clean ISA, a well-documented kernel, and structured laboratory exercises enables students to gain deep practical understanding of operating system internals. The Lateralus implementation demonstrates the language's suitability for systems programming in educational contexts.

2.1 RISC-V Extensions Used

The RV64I base instruction set provides integer arithmetic, logical operations, loads, stores, branches, and jumps. All kernel code uses this base set, ensuring compatibility with the simplest RISC-V implementations. The simplicity of the base ISA means students can read disassembled

kernel code without extensive ISA knowledge.

The M extension adds integer multiplication and division instructions. While the kernel can function without hardware multiply, the M extension significantly improves performance for address calculation and scheduling arithmetic. The compiler automatically uses M-extension instructions when targeting a compatible machine.

The A extension provides atomic memory operations including load-reserved/store-conditional pairs and atomic swap. These operations implement spinlocks, mutexes, and lock-free data structures in the kernel. The A extension is essential for multiprocessor support.

The C extension adds compressed 16-bit instruction encodings for common operations. Compressed instructions reduce code size by approximately 30 percent, improving instruction cache utilization. The extension is transparent to the programmer as the assembler automatically selects compressed encodings.

The S extension defines supervisor-mode CSRs (Control and Status Registers) for page table management, trap handling, and timer configuration. FRISC-OS uses sstatus, stvec, sepc, scause, stval, satp, and sip/sie registers for kernel operations.

The Zicsr extension provides CSR access instructions (csrr, csrw, csrs, csrc) that read, write, set, and clear bits in control registers. These instructions are the primary interface between the kernel and the RISC-V privilege architecture.

The Zifencei extension provides the fence.i instruction for instruction cache synchronization. This instruction is necessary after modifying code pages (for example, when loading a new process) to ensure the instruction cache reflects the updated memory contents.

The F and D extensions add single-precision and double-precision floating-point support. FRISC-OS saves and restores floating-point registers during context switches when a process has used floating-point instructions, using lazy save to avoid overhead for integer-only processes.

Timer support uses the stimecmp CSR (or SBI timer calls on older implementations) to program one-shot timer interrupts. The timer drives the preemptive scheduler and provides the time base for the sleep system call.

Performance counter extensions provide cycle and instruction counters accessible from supervisor mode. FRISC-OS exposes these counters through a system call, allowing user-space programs to measure execution performance for benchmarking exercises.

5.1 Address Space Layout

The Sv39 virtual address space is divided into two halves: the lower half (addresses 0x0 to 0x3FFFFFFFF) is available for user processes, and the upper half (addresses 0xFFFFFFC000000000 and above) is reserved for the kernel. This split provides 256 GB for each half, which is more than sufficient for educational use.

The user address space layout places the text segment at 0x10000, the data and BSS segments immediately following, the heap growing upward from the end of BSS, and the stack at the top of the user half growing downward. This layout matches the conventional Unix process memory model.

The kernel address space maps physical memory starting at 0xFFFFFFC000000000 with an identity offset. Kernel code, data, and the heap are accessed through this mapping. Device MMIO regions are mapped at fixed virtual addresses above the physical memory mapping.

Guard pages between the user stack and heap prevent stack overflow from silently corrupting heap data. A guard page is a virtual page with no physical mapping; accessing it triggers a page fault that the kernel handles by terminating the process with a descriptive error message.

Shared memory regions allow processes to map the same physical pages at agreed-upon virtual addresses. The shared memory implementation teaches students about physical-to-virtual mapping independence and the synchronization challenges of shared memory programming.

The trampoline page is mapped at the same virtual address in every address space (both user and kernel). It contains the trap entry and return code that must be accessible during the address space switch that occurs on every trap. Mapping it at a fixed address avoids the chicken-and-egg problem of switching page tables during trap handling.

Per-process kernel stacks are allocated in the kernel address space with guard pages between them. The guard pages prevent one process's kernel stack overflow from corrupting another process's stack. The kernel stack size is one page (4 KB), which is sufficient for the simplified FRISC-OS call depth.

The page table walker traverses the three-level page table hierarchy to translate virtual addresses to physical addresses. The walker is implemented as a simple function that indexes into each level using the corresponding bits of the virtual address. Students trace through this function to understand virtual memory translation.

Address space switching occurs during context switches by writing the new process's root page table address to the satp CSR. The sfence.vma instruction flushes the TLB after the switch to ensure subsequent memory accesses use the new page table. On multiprocessor systems, TLB flushes must be broadcast to all CPUs.

The kernel direct map provides access to all physical memory through the kernel address space without requiring per-page mapping. This direct map simplifies kernel memory management and allows the kernel to access any physical address by adding the kernel base offset.

6.1 Process Synchronization

Sleep locks provide mutual exclusion for operations that may block, such as disk I/O. Unlike spinlocks, sleep locks allow the holding process to be descheduled, preventing priority inversion and CPU waste. The sleep lock implementation uses a wait channel for blocked processes.

Spinlocks protect short critical sections where blocking is not permitted, such as scheduler data

structures and interrupt handlers. The spinlock implementation uses RISC-V atomic instructions and includes debugging support that detects recursive locking and lock-order violations.

Condition variables allow processes to wait for specific conditions within a critical section. The wait operation atomically releases the associated lock and blocks the process. The signal operation wakes one waiting process, and broadcast wakes all waiters.

Semaphores implement counting-based synchronization for resource management. The kernel uses semaphores for limiting concurrent access to shared resources like the buffer cache. The semaphore implementation is built on top of spinlocks and wait channels.

Deadlock detection uses a wait-for graph that tracks which process is waiting for which lock. The detection algorithm runs periodically and reports cycles in the graph. While deadlock detection adds overhead, it is invaluable for debugging student implementations.

Signal handling allows asynchronous notification of processes. The kernel supports a simplified signal model with kill and sigaction system calls. Signal delivery modifies the process's trap frame to execute the signal handler on the next return to user mode.

Pipe implementation provides unidirectional byte streams between processes. Pipes use a kernel buffer with reader and writer counts. Writing to a pipe with no readers delivers SIGPIPE. Reading from an empty pipe blocks until data is available or all writers close.

Process groups organize related processes for job control. The shell creates a process group for each pipeline, allowing signals to be delivered to all processes in the group. Process group management teaches students about the relationship between shells, jobs, and signals.

The wait system call blocks the parent until a child process exits. Wait returns the child's exit status and frees the child's process table entry. Orphaned processes are re-parented to init, which periodically waits for them to prevent zombie accumulation.

Atomic operations are exposed to user space through a library that wraps RISC-V A-extension instructions. Students use these operations in later labs to implement user-space synchronization primitives, reinforcing the connection between hardware capabilities and software abstractions.

8.1 Filesystem Implementation Details

The superblock is located at block 1 (block 0 is the boot block) and contains filesystem metadata including the total number of blocks, inodes, free blocks, and free inodes. The superblock is read during filesystem initialization and cached in memory for the lifetime of the kernel.

The block bitmap uses one bit per data block, where 1 indicates allocated and 0 indicates free. Block allocation scans the bitmap for the first free bit, sets it, and returns the corresponding block number. The scan starts from the last allocated block to improve locality.

Inode allocation similarly scans the inode bitmap for a free inode. New inodes are initialized with the specified type (regular file, directory, or symbolic link), permissions, and timestamps. The inode is

written to disk before being returned to the caller.

File read operations translate byte offsets to block numbers using the inode's block pointer structure. Direct pointers map the first 48 KB, indirect pointers map the next 4 MB, and double-indirect pointers extend capacity to 4 GB. Block numbers are resolved to physical addresses for disk read operations.

File write operations allocate new blocks as needed when writing beyond the current file size. The write path handles partial block writes by reading the existing block, modifying the affected bytes, and writing the block back. Block allocation failures are reported as out-of-space errors.

Directory operations create and remove directory entries by manipulating the linked list structure within directory blocks. Entry creation searches for a free slot (entry with inode number 0) or appends to the directory. Removal sets the entry's inode number to 0 and merges adjacent free entries.

Path resolution walks the directory hierarchy from the root inode (or current directory for relative paths) by repeatedly looking up each component name in the current directory. Path resolution handles the special entries . (current directory) and .. (parent directory).

The namei function implements complete path resolution including symbolic link following. Symbolic links are resolved by reading the link target and recursively resolving the resulting path. A recursion limit prevents infinite loops from circular symbolic links.

File descriptor management tracks open files per process. Each file descriptor points to a file object that records the inode, current offset, and open mode. Dup creates a new file descriptor pointing to the same file object, sharing the offset.

Filesystem consistency is maintained through careful ordering of disk writes. New blocks are allocated and written before the inode is updated to reference them. This ordering ensures that a crash cannot leave the filesystem referencing uninitialized blocks.

9.1 Assessment and Evaluation

Assessment uses a combination of automated testing (60 percent), code review (25 percent), and a final presentation (15 percent). Automated tests verify functional correctness, code review evaluates code quality and design decisions, and presentations assess conceptual understanding.

The automated test suite runs approximately 200 tests per lab, covering normal operation, edge cases, and error handling. Tests are designed to be deterministic and produce consistent results across different host platforms. The test harness reports pass/fail status with descriptive messages.

Code review rubrics evaluate naming conventions, function decomposition, comment quality, error handling, and adherence to the project's coding style. Students receive written feedback on their code reviews, helping them develop professional coding habits.

Peer code review is introduced in later labs. Students review each other's implementations and provide written feedback. The peer review process teaches students to read and evaluate code written by others, a skill essential for professional software development.

Performance benchmarks are included in later labs as bonus objectives. Students optimize their implementations to meet performance targets for operations per second in memory allocation, context switching, and file I/O. Optimization exercises reinforce understanding of the performance implications of design choices.

Collaboration policy allows students to discuss concepts and approaches but requires individual implementation. An academic integrity system detects code similarity between submissions and flags potential violations for instructor review. The system distinguishes between legitimate similarity from shared specifications and illegitimate copying.

Office hours provide one-on-one debugging assistance using GDB connected to QEMU. Teaching assistants guide students through debugging techniques including breakpoints, watchpoints, and memory inspection. The debugging sessions are among the most valuable learning experiences.

Course evaluations consistently rate the lab exercises as the most effective component for learning operating system concepts. Students report that implementing the concepts themselves provides deeper understanding than any amount of lecture or reading.

Post-course surveys show that students who complete all labs perform significantly better on technical interviews involving systems programming questions. The hands-on experience with memory management, scheduling, and synchronization translates directly to professional competence.

The course materials are openly available under a Creative Commons license. Several universities have adopted FRISC-OS for their operating systems courses, providing feedback that drives improvements to the labs and documentation.

3.1 Kernel Debugging Infrastructure

GDB remote debugging connects to QEMU's built-in GDB server for source-level kernel debugging. Students set breakpoints, inspect variables, and step through kernel code using familiar GDB commands. The debug configuration loads kernel symbols automatically.

Printf-style debugging uses the UART console for diagnostic output. The kernel's println macro formats and prints messages to the serial console. This low-tech approach is reliable and works even when the kernel is too broken for GDB to connect.

Stack trace generation walks the frame pointer chain to produce human-readable call stacks. Each frame's return address is resolved to a function name using the kernel symbol table. Stack traces are printed automatically on kernel panics and available on demand via a debug key.

Memory dump utilities display hexadecimal and ASCII representations of memory regions. Students use memory dumps to inspect page tables, trap frames, and data structures during debugging. The dump format is designed for easy visual scanning.

Kernel assertions verify invariants at runtime and halt the kernel with a descriptive message when violated. Assertions check conditions like non-null pointers, valid array indices, and lock states. The

assertion macro includes the file name, line number, and condition expression in the panic message.

The kernel debugger (kdb) provides an interactive debug shell accessible via a keyboard shortcut. From kdb, students can inspect process tables, view memory maps, dump page tables, and examine device registers. The debugger is implemented as a module that can be excluded from release builds.

Trace points record events (context switches, system calls, page faults) to a circular buffer. The trace buffer is dumped on demand or on panic, providing a history of recent kernel events. Trace analysis helps students understand the temporal relationships between kernel operations.

QEMU monitor commands complement kernel-level debugging. Students use info registers to view CPU state, info mem to view QEMU's view of page tables, and info mtree to view the memory-mapped device layout. These commands verify that kernel configuration matches the expected hardware state.

Sanitizer support for the kernel catches memory errors at runtime. The address sanitizer detects out-of-bounds accesses and use-after-free bugs. Sanitizer instrumentation is added by the compiler and checked at runtime, providing precise error reports with allocation context.

Logging levels (trace, debug, info, warn, error) allow students to control the verbosity of kernel output. During initial development, trace-level logging shows every operation. As the kernel stabilizes, students reduce the log level to focus on warnings and errors.

7.1 Networking Stack

The virtio-net driver provides Ethernet frame transmission and reception. The driver uses two virtqueues: one for receiving frames and one for transmitting frames. Receive buffers are pre-allocated and replenished after each frame is consumed by the network stack.

The Ethernet layer handles frame encapsulation and decapsulation. Outgoing frames are prepended with Ethernet headers containing source and destination MAC addresses and the EtherType field. Incoming frames are parsed and dispatched to the appropriate protocol handler based on EtherType.

The ARP (Address Resolution Protocol) implementation maintains a cache mapping IP addresses to MAC addresses. ARP requests are broadcast when the cache does not contain an entry for the destination IP. ARP replies update the cache and wake processes waiting for address resolution.

The IP layer implements IPv4 packet handling including fragmentation and reassembly. The IP header checksum is computed on transmission and verified on reception. TTL decrementing and ICMP time-exceeded generation are implemented for completeness.

The ICMP implementation handles echo requests (ping) and generates echo replies. ICMP is the first network protocol students implement, providing immediate visual feedback through ping tests. The implementation teaches students about protocol layering and packet encapsulation.

The UDP implementation provides connectionless datagram delivery. UDP sockets bind to ports,

send datagrams, and receive datagrams. The simplicity of UDP makes it an ideal protocol for teaching socket programming concepts before introducing the complexity of TCP.

The TCP implementation provides reliable byte stream delivery with connection establishment (three-way handshake), flow control (sliding window), and connection termination. The TCP state machine is one of the more complex components in FRISC-OS and occupies an entire lab.

The socket interface provides the system call layer for network programming. Socket operations include socket, bind, listen, accept, connect, send, recv, and close. The interface follows the Berkeley sockets API, allowing students to write familiar network programs.

The network stack uses a layered architecture where each layer adds or removes protocol headers. This architecture directly mirrors the OSI model taught in networking courses, providing a concrete implementation of the theoretical concepts.

Network testing uses a tap device in QEMU that bridges the virtual network to the host. Students can ping the FRISC-OS kernel from the host system and run network client/server programs. The tap configuration is automated by the build system.

10.1 Comparison with Other Teaching OSes

Xv6 from MIT is the closest comparable project. Xv6 is written in C and targets x86 or RISC-V. FRISC-OS differentiates itself through its use of Lateralus, which provides stronger type safety and prevents entire classes of memory bugs that C allows. This reduces frustrating debugging time and allows students to focus on OS concepts.

MINIX, originally developed by Andrew Tanenbaum, uses a microkernel architecture. While microkernel design is pedagogically interesting, the message-passing overhead and distributed debugging complexity make MINIX challenging for a single-semester course. FRISC-OS's monolithic design is simpler to understand and debug.

PintOS from Stanford targets x86 and is widely used in operating systems courses. FRISC-OS offers the advantage of RISC-V's simpler and more regular ISA, which reduces the time students spend understanding hardware quirks and increases the time available for OS concepts.

GeekOS targets x86 with a focus on project-based learning. FRISC-OS follows a similar philosophy but updates the approach with a modern ISA, a memory-safe language, and contemporary development tools including Git-based submission and automated testing.

NachOS uses a simulated hardware platform, while FRISC-OS runs on QEMU's emulated RISC-V hardware. The advantage of real (emulated) hardware is that students work with actual ISA documentation and tools, preparing them for real-world systems programming.

The choice of Lateralus over C is the most significant differentiator. Students spend less time debugging segmentation faults and buffer overflows, and more time understanding operating system algorithms and data structures. The compiler catches many bugs before they manifest at runtime.

FRISC-OS's lab structure is inspired by MIT's xv6 labs but adapted for the Lateralus language and RISC-V architecture. Each lab is self-contained with clear objectives, detailed specifications, and comprehensive automated tests. The lab difficulty is calibrated for undergraduate students with one semester of systems programming experience.

Instructor support materials include lecture slides, exam questions, solution keys, and a teaching guide with common student misconceptions and debugging tips. The materials are updated annually based on instructor feedback from adopting institutions.

Extension projects provide additional challenges for advanced students. Projects include implementing a simple shell, adding user-space threading, writing a network application, or porting the kernel to physical RISC-V hardware. Extension projects are optional and graded separately.

The FRISC-OS community maintains a discussion forum where instructors share experiences, report issues, and propose improvements. The community-driven development model ensures that the project continues to evolve based on teaching experience across diverse educational settings.

4.1 System Call Implementation

System calls use the RISC-V ecall instruction to transition from user mode to supervisor mode. The ecall triggers a synchronous exception with cause code 8 (environment call from U-mode). The trap handler reads the system call number from register a7 and arguments from a0 through a5.

The system call table maps call numbers to handler functions. FRISC-OS implements approximately 25 system calls covering process management (fork, exec, exit, wait, getpid), file operations (open, close, read, write, seek, stat), memory management (brk, mmap), and miscellaneous operations (sleep, uptime).

Return values are passed back to user space through the a0 register. Error codes follow the convention of returning negative errno values. The user-space library wraps system calls in functions that set errno and return -1 on error, matching the POSIX convention.

System call argument validation ensures that user-provided pointers reference valid user-space memory. The kernel verifies that each pointer argument falls within the process's mapped virtual memory regions before dereferencing it. Invalid pointers cause the system call to return an EFAULT error.

The open system call resolves the path name to an inode, allocates a file object with the specified mode, and assigns it to the lowest available file descriptor. The implementation handles flags for read-only, write-only, read-write, create, and truncate modes.

The mmap system call maps files or anonymous memory into the process's address space. File-backed mappings use demand paging to load pages from disk on first access. Anonymous mappings provide zero-filled pages for heap expansion and large allocations.

The pipe system call creates a unidirectional communication channel between processes. The kernel allocates a buffer and two file descriptors: one for reading and one for writing. Pipe I/O blocks when

the buffer is full (write) or empty (read).

System call tracing logs all system calls made by a process, including arguments and return values. Tracing is enabled per-process through a debug flag. The trace output helps students verify that their user-space programs interact correctly with the kernel.

The dup and dup2 system calls duplicate file descriptors. Dup assigns the lowest available descriptor, while dup2 assigns a specific descriptor number. Both operations increment the reference count on the underlying file object.

The sbrk system call adjusts the process heap boundary. Positive values grow the heap by allocating new pages, while negative values shrink it by unmapping pages. The implementation validates that the new boundary does not overlap with the stack or other mapped regions.

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., Kaashoek, F., and Morris, R. xv6: A Teaching Operating System. MIT, 2023.
[5] Tanenbaum, A.S. and Bos, H. Modern Operating Systems, 4th Ed. Pearson, 2014.
[6] OpenSBI Documentation. https://github.com/riscv-software-src/opensbi, 2024.