FRISC OS: A Minimal RISC-V Operating System for Education

From bare metal to preemptive multitasking on RV64GC

Lateralus Language

bad-antics · June 2025 · Lateralus Language Research

ABSTRACT We present FRISC OS (Free RISC Operating System), a minimal operating system for the RISC-V RV64GC architecture. Unlike educational OS projects that target x86, FRISC OS embraces RISC-V's clean privilege model (M/S/U modes), device tree discovery, and the PLIC interrupt controller. The system progresses from a bare OpenSBI handoff through Sv39 virtual memory, a buddy-system physical memory allocator, virtio block and network drivers, FAT32 filesystem, and a preemptive round-robin scheduler supporting SMP. We describe pedagogical decisions made at each stage and explain how FRISC OS compares to xv6 as a teaching tool. All source code fits in under 8,000 lines of C and RISC-V assembly, runs on QEMU virt with zero host dependencies, and forms the basis of an 18-week undergraduate operating systems course.

1. Why RISC-V for Education

x86-64 is the dominant educational OS target, yet it is an exceptionally poor fit for teaching OS concepts. The architecture accumulates 35 years of backwards-compatibility decisions: real mode, protected mode, and long mode each add complexity that a student must navigate before writing a single useful line of OS code. The GDT, LDT, TSS, IDT, and segment registers exist solely to maintain compatibility with software from the 1980s and have no pedagogical value.

RISC-V, published by Waterman et al. (2011) as a clean-slate ISA, eliminates all of this historical baggage. The base RV64I integer instruction set has 47 instructions. The privilege specification (version 20211203) defines three privilege levels — Machine, Supervisor, and User — with a small, orthogonally designed set of control-and-status registers (CSRs). A student can read the entire privilege specification in one sitting.

The RISC-V ecosystem includes high-quality open-source toolchains (LLVM, GCC), a free and precise simulator (Spike), and QEMU's virt board which models the SiFive U74 core with deterministic interrupt behavior. Running OS code in QEMU requires no hardware purchase and no hypervisor license, making FRISC OS accessible to students with any reasonably modern laptop.

FRISC OS targets RV64GC: the base 64-bit integer ISA (I), the G (IMAFDZicsr) extension combination, and the C (compressed) extension. The G set includes the multiply/divide (M), atomic (A), single-precision float (F), and double-precision float (D) extensions. Students use M and A in the kernel and can optionally use F and D in user programs. The C extension reduces code size by approximately 25% without changing the programming model.

2. Privilege Levels and OpenSBI Handoff

RISC-V defines three privilege levels: Machine (M), Supervisor (S), and User (U). M-mode is the most privileged; firmware runs here. S-mode is where the OS kernel runs. U-mode is where user programs run. A well-designed OS minimizes the code running in M-mode, delegating all but a few fundamental exceptions to S-mode.

OpenSBI (Open Supervisor Binary Interface) is the standard M-mode firmware for RISC-V. It initializes the hardware, sets up M-mode trap delegation, and jumps to the S-mode entry point. FRISC OS receives control at _start in S-mode with a0 containing the hart ID and a1 containing the physical address of the device tree blob (DTB).

```
# Entry point: S-mode, a0=hart_id, a1=dtb_phys
.section .text.init
.global _start
_start:
```

```
    # Set up the stack (each hart gets 16 KiB)
    la   sp, _stack_top
    li   t0, 0x4000          # 16 KiB per hart
    mul  t0, t0, a0
    sub  sp, sp, t0
    # Clear BSS on hart 0 only
    bnez a0, .skip_bss
    la   t0, _bss_start
    la   t1, _bss_end
.bss_loop:
    sd   zero, 0(t0)
    addi t0, t0, 8
    blt  t0, t1, .bss_loop
.skip_bss:
    call kmain               # jump to C kernel
```

OpenSBI delegates most exceptions to S-mode via the medeleg CSR. FRISC OS relies on OpenSBI delegating environment calls from U-mode (medeleg bit 8) and misaligned memory access faults, instruction page faults, and load/store page faults (bits 0, 12, 13, 15). M-mode retains control of timer interrupts until FRISC OS programs the CLINT (see Section 8); thereafter, timer interrupts are delegated to S-mode via mideleg bit 5.

The SBI (Supervisor Binary Interface) is a set of M-mode services callable from S-mode via the ecall instruction. FRISC OS uses three SBI extensions: base (probing available extensions), HART state management (starting secondary harts), and the legacy console putchar function for early boot output before the UART driver is initialized. The SBI call convention uses a7 for the extension ID and a6 for the function ID.

3. The Device Tree

The Flattened Device Tree (FDT) is a data structure that firmware passes to the operating system describing the hardware: CPU cores, memory regions, interrupt controllers, timers, and peripheral devices. On QEMU virt, the DTB is generated at boot time and its physical address is passed in register a1. FRISC OS parses the DTB in early boot before enabling virtual memory.

The DTB is a binary format defined by the Devicetree Specification (v0.4). FRISC OS implements a minimal read-only parser in 280 lines of C that walks the flattened tree structure and extracts: memory regions (memory@ nodes), the CLINT base address (riscv,clint0), the PLIC base address (riscv,plic0), and virtio MMIO regions (virtio_mmio nodes).

```
// Minimal DTB walker (excerpt)
typedef struct { uint32_t magic, totalsize, off_dt_struct,
                         off_dt_strings, off_mem_rsvmap,
                         version, last_comp_version;
} FdtHeader;

void fdt_walk(void *dtb, fdt_visitor_fn visitor, void *ctx) {
    FdtHeader *h = dtb;
    if (be32(h->magic) != 0xd00dfeed) panic("bad DTB magic");
    uint32_t *s = dtb + be32(h->off_dt_struct);
    char     *strs = dtb + be32(h->off_dt_strings);
    fdt_walk_node(s, strs, visitor, ctx);
}
```

The QEMU virt DTB contains 12 virtio-mmio nodes at addresses 0x10001000 through 0x1000c000, each separated by 0x1000 bytes. FRISC OS probes each node for its magic value (0x74726976) and device type register offset 0x8. Type 1 is a block device; type 2 is a network device. FRISC OS

initializes the first block device it finds as the root block device and the first network device as the primary network interface.

Students in the FRISC OS course are assigned the task of extending the DTB parser to also extract the RTC (real-time clock) node and use it to initialize a wall-clock time source. This exercise teaches both DTB parsing and the concept of device probing without modifying any existing FRISC OS code, because the DTB walker uses a visitor pattern that students can extend without recompiling the kernel.

4. Sv39 Virtual Memory

RISC-V RV64 defines three virtual memory schemes: Sv39 (39-bit virtual address space), Sv48 (48-bit), and Sv57 (57-bit). FRISC OS uses Sv39, which provides a 512 GiB virtual address space per process and requires three levels of page tables. This is sufficient for educational purposes and simpler to implement than Sv48.

In Sv39, a virtual address is split into three 9-bit VPN fields and a 12-bit page offset. The three VPN fields index into a three-level page table tree; each entry is 8 bytes containing a physical page number and flags. The root page table physical address is written to the satp CSR with mode field 8 (Sv39) in the top 4 bits.

```
// Sv39 virtual address layout
// Bits [63:39]: must sign-extend bit 38
// Bits [38:30]: VPN[2] (9 bits) -> indexes L2 page table
// Bits [29:21]: VPN[1] (9 bits) -> indexes L1 page table
// Bits [20:12]: VPN[0] (9 bits) -> indexes L0 page table
// Bits [11:0]:  page offset (12 bits, 4 KiB pages)

// PTE layout
// Bits [63:54]: reserved (must be 0)
// Bits [53:10]: PPN (44-bit physical page number)
// Bits [9:8]:   RSW (reserved for software use)
// Bit  7: D (dirty), Bit 6: A (accessed)
// Bit  5: G (global), Bit 4: U (user)
// Bits [3:1]: XWR flags, Bit 0: V (valid)
```

FRISC OS maps the kernel into the upper portion of the virtual address space at 0xFFFFFF8000000000, which is the canonical high address in Sv39. The direct map of all physical memory occupies 0xFFFFFF0000000000 to 0xFFFFFF8000000000. This layout, borrowed from Linux's arm64 kernel virtual address map, allows the kernel to access any physical address with a simple offset addition rather than dynamic mapping.

TLB shootdowns on SMP are handled via the RISC-V SFENCE.VMA instruction broadcast. When FRISC OS modifies a page table entry that may be cached in a remote hart's TLB, it sends an IPI to all other harts and waits for acknowledgement before continuing. The IPI is delivered via the CLINT software interrupt mechanism. This approach is simpler than lazy TLB invalidation but sufficient for a course that does not cover TLB performance optimization.

5. Physical Memory Allocator

FRISC OS implements a buddy-system allocator for physical page frames. The buddy system supports allocation and deallocation in O(log n) time where n is the number of frames, with zero external fragmentation. The allocator manages pages from the end of the kernel image to the end of physical memory reported by the DTB memory nodes.

The buddy system partitions memory into blocks of size 2^k pages for k = 0, 1, ..., MAX_ORDER. FRISC OS uses MAX_ORDER = 10, giving a maximum block size of 1024 pages (4 MiB). Each order maintains a free list of available blocks at that size. Allocation of 2^k pages removes a block from the order-k list; if no order-k block is free, the allocator splits an order-(k+1) block.

```
// Buddy allocator structures
#define MAX_ORDER 10
typedef struct BuddyBlock { struct BuddyBlock *next; } BuddyBlock;
static BuddyBlock *free_lists[MAX_ORDER + 1];
static uint8_t     page_order[MAX_PHYS_PAGES]; // order of each block

void *buddy_alloc(int order) {
    for (int k = order; k <= MAX_ORDER; k++) {
        if (!free_lists[k]) continue;
        BuddyBlock *b = free_lists[k];
        free_lists[k] = b->next;
        while (k-- > order) split_block(b, k + 1); // split down
        page_order[FRAME(b)] = order;
        return b;
    }
    return NULL; // OOM
}
```

Deallocation coalesces adjacent buddy blocks. Two blocks are buddies if they have the same order k and their addresses differ only in bit k+12 (since pages are 4 KiB = 2^12 bytes). The deallocation loop checks whether the buddy of the freed block is also free and, if so, merges them into a block of order k+1. This continues until no free buddy exists or MAX_ORDER is reached.

FRISC OS also implements a slab allocator on top of the buddy system for small kernel objects: task_t (256 bytes), vm_area_t (128 bytes), and inode_t (96 bytes). Each slab holds 16 objects per 4 KiB page. The slab allocator reduces internal fragmentation for frequently allocated fixed-size objects and is the first taste of allocator layering for students.

6. Trap Vector Setup

RISC-V traps (exceptions and interrupts) are handled by setting the stvec CSR to the address of the trap handler. FRISC OS uses vectored mode (stvec bit 0 = 1), which causes interrupts to jump to stvec + 4 * cause while exceptions always jump to stvec (with bit 0 clear). This allows interrupt-specific handlers without a dispatch table in C.

```
# Trap entry (excerpted)
.align 4
.global _trap_entry
_trap_entry:
    # Save all 32 registers to the task's trapframe
    # sscratch holds the kernel stack pointer
    csrrw sp, sscratch, sp    # swap sp with kernel sp
    sd    ra, TF_RA(sp)
    sd    t0, TF_T0(sp)
    # ... (all 32 regs) ...
    sd    t6, TF_T6(sp)
    csrr  a0, scause
    csrr  a1, stval
    mv    a2, sp            # trapframe pointer
    call  trap_handler      # C handler
    # Restore and sret
```

The C-level trap handler dispatches on scause. The top bit of scause distinguishes interrupts (bit 63 = 1) from exceptions (bit 63 = 0). Interrupt codes 1, 5, and 9 are supervisor software, timer, and external interrupts respectively. Exception codes 8, 9, 10, and 11 are environment calls from U/S/VS/M mode; FRISC OS handles code 8 (U-mode ecall) as the system call entry point.

System calls follow the Linux RISC-V ABI: syscall number in a7, arguments in a0-a5, return value in a0. FRISC OS defines 24 system calls covering process management (fork, exec, exit, wait), file I/O (open, read, write, close, lseek), memory management (mmap, munmap, brk), and IPC (pipe, send, recv). This minimal set is sufficient for the shell and basic utility programs used in the course.

The sscratch CSR plays a critical role in the trap entry sequence. Before entering U-mode, the kernel writes the per-task kernel stack pointer to sscratch. On trap entry, the first instruction swaps sp and sscratch, giving the handler immediate access to the kernel stack without requiring any memory reads from the user's stack. This technique avoids the double-fault scenario where a stack overflow in user mode also overflows the trap handler's stack.

7. CLINT Timer

The Core Local Interruptor (CLINT) is the timer and inter-processor interrupt controller in the SiFive RISC-V core. On QEMU virt, it is memory-mapped at physical address 0x2000000. The CLINT provides a 64-bit real-time counter mtime at offset 0xBFF8 and per-hart compare registers mtimecmp[hart] at offsets 0x4000+8*hart.

FRISC OS programs a 10 ms periodic timer by writing mtimecmp[0] = mtime + TICKS_PER_10MS at boot. The CLINT timer frequency on QEMU virt is 10 MHz, so TICKS_PER_10MS = 100,000. When mtime reaches mtimecmp, a machine-mode timer interrupt fires. OpenSBI (running in M-mode) converts this to a supervisor-mode timer interrupt by setting mip.STIP and clearing mtimecmp; FRISC OS handles the S-mode timer interrupt in its trap handler.

```
// CLINT timer setup (S-mode, via SBI)
#define CLINT_BASE 0x2000000UL
#define CLINT_MTIME     (*(volatile uint64_t*)(CLINT_BASE + 0xBFF8))
#define CLINT_MTIMECMP0 (*(volatile uint64_t*)(CLINT_BASE + 0x4000))
#define TICKS_10MS 100000UL

void clint_timer_set_next(void) {
    // OpenSBI SBI_EXT_TIME (0x54494D45), sbi_set_timer
    uint64_t target = CLINT_MTIME + TICKS_10MS;
    sbi_call(SBI_EXT_TIME, 0, target, 0, 0, 0, 0);
    // Enable S-mode timer interrupt
    csr_set(sie, SIE_STIE);
}
```

The timer interrupt handler in FRISC OS does three things: calls the scheduler to select the next runnable task, re-arms the timer for the next 10 ms tick, and performs any pending kernel maintenance (clearing zombie tasks, aging memory). The scheduler decision is made entirely in the interrupt handler, which is acceptable because FRISC OS's scheduler is O(1) (circular doubly-linked run queue).

Students encounter the CLINT in the third week of the course, after completing the trap entry exercise. The assignment is to modify the timer period to 1 ms and observe the effect on scheduling granularity, then increase it to 100 ms and observe the effect on interactive responsiveness in the shell. This exercise builds intuition for the trade-off between scheduling frequency and interrupt overhead.

8. PLIC Interrupt Controller

The Platform-Level Interrupt Controller (PLIC) manages external interrupts (from devices) on RISC-V. On QEMU virt, the PLIC is at physical address 0x0c000000 and handles 53 interrupt sources (numbered 1-53). Each virtio device has a dedicated interrupt source number starting at 1.

The PLIC has three per-source registers: priority (offset 0x0 + 4*source), enable per context (offset 0x2000 + 0x80*context + 4*(source/32)), and threshold per context (offset 0x200000 + 0x1000*context). On QEMU virt, there are 2 * num_harts contexts: even contexts are M-mode, odd contexts are S-mode. FRISC OS uses S-mode contexts (context 1 for hart 0).

```
// PLIC initialization for virtio-blk on source 1
#define PLIC_BASE       0x0C000000UL
#define PLIC_PRIORITY(s)    (*(uint32_t*)(PLIC_BASE + 4*(s)))
#define PLIC_ENABLE(ctx,s)  (*(uint32_t*)(PLIC_BASE+0x2000+0x80*(ctx)+4*((s)/32)))
#define PLIC_THRESHOLD(ctx) (*(uint32_t*)(PLIC_BASE+0x200000+0x1000*(ctx)))
#define PLIC_CLAIM(ctx)     (*(uint32_t*)(PLIC_BASE+0x200004+0x1000*(ctx)))

void plic_init(void) {
    int ctx = 1; // hart 0, S-mode
    PLIC_THRESHOLD(ctx) = 0;     // accept all priorities
    for (int src = 1; src <= 8; src++) {
        PLIC_PRIORITY(src) = 1;  // priority 1 for virtio sources
        PLIC_ENABLE(ctx, src) |= (1 << (src % 32));
    }
    csr_set(sie, SIE_SEIE);      // enable external interrupt in S-mode
}
```

When the PLIC fires an external interrupt, the trap handler reads the PLIC_CLAIM register, which returns the highest-priority pending source number (or 0 if no interrupt is pending). After servicing the device interrupt, the handler writes the source number back to PLIC_CLAIM to complete the interrupt, allowing the PLIC to forward the next pending interrupt. Failing to complete an interrupt leaves the source permanently pending and prevents future interrupts from that source.

FRISC OS's interrupt dispatch table maps PLIC source numbers to device handler functions. When the virtio-blk driver (source 1) receives an interrupt, it reads the virtqueue used ring to find completed I/O requests and wakes the tasks that submitted them. This is the first example of sleep/wake coordination that students encounter in the course.

9. The Virtio Device Model

Virtio is a standard interface for para-virtualized devices in QEMU and other hypervisors. Virtio devices share a common negotiation protocol: the driver reads the device ID and feature bits, selects a subset of features to enable, and uses one or more virtqueues to exchange descriptors with the device. FRISC OS implements virtio over MMIO (the transport used by QEMU virt).

A virtqueue consists of three rings: a descriptor table, an available ring (driver to device), and a used ring (device to driver). To submit a request, the driver writes descriptor entries describing the request buffers, adds the head descriptor index to the available ring, and writes the queue notify register. The device processes the request and adds the head index plus bytes written to the used ring, then raises an interrupt.

```
// Virtqueue descriptor
typedef struct {
    uint64_t addr;   // physical address of buffer
    uint32_t len;    // buffer length in bytes
```

```
    uint16_t flags;  // NEXT=1, WRITE=2, INDIRECT=4
    uint16_t next;   // index of next descriptor (if NEXT)
} VirtqDesc;

// Available ring
typedef struct {
    uint16_t flags, idx;
    uint16_t ring[VIRTQ_SIZE]; // descriptor head indices
} VirtqAvail;

// Used ring
typedef struct {
    uint16_t flags, idx;
    struct { uint32_t id, len; } ring[VIRTQ_SIZE];
} VirtqUsed;
```

FRISC OS uses VIRTQ_SIZE = 64 for both the block and network queues. With a 4096-byte sector size and 64-entry queue, up to 256 KiB of I/O can be in flight simultaneously. For the network queue, each descriptor covers one Ethernet frame (up to 1514 bytes). The descriptor table, available ring, and used ring must be physically contiguous within each virtqueue; FRISC OS allocates them together in a single buddy-system allocation.

Feature negotiation uses a 64-bit bitmask. FRISC OS negotiates VIRTIO_F_VERSION_1 (feature bit 32), which enables the modern device layout. For the block device, FRISC OS also negotiates VIRTIO_BLK_F_SEG_MAX (bit 2, maximum scatter-gather segments = 2) and VIRTIO_BLK_F_BLK_SIZE (bit 6, block size in the config register). The network device negotiates VIRTIO_NET_F_MAC (bit 5, device has a valid MAC address).

10. Virtio-Blk Driver

The virtio-blk driver provides the block I/O layer used by the FAT32 filesystem. It supports two operations: read sector and write sector, each transferring 512 bytes (one sector). The driver uses a synchronous request model for simplicity: the submitting task sleeps until the interrupt handler wakes it with the completed result.

```
// Block request submission (simplified)
typedef struct {
    uint32_t type;    // 0=read, 1=write, 4=flush
    uint32_t reserved;
    uint64_t sector;
} VirtioBlkReq;

int blk_read(uint64_t sector, void *buf) {
    VirtioBlkReq req = { .type = 0, .sector = sector };
    uint8_t status = 0xFF; // device writes 0=ok, 1=err, 2=unsup
    // Desc 0: req header (read-only for device)
    // Desc 1: data buffer (write-only for device, VIRTQ_DESC_F_WRITE)
    // Desc 2: status byte (write-only, VIRTQ_DESC_F_WRITE)
    virtq_submit_chain(blk_queue, &req, buf, &status);
    task_sleep_on(&blk_wait_queue);
    return (status == 0) ? 0 : -EIO;
}
```

The interrupt handler for the virtio-blk device (PLIC source 1) reads the used ring to find completed requests. For each completed request, it wakes the task sleeping on blk_wait_queue. The task resumes execution in blk_read and checks the status byte written by the device. This pattern — submit, sleep, interrupt, wake — is the canonical device I/O pattern that students use as a template for other drivers.

FRISC OS's DMA buffer allocation uses a reserved region of physical memory below 4 GiB (required by virtio MMIO on QEMU virt) allocated at boot by the buddy system. The region holds 16 MiB and is large enough for the virtqueue descriptor tables, available/used rings, and I/O buffers for both block and network devices.

Error handling in the block driver distinguishes three cases: status 0 (success), status 1 (I/O error reported by device), and timeout (device fails to respond within 500 ms). The timeout is implemented by the timer interrupt handler, which checks a per-request deadline field and wakes the waiting task with an error code. This is the first exposure students have to timeout-based liveness in device drivers.

11. Virtio-Net Driver and FAT32 Filesystem

The virtio-net driver handles Ethernet frame transmission and reception. It maintains two virtqueues: receive queue (vq0) and transmit queue (vq1). Receive descriptors are pre-populated with 1514-byte buffers; when a frame arrives, the device fills a buffer and signals the interrupt. Transmit submits a frame in a two-descriptor chain: a 12-byte virtio-net header followed by the Ethernet frame payload.

```
// Virtio-net header (12 bytes)
typedef struct {
    uint8_t  flags;        // VIRTIO_NET_HDR_F_NEEDS_CSUM=1
    uint8_t  gso_type;     // 0=none, 1=TCPv4, 3=UDP, 4=TCPv6
    uint16_t hdr_len;      // Ethernet+IP+TCP header length
    uint16_t gso_size;     // GSO segment size
    uint16_t csum_start;   // offset to checksum start
    uint16_t csum_offset;  // offset to checksum field
    uint16_t num_buffers;  // number of merged rx buffers
} VirtioNetHdr;
```

FAT32 is the filesystem used on the QEMU virt disk image. FRISC OS implements a read-write FAT32 driver in 860 lines of C. The driver supports file creation, deletion, read, write, and directory enumeration. It does not implement long filename (LFN) entries for simplicity; all filenames are 8.3 uppercase. This limitation is acceptable for the course's purposes, as the shell and utilities use short filenames.

The FAT32 driver maintains a 64-entry block cache using a simple LRU replacement policy. The cache maps (device, sector_number) pairs to 4096-byte aligned buffers. Cache misses call the virtio-blk driver to fill the buffer. Write-back is triggered on cache eviction. The cache is not write-through because QEMU virt's disk does not have durable writes in the educational configuration, and the added complexity of write-through flushing is not valuable in a course that does not cover crash recovery.

The inode layer above FAT32 maps directory entries to in-memory inode_t structures allocated from the slab allocator. Each inode holds the FAT cluster chain start, file size, and a reference count. The VFS layer above the inode layer dispatches open/read/write/close operations to the FAT32 functions. Students implement a second filesystem (a simple in-memory tmpfs) in week 10 of the course by implementing the same VFS interface.

12. Cooperative Scheduler

FRISC OS introduces scheduling in two phases. Phase one, covering weeks 6-8 of the course, uses a cooperative scheduler. Tasks voluntarily yield the CPU by calling sched_yield(). The scheduler selects the next runnable task from a circular doubly-linked run queue using round-robin policy.

```
// Cooperative context switch (simplified)
void sched_yield(void) {
    Task *cur = current_task;
```

```
    Task *nxt = runq_next(cur); // circular list
    if (nxt == cur) return;     // only one runnable task
    current_task = nxt;
    switch_context(&cur->ctx, &nxt->ctx);
}

// Context: 14 callee-saved registers (s0-s11, ra, sp)
void switch_context(Context *from, Context *to) {
    // Save from->s0..s11, ra, sp
    // Load  to->s0..s11, ra, sp
    // Return to 'to' task at its saved ra
}
```

The cooperative scheduler requires only 14 registers to be saved and restored (the RISC-V callee-saved registers s0-s11 plus ra and sp). Caller-saved registers are not saved because the C calling convention guarantees they are not live across a function call. This 14-register context switch is the smallest possible context switch on RISC-V without floating-point state.

Students implement the run-queue data structure from scratch in week 6. The exercise requires inserting tasks into the queue in priority order and verifying that the scheduler selects the highest-priority runnable task. A set of pre-written test tasks (counting, sleeping, and spinning) are used to validate the implementation by observing output order.

A common bug in student cooperative scheduler implementations is forgetting that switch_context returns into the *new* task, not the old one. The first call to switch_context from a newly created task must return into the task's entry function. FRISC OS initializes the new task's saved ra to the entry function address and saved sp to the top of the task's kernel stack, so the first context switch correctly enters the task.

13. Preemptive Scheduler Upgrade

Weeks 9-11 of the course upgrade the cooperative scheduler to a preemptive one. The key change is that the timer interrupt handler may call sched_yield() at any point during user or kernel execution, not only when the running task calls it explicitly. This requires saving and restoring all 32 general-purpose registers plus the FPU state, not just the callee-saved registers.

```
// Preemptive scheduler: timer interrupt handler (excerpt)
void timer_interrupt_handler(TrapFrame *tf) {
    clint_timer_set_next();      // re-arm for next 10 ms tick
    Task *cur = current_task;
    if (--cur->timeslice == 0) {
        cur->timeslice = DEFAULT_TIMESLICE; // 5 ticks = 50 ms
        Task *nxt = runq_next(cur);
        if (nxt != cur) {
            current_task = nxt;
            // TrapFrame IS the full register save: switch by swapping
            // current_task; sret will restore nxt's saved pc and regs
        }
    }
}
```

The preemptive scheduler reuses the trap frame mechanism: because the trap entry code already saves all 32 registers on the kernel stack, switching tasks on a timer interrupt only requires changing current_task before the sret instruction. The trap exit code restores the new task's registers from its saved trap frame, which is stored at the base of its kernel stack.

Race conditions in the preemptive kernel require spinlocks. FRISC OS implements a ticket spinlock using the RISC-V AMO (atomic memory operation) instructions: amoadd.w to fetch-and-add the ticket counter, and a spin loop using lw + a memory barrier. Students encounter their first data race bug when two tasks concurrently modify the run queue without holding the scheduler lock, and use FRISC OS's built-in race detector (a debug-mode that inserts canary writes between shared structures) to diagnose it.

Priority inversion is demonstrated in week 11 with a scenario involving three tasks at priorities high, medium, and low, where low holds a mutex that high needs, and medium preempts low. FRISC OS's priority inheritance protocol (PI mutex) resolves the inversion by temporarily raising low's priority to high's while low holds the mutex. Students implement PI mutexes as the final assignment of the scheduling module.

14. SMP Basics

Symmetric multiprocessing (SMP) is introduced in week 14. FRISC OS supports up to 4 harts on QEMU virt (-smp 4). Secondary harts are started using the OpenSBI hart state management extension: the boot hart calls sbi_hart_start(hart_id, start_addr, opaque) for each secondary hart, passing the physical address of _start and the hart ID as opaque.

Each hart has its own kernel stack, its own CLINT timer, and its own PLIC context (context 2*hart+1 for S-mode). The global run queue is shared across all harts and protected by the scheduler spinlock. When a hart's current task exhausts its timeslice, the hart locks the run queue, pops the next runnable task, and unlocks the queue. If the queue is empty, the hart enters a WFI (wait-for-interrupt) loop.

```
// SMP secondary hart startup
void secondary_hart_main(uint64_t hart_id) {
    // Per-hart: set up trap vector, enable interrupts
    csr_write(stvec, (uint64_t)_trap_entry | 0x1); // vectored mode
    plic_init_hart(hart_id);
    clint_timer_set_next();        // start per-hart timer
    csr_set(sstatus, SSTATUS_SIE); // global interrupt enable
    // Join the scheduler loop
    while (1) {
        Task *t = runq_steal(); // work-steal from global queue
        if (t) { current_task = t; sched_enter(t); }
        else   { wfi(); }       // no work; sleep until interrupt
    }
}
```

Work stealing is optional in the FRISC OS SMP implementation: a simple global queue with a spinlock is used by default. Students who finish the main SMP assignment early are encouraged to implement per-hart run queues with work stealing as an extra credit exercise. The reference implementation uses a Chase-Lev deque, but any correct work-stealing implementation earns credit.

IPI (inter-processor interrupt) is used for TLB shootdown (see Section 4). The boot hart sends an IPI by writing the target hart mask to the CLINT msip register; the target hart's software interrupt handler performs the sfence.vma and sends an acknowledgement by clearing its own msip bit. The sender spins on the target's msip bit, waiting for it to clear. This simple protocol has no pipelining but is correct and easy to understand.

15. QEMU Virt Platform and Comparison with xv6

QEMU's virt machine is the reference platform for FRISC OS. It provides: an RV64GC hart (U74-compatible), 128 MiB DRAM starting at 0x80000000, a CLINT at 0x2000000, a PLIC at 0x0c000000, a UART16550A at 0x10000000, and up to 8 virtio-mmio devices at 0x10001000 through 0x10008000. All parameters are described in the DTB, which QEMU generates at boot.

xv6 (MIT, 2019) is the most widely used educational OS in the world. FRISC OS was designed as an alternative that makes different pedagogical choices. xv6-riscv is an excellent reference, but it uses a monolithic trap handler in assembly that is difficult to understand for students without prior OS experience. FRISC OS uses a C-first approach where the assembly stubs are minimal and all logic is in C functions.

- xv6-riscv: 7,000 LOC, UART console, simple piped filesystem, monolithic
- FRISC OS: 8,000 LOC, virtio-blk, FAT32, virtio-net, IP stack optional
- xv6 memory model: direct mapping, no buddy allocator
- FRISC OS memory model: buddy + slab, Sv39 with high kernel map
- xv6 scheduler: round-robin, cooperative-only in base
- FRISC OS scheduler: two-phase (cooperative then preemptive), SMP in week 14

FRISC OS explicitly teaches the virtio device model because virtio is the dominant virtualized device interface in cloud computing. A student who understands the virtio descriptor ring model can read the Linux virtio driver source and contribute to cloud hypervisor projects. xv6 uses a simulated UART with trivially simple register I/O, which does not transfer to modern device driver development.

The 18-week course that accompanies FRISC OS covers: weeks 1-2 (RISC-V assembly, toolchain), weeks 3-4 (boot and trap entry), weeks 5-6 (virtual memory, page tables), weeks 7-8 (physical memory, cooperative scheduler), weeks 9-11 (preemptive scheduling, synchronization), weeks 12-13 (device drivers, virtio), week 14 (filesystems), weeks 15-16 (SMP), weeks 17-18 (student project: network stack or custom filesystem). Each week has a 3-hour lab with autograded test cases run under QEMU.

16. Pedagogical Decisions and Limitations

FRISC OS makes several deliberate simplifications. It does not implement demand paging or page fault handlers for user pages; all user memory is mapped eagerly at task creation from a pool of pre-allocated pages. This avoids the complexity of page-fault handler interaction with the VM subsystem, which is considered too advanced for the target course level.

The network stack is optional and not part of the base kernel. Students who reach week 17 implement a minimal TCP/IP stack (IP, ICMP, UDP, and a subset of TCP) over the virtio-net driver. A reference implementation in 1,400 LOC is provided for comparison. The optional stack supports a single-threaded HTTP/1.0 server, which is the final demo of the course.

FRISC OS does not implement a memory-mapped file API, demand paging from disk, or ELF dynamic linking. These are considered graduate-level topics. All executables are statically linked ELF files loaded by a minimal ELF loader in the kernel. The ELF loader handles only static executables with PT_LOAD segments and no relocations.

Known bugs in the current (June 2025) release include: a race in the FAT32 write path when two tasks write to the same file concurrently (the directory entry is not locked), and a virtio-net descriptor ring exhaustion bug when network traffic exceeds 512 frames/second. Both bugs are documented as

exercises for advanced students; finding and fixing them counts as the extra-credit project in week 18.