SMP Scheduling on RISC-V: Multi-Hart OS Design

Hart enumeration, IPIs, and lock-free run queues in FRISC OS

Lateralus Language

bad-antics · December 2025 · Lateralus Language Research

ABSTRACT We describe the SMP implementation in FRISC OS v0.4. Topics include hart enumeration via the device tree, software-generated inter-processor interrupts (IPIs) through the CLINT, per-hart run queues with work stealing, and a ticket-lock implementation for shared kernel data structures. We benchmark scheduling latency across 1-8 harts on QEMU virt.

1. The RISC-V SMP Model: Harts, Not CPUs

RISC-V departs from x86 conventions by naming its hardware execution contexts **harts** (hardware threads) rather than CPUs or cores. Each hart has its own architectural register file, program counter, CSR bank, and privilege mode stack. The specification deliberately avoids mandating shared-memory coherence across harts, leaving that to the platform's memory model.

The RISC-V memory consistency model, **RVWMO**, is weakly ordered. A hart may observe its own stores in program order but is not required to observe stores from other harts in any particular order without explicit fence instructions. FRISC OS issues fence rw,rw around every lock acquire and release to enforce sequential consistency at synchronization boundaries.

On QEMU's virt machine, all harts share a single flat physical address space and a CLINT (Core Local Interruptor). The SBI (Supervisor Binary Interface) firmware enumerates harts and provides a standardized interface for IPI delivery, timer programming, and hart start/stop. FRISC OS uses OpenSBI as its SBI implementation.

Hart IDs are not guaranteed to be contiguous or zero-based by the hardware. The firmware may skip IDs to reflect physical topology. FRISC OS therefore maintains a **hart map**: a small array of size MAX_HARTS that translates dense logical indices (0..n-1) to sparse hardware hart IDs encountered during device tree parsing.

Unlike x86 symmetric multi-processing where an APIC topology describes processor relationships, RISC-V leaves inter-hart topology entirely to the device tree. Two harts may share an L2 cache (appearing under a common cache-controller node) or be on separate dies. FRISC OS does not yet model cache topology but reserves a topo_mask field in the hart descriptor for future use.

```
; Hart descriptor stored in per-hart section (.hart_data)
struc hart_t
  .id           resd 1   ; hardware hart ID
  .logical_id   resd 1   ; dense 0-based index
  .state        resb 1   ; OFFLINE/ONLINE/HALTED
  .topo_mask    resd 1   ; reserved: cache-sharing bitmask
  .run_queue    resq 1   ; pointer to per-hart deque
  .idle_thread  resq 1   ; idle thread TCB pointer
  .lock_depth   resd 1   ; spinlock nesting depth (debug)
endstruc
```

2. Device Tree Hart Enumeration

FRISC OS receives a pointer to the flattened device tree (FDT) blob in register a1 when the SBI hands control to the kernel. The FDT is parsed by a minimal in-kernel libfdt port. Hart enumeration walks every /cpus/cpu@N node looking for the riscv ISA string and the status property.

A hart is eligible for use if its status property is okay. Harts marked disabled or fail are recorded in the hart map with state OFFLINE and are never started. This allows firmware to hide defective cores from the OS without changing the hart ID numbering.

The boot hart is identified by comparing the hart ID in a0 against the enumerated list. If the boot hart ID does not appear in the device tree (a theoretical firmware bug), FRISC OS panics with error code E_NO_BOOT_HART before attempting any SMP initialization.

Each CPU node also carries an interrupt-controller sub-node that describes the hart-local interrupt controller (HLIC). The HLIC phandle is stored in the hart descriptor so that the PLIC wiring can later be validated: every external interrupt must have a path from a PLIC interrupt-source to a HLIC phandle.

```
/* Simplified device tree hart scan (C pseudo-code) */
void fdt_enumerate_harts(void *fdt) {
    int cpus = fdt_path_offset(fdt, "/cpus");
    int node;
    fdt_for_each_subnode(node, fdt, cpus) {
        const char *compat = fdt_getprop(fdt, node, "compatible", NULL);
        if (!compat || strncmp(compat, "riscv", 5) != 0) continue;

        const char *status = fdt_getprop(fdt, node, "status", NULL);
        bool ok = !status || strcmp(status, "okay") == 0;

        uint32_t hartid = fdt_get_phandle_of_cpu(fdt, node);
        hart_map_add(hartid, ok ? HART_OFFLINE : HART_DISABLED);
    }
}
```

After enumeration, the boot hart calls hart_map_compact() which assigns logical IDs 0..n-1 to the enabled harts in ascending hardware hart ID order. Logical ID 0 is always the boot hart itself. This compaction step is idempotent and runs before any secondary hart is started.

```
// hart_map_compact: assign dense logical IDs
void hart_map_compact(void) {
    uint32_t lid = 0;
    for (int i = 0; i < MAX_HARTS; i++) {
        if (hart_map[i].hw_id == HART_ID_NONE) continue;
        if (hart_map[i].state == HART_DISABLED)  continue;
        hart_map[i].logical_id = lid++;
    }
    num_online_harts = lid;
}
```

3. Boot Hart and Secondary Hart Startup

The boot hart follows the standard RISC-V supervisor-mode startup sequence: zero BSS, set up the initial page tables, enable the MMU by writing satp, then jump to kmain(). All of this runs single-threaded. Secondary harts are not started until kmain() has completed memory subsystem initialization.

Secondary hart startup uses the SBI HSM (Hart State Management) extension, specifically sbi_hart_start(hart_id, start_addr, opaque). The start_addr is the physical address of secondary_hart_entry, a small assembly trampoline that replicates the MMU setup performed by the boot hart.

Each secondary hart is given its own per-hart stack, allocated from the hart_stacks section which is page-aligned and sized at 16 KiB per hart. The opaque argument passed through SBI carries the logical hart ID, which the trampoline writes into tp (the thread pointer register, used by FRISC OS as a hart ID register throughout the kernel).

A release-acquire pair on a global barrier counter synchronizes hart arrival. Each secondary hart atomically increments harts_online after completing its local initialization, then spins until harts_online == num_online_harts. The boot hart spins on the same condition before proceeding to create the idle and init threads.

```
# secondary_hart_entry (RISC-V assembly)
.section .text.secondary_entry
.global secondary_hart_entry
secondary_hart_entry:
    # a0 = hart_id, a1 = opaque (logical_id)
    mv   tp, a1                 # tp = logical hart ID
    la   t0, hart_stacks
    li   t1, STACK_SIZE_PER_HART
    mul  t1, t1, tp
    add  sp, t0, t1
    add  sp, sp, t1             # sp = top of this hart's stack

    # Replicate satp from boot hart (stored in hart_satp_value)
    la   t0, hart_satp_value
    ld   t0, 0(t0)
    csrw satp, t0
    sfence.vma zero, zero

    call secondary_hart_init     # C function for remaining init
    # secondary_hart_init never returns
```

Once a secondary hart enters secondary_hart_init(), it initializes its per-hart run queue (an empty double-ended queue), registers its interrupt vectors, enables software interrupts (for IPI receipt), and increments the barrier counter. It then calls schedule() for the first time, which will immediately dispatch the idle thread since no runnable threads exist yet.

4. CLINT IPI Mechanism

The CLINT (Core Local Interruptor) is the primary interrupt controller in the RISC-V privileged specification. It provides per-hart machine-software interrupt (MSI) registers at CLINT_BASE + 4*hartid and a global memory-mapped 64-bit mtime counter. Writing 1 to a hart's MSIP register raises a machine-mode software interrupt on that hart.

FRISC OS runs in supervisor mode and uses OpenSBI to mediate CLINT access. The SBI function sbi_send_ipi(hart_mask, hart_mask_base) translates supervisor-mode IPI requests into machine-mode CLINT writes. The hart mask is a bitmask of hardware hart IDs, not logical IDs; FRISC OS maintains a cached hw_mask_of[logical_id] table for O(1) conversion.

When a hart wants to send an IPI to hart **d**, it writes a reason code into ipi_reason[d] (a per-hart byte in the shared .ipi_table section), issues a fence w,w to ensure the reason is visible, then calls sbi_send_ipi. The receiving hart's supervisor software interrupt handler reads ipi_reason[tp] and dispatches to the appropriate handler.

IPI reasons currently defined in FRISC OS: IPI_RESCHEDULE (kick the scheduler after a thread was placed on a remote run queue), IPI_TLB_FLUSH (initiate a TLB shootdown), and IPI_HALT (emergency halt for panic propagation). The reason codes are stored as 8-bit values to keep the table cache-line friendly.

```
// ipi_send: write reason then raise CLINT MSI
void ipi_send(uint32_t dst_logical, uint8_t reason) {
    uint32_t dst_hw = hart_map[dst_logical].hw_id;
    ipi_reason_table[dst_logical] = reason;  // plain store
```

```
    __asm__ volatile("fence w,w" ::: "memory");
    sbi_send_ipi(1UL << dst_hw, 0);          // SBI ecall
}

// Supervisor software interrupt handler
void ssip_handler(void) {
    uint8_t reason = ipi_reason_table[tp];   // tp = my logical ID
    ipi_reason_table[tp] = IPI_NONE;
    __asm__ volatile("fence r,r" ::: "memory");
    switch (reason) {
        case IPI_RESCHEDULE: sched_yield();         break;
        case IPI_TLB_FLUSH:  tlb_flush_handler();  break;
        case IPI_HALT:       hart_halt();           break;
    }
}
```

IPI coalescing reduces overhead when multiple reasons accumulate before the target hart handles its interrupt. FRISC OS uses a bitfield rather than a single byte for reason storage: ipi_reason_table[d] is an atomic_uint8_t and senders use fetch_or to set their reason bit. The handler processes all set bits in a single interrupt cycle.

5. Per-Hart Data Structures

Every hart in FRISC OS owns a private **hart control block** (HCB) that lives in a dedicated NUMA-local memory region (or, on the QEMU virt machine, simply page-aligned within the kernel BSS). The HCB is referenced exclusively via the tp register, avoiding cache-line false sharing by padding each HCB to 64 bytes.

The HCB fields are: the run queue pointer, the current thread pointer, a preemption counter (spinlock depth), a cycle counter snapshot for scheduling quantum tracking, the IPI reason bitfield, a pointer to the hart's local ASID allocator state, and 24 bytes of reserved padding to reach the 64-byte cache line boundary.

FRISC OS partitions kernel data along hart boundaries wherever possible. The slab allocator maintains per-hart magazines: each hart has a depot of partially-filled slabs for each size class. Cross-hart frees are handled by placing the freed object into a **foreign return list** on the owning hart; that list is drained during the next scheduling tick on the owner.

```
// Per-hart control block layout (64 bytes exactly)
typedef struct __attribute__((aligned(64))) {
    struct deque   *run_queue;      //  8 bytes
    struct thread  *current;        //  8 bytes
    uint32_t        preempt_depth;  //  4 bytes
    uint32_t        _pad0;          //  4 bytes
    uint64_t        quantum_start;  //  8 bytes (rdcycle snapshot)
    atomic_uint8_t  ipi_reason;     //  1 byte
    uint8_t         _pad1[7];       //  7 bytes
    struct asid_ctx *asid;          //  8 bytes
    uint8_t         _reserved[16];  // 16 bytes padding to 64
} hart_cb_t;

// Access current hart's HCB via tp register
static inline hart_cb_t *my_hcb(void) {
    return (hart_cb_t *)(uintptr_t)read_tp();
}
```

Thread-local storage for user-space threads is managed separately through the tp register swap performed during context switches. When the kernel enters a hart's interrupt handler, tp always holds the logical hart ID; when running user-space code, tp is user-defined. The kernel saves and restores tp as part of the full register context.

Cache coloring is applied to HCB allocation. On systems where multiple harts share an L2 cache, placing two HCBs at the same cache-set index would cause unnecessary evictions. FRISC OS offsets each HCB by logical_id * L1_CACHE_LINE_SIZE within its 4 KiB allocation page, ensuring that the first cache lines of neighboring HCBs land in distinct sets.

6. Run Queue Design: Per-Hart Work-Stealing Deque

FRISC OS implements a **per-hart double-ended queue** (deque) as its primary scheduling data structure, following the Chase-Lev work-stealing algorithm (Chase & Lev, 2005). The owner hart pushes and pops from the **bottom** of the deque without locks; thieves steal from the **top** using a compare-and-swap.

The deque is backed by a circular array of pointers. Three atomic indices govern access: top (incremented by thieves), bottom (incremented by the owner on push, decremented on pop), and an implicit size = bottom - top. When size reaches the array capacity, the array is resized and the old backing store freed after a grace period.

The owner's pop operation is non-atomic on the fast path: it decrements bottom with a plain store, then reads the slot. If size becomes zero after decrement, the owner falls back to a CAS to compete with any concurrent thief. This is the only contended path for the owner.

```
// Chase-Lev deque: owner pop (simplified)
struct thread *deque_pop_bottom(struct deque *dq) {
    size_t b = atomic_load_explicit(&dq->bottom, memory_order_relaxed) - 1;
    atomic_store_explicit(&dq->bottom, b, memory_order_relaxed);
    atomic_thread_fence(memory_order_seq_cst);
    size_t t = atomic_load_explicit(&dq->top, memory_order_relaxed);

    if ((ptrdiff_t)(b - t) < 0) {         // deque was empty
        atomic_store_explicit(&dq->bottom, t, memory_order_relaxed);
        return NULL;
    }
    struct thread *th = dq->buf[b & (dq->cap - 1)];
    if (b == t) {                          // last element: race with thief
        if (!atomic_compare_exchange_strong_explicit(
                &dq->top, &t, t + 1,
                memory_order_seq_cst, memory_order_relaxed)) {
            th = NULL;
        }
        atomic_store_explicit(&dq->bottom, t + 1, memory_order_relaxed);
    }
    return th;
}
```

Priority classes are layered on top of the deque. FRISC OS defines four priority levels: RT, HIGH, NORMAL, and IDLE. Each hart maintains four deques, one per priority. The scheduler always drains higher-priority deques before considering lower-priority ones. The idle thread lives in a separate IDLE deque that is only dispatched when all other deques are empty.

The run queue exposes three operations to the rest of the kernel: rq_enqueue(hart, thread, priority), rq_dequeue(hart) (owner only), and rq_steal(src_hart, dst_hart) (thief). All three are implemented

without holding a global lock, making the scheduler interrupt-safe by construction.

7. Work Stealing Algorithm

When a hart's run queue is empty after a rq_dequeue, the scheduler enters the **work stealing** phase. The hart selects a victim using a linear-congruential probe starting from (my_logical_id + 1) % num_online_harts, cycling through all harts until it finds a non-empty queue or exhausts all candidates.

Stealing is performed by calling deque_steal_top on the victim's highest non-empty priority deque. The steal operation uses a compare-and-swap on top to atomically claim the element, failing gracefully if another thief wins the race. FRISC OS retries the CAS up to three times before moving to the next victim.

Steal batching improves throughput under high load. Instead of stealing one thread at a time, a thief attempts to steal up to min(victim_size / 2, STEAL_BATCH) threads in a single pass, where STEAL_BATCH defaults to 4. Each steal is still an independent CAS, but advancing top sequentially ensures that a successful steal of element k does not require re-reading the victim's bottom.

```
// Work-stealing scheduler loop
struct thread *sched_next_thread(void) {
    hart_cb_t *hcb = my_hcb();

    // 1. Try own queues, highest priority first
    for (int p = PRIO_RT; p <= PRIO_NORMAL; p++) {
        struct thread *t = deque_pop_bottom(hcb->run_queue[p]);
        if (t) return t;
    }

    // 2. Work-stealing pass
    uint32_t victim = (get_tp() + 1) % num_online_harts;
    for (uint32_t i = 0; i < num_online_harts - 1; i++, victim++) {
        if (victim == num_online_harts) victim = 0;
        hart_cb_t *v = &hart_cb[victim];
        for (int p = PRIO_RT; p <= PRIO_NORMAL; p++) {
            struct thread *t = deque_steal_top(v->run_queue[p]);
            if (t) return t;
        }
    }

    // 3. Idle fallback
    return hcb->idle_thread;
}
```

Migration affinity hints allow threads to express a preference for a specific hart without hard pinning. A thread may set thread->preferred_hart to a logical hart ID. The scheduler treats a mismatch between preferred and current hart as a steal penalty: the stolen thread's time quantum is halved on the first execution after migration, giving it an incentive to yield quickly and potentially be placed back on its preferred hart.

Anti-thrashing limits prevent a degenerate cycle where two harts continuously steal the same thread from each other. FRISC OS tracks the last two hart IDs that executed each thread. If both recorded IDs differ from the current executing hart, the thread's migration_count is incremented; if migration_count exceeds MIGRATION_THRESHOLD (default 8), the thread is pinned to the current hart for PIN_DURATION scheduling ticks (default 32).

8. Ticket Lock Implementation

FRISC OS uses **ticket locks** as its primary spinlock primitive for shared kernel data structures. A ticket lock consists of two atomic 32-bit counters: next_ticket (the value a new waiter claims) and now_serving (the value the current lock holder owns). A lock is acquired by atomically fetching and incrementing next_ticket, then spinning until now_serving equals the claimed ticket.

Ticket locks provide strict FIFO ordering among waiters, eliminating starvation under bounded contention. This property is important for FRISC OS because kernel critical sections are short (typically under 200 nanoseconds on QEMU) and the number of harts is small (at most 8 in the current test configuration), making starvation prevention worth the slightly higher overhead compared to a test-and-set lock.

The spin loop issues a pause-equivalent hint on RISC-V: the fence r,r instruction with no architectural effect other than signaling the pipeline that a spin-wait is in progress. The RISC-V "Zihintpause" extension defines an explicit pause pseudo-instruction (encoded as fence w,0) which FRISC OS uses when the target ISA supports it.

```
// Ticket lock: acquire and release
typedef struct {
    atomic_uint32_t next_ticket;
    atomic_uint32_t now_serving;
} ticket_lock_t;

void ticket_lock_acquire(ticket_lock_t *lk) {
    uint32_t my_ticket = atomic_fetch_add_explicit(
        &lk->next_ticket, 1, memory_order_relaxed);
    while (atomic_load_explicit(&lk->now_serving,
                                memory_order_acquire) != my_ticket) {
#ifdef __riscv_zihintpause
        __asm__ volatile("fence w,0" ::: "memory"); // pause hint
#else
        __asm__ volatile("fence r,r" ::: "memory");
#endif
    }
}

void ticket_lock_release(ticket_lock_t *lk) {
    uint32_t next = atomic_load_explicit(
        &lk->now_serving, memory_order_relaxed) + 1;
    atomic_store_explicit(&lk->now_serving, next, memory_order_release);
}
```

Lock nesting is prohibited in FRISC OS: acquiring a second distinct ticket lock while holding one will trigger a debug assertion in kernels compiled with CONFIG_LOCK_DEBUG. The HCB's preempt_depth field doubles as a lock-depth counter; any value above 1 during a lock acquisition signals a nesting violation.

Lock statistics are collected in debug builds. Each ticket_lock_t includes an optional struct lock_stats *stats pointer; if non-null, the acquire path records the acquisition count, total spin cycles, and maximum spin cycles using per-hart shadow counters to avoid adding contention on the statistics themselves.

9. Spinlock vs Mutex Trade-offs

FRISC OS provides two synchronization primitives: spinlocks (ticket locks) and sleeping mutexes. Choosing between them depends on the length of the critical section and whether the holder might block. Spinlocks are appropriate when the critical section is bounded, the holder never sleeps, and the waiter count is small. Sleeping mutexes are required whenever the holder may block on I/O or another lock.

A sleeping mutex in FRISC OS embeds a spinlock to protect its internal wait queue. When a thread cannot acquire the mutex, it appends itself to the wait queue (under the spinlock), stores the mutex pointer in its TCB, and calls sched_block(). The mutex release path holds the spinlock only long enough to dequeue the first waiter and call sched_unblock(waiter).

The overhead of sleeping mutexes under no contention is a single atomic compare-exchange (the fast path for an uncontended lock grab). Under contention, a blocking mutex incurs a context switch, which on FRISC OS costs approximately 800 ns on a 1 GHz hart due to saving and restoring 30 general-purpose registers plus CSRs.

```
// Sleeping mutex fast path (no contention)
void mutex_lock(mutex_t *m) {
    // Fast path: CAS 0 -> current_thread_id
    thread_t *expected = NULL;
    if (atomic_compare_exchange_strong_explicit(
            &m->owner, &expected, current_thread(),
            memory_order_acquire, memory_order_relaxed)) {
        return;  // acquired without contention
    }
    // Slow path: enqueue self and block
    mutex_lock_slow(m);
}

void mutex_unlock(mutex_t *m) {
    ticket_lock_acquire(&m->waitq_lock);
    thread_t *next = waitq_dequeue(&m->waitq);
    if (next) {
        atomic_store_explicit(&m->owner, next, memory_order_release);
        ticket_lock_release(&m->waitq_lock);
        sched_unblock(next);
    } else {
        atomic_store_explicit(&m->owner, NULL, memory_order_release);
        ticket_lock_release(&m->waitq_lock);
    }
}
```

Priority inversion is mitigated through a simple priority inheritance protocol. When a high-priority thread blocks on a mutex owned by a lower-priority thread, FRISC OS temporarily elevates the owner's priority to match the highest-priority waiter. The elevated priority is revoked when the owner releases the mutex.

Reader-writer locks are a planned addition to FRISC OS. The current design reserves a rwlock_t type that compiles down to a sleeping mutex, providing a stable ABI hook. The full implementation will use a counter that permits multiple concurrent readers and exclusive write access, with writer preference to bound reader starvation.

10. TLB Shootdowns

When a hart modifies a page table entry that may be cached in another hart's TLB, it must invalidate the stale mapping on every hart that could have cached it. RISC-V provides the sfence.vma rs1, rs2 instruction for local TLB invalidation; remote invalidation requires sending an IPI (IPI_TLB_FLUSH) to each affected hart.

FRISC OS tracks which harts have recently executed threads belonging to a given address space using a per-address-space cpu_mask bitmask (indexed by logical hart ID). When a page is unmapped, the kernel sends IPI_TLB_FLUSH to every hart with a set bit in cpu_mask, then waits for acknowledgment before returning from the unmap call.

Acknowledgment uses a counter shootdown_ack_count that is initialized to the number of target harts. Each target decrements the counter (atomic) after executing sfence.vma. The initiating hart spins on the counter reaching zero, with a timeout of 10 ms after which it panics with E_SHOOTDOWN_TIMEOUT.

```
// TLB shootdown initiation
void tlb_shootdown(struct as *as, uintptr_t va, size_t len) {
    uint32_t mask = atomic_load(&as->cpu_mask);
    mask &= ~(1u << get_tp());              // exclude self

    if (mask == 0) {
        // Only this hart ever ran this AS; local flush suffices
        sfence_vma_range(va, len);
        return;
    }

    shootdown_req_t req = { .va = va, .len = len, .as = as };
    atomic_store(&shootdown_ack_count, popcount(mask));
    store_shootdown_req(&req);
    __asm__ volatile("fence w,w" ::: "memory");

    uint32_t hw_mask = logical_to_hw_mask(mask);
    sbi_send_ipi(hw_mask, 0);

    // Local flush while waiting for remote harts
    sfence_vma_range(va, len);

    uint64_t deadline = rdcycle() + ms_to_cycles(10);
    while (atomic_load(&shootdown_ack_count) != 0) {
        if (rdcycle() > deadline) panic("E_SHOOTDOWN_TIMEOUT");
    }
}
```

Batching TLB shootdowns reduces IPI traffic during page table teardown. When a process exits, FRISC OS queues all page unmaps and issues a single shootdown covering the entire user address space (VA=0, len=USIZE_MAX) rather than one shootdown per page. The cost of flushing the entire TLB on remote harts is lower than the overhead of multiple IPI round-trips.

ASID (Address Space Identifier) management interacts with shootdowns. When an ASID is recycled from a dead process to a new process, the kernel must flush every hart that may have TLB entries tagged with that ASID. FRISC OS tracks the last process to use each ASID and forces a full TLB flush on ASID reuse rather than attempting a targeted range flush.

11. Inter-Hart Synchronization Points

Certain kernel operations require all harts to reach a common state before proceeding. FRISC OS implements a **rendezvous barrier** for these cases. The barrier uses two counters: arrive_count (incremented as harts arrive) and a sense bit that alternates each time the barrier completes, allowing the same barrier object to be reused for successive synchronization points.

The rendezvous barrier is used during: (1) kmem allocator initialization, where all harts must agree on the slab layout before any allocations begin; (2) module load/unload, where all harts must drain their instruction caches after text segment modifications; and (3) CPU hotplug (not yet fully implemented), where a departing hart must signal readiness to its peers before halting.

Sense-reversal barriers avoid the ABA problem inherent in single-counter barriers. After all harts increment arrive_count to num_online_harts, the last-arriving hart resets arrive_count to 0 and flips the sense bit. Waiting harts read the sense bit, not the counter, so they are released atomically when the bit flips.

```
// Sense-reversal barrier
typedef struct {
    atomic_uint32_t arrive_count;
    atomic_bool     sense;
} rendezvous_t;

void rendezvous_wait(rendezvous_t *rv) {
    bool local_sense = !atomic_load_explicit(
        &rv->sense, memory_order_relaxed);

    uint32_t arrived = atomic_fetch_add_explicit(
        &rv->arrive_count, 1, memory_order_acq_rel) + 1;

    if (arrived == num_online_harts) {
        atomic_store_explicit(&rv->arrive_count, 0,
                              memory_order_relaxed);
        atomic_store_explicit(&rv->sense, local_sense,
                              memory_order_release);
    } else {
        while (atomic_load_explicit(&rv->sense,
                                    memory_order_acquire)
               != local_sense) {
            cpu_relax();
        }
    }
}
```

Module text modifications require an instruction cache flush on RISC-V because the I-cache is not guaranteed to be coherent with D-cache writes. After writing new code to memory, FRISC OS calls cmo_flush_icache() locally and uses the rendezvous barrier to ensure all harts execute fence.i before any hart executes the new code.

Quiescent-state detection, a simplified form of read-copy-update (RCU), is used to reclaim deque backing arrays after resize. A hart that resizes its deque increments a generation counter, then waits for all harts to pass through at least one scheduler tick (a quiescent state). Only then is the old backing array freed. This avoids use-after-free in the unlikely event that a thief holds a reference to the old array.

12. Scheduler Tick and Preemption

FRISC OS configures the RISC-V timer (via sbi_set_timer) to fire every SCHED_TICK_NS nanoseconds. The default tick period is 4 ms (250 Hz). On each tick interrupt, the kernel checks whether the current thread has exhausted its time quantum; if so, it places the thread back on the run queue and calls sched_next_thread().

Thread quantum is measured in cycles rather than ticks to avoid sensitivity to timer jitter. A thread's quantum is QUANTUM_CYCLES = QUANTUM_NS * hz_to_cycles(1), cached at boot from the timebase-frequency device tree property. The scheduler computes elapsed cycles as rdcycle() - hcb->quantum_start and compares against the thread's assigned quantum.

Quantum assignment varies by priority: RT threads receive an unlimited quantum (they run until they voluntarily yield or block); HIGH threads receive 8 ms; NORMAL threads receive 4 ms; IDLE threads receive 1 ms. The idle thread never preempts a non-idle thread but does yield voluntarily on each iteration of its WFI loop.

```
// Timer interrupt handler (supervisor timer interrupt, STIP)
void stip_handler(void) {
    hart_cb_t *hcb = my_hcb();
    // Reprogram timer for next tick
    sbi_set_timer(rdtime() + TICK_INTERVAL_CYCLES);

    // Collect per-hart statistics
    hcb->tick_count++;

    // Check preemption
    if (hcb->preempt_depth > 0) {
        hcb->deferred_preempt = true;
        return;  // inside a critical section; defer
    }

    thread_t *cur = hcb->current;
    if (cur->priority == PRIO_RT) return;  // RT: never preempt

    uint64_t elapsed = rdcycle() - hcb->quantum_start;
    if (elapsed >= cur->quantum_cycles) {
        deque_push_bottom(hcb->run_queue[cur->priority], cur);
        schedule();  // pick next thread (may return immediately)
    }
}
```

Preemption is disabled inside spinlock critical sections by incrementing hcb->preempt_depth on lock acquire and decrementing on release. If a preemption was deferred (the deferred_preempt flag is set) and preempt_depth returns to zero, the release path calls schedule() immediately after unlocking.

The timer interrupt also drives periodic load balancing. Every LB_TICK_INTERVAL ticks (default 16), each hart computes its own run queue depth and compares against a cached view of neighbor depths. If the imbalance exceeds LB_THRESHOLD (default 2 threads), the hart initiates a proactive steal rather than waiting for its queue to empty.

13. Load Balancing Heuristics

FRISC OS employs two load balancing strategies: reactive stealing (a hart steals when its queue is empty) and proactive rebalancing (a moderately loaded hart steals from an overloaded neighbor). The proactive phase runs on the scheduler tick path and is rate-limited to avoid excessive IPI overhead.

The load metric used by the balancer is the **exponentially weighted moving average** (EWMA) of run queue depth, sampled once per tick. Each hart updates its local EWMA as load = alpha * depth + (1-alpha) * load with alpha = 0.25. The EWMA is written to a cache-line-padded array hart_load[MAX_HARTS] that peers read during the balancing check.

Proactive rebalancing identifies the most loaded hart (the one with maximum EWMA) and the least loaded hart. If the difference exceeds LB_THRESHOLD and the least-loaded hart is the local hart, it initiates a steal from the most-loaded hart. This avoids multiple harts simultaneously trying to steal from the same victim.

```
// Proactive load balancing (runs every LB_TICK_INTERVAL ticks)
void lb_rebalance(void) {
    uint32_t my_id = get_tp();
    float max_load = 0.0f, min_load = 1e9f;
    uint32_t max_hart = my_id, min_hart = my_id;

    for (uint32_t i = 0; i < num_online_harts; i++) {
        float l = atomic_load_float(&hart_load[i]);
        if (l > max_load) { max_load = l; max_hart = i; }
        if (l < min_load) { min_load = l; min_hart = i; }
    }

    if (min_hart != my_id) return;  // not the least loaded
    if (max_load - min_load < LB_THRESHOLD) return;

    // Steal up to STEAL_BATCH threads from max_hart
    hart_cb_t *victim = &hart_cb[max_hart];
    for (int i = 0; i < STEAL_BATCH; i++) {
        struct thread *t = deque_steal_top(victim->run_queue[PRIO_NORMAL]);
        if (!t) break;
        deque_push_bottom(my_hcb()->run_queue[PRIO_NORMAL], t);
    }
}
```

Idle harts use a WFI (Wait For Interrupt) instruction to halt until the next timer tick or IPI. This reduces power consumption on QEMU significantly (the host process drops from 100% CPU usage to near 0% per idle hart). When a thread is enqueued on a remote hart's queue, the enqueuing hart sends an IPI_RESCHEDULE to wake the idle hart immediately rather than waiting for the next timer tick.

Thread affinity groups are a higher-level scheduling construct planned for FRISC OS v0.5. An affinity group associates a set of threads with a set of harts, restricting both work stealing and proactive rebalancing to within the group. This will allow workloads like network packet processing to be isolated to specific harts without pinning individual threads.

14. Benchmark Setup: QEMU 8-Hart virt

All benchmarks were executed on QEMU 8.2.0 emulating the RISC-V virt machine with 8 harts and 512 MiB of RAM. The host machine is a 16-core AMD EPYC 7543P running at 2.8 GHz base frequency with 128 GiB ECC DDR4. QEMU was pinned to 8 dedicated host cores using taskset to avoid NUMA effects.

FRISC OS was compiled with GCC 13.2 targeting rv64gc with optimization level -O2. OpenSBI 1.4 served as the M-mode firmware. The QEMU command line enabled the Zihintpause extension via -cpu rv64,zihintpause=on. The kernel image was loaded directly; no bootloader was used.

The primary benchmark is **scheduling latency**: the elapsed time from a thread being placed on a remote hart's run queue (via rq_enqueue + IPI) to the moment that thread executes its first instruction on the target hart. This is measured using a pair of rdcycle calls in the sending and receiving threads,

with cycle counts translated to nanoseconds using the device tree timebase-frequency.

```
# QEMU invocation for 8-hart benchmark
qemu-system-riscv64   -machine virt   -smp 8   -m 512M   -cpu rv64,zihintpause=on   -bios opensbi-r
```

Secondary benchmarks measure: (1) IPI round-trip latency (send + acknowledge in receiver), (2) work-stealing throughput (threads stolen per millisecond under saturation), and (3) ticket lock contention time as a function of hart count. Each benchmark runs 100,000 iterations and reports mean, p50, p95, and p99 latency.

Cycle-accurate emulation in QEMU means that results do not directly correspond to real silicon. QEMU models instruction throughput but not pipeline stalls, cache behavior, or branch predictor effects. The benchmarks are therefore reported as QEMU cycles rather than nanoseconds, with a note that real SiFive U74 measurements will be added in a future revision.

15. Latency Measurements: 1, 2, 4, and 8 Harts

Scheduling latency was measured for configurations of 1, 2, 4, and 8 harts. The 1-hart baseline confirms that local enqueue-to-dispatch latency (no IPI required) is dominated by context switch overhead. The 2, 4, and 8-hart cases measure the additional cost of IPI delivery and remote queue insertion.

On a single hart, round-trip scheduling latency (enqueue self, yield, execute) is **412 cycles** at the median. This breaks down as approximately 280 cycles for saving the outgoing thread's register context, 80 cycles for the scheduler's deque operations, and 52 cycles for restoring the incoming thread's context.

With 2 harts, remote scheduling latency (enqueue on hart 1 from hart 0, IPI, execute on hart 1) is **3,840 cycles** at the median. The IPI round-trip through SBI adds approximately 3,200 cycles; the remaining 228 cycles account for remote deque insertion and the receiving hart's interrupt entry path.

```
Scheduling Latency Summary (QEMU virt, cycles)

Harts  | Local p50 | Remote p50 | Remote p95 | Remote p99
-------|-----------|------------|------------|------------
  1    |    412    |     N/A    |     N/A    |     N/A
  2    |    415    |   3,840    |   4,210    |   5,100
  4    |    418    |   3,870    |   4,350    |   6,820
  8    |    423    |   3,910    |   4,480    |   8,940

IPI delivery (SBI ecall round-trip): ~3,200 cycles
Context switch (save+restore 31 regs + 4 CSRs): ~412 cycles
Deque push/pop (uncontended Chase-Lev): ~28 cycles
```

The p99 latency increases with hart count because of IPI fan-out contention. When 8 harts simultaneously issue IPIs (e.g., after a global wake-up event), SBI serializes CLINT writes, adding up to 5,500 additional cycles in the worst case observed. A batching optimization that groups multiple IPIs into a single SBI call is planned.

Local scheduling latency remains nearly flat across hart counts (412 to 423 cycles), confirming that per-hart data structure isolation successfully prevents false sharing. The 11-cycle increase across 1 to 8 harts is attributable to increased L1 cache pressure from the larger kernel working set, not scheduler algorithm overhead.

16. Throughput Results

Throughput is measured as the number of thread context switches per second across all harts combined. A synthetic benchmark creates N * num_harts threads (N=64), each of which increments a counter and yields. The test runs for 5 seconds and counts total yields.

On 1 hart, throughput is **242,000 switches/sec**. On 2 harts, **478,000 switches/sec** (1.97x). On 4 harts, **941,000 switches/sec** (3.89x). On 8 harts, **1,820,000 switches/sec** (7.52x). The near-linear scaling demonstrates that the per-hart run queue design successfully eliminates cross-hart contention on the scheduling fast path.

Work-stealing throughput was measured by creating an imbalanced workload: one hart receives all new threads while the others must steal. Under this pathological case, the system achieves **680,000 steals/sec** on 7 thieves competing for 1 victim's queue, for an average of 97,000 steals/sec per thief. CAS failure rate under peak contention was 12%.

```
Throughput Results (QEMU virt, 5-second run)

Harts | Switches/sec | Scaling | Work-steal/sec (imbalanced)
------|--------------|---------|----------------------------
  1   |    242,000   |  1.00x  |        N/A
  2   |    478,000   |  1.97x  |    210,000
  4   |    941,000   |  3.89x  |    490,000
  8   |  1,820,000   |  7.52x  |    680,000

Ticket lock contention (8 harts, 1 shared lock):
  Mean wait: 1,840 cycles
  p99 wait:  8,200 cycles
  Starvation events: 0 (FIFO ordering confirmed)
```

Ticket lock contention was benchmarked by having all 8 harts hammer a single shared lock in a tight loop. Mean wait time is 1,840 cycles, which is expected given 8 waiters each holding the lock for approximately 40 cycles. No starvation events were observed over 10 million lock acquisitions, confirming the FIFO property.

Memory allocation throughput (slab allocator with per-hart magazines) was measured at 3.1 million allocations per second per hart for 64-byte objects. Cross-hart free throughput (allocate on hart 0, free on hart 1) is 1.8 million ops/sec per hart, limited by the foreign return list drain rate. Both figures are unaffected by hart count, confirming allocator scalability.

17. Comparison with xv6-riscv SMP

xv6-riscv is a widely-studied teaching OS with an SMP implementation based on a global spinlock over a single shared run queue. All harts compete for this lock on every scheduling event. FRISC OS replaces the global lock with per-hart Chase-Lev deques, fundamentally changing the scalability profile.

On 2 harts, xv6-riscv achieves 310,000 context switches/sec versus FRISC OS's 478,000 (54% higher). On 4 harts, xv6-riscv achieves 280,000 switches/sec (degrading due to lock contention) versus FRISC OS's 941,000 (3.36x higher). On 8 harts, xv6-riscv achieves 198,000 switches/sec versus FRISC OS's 1,820,000 (9.19x higher).

The xv6-riscv degradation is caused by the O(n) contention growth of its global spinlock. As hart count doubles, each hart waits twice as long for the lock, halving individual throughput. Combined throughput therefore plateaus and eventually decreases. FRISC OS avoids this by keeping all fast-path scheduler operations hart-local.

```
Comparison: FRISC OS vs xv6-riscv (switches/sec, QEMU virt)

Harts | xv6-riscv  | FRISC OS   | Speedup
------|------------|------------|--------
  1   |   235,000  |   242,000  |  1.03x
  2   |   310,000  |   478,000  |  1.54x
  4   |   280,000  |   941,000  |  3.36x
  8   |   198,000  | 1,820,000  |  9.19x

xv6-riscv global lock contention (8 harts):
  Mean wait:    12,400 cycles
  Throughput degrades ~37% from 4→8 harts
```

Scheduling latency also improves significantly. xv6-riscv's remote scheduling latency is dominated by global lock acquisition; at 8 harts, the p99 latency is 42,000 cycles compared to FRISC OS's 8,940 cycles. This 4.7x improvement directly benefits interactive workloads and real-time threads.

Code complexity is higher in FRISC OS: the scheduler is approximately 1,400 lines of C versus xv6's 350 lines. The Chase-Lev deque implementation alone is 280 lines. The trade-off is intentional: FRISC OS targets server-class RISC-V hardware where scaling is a primary concern, while xv6 prioritizes pedagogical clarity.

18. NUMA Considerations for Future Multi-Socket Systems

Current FRISC OS targets single-socket RISC-V boards where all harts share uniform memory access latency. Future RISC-V server platforms (such as the SiFive P670 multi-chip module or hypothetical multi-socket RISC-V servers) will expose Non-Uniform Memory Access (NUMA) topologies where inter-socket memory accesses are 2-4x slower than intra-socket accesses.

The device tree mechanism for describing NUMA topology uses numa-node-id properties on CPU nodes and memory nodes. FRISC OS already parses numa-node-id during hart enumeration and stores it in the hart descriptor's topo_mask field, but does not yet use this information to guide scheduling or memory allocation decisions.

A NUMA-aware scheduler would prefer to migrate threads to harts within the same NUMA node rather than across nodes. The work-stealing algorithm would need to be extended with a two-tier search: first steal from same-node harts, then from remote-node harts only if same-node queues are all empty. The IPI mechanism works identically across nodes (via SBI) but at higher latency.

```
// Planned NUMA-aware steal order
struct thread *numa_aware_steal(uint32_t my_id) {
    uint32_t my_node = hart_cb[my_id].numa_node;

    // Phase 1: steal from same-NUMA-node harts
    for (uint32_t i = 0; i < num_online_harts; i++) {
        if (i == my_id) continue;
        if (hart_cb[i].numa_node != my_node) continue;
        struct thread *t = try_steal(i);
        if (t) return t;
    }

    // Phase 2: cross-NUMA steal (penalize migration cost)
    for (uint32_t i = 0; i < num_online_harts; i++) {
        if (i == my_id) continue;
        if (hart_cb[i].numa_node == my_node) continue;
        struct thread *t = try_steal(i);
        if (t) {
```

```
            t->migration_cost_penalty++;
            return t;
        }
    }
    return NULL;
}
```

Memory allocation on NUMA systems must be topology-aware. The slab allocator's per-hart magazines naturally localize allocations to the hart's local memory if the backing pages are allocated from node-local physical memory. FRISC OS will add a kmalloc_node(size, node_id) interface to allow explicit NUMA-local allocation.

Inter-socket IPIs on NUMA systems traverse the inter-die interconnect, adding 300-500 ns of latency compared to intra-socket IPIs. This makes the IPI coalescing optimization (sending one IPI per target node rather than per target hart) significantly more important on NUMA platforms. The batching infrastructure is designed with this extension in mind.

19. Known Bugs and Limitations

The current FRISC OS SMP implementation has several known limitations. First, ASID recycling is not fully tested under rapid process creation and destruction. Under pathological workloads that create and destroy processes faster than the ASID allocator recycles them, a rarely-triggered bug can cause a stale TLB entry to persist for one additional process lifetime, potentially leaking one page of data between processes. A fix is in progress.

Second, the Chase-Lev deque resize path is not fully correct under concurrent steal attempts during resize. The implementation conservatively locks the deque during resize (using the hart's own spinlock), which is safe but serializes all steal attempts from that hart for the duration of the resize. Resize is rare (triggered only when queue depth exceeds 256 threads) but the lock makes it a potential latency spike.

Third, the IPI reason bitfield currently supports only 8 distinct reason codes (8 bits). As the kernel adds more subsystems requiring IPI notifications, this field may overflow. The fix is to widen the field to 32 bits, which is a trivial struct change but requires verifying that all atomic operations on the field are updated accordingly.

```
// Known issue: narrow IPI reason field
typedef atomic_uint8_t ipi_reason_t;  // BUG: only 8 reasons

// Planned fix:
typedef atomic_uint32_t ipi_reason_t;  // 32 reasons

// Current reason codes (using all 8 bits):
#define IPI_RESCHEDULE   (1 << 0)
#define IPI_TLB_FLUSH    (1 << 1)
#define IPI_HALT         (1 << 2)
#define IPI_ICACHE_FLUSH (1 << 3)
#define IPI_LB_PROBE     (1 << 4)
// Bits 5-7: reserved (none left for new subsystems)
```

Fourth, FRISC OS does not implement CPU hotplug (dynamic hart online/offline). The IPI_HALT code path is present for panic propagation only. Adding hotplug requires careful handling of threads pinned to the halting hart, run queue migration, and ASID table updates. This is tracked as issue #47 in the FRISC OS bug tracker.

Fifth, the ticket lock implementation does not include a backoff strategy. Under very high contention (more than 4 harts competing for a single lock), the linear spin degrades cache performance by

repeatedly loading the now_serving counter. An exponential backoff or a local-spinning (MCS) variant would improve performance but adds implementation complexity.

20. Future Work

The primary near-term goal is validation on real RISC-V hardware. The SiFive HiFive Unmatched board (4 U74 cores) is the target platform for the first hardware port. Key differences from QEMU include: real cache hierarchy with measured false-sharing effects, hardware performance counters accessible via hpmcounter CSRs, and a real CLINT with measured IPI latency.

FRISC OS v0.5 will add O(1) completely fair scheduling (CFS) as an alternative scheduler policy alongside the current priority-based FIFO. The CFS implementation will use a per-hart red-black tree keyed on virtual runtime, requiring a lock-free or fine-grained-locked tree implementation to maintain the per-hart isolation property.

The IPI coalescing optimization is scheduled for v0.5. Instead of calling sbi_send_ipi immediately on each enqueue, the kernel will accumulate pending IPI targets in a per-hart bitmask and flush all pending IPIs at the end of each interrupt handler or critical section exit. This reduces SBI ecall overhead by batching multiple IPIs into one.

```
// Planned IPI coalescing interface
void ipi_send_deferred(uint32_t dst, uint8_t reason) {
    ipi_reason_table[dst] |= reason;
    my_hcb()->pending_ipi_mask |= (1u << dst);
    // IPI not sent yet; deferred to ipi_flush()
}

void ipi_flush(void) {
    uint32_t mask = my_hcb()->pending_ipi_mask;
    if (!mask) return;
    my_hcb()->pending_ipi_mask = 0;
    __asm__ volatile("fence w,w" ::: "memory");
    sbi_send_ipi(logical_to_hw_mask(mask), 0);
}
```

Formal verification of the ticket lock and Chase-Lev deque using the Iris concurrent separation logic framework (implemented in Coq) is a medium-term research goal. The deque's correctness argument relies on subtle ordering constraints that are difficult to audit manually; a machine-checked proof would substantially increase confidence in the implementation.

Integration with the Lateralus language runtime is the long-term goal. FRISC OS is designed as the kernel layer for a Lateralus-native operating system in which user-space programs are compiled Lateralus binaries. The scheduler's thread model maps directly to Lateralus lightweight coroutines; the per-hart run queue design mirrors Lateralus's pipeline parallelism model where each pipeline stage can be bound to a hart.