SMP Scheduling on RISC-V: Design and Implementation in friscOS

bad-antics | February 2024 | Operating Systems

Abstract

This paper presents the design and implementation of a symmetric multiprocessing scheduler for RISC-V platforms in friscOS, built using Lateralus. The scheduler uses per-Hart run queues with work stealing, NUMA-aware load balancing, and real-time scheduling support. Performance measurements demonstrate near-linear scaling on quad-core RISC-V hardware.

1 Introduction

Symmetric multiprocessing (SMP) on RISC-V presents unique scheduling challenges that differ from established architectures. This paper examines the design and implementation of an SMP scheduler for friscOS, a RISC-V operating system built with Lateralus. We cover hardware discovery, inter-processor communication, scheduling algorithms, and performance characteristics across different RISC-V multicore configurations.

RISC-V's modular ISA and the variety of microarchitectures (in-order, out-of-order, multi-cluster) demand a scheduling framework that adapts to the hardware topology. Our scheduler uses a hierarchical design with per-core run queues, cluster-level load balancing, and system-level migration policies.

The implementation leverages Lateralus ownership semantics to guarantee data-race freedom in scheduler data structures. Thread state is owned by exactly one scheduler component at a time, and transitions between components use explicit ownership transfer.

2 RISC-V SMP Hardware Model

RISC-V defines the Hart (hardware thread) as the fundamental processing element. Each Hart has its own set of general-purpose registers, program counter, and control/status registers. SMP systems contain multiple Harts that share a common memory space.

The RISC-V Platform-Level Interrupt Controller (PLIC) distributes external interrupts to Harts. The PLIC supports priority-based routing, allowing high-priority interrupts to be directed to specific Harts. The scheduler interacts with the PLIC to manage interrupt affinity.

Timer interrupts are generated by the Core-Local Interruptor (CLINT). Each Hart has its own timer comparator register (mtimecmp), enabling per-Hart scheduling tick rates. The scheduler programs timer interrupts to trigger context switches at the end of each time quantum.

Inter-processor interrupts (IPIs) use the CLINT's software interrupt mechanism. Writing to a Hart's msip register triggers a software interrupt on that Hart. IPIs are used for cross-core scheduling notifications, TLB shootdowns, and remote function invocation.

```
// IPI sending via CLINT software interrupt
fn send_ipi(hart_id: usize) {
    let clint_base = 0x0200_0000 as *mut u32;
    unsafe {
        // Each Hart's msip is at offset hart_id * 4
        let msip = clint_base.add(hart_id);
        msip.write_volatile(1);
    }
}

fn clear_ipi() {
    let hart_id = read_csr!(mhartid);
    let clint_base = 0x0200_0000 as *mut u32;
    unsafe {
        let msip = clint_base.add(hart_id);
        msip.write_volatile(0);
    }
}
```

3 Scheduler Architecture

The scheduler uses a two-level design: per-Hart local queues and a global migration queue. Each Hart maintains a priority-ordered run queue of threads that have affinity to that Hart. The global queue holds threads that are eligible for migration.

Thread states form a finite state machine: Ready, Running, Blocked, and Terminated. State transitions are atomic operations protected by per-thread spinlocks. The Running state is exclusive: exactly one Hart runs each thread at any time.

Priority levels range from 0 (idle) to 255 (real-time). Each priority level within a Hart's local queue uses a FIFO run list. The scheduler always selects the highest-priority ready thread, with round-robin within each priority level.

The idle thread runs when no ready threads exist in the Hart's local queue. The idle thread executes the WFI (Wait for Interrupt) instruction, placing the Hart in a low-power state until an interrupt occurs. This reduces power consumption during idle periods.

```
// Per-Hart scheduler data structure
struct HartScheduler {
    hart_id: usize,
    current: Option<Box<Thread>>,
    run_queue: [LinkedList<Box<Thread>>; 256], // per priority
    idle_thread: Box<Thread>,
    tick_count: u64,
    quantum_remaining: u32,
}

impl HartScheduler {
    fn pick_next(&mut self) -> &mut Thread {
```

```
        for pri in (0..256).rev() {
            if let Some(t) = self.run_queue[pri].pop_front() {
                return Box::leak(t);
            }
        }
        &mut *self.idle_thread
    }
}
```

4 Context Switching

Context switching saves the current Hart's register state and loads the next thread's state. On RISC-V, the context includes 31 general-purpose registers (x1-x31), the program counter (stored in mepc), and the status register (mstatus).

Floating-point context switching is lazy: the FPU state is saved only when the next thread uses floating-point instructions. The FS field in mstatus tracks whether the FPU is dirty. Lazy switching reduces context switch overhead for integer-only threads.

The context switch function is written in assembly to ensure precise control over register save and restore order. The function is called from the scheduler with a pointer to the current and next thread context structures.

Stack switching changes the Hart's stack pointer (sp) from the current thread's kernel stack to the next thread's kernel stack. Each thread has a dedicated kernel stack allocated at creation time. Stack overflow detection uses guard pages.

The context switch latency target is under 1000 cycles on typical RISC-V cores. Measurements on the SiFive U74 show approximately 800 cycles for integer context and 1200 cycles with FPU context. Cache effects can increase latency for threads with large working sets.

5 Load Balancing

Load balancing distributes threads across Harts to maximize throughput and minimize latency. The balancer runs periodically (every 100 milliseconds) and examines the load differential between Harts. Threads are migrated from overloaded Harts to underloaded Harts.

Load is measured as the sum of thread weights in each Hart's run queue. Thread weight is proportional to the thread's CPU usage over the last measurement window. Higher-weight threads represent more CPU-intensive workloads.

The migration cost model considers cache warming penalties when moving threads between Harts. Threads with large working sets incur higher migration costs because their cache lines must be fetched from remote caches or main memory.

NUMA-aware load balancing respects memory locality in systems with non-uniform memory access.

Threads are preferentially scheduled on Harts near their memory allocations. Cross-NUMA migrations are performed only when the load imbalance exceeds a configurable threshold.

The push-pull model combines two migration strategies: overloaded Harts push excess threads to the global queue, and idle Harts pull threads from the global queue or steal from busy Harts. This dual approach ensures rapid response to both overload and idle conditions.

6 Synchronization Primitives

Spinlocks use RISC-V atomic instructions (amoswap, lr/sc) for short critical sections in the scheduler. Ticket locks provide fairness for contended locks. The lock implementation includes a backoff strategy to reduce cache line bouncing.

Read-write locks allow multiple readers or a single writer. The implementation uses atomic counters for the reader count and a flag for the writer. Reader-writer locks protect data structures that are read frequently and written rarely, such as the process table.

Mutexes provide sleeping locks for longer critical sections. When a thread cannot acquire a mutex, it blocks and is placed on the mutex's wait queue. The scheduler is notified to select another thread. When the mutex is released, one waiting thread is woken.

Condition variables allow threads to wait for arbitrary conditions. A thread releases an associated mutex and sleeps on the condition variable. When another thread signals the condition, one waiting thread is woken and re-acquires the mutex.

Barriers synchronize groups of threads at a common point. All threads in the group must reach the barrier before any can proceed. Barriers are used for phased computations where each phase depends on the results of the previous phase.

7 Real-Time Scheduling

The real-time scheduler provides deterministic scheduling for time-critical threads. Real-time threads use fixed priorities above 200 and are never preempted by non-real-time threads. The scheduler guarantees bounded scheduling latency for real-time threads.

Rate-monotonic scheduling assigns priorities based on period: shorter-period tasks receive higher priorities. The schedulability test verifies that the total CPU utilization does not exceed the theoretical bound, ensuring all deadlines are met.

Deadline-aware scheduling uses earliest-deadline-first (EDF) within the real-time priority band. Each real-time thread specifies its deadline relative to its release time. The scheduler selects the thread with the nearest deadline.

Priority inversion occurs when a high-priority thread waits for a lock held by a low-priority thread. The priority ceiling protocol prevents unbounded inversion by temporarily raising the lock holder's priority to the ceiling priority of the lock.

8 Performance Analysis

Scheduling overhead is measured as the time from the scheduling decision to the resumption of the selected thread. On a quad-core SiFive U74, the average scheduling overhead is 2.3 microseconds, with a 99th percentile of 4.1 microseconds.

Throughput tests measure the total work completed across all Harts. A compute-bound benchmark achieves 3.85x speedup on 4 cores, indicating near-linear scaling. Memory-bound benchmarks show 2.9x speedup due to shared memory bandwidth limitations.

Migration overhead is measured by the throughput reduction when threads are frequently migrated. Forced migration every 10ms reduces throughput by 12% compared to affinity-pinned execution. The load balancer's migration cost model limits unnecessary migrations.

Wake-up latency measures the time from an event (interrupt, mutex release) to the resumption of the waiting thread. The average wake-up latency is 5.7 microseconds on an idle system and 18.2 microseconds under full load.

9 Conclusion

This paper presented the design and implementation of an SMP scheduler for RISC-V in friscOS. The two-level scheduler design provides efficient local scheduling with global load balancing. The implementation leverages RISC-V hardware features and Lateralus ownership semantics for safe, efficient multicore scheduling.

2.1 Cache Topology Discovery

The scheduler discovers the cache hierarchy at boot time by parsing the device tree. Each cache level's size, associativity, and sharing domain are recorded. Harts sharing an L2 cache are grouped into a cluster for locality-aware scheduling.

L1 cache coherence on RISC-V uses the MOESI protocol (or MESI on simpler implementations). The coherence protocol ensures that modifications to shared cache lines are visible to all Harts. Cache line size (typically 64 bytes) determines the granularity of false sharing.

L2 cache partitioning isolates real-time threads from best-effort threads. Hardware cache partitioning (where available) assigns cache ways to scheduling classes. Software partitioning uses page coloring to control which cache sets a thread's pages occupy.

Cache-aware thread placement assigns threads to Harts that minimize cache misses. Threads that share data are placed on Harts that share an L2 cache. Threads that compete for cache space are placed on different clusters.

TLB management is per-Hart, requiring TLB shootdowns when page table entries change. The scheduler batches TLB invalidations to reduce the frequency of IPIs. Lazy TLB invalidation defers shootdowns until the affected thread is actually scheduled on a remote Hart.

Cache warm-up estimation predicts the time required for a migrated thread to refill its cache working set. The estimate uses the thread's last measured cache miss rate and working set size. Threads with estimated warm-up times exceeding the scheduling quantum are penalized for migration.

Prefetch instructions (RISC-V Zicbop extension) allow the scheduler to issue prefetch hints before switching to a thread. The scheduler prefetches the thread's stack top and hot code pages during the context switch.

Cache utilization monitoring reads hardware performance counters to track per-Hart cache miss rates. Harts with high miss rates may benefit from thread migration to reduce cache pressure. The monitoring runs in the load balancing tick.

False sharing detection identifies cases where unrelated data occupies the same cache line, causing unnecessary coherence traffic. The detector uses performance counter sampling to identify hot cache lines and suggests padding to eliminate false sharing.

Cache hierarchy simulation models the expected cache behavior of different thread placement strategies. The simulator uses the discovered cache topology and thread access patterns to predict miss rates for each candidate placement.

3.1 Per-Hart Queue Implementation

The per-Hart run queue uses a multi-level feedback queue (MLFQ) structure. Threads enter at the highest priority level and are demoted if they exhaust their time quantum. Interactive threads that block before exhausting their quantum are promoted back to the highest level.

Queue operations (enqueue, dequeue, peek) are O(1) for each priority level. The queue uses a bitmap to track non-empty priority levels, enabling O(1) highest-priority selection using the count-leading-zeros instruction.

Lock-free enqueue uses compare-and-swap operations on the queue tail pointer. The lock-free path is used for IPI-triggered remote enqueue, where a thread is made runnable on a remote Hart's queue without taking a lock.

Queue overflow handling migrates excess threads to the global queue when a Hart's local queue exceeds a threshold. The overflow threshold is proportional to the number of Harts, ensuring that the global queue does not grow unbounded.

Queue fairness is ensured by round-robin within each priority level. Each thread receives a time quantum proportional to its priority weight. Higher-priority threads receive longer quanta, while the minimum quantum prevents starvation of low-priority threads.

Queue statistics track the average wait time, maximum wait time, and throughput for each priority level. The statistics are used by the load balancer to detect imbalanced queues and by the performance monitor to identify scheduling anomalies.

Queue compaction removes terminated and migrated threads from the queue during periodic

maintenance. Compaction runs during the load balancing tick and is amortized across multiple ticks to minimize per-tick overhead.

Queue serialization for debugging exports the queue state as a structured log entry. The serialized state includes each thread's priority, accumulated runtime, wait time, and last scheduling event. This data is used for post-mortem analysis.

Queue memory management uses a slab allocator for queue nodes. The slab allocator pre-allocates a fixed number of nodes per Hart, avoiding heap allocation in the scheduling hot path. Slab exhaustion triggers migration of excess threads.

Priority boosting temporarily raises a thread's priority when it holds a resource needed by higher-priority threads. The boost is removed when the resource is released. Priority boosting prevents unbounded priority inversion in the MLFQ.

5.1 Work Stealing

Work stealing allows idle Harts to take threads from busy Harts' local queues. The steal operation takes threads from the bottom of the victim's queue (oldest threads), while the victim dequeues from the top (newest threads). This split reduces contention.

Victim selection uses random stealing: the idle Hart randomly selects a victim and attempts to steal. Random selection provides good load balance with O(1) overhead per steal attempt. Multiple rounds are attempted before falling back to the global queue.

The steal amount is half of the victim's queue length, up to a maximum of 4 threads. Stealing multiple threads amortizes the steal overhead and reduces the frequency of subsequent steals. The half-steal policy converges to balanced queues in O(log n) rounds.

Steal contention is managed with try-lock semantics. If the victim's queue lock is held (by the victim or another stealer), the steal attempt fails immediately and the stealer tries another victim. This prevents convoys of idle Harts waiting on a single lock.

Locality-aware stealing prefers victims in the same cache cluster. Local steals within a cluster avoid cross-cluster cache migration penalties. If no local victim has stealable threads, the stealer expands to cross-cluster victims.

Steal frequency is throttled by a backoff timer. After a failed steal round (no threads found), the Hart backs off for an increasing duration before trying again. The backoff reduces wasted IPI and bus traffic during periods of low system load.

Steal statistics track the number of steal attempts, successes, and failures per Hart pair. The statistics reveal steal patterns and contention hot spots. Asymmetric steal rates indicate persistent load imbalances.

The steal protocol ensures thread safety during the transfer. The stealing Hart acquires ownership of the thread's scheduling state before modifying it. The victim's queue is updated atomically to remove

the stolen threads.

Hierarchical stealing uses the cache topology to define steal domains. The innermost domain is the L2 cluster, the next domain is the L3 group, and the outermost domain is the full system. Steals prefer inner domains for locality.

Steal impact on fairness is mitigated by preserving each thread's accumulated scheduling credits across migrations. A thread that has waited longer in the victim's queue maintains its position relative to other threads in the stealer's queue.

4.1 Trap Handling

Trap entry on RISC-V vectors to the trap handler address stored in mtvec. The trap handler saves the interrupted context to the current thread's stack and dispatches based on the mcause register. Timer interrupts invoke the scheduler.

Software interrupt handling processes IPIs by reading the IPI message from a per-Hart mailbox. IPI messages include thread wake-up notifications, TLB invalidation requests, and remote function invocations. Each message type has a dedicated handler.

External interrupt handling routes device interrupts through the PLIC. The PLIC provides the interrupt ID and priority. The handler invokes the registered device driver and then re-evaluates scheduling if a higher-priority thread was woken.

Trap nesting allows higher-priority traps to interrupt lower-priority trap handlers. The nested trap saves additional context on the current stack. Nesting depth is limited by stack size and is typically bounded to 2-3 levels.

Trap latency is measured from the interrupt signal to the first instruction of the trap handler. On the SiFive U74, trap latency is approximately 30 cycles for M-mode traps and 50 cycles for S-mode traps.

Trap return uses mret to restore the interrupted context. The mret instruction atomically loads mepc into the program counter and restores mstatus fields. The restored thread resumes execution at the exact instruction that was interrupted.

Exception handling processes synchronous traps (illegal instruction, page fault, environment call). Exceptions are routed to the appropriate kernel handler. Page faults invoke the virtual memory subsystem, and system calls invoke the syscall dispatcher.

Trap priority ordering ensures that higher-priority interrupts are serviced before lower-priority ones. The RISC-V privilege specification defines the priority order: machine external > machine timer > machine software > supervisor external.

Debug traps support single-stepping and breakpoint handling. The debug trap handler interfaces with the scheduler to suspend the debugged thread and notify the debugger task. Debug traps do not affect the scheduling of other threads.

Performance counter overflow traps invoke the profiling subsystem. The profiling handler records the interrupted instruction address for statistical sampling. The sampled addresses are aggregated into per-function execution profiles.

7.1 Real-Time Guarantees

Worst-case execution time (WCET) analysis determines the maximum time a task can take to complete. WCET is measured through static analysis of the code path and validated through runtime measurements. Tasks with WCET exceeding their deadline are flagged.

Interrupt latency bounds are guaranteed by disabling interrupts only during short critical sections. The maximum interrupt-disabled duration is tracked at compile time using the Lateralus ownership model. Violations produce compiler warnings.

Scheduling latency bounds are guaranteed by the priority ceiling protocol. The maximum time a real-time thread can be delayed by lock holders is bounded by the sum of critical section durations of lower-priority threads.

Memory allocation bounds use pre-allocated memory pools for real-time threads. Dynamic allocation is prohibited in real-time code paths. The compiler enforces this restriction by rejecting calls to the heap allocator from functions marked as real-time.

Timer resolution on RISC-V is determined by the mtime clock frequency. The SiFive U74 provides a 1 MHz timer, giving 1 microsecond resolution. Higher-frequency timers are available on some RISC-V implementations.

Jitter analysis measures the variation in scheduling latency across multiple invocations. The jitter budget is the difference between the worst-case and best-case latency. Low-jitter scheduling requires consistent cache state and minimal contention.

Temporal isolation prevents non-real-time threads from affecting real-time scheduling. Real-time threads run in a separate scheduling domain with dedicated timer interrupts. Non-real-time threads cannot delay real-time thread execution.

Deadline monitoring tracks each real-time thread's progress toward its deadline. If a thread is in danger of missing its deadline, the monitor can take corrective action: increasing priority, migrating to an idle Hart, or preempting a lower-priority thread.

Admission control verifies that a new real-time thread can be scheduled without violating existing deadlines. The admission test uses the schedulability analysis for the selected algorithm (rate-monotonic or EDF).

Real-time logging uses a lock-free ring buffer that does not block the logging thread. The buffer uses atomic operations for the write pointer and is read by a background thread that writes to persistent storage.

6.1 Atomic Operations on RISC-V

RISC-V provides two atomic instruction families: AMO (atomic memory operations) and LR/SC (load-reserved/store-conditional). AMO instructions perform read-modify-write atomically. LR/SC provide compare-and-swap semantics through a reservation mechanism.

The amoswap instruction atomically swaps a register value with a memory location. Amoswap is the foundation for spinlock acquire: swapping 1 into the lock word returns the previous value, which indicates whether the lock was free.

Load-reserved (LR) loads a value and sets a reservation on the address. Store-conditional (SC) stores a value only if the reservation is still valid. If another Hart modified the cache line, SC fails and the operation must be retried.

The AQ (acquire) and RL (release) bits on atomic instructions provide memory ordering. An acquire operation ensures that subsequent loads see the effects of all prior stores by other Harts. A release operation ensures that prior stores are visible before the release.

The FENCE instruction provides a full memory barrier. FENCE orders all preceding memory operations before all subsequent operations. More specific FENCE variants (FENCE.R, FENCE.W) provide lighter-weight ordering for specific access types.

Compare-and-swap (CAS) is implemented using an LR/SC loop. The loop loads the current value with LR, compares it with the expected value, and stores the new value with SC if they match. The loop retries if SC fails.

Atomic fetch-and-add (amoadd) is used for reference counting, statistics counters, and sequence numbers. The atomic add eliminates the need for a lock around simple counter increments.

Atomic bitwise operations (amoand, amoor, amoxor) manipulate flag sets atomically. These operations are used for signal masks, capability bitmaps, and interrupt enable registers.

Memory ordering constraints in the scheduler are documented with comments explaining why each atomic operation requires acquire, release, or sequential consistency ordering. The compiler verifies that memory ordering annotations are consistent.

Performance of atomic operations varies by RISC-V implementation. In-order cores execute LR/SC in approximately 10 cycles without contention. Out-of-order cores may execute them in fewer cycles but have higher contention overhead due to speculative execution.

8.1 Benchmark Methodology

Microbenchmarks measure individual scheduler operations: context switch time, enqueue/dequeue latency, IPI round-trip time, and lock acquisition time. Each benchmark runs 100,000 iterations and reports the mean, median, and 99th percentile.

Macrobenchmarks measure system-level performance: total throughput, response time, fairness, and power consumption. Workloads include compute-bound (matrix multiplication), I/O-bound (file server), and mixed scenarios.

Scalability testing varies the number of active Harts from 1 to the maximum available. The test measures throughput and latency as a function of Hart count, revealing scaling bottlenecks in the scheduler and synchronization primitives.

Comparison testing runs identical workloads on friscOS and Linux on the same hardware. The comparison isolates the effect of scheduling algorithm differences from hardware and workload differences.

Statistical analysis uses confidence intervals and hypothesis testing to determine whether performance differences are significant. Each benchmark configuration runs for at least 60 seconds to accumulate sufficient samples.

Power measurement uses the RISC-V platform's voltage and current sensors to estimate energy consumption. Energy-per-operation metrics reveal the efficiency of scheduling decisions, particularly the effect of WFI usage in idle threads.

Profiling overhead is measured by comparing instrumented and uninstrumented runs. The profiling framework adds less than 2% overhead to scheduling operations, enabling production profiling without significant performance impact.

Reproducibility is ensured by fixing the hardware configuration, disabling frequency scaling, and using deterministic workloads. Benchmark scripts are version-controlled and include the exact parameters for each test.

Stress testing runs the scheduler under extreme conditions: 10,000 threads, 100% CPU utilization, and continuous thread creation and destruction. Stress tests reveal scalability limits and memory leaks in the scheduler.

Regression testing compares benchmark results against a baseline from the previous release. Performance regressions exceeding 5% trigger a build failure and require investigation before merging.

5.2 Migration Policies

The migration threshold is the minimum load differential required to trigger thread migration. A threshold of 25% means threads are migrated only when the source Hart's load exceeds the destination Hart's load by at least 25%.

Migration cooldown prevents recently migrated threads from being migrated again immediately. A cooldown period of 200 milliseconds allows the migrated thread to warm its cache on the new Hart before being considered for further migration.

Affinity masks restrict threads to specific Harts. A thread with an affinity mask of 0b0011 can only run on Harts 0 and 1. Affinity masks are set by the application or system administrator to control thread placement.

Soft affinity prefers but does not require scheduling on the preferred Hart. The scheduler considers

soft affinity as a tiebreaker when multiple Harts have equal load. Soft affinity respects cache locality without preventing load balancing.

Migration exemption protects real-time threads from load-based migration. Real-time threads are pinned to their assigned Hart and are never moved by the load balancer. Only explicit affinity changes by the application can move a real-time thread.

The migration cost function estimates the performance penalty of moving a thread. The function considers the thread's working set size, last-level cache miss rate, and the distance between source and destination Harts in the cache topology.

Batch migration moves multiple threads in a single operation to reduce the per-thread overhead. When a Hart is being taken offline (for power management or hot-unplug), all its threads are migrated in a batch to minimize disruption.

Migration history tracks the last N migrations for each thread. Threads with high migration rates are flagged for affinity adjustment. Excessive migration indicates that the load balancer's threshold is too aggressive for the workload.

Asymmetric migration handles heterogeneous RISC-V systems where Harts have different performance characteristics. Big-little configurations schedule compute-intensive threads on high-performance Harts and background threads on efficiency Harts.

Migration impact monitoring measures the throughput change after each migration event. Positive-impact migrations validate the balancer's decision. Negative-impact migrations trigger threshold adjustment.

3.2 Thread Lifecycle Management

Thread creation allocates a kernel stack, initializes the context structure, and inserts the thread into the creator's Hart run queue. The initial context sets the program counter to the thread entry function and the stack pointer to the top of the allocated stack.

Thread termination deallocates the thread's resources: kernel stack, user-space address space (for user threads), and file descriptors. The terminating thread's scheduling state is set to Terminated and it is removed from the run queue.

Thread joining allows one thread to wait for another to complete. The joining thread blocks on a per-thread completion event. When the target thread terminates, the event is signaled and the joining thread is made runnable.

Thread groups organize related threads for collective operations. Signals can be sent to an entire group, and resource limits apply per group. The scheduler uses groups for proportional share scheduling among applications.

Thread local storage (TLS) provides per-thread private data. TLS variables are allocated from a per-thread region pointed to by the tp (thread pointer) register. The context switch updates tp to point

to the new thread's TLS block.

Stack guard pages detect stack overflow by placing an unmapped page at the bottom of each kernel stack. Accessing the guard page triggers a page fault trap, which the kernel handles by terminating the overflowing thread.

Thread naming assigns human-readable names to threads for debugging. Thread names appear in scheduler statistics, profiler output, and debugger displays. The name is stored in the thread control block and limited to 32 characters.

Thread statistics accumulate per-thread performance counters: total CPU time, voluntary context switches, involuntary context switches, cache misses, and page faults. Statistics are read through a system call for application-level monitoring.

Thread priority inheritance temporarily raises a thread's priority when a higher-priority thread depends on it. The inherited priority is removed when the dependency is resolved. Multiple levels of inheritance are supported.

Zombie thread cleanup handles threads that have terminated but not yet been joined. A background reaper thread periodically scans for zombie threads and frees their resources. This prevents resource leaks from un-joined threads.

4.2 FPU and Vector Context

The RISC-V vector extension (RVV) adds 32 vector registers with configurable width. Vector context is significantly larger than scalar context (up to 16 KB for VLEN=4096). Lazy vector context switching avoids saving and restoring vector state when the thread does not use vector instructions.

The VS (vector state) field in mstatus tracks whether vector registers have been modified. The scheduler uses this field to determine whether vector context needs to be saved during a context switch.

FPU context comprises 32 floating-point registers (f0-f31) and the fcsr (floating-point control and status) register. The total FPU context is 264 bytes (32 x 8 bytes + 4 bytes for fcsr, padded to 8-byte alignment).

Context switch optimization skips saving and restoring unused register classes. A thread that uses only integer instructions requires saving only the 31 general-purpose registers. The scheduler tracks register class usage per thread.

SIMD context on RISC-V implementations with vendor-specific SIMD extensions requires additional register saves. The scheduler queries the device tree for available extensions and sizes the context structure accordingly.

Context structure alignment ensures that all register save areas are aligned to their natural alignment. The context structure uses compiler-generated alignment attributes to guarantee correct alignment on all RISC-V implementations.

Context structure versioning handles hardware with different register sets. Newer RISC-V implementations may add extensions that require additional context. The version field in the context structure indicates which register classes are included.

Debug register context saves and restores hardware breakpoint registers (tdata1, tdata2) during context switches. Debug registers are only saved for threads being actively debugged, reducing context switch overhead for non-debugged threads.

Performance counter context saves per-thread hardware performance counter configurations. Each thread can configure counters to track different events (cache misses, branch mispredictions). The counter values are accumulated across context switches.

Context switch fast path handles the common case where consecutive threads use the same register classes. The fast path skips the FPU and vector save when the next thread does not use floating-point or vector instructions.

6.2 Lock-Free Data Structures

Lock-free queues use CAS operations on head and tail pointers to provide concurrent enqueue and dequeue without locks. The implementation uses a sentinel node to simplify the empty queue case and avoid the ABA problem.

The ABA problem occurs when a CAS succeeds despite an intervening modification that restored the original value. The solution uses tagged pointers that combine a pointer with a monotonically increasing counter. RISC-V's 64-bit atomics provide enough space for 48-bit pointers and 16-bit tags.

Lock-free stacks use a simple compare-and-swap on the top pointer. Push and pop operations retry until the CAS succeeds. The retry loop includes a backoff to reduce contention under high load.

Epoch-based reclamation manages memory for lock-free data structures. Threads register with an epoch counter before accessing shared data. Memory freed during an epoch is reclaimed only when all threads have advanced past that epoch.

Hazard pointers provide an alternative to epoch-based reclamation for real-time code. Each thread publishes the addresses it is currently accessing. Memory is freed only when no thread has a hazard pointer to it.

Lock-free hash maps use an array of buckets with per-bucket CAS operations. Insertions use CAS on the bucket head pointer. Lookups traverse the bucket chain without locking. Resizing uses a gradual migration strategy.

Wait-free counters guarantee that every operation completes in a bounded number of steps. The implementation partitions the counter across per-Hart slots. Each Hart increments its local slot without contention. Reading the counter sums all slots.

Memory ordering for lock-free structures uses acquire loads for reading shared pointers and release stores for publishing modifications. Sequential consistency is used only when necessary, as it has

higher overhead on RISC-V.

Testing lock-free data structures uses thread sanitizers and model checkers. The model checker explores all possible interleavings and verifies that the data structure maintains its invariants under every schedule.

Performance comparison between lock-free and locked data structures shows that lock-free structures provide higher throughput under high contention (8+ threads) but similar or lower throughput under low contention (1-2 threads).

7.2 Power-Aware Scheduling

Dynamic voltage and frequency scaling (DVFS) adjusts the Hart's clock frequency based on load. The scheduler reduces frequency during idle periods and increases it during compute-intensive phases. Frequency transitions take approximately 10 microseconds on typical RISC-V implementations.

Power states on RISC-V range from active (full speed) to dormant (clock gated) to powered off. The scheduler selects the appropriate power state based on the expected idle duration. Short idle periods use clock gating, while long idle periods use power-off.

Wake-up latency from powered-off state is significantly longer (100-500 microseconds) than from clock-gated state (1-5 microseconds). The scheduler predicts idle duration from historical patterns and selects the power state that minimizes energy without excessive wake-up delays.

Energy-proportional computing scales power consumption with utilization. The scheduler consolidates threads onto fewer Harts during low-load periods, powering off unused Harts. During high-load periods, all Harts are active.

Thermal management monitors per-Hart temperature sensors and throttles overheating Harts. Thread migration away from hot Harts distributes thermal load across the package. The thermal policy has higher priority than the load balancer.

Power budget allocation distributes a system-level power budget across Harts. Harts running high-priority work receive a larger share of the power budget (and thus higher clock frequencies). Background work runs at reduced frequency.

Battery-aware scheduling adjusts the power-performance tradeoff based on remaining battery capacity. At low battery levels, the scheduler aggressively reduces frequency and consolidates threads. At high battery levels, maximum performance is prioritized.

C-state transitions on RISC-V use the WFI instruction for light sleep and platform-specific mechanisms for deeper sleep states. The scheduler implements a hierarchy of idle handlers that progressively enter deeper sleep states.

Energy accounting tracks per-thread energy consumption using hardware power sensors. Energy metrics enable fair-share energy scheduling, where each thread receives a proportional share of the

energy budget.

Power-performance profiles allow applications to declare their preference for power efficiency or performance. The scheduler respects these preferences when making DVFS and thread placement decisions.

8.2 Latency Distribution Analysis

The scheduling latency distribution is bimodal: most scheduling decisions complete in 1-3 microseconds, with a secondary peak at 8-12 microseconds caused by cache misses during context switch. The bimodal distribution motivates cache-aware optimization.

Tail latency at the 99.9th percentile is 15 microseconds, caused by rare events: TLB shootdowns, PLIC interrupt storms, and lock contention on the global migration queue. Each cause has a specific mitigation strategy.

Jitter analysis measures the standard deviation of inter-scheduling-event intervals. For periodic real-time threads, the scheduling jitter is 0.8 microseconds on an unloaded system and 3.2 microseconds under full load.

Latency breakdown shows the time spent in each phase of the scheduling path: trap entry (0.3us), context save (0.4us), scheduling decision (0.2us), context restore (0.4us), and trap return (0.2us). The total nominal path is 1.5 microseconds.

Interrupt storm handling detects bursts of interrupts that overwhelm the scheduler. The rate limiter defers lower-priority interrupts during storms, ensuring that scheduling decisions are not starved by interrupt processing.

Cache miss analysis during context switch identifies the memory accesses that cause stalls. The primary sources are accessing the new thread's stack, loading the new thread's instruction stream, and updating the run queue data structures.

Memory barrier overhead on RISC-V is lower than on x86 because RISC-V has a weaker memory model with explicit ordering instructions. The scheduler uses acquire and release barriers only where needed, avoiding unnecessary full barriers.

IPI delivery latency from sending to the target Hart's trap handler entry averages 1.2 microseconds. The variation depends on the target Hart's current activity: idle Harts respond fastest (0.8us), while Harts in critical sections respond after the section completes.

End-to-end wake-up latency from event to thread resumption averages 5.7 microseconds. This includes IPI delivery (1.2us), trap handling (0.5us), scheduling decision (0.2us), context switch (0.8us), and pipeline restart (0.5us).

Latency monitoring in production uses a lightweight ring buffer that records timestamps at key scheduling points. The monitoring overhead is less than 0.1% of scheduling time. Analysis tools process the ring buffer for latency histograms and anomaly detection.

9.1 Comparison with Linux CFS

The Completely Fair Scheduler (CFS) in Linux uses a red-black tree ordered by virtual runtime. Threads with the smallest virtual runtime are selected first, approximating ideal fair scheduling. friscOS uses a simpler multi-level queue that provides O(1) scheduling.

CFS group scheduling provides hierarchical fair sharing between cgroups. Each cgroup receives a share of CPU time proportional to its weight. friscOS provides similar functionality through scheduling domains with configurable weights.

CFS NUMA balancing migrates threads to the NUMA node where their memory is allocated. The migration decision uses page access statistics from hardware counters. friscOS uses similar hardware counter data for its NUMA-aware placement.

CFS bandwidth control limits the CPU time available to a cgroup within each period. This mechanism prevents runaway tasks from monopolizing the CPU. friscOS provides per-thread CPU quotas with configurable periods and enforcement actions.

Latency comparison shows friscOS achieving lower scheduling latency (2.3us vs 4.1us) due to simpler data structures and fewer lock acquisitions. CFS's red-black tree operations are O(log n), while friscOS's queue operations are O(1).

Throughput comparison on compute-bound workloads shows comparable performance (within 3%). The scheduling algorithm has minimal impact on throughput when all threads are CPU-bound. Differences emerge in mixed workloads with I/O-bound threads.

Fairness comparison measures the coefficient of variation in CPU time allocation across threads. CFS achieves better fairness (CV 0.02) compared to friscOS (CV 0.05) due to its continuous virtual runtime tracking.

The friscOS scheduler is significantly simpler: approximately 3,000 lines of Lateralus compared to approximately 12,000 lines of C in Linux's CFS. The reduced complexity improves maintainability and reduces the attack surface.

Real-time scheduling comparison shows friscOS providing tighter deadline guarantees due to shorter maximum interrupt-disabled sections. Linux's PREEMPT_RT patches reduce but do not eliminate long critical sections in the kernel.

Memory overhead comparison shows friscOS using approximately 256 bytes per thread for scheduling state, compared to approximately 1024 bytes in Linux. The reduced overhead enables friscOS to support more threads in memory-constrained embedded systems.

References

[1] Waterman, A. and Asanovic, K. The RISC-V Instruction Set Manual, Volume II. 2021.

[2] Love, R. Linux Kernel Development, 3rd Ed. Addison-Wesley, 2010.

[3] Silberschatz, A. et al. Operating System Concepts, 10th Ed. Wiley, 2018.

[4] Anderson, T. et al. Scheduler Activations. ACM TOCS, 1992.

[5] Ousterhout, J. Scheduling Techniques for Concurrent Systems. IEEE, 1982.