Lion: Modular Verification of Async Runtime Liveness

Ti Zhou, Zihao Zhang, Omar Chowdhury and Shuai Mu
Stony Brook University

Abstract

Asynchronous programming (*async*) is the dominant paradigm for developing high-performance systems, where *concurrent tasks* voluntarily yield control to an *async runtime* when resources are unavailable. These programs expect a fundamental *liveness promise* from the runtime: every task that yields will eventually be resumed when its awaited event fires. A single scheduling bug can silently break this contract, leaving tasks permanently stalled with no observable symptoms, such as a crash or error trace, to signal the failure. We present a general parametric specification framework and methodology for discharging this promise in runtimes composed of multiple interacting modules. Our approach introduces an append-only *logical event log* that abstracts implementation details and enables a Kamp-style translation of Linear Temporal Logic (LTL) into First-Order Logic. This enables deductive program verifiers to reason about progress via monotonic timestamps without needing native temporal logic support. Our main liveness theorem abstracts over the sufficient conditions, such as environmental assumptions and calling conventions, allowing these proof obligations to be instantiated for different runtime implementations. We demonstrate this framework via *Lion*, a Tokio-compatible runtime in Rust verified using *Verus*. By establishing a bijective mapping between log events and concrete actions, we prove that Lion maintains the liveness contract and measure at least 95% of unverified-runtime throughput on nine real-world workloads.

1 Introduction

Asynchronous programming (*async*) has become a dominant paradigm for building high-performance systems software. In this model, many lightweight tasks are multiplexed onto a small number of threads, and tasks yield control when the resource they need is not yet available, rather than blocking the underlying thread. This style of execution is enabled by an *async runtime*, typically provided as a library such as libevent [55], libuv [45], or Tokio [66], built on top of OS primitives such as `epoll` and `io_uring` [3]. Async runtimes now sit on the critical path of web servers, databases, messaging systems, and application frameworks.

Every async program relies on a liveness promise from its runtime: *if a task yields waiting for an event, then the runtime will resume that task once the awaited event fires.* Unlike a preemptive kernel, which eventually schedules every runnable thread, an async runtime offers no safety net once a wakeup is lost or a registration goes missing. A violation of this promise can silently stall a task forever, often with no crash, error trace, or obvious local symptom. Bugs of this form can arise from misplaced registrations, lost wakeup notifications, incorrect queue handling, or other subtle errors in scheduling and dispatch. Even mature async runtimes have exhibited such bugs in practice, underscoring that this is a real and difficult correctness problem. For example, Tokio, Rust's most widely used async runtime, has had 11 confirmed liveness bugs (§2.2). Some took over a year to fix, reflecting how subtle they are and how deeply they are entangled with the runtime's internal scheduling logic.

When an async application stalls, isolating the root cause is notoriously difficult. The failure may be due to a bug in the application, a violation of the runtime's calling conventions, a bug in the runtime itself, or a mismatch between the runtime's expectations and the guarantees provided by the underlying OS primitives. Our goal in this work is to remove the runtime implementation from this set of plausible root causes by formally verifying its liveness guarantees.

At a high level, the desired guarantee can be *conceptually* expressed as the temporal property: $\Box \forall t.\ \mathit{event_fired}(t) \rightarrow \Diamond\ \mathit{is_completed}(t)$. Here, $\Box$ informally means "always," and $\Diamond$ means "eventually." This end-to-end formulation captures the operational promise described above: when a yielded task's awaited event fires, the runtime eventually resumes it, and repeated discharge of such obligations yields eventual completion. The difficulty is not only stating such a property at an abstract level while hiding implementation details, but also proving it in a deductive verifier such as Verus [35]. Verus provides strong support for functional correctness through preconditions, postconditions, invariants, and first-order reasoning, but it does not treat temporal operators such as $\Box$ and $\Diamond$ as first-class proof constructs. The property we care about spans an unbounded sequence of interactions across multiple runtime components, while the verifier fundamentally reasons about assertions attached to individual program points and procedure boundaries.

Our key idea is to introduce an append-only *logical event log* that records abstract runtime events while hiding implementation details. Within a single module log, eventuality

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
Manuscript submitted to ACM

can be expressed using later log indices; across logs, monotonic timestamps let us relate the ordering of events produced by different components. Intuitively, a statement such as "after event A, event B eventually occurs" can be re-expressed as a first-order statement that there exists a later relevant log position whose event is B. This yields a Kamp-style translation of the relevant temporal obligations into first-order logic [27], enabling deductive verifiers to reason about progress without native temporal logic support. For any concrete runtime instantiation, one must then establish a bijective correspondence between abstract log events and concrete runtime actions so that proofs over the log faithfully capture the behavior of the implementation.

The second challenge is modularity. An async runtime is not a single scheduler loop but a composition of interacting modules. To address this, we introduce *async contracts*: module-level acceptance and fulfillment conditions that specify what progress obligation each component assumes and what progress it guarantees. The fulfillment of one module's contract can then trigger the acceptance of the next, yielding the usual *contract chaining* proof pattern. This lets us decompose global liveness into local proof obligations while still obtaining an end-to-end guarantee.

A third challenge is identifying the assumptions under which such a guarantee is valid. The runtime must react correctly not only to events from the OS, but also to events produced by other runtime modules and to application-level events induced by well-formed clients. The liveness argument therefore depends on environmental assumptions about OS-level signaling mechanisms and on calling conventions that clients must respect. These assumptions play a role analogous to fairness or progress assumptions in traditional liveness proofs: they capture, for example, that relevant external events are eventually delivered and that enabled progress steps are not postponed forever. We make these assumptions explicit as specificational predicates. In particular, if one wishes to port the runtime to a new environment, these explicit assumptions identify what must be checked, manually, against the semantics and design documentation of the target OS primitives to determine whether the runtime's guarantees would continue to hold.

We realize this methodology in *Lion*, a Tokio-compatible async runtime written in Rust and mechanically verified in Verus. Lion consists of core modules, including an executor and a reactor, together with utility modules for async I/O, timers, and channels. Its design uses clear ownership boundaries to support modular specification and verification. For Lion, we establish the required correspondence between abstract log events and concrete runtime actions and prove that Lion satisfies the runtime liveness contract. Our evaluation on microbenchmarks and three real-world systems, rumqtt [5], Pingora [16], and Axum [65], shows that Lion can replace Tokio with only minor code changes and reaches at least 95% of Tokio's throughput on nine real-world workloads, showing that verified async runtimes need not incur prohibitive overhead.

In summary, this paper makes the following contributions:

- We present **Lion**, a Tokio-compatible async runtime in Rust with mechanically verified liveness guarantees.
- We introduce a **logical-log-based verification methodology** that abstracts away implementation details behind log events and reduces liveness obligations into first-order reasoning supported by existing deductive verifiers.
- We develop **modular async contracts** and a **contract-chaining** proof pattern for establishing end-to-end liveness across multiple interacting runtime components.
- We make explicit the **environmental assumptions and calling conventions** needed to discharge the liveness proof, thereby clarifying the boundary of the guarantee and providing a basis for portability analysis.
- We provide an **empirical evaluation** on microbenchmarks and three real-world systems, showing that Lion's verified design costs at most 5% of Tokio's throughput.

2 Background and Motivation

We introduce async programming (§2.1), present the challenges of liveness failures in production async runtimes (§2.2), and identify the gap in existing verified systems that Lion addresses (§2.3).

2.1 Async Programming and Runtimes

High-performance systems software often must handle large numbers of concurrent, mostly I/O-bound tasks (e.g., network requests, file operations, and timer expirations). In such workloads, tasks spend much of their time waiting for external events rather than using the CPU. A traditional synchronous design, where each task occupies its own thread, is often inefficient because stack management and kernel-level context switching impose substantial overhead. This limitation is reflected in the well-known C10K problem [28].

Asynchronous programming addresses this problem by representing concurrent work as lightweight *tasks* that are intended not to block the underlying execution thread. Conceptually, each task can be viewed as a *state machine* with explicit suspension points as states where transitions are triggered by asynchronous events. These events arise at multiple layers: low-level *system events* signaled by OS primitives such as `epoll` or `io_uring`, and higher-level *application events* corresponding to the completion of logical operations.

This model is enabled by an async runtime, typically implemented as a library. When a task reaches a suspension point, it registers interest in a relevant event and yields control. The runtime records the task as waiting, reacts to event notifications, and later reschedules the task when the awaited event occurs. By handling scheduling and most context switching in user space, the runtime can multiplex many logical tasks onto a small number of threads with lower overhead.

Although async systems are often exposed through higher-level abstractions, such as callback-based APIs, `async`/`await`, or goroutines, these abstractions ultimately rely on the same underlying runtime machinery for managing tasks, suspension, and event delivery.

2.2 Liveness Failures in Async Runtimes

As discussed in the introduction, async runtimes make a fundamental liveness promise: when a task yields waiting for an event, the runtime should eventually resume it once that event fires. We now ground this discussion in practice by examining concrete liveness failures from Tokio, Rust's most widely used async runtime. These bugs show that even mature runtimes can violate this promise in subtle ways that are difficult to diagnose and repair.

Representative liveness bugs in Tokio's runtime. Issue 5020 is a Tokio bug in which a spawned task never completes [74]. Normally, spawning or waking a task must notify the executor that new work is available, since the executor may be idle waiting for events. Tokio optimized this path by suppressing the notification when a flag indicated that the executor was already polling tasks. However, a later API also set this flag in a context where no executor was actually polling. As a result, the notification was suppressed when the task's timer later woke it, and the task remained queued indefinitely with no executor aware of it. The issue remained open for over a year, and its fix required 18 commits and 38 review comments over 26 days before the maintainers were confident in its correctness.

Issue 3069 is a Tokio bug in which a registered timer occasionally fails to fire [70]. In the reported behavior, a short sleep inside a loop succeeds for several iterations and then hangs indefinitely. The likely cause is a boundary-condition error in the timer wheel: a timer whose deadline coincides with, or falls just behind, the driver's current position may be inserted into a slot that has already been scanned, causing it to be silently skipped. Thus, registration succeeds with no error indication, yet the callback is never delivered. The maintainers never identified the exact failure path; the issue was ultimately resolved as a side effect of a later rewrite [72].

Tokio has 9 more confirmed liveness bugs similar to these two. We also studied liveness bugs in libevent and libuv; due to space limitations, we discuss them in Appendix A.

The bug tracker also reveals a broader pattern: when user code silently hangs, developers often ask whether the fault lies in their code or in the runtime [68, 69, 75]. Although such reports often turn out to be user misuse, the possibility of runtime bugs makes root-cause analysis difficult for both users and maintainers. This further motivates verifying the liveness of the runtime core: with a verified core, users investigating a hang can rule out runtime-level causes and focus on application logic.

2.3 Verified Systems and the Async Gap

Program verification has produced verified compilers [38], kernels [15, 21, 31], file systems [10, 11, 14], distributed systems [25, 88, 96], key-value stores [23, 36], and cluster controllers [63]. Industrial experience at Amazon [50] further demonstrates the practical value of formal methods for critical systems. These systems verify application-level logic (e.g., IronFleet [25] verifies a Paxos-based protocol) but typically rely on a simple I/O layer such as a thread pool. Replacing this layer with a modern async runtime would improve their performance, but as our bug study in §2.2 shows, doing so introduces the risk of subtle liveness failures.

Some of these verified systems do reason about liveness, but at a different level of abstraction. IronFleet proves that its replicated state machine eventually makes progress, and Anvil [63] proves that cluster controllers eventually reconcile desired and actual state. In both cases, the liveness argument assumes that all registered callbacks or messages are eventually delivered and that each processing step executes to completion once invoked. This is a natural and reasonable simplification: the I/O layer has traditionally been simple enough that the assumption appears self-evident. However, as async runtimes grow in complexity and adopt increasingly sophisticated scheduling, the assumption becomes a quietly error-prone part of the verification story. As our study of 11 Tokio bugs demonstrates (§2.2), satisfying this assumption in a real async runtime is far from trivial.

Lion addresses this overlooked layer. It provides an async runtime with machine-checked liveness, offering a verified foundation beneath existing verified software and closing a gap in the verification stack previously bridged only by trust. At the same time, it replaces a synchronous I/O layer with an efficient async runtime while formally guaranteeing the key assumption such systems rely on: every registered callback is eventually triggered.

3 Lion System Design

We now present Lion, an async runtime with formally verified liveness properties. Lion's design principle is to make module boundaries explicit; each module exclusively owns its internal state, and all cross-module communication uses indirection rather than shared mutable data structures. This clarifies module interfaces, simplifies reasoning, and facilitates the modular verification presented in §4–§5.

Rust's Async Model. To understand Lion's architecture, we first describe how Rust realizes async. As discussed in §2, async computation conventionally uses event loops and callbacks, but writing callbacks manually is error-prone and fragments control flow. Rust addresses this with `async/await` syntax, letting developers write sequential-looking code (upper-right in Figure 1) that executes cooperatively.

Hereafter, a *task* is an `async` function that contains one or more `.await` points (analogous to callback registration points in other languages). At compile time, Rust transforms each task into a state machine (upper-left in Figure 1), where each `.await` point becomes a *state*. The runtime drives a task by calling `poll()`. If the awaited resource is available, execution advances; otherwise it returns `Pending`. A task may yield and resume many times before reaching `Ready`. This mirrors what `rustc` actually does: it lowers each `async fn` to an anonymous `Future` whose fields hold the values live across an `.await` and whose `poll` resumes at the current await point; Figure 1 simplifies only the anonymous naming and `Pin` details.

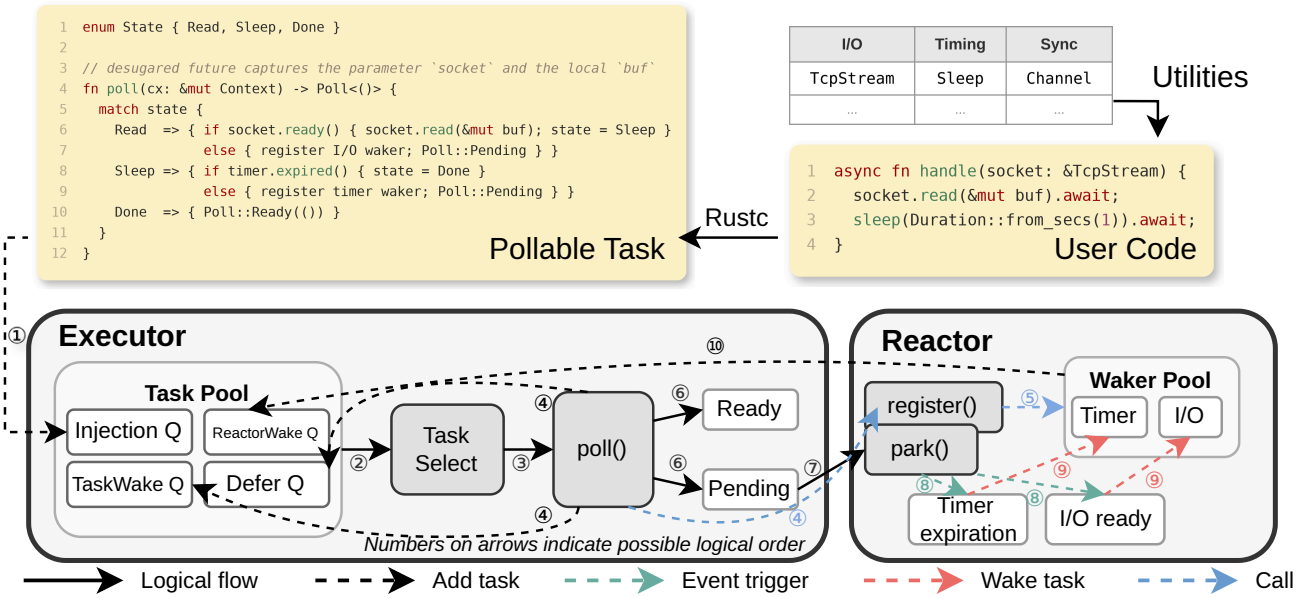


Figure 1. Lion runtime architecture.

Lion's Architecture. Lion has two core modules—an executor and a reactor—together with a layer of utility modules (Figure 1), each owning its own state. The *executor* drives the runtime. It maintains the task queues, selects tasks to run, and calls `poll()` to advance each task's state machine. When a task returns `Pending`, the executor sets it aside and moves on to the next; when a task returns `Ready`, it is complete. The *reactor* is an event monitor that accepts registration requests for I/O and timer events, waits for them, and notifies the executor via *wakers*, which re-enqueue the tasks for polling. *Utility modules* (e.g., `TcpStream`) interface user tasks with the core modules, hiding event registration and waker management behind async APIs.

Core Execution Loop. These modules interact through the *poll-register-wake cycle*, illustrated in Figure 1:

1. The executor pops a task from its queues and calls `poll()` (steps 2–3).
2. If the task needs an unavailable resource, it registers the event and a waker with the reactor (steps 4–5) via utility modules, then returns `Pending`.
3. When no runnable tasks remain, the executor parks on the reactor, until an event arrives (step 7).
4. When an event becomes ready (step 8), the reactor invokes the associated waker (step 9), which re-enqueues the task to the executor (step 10).
5. The executor polls the task, advancing it to the next state.

Additional Scheduling Paths. The core poll-register-wake cycle handles I/O and timer events through the reactor, but real-world runtimes need additional scheduling paths. Lion introduces dedicated queues for these paths, each adding a new suspension/resumption path:

- *Cooperative yielding* (step 4 → Defer Queue): Because the executor switches tasks only at await points, long computations can starve others. To prevent this, a task can call `yield_now().await` to move to the back of the Defer Queue and let other runnable tasks run first.
- *Utility modules* (step 4 → TaskWake Queue): The reactor handles OS events, but tasks may also wait on conditions produced by other tasks. These conditions are internal, so the waiting task stores its waker in the utility module, which enqueues the task into the TaskWake Queue when the condition is fulfilled, avoiding reactor overhead.

Every such path is a new place where the async contract (§2.2) can fail. A misplaced waker, forgotten enqueue, or incorrect queue drain order can silently stall tasks. Our verification must cover all paths so tasks stay live in every feature combination.

4 Specifying and Verifying Async Liveness

Liveness [2]—the guarantee that something good eventually happens—is fundamentally harder to verify than safety. The central liveness property we verify for Lion is that for any task spawned into the executor, there exists a bounded number of driving cycles within which the task is guaranteed to reach the `Ready` state, provided the task itself eventually finishes its own work rather than looping forever.

4.1 Overview

Proving this property is challenging for two reasons. First, the property is inherently temporal: it spans multiple function calls over an unbounded execution. Verus provides Hoare-logic contracts (pre- and postconditions per function) but no temporal operators such as TLA+'s ◇ (eventually) or □ (always). A function's contract can state "if X holds before the call, then Y holds after," but liveness requires stating "Z will *eventually* hold": a property that spans across many function calls. Second, the runtime comprises multiple modules (executor, reactor, utility modules), each with its own internal logic.

Verifying them as a monolith would be intractable; verifying them independently requires a composition mechanism that chains their guarantees into an end-to-end property.

Our key insight is that in a single-threaded runtime, execution forms a deterministic sequence of events. We model this sequence as an append-only *logical event log*, so that "event A at index i eventually leads to event B" becomes $\exists\, j > i.\ L[j] = B$, a first-order assertion expressible directly in Verus without language extensions.

We formalize each module's liveness obligation as an *async contract*: a pair of predicates (acceptance, fulfillment) over the event log, together with an assumption capturing environmental preconditions. End-to-end liveness is obtained by *contract chaining*: the fulfillment of one module's contract triggers the acceptance of the next.

The guarantee this yields is conditional, and deliberately so. We do not attempt to prove that user code is correct or that the environment cooperates; the framework instead takes both as explicit parameters of its liveness theorem, to be discharged when it is instantiated for a particular runtime. Two kinds of condition appear. The environment must behave: the clock keeps advancing, and a wake source that a task registers eventually fires. User tasks must be well behaved: a task eventually returns `Ready` rather than looping forever, and it keeps a resource it awaits registered until it is woken. What remains, and what we prove, is that the runtime itself never forgets a task. §5.2 states these conditions precisely and shows how Lion instantiates them.

The rest of this section develops the methodology progressively. §4.2 introduces event logs, progress invariants, and ghost instrumentation through a simple executor example. §4.3 wraps this machinery into async contracts and defines liveness. §4.4 shows how multiple contracts compose, both within one module and across modules.

4.2 Event Logs and Progress Invariants

We develop the core mechanism using a simplified version of Lion's executor.

Example 1 (Simplified Executor)**.** The executor accepts tasks through a single input queue. It is periodically driven; on each driving cycle (called a *tick*) it pops tasks from the input queue into an internal cache (a FIFO queue in this simplified example), then polls (executes) tasks from the cache.

The liveness property for this example is: *every task the executor accepts from the input queue is eventually polled.* In other words, the executor must not "lose" a task between accepting it from the input queue and executing it.

Modeling with a Logical Event Log. To formally verify this property, we need to track what the executor does over time. We capture its behavior as a logical *event log* L: a specification-level, append-only list that records actions visible at the module's interface. The log does not exist at runtime; it is maintained as a *ghost variable* in the executor's state, updated by each operation but erased during compilation (Figure 2).

In Example 1, each tick is bracketed by `Tick(Begin)` and `Tick(End)` events. Between them, the executor performs *cycle actions*: `Pop(Option`$\langle T \rangle$`)` (dequeue a task from the input queue into the cache, or `None` if empty) and `Poll(`T`)` (execute a task from the cache). In the implementation, the `tick` function appends cycle delimiters at entry and exit, and each cycle action (e.g., `poll`) appends its corresponding event. An example log trace:

$$\underbrace{\texttt{Tick(B)},\ \texttt{Pop}(T_1),\ \texttt{Poll}(T_1),\ \texttt{Tick(E)},}_{\text{pop one, poll one}}$$

$$\underbrace{\texttt{Tick(B)},\ \texttt{Pop}(T_2),\ \texttt{Pop}(T_3),\ \texttt{Poll}(T_2),\ \texttt{Poll}(T_3),\ \texttt{Tick(E)},}_{\text{pop two, poll two}}$$

$$\underbrace{\texttt{Tick(B)},\ \texttt{Pop(None)},\ \texttt{Tick(E)},}_{\text{empty queue}}\ \ldots$$

Because the log is append-only, log indices serve as logical timestamps: "event A happens before event B" becomes $i_A < i_B$, a first-order assertion. The system's behavioral policies are predicates over L, expressible with first-order quantifiers.

Step. To reason about liveness, we need to count how many driving cycles the module has executed. In Example 1, one step is one complete tick cycle:

$$\texttt{step}(L, L') \ \triangleq\ L \sqsubseteq L' \ \wedge\ \text{ticks}(L, L') = 1$$

where $L \sqsubseteq L'$ denotes that L' extends L, and $\text{ticks}(L, L')$ counts complete tick cycles in $L' \setminus L$.

Progress Invariant. We need an invariant over the log that guarantees the module makes progress each step toward completing its obligation (e.g., polling a cached task). We call this `progress_inv(`L`)`: a conjunction of *local liveness* properties, each stating that when a certain event occurs in the log, a corresponding outcome follows within a bounded number of events. For the executor in Example 1, the key local liveness property is *cached* $\Rightarrow$ *poll*: if the cache is non-empty after popping from the input queue, then at least one `Poll` occurs before the tick ends.

Crucially, `progress_inv` is an *inductive invariant*: if it holds before a step, it holds after. Each function proves that the events it appends preserve `progress_inv`.

From progress to liveness. The *cached* $\Rightarrow$ *poll* property guarantees that the executor makes progress each tick (at least one task is polled when the cache is non-empty). But this alone is *module-level progress*, not *task-level liveness*: it says "some task advances" but not "task T in particular advances." For liveness, we need to show that a specific task T, once it enters the cache, is eventually polled.

In our simplified example, the cache is a FIFO queue, so the argument is straightforward: each tick polls the task at the head of the cache, and T moves one position closer to the head each tick. If T enters at position k, it is polled within k ticks. In the full Lion executor (§5), additional invariants (such as FIFO ordering across multiple queue sources) ensure the same property for all scheduling paths.

This distinction between module-level progress and task-level liveness is central to how async contracts work (§4.3):

```
struct Executor {
  log: Ghost<Log>, // ghost log
  cache: Queue<Task>,
  // ... other fields
}

fn tick(&mut self)
  requires
    progress_inv(cached_implies_poll, old(self).log@),
    queue_matches_log(old(self).cache@, old(self).log@)
  ensures
    progress_inv(cached_implies_poll, self.log@),
    queue_matches_log(self.cache@, self.log@)
{
  self.log = Ghost(self.log@.push(Tick(Begin)));
  ... // pop from input queue into cache
  if !self.cache.is_empty() {
    // helper inv: cache non-empty
    // means acceptance is triggered at this tick
    let task = self.cache.pop_front();
    poll_task(task);
    self.log = Ghost(self.log@.push(Poll(task)));
    // Poll before Tick(End) discharges
    // obligation discharged: task polled
  }
  ...
  self.log = Ghost(self.log@.push(Tick(End)));
  // prefix monotonicity: old pairs preserved
}
```

Figure 2. Ghost-instrumented executor (simplified). The ghost log is maintained alongside executable code; each function's contract specifies how the log is extended and that `progress_inv` is preserved.

each contract captures a task-level liveness guarantee, built on top of the module's progress invariant.

We now walk through how `progress_inv` is proved in the `tick` function (Figure 2). The key idea is a *helper invariant*, `queue_matches_log`, that ties the cache to the ghost log: the runtime's `cache` always equals the sequence of tasks that have been popped from the input queue but not yet polled, according to the log.

The proof proceeds as follows. After appending `Tick(Begin)` and popping tasks from the input queue into the cache, the function checks whether the cache is empty. By the helper invariant, a non-empty cache means the acceptance condition of *cached* $\Rightarrow$ *poll* is triggered at this tick. The function takes the first task from the cache and calls `poll`, appending a `Poll` event to the log. Because this `Poll` occurs before `Tick(End)`, the obligation is discharged (the task is polled before the tick ends). All previously satisfied pairs carry over to the extended log by prefix monotonicity.

Soundness of the encoding. Figure 2 inlines the log update to keep the example readable, which raises the question of whether a `tick` could satisfy this contract without actually advancing the runtime. Two properties rule this out, both discharged by the verifier. A tick cannot be a *no-op* that logs progress it never made: `progress_inv` requires each step to grow the log strictly and to append exactly one complete `Tick(Begin)`…`Tick(End)` cycle, while *cached* $\Rightarrow$ *poll* forces the tick to poll a runnable task before it ends. Nor can a tick *never return*, which would leave the runtime stalled while satisfying its postcondition vacuously: the inner loops of `tick` carry decreasing measures, the sole exception, draining the injection queue, resting on the per-tick arrival set being finite (§5.2). A third property, that the log is append-only and cannot be rewritten at all, rests on a small trusted base rather than on the proof; we return to it in §5.2.

4.3 Async Contracts

The event log and progress invariant from §4.2 give us a mechanism for proving that a module makes progress each step. We now wrap this machinery into reusable abstractions.

We further wrap the progress invariant into an *async contract*, which packages a module's liveness obligation into a form suitable for composition.

Definition 1 (Async Contract)**.** An *async contract* has three components. We illustrate each using Example 1:

- **Acceptance**: the trigger event at which the module takes on an obligation, e.g., the executor pops task T from the input queue into the cache.
- **Fulfillment**: the discharge of that obligation, e.g., the executor polls (executes) T.
- **Assumption**: additional system-specific preconditions, e.g., task IDs are unique.

The contract is satisfied if, given the assumption, every acceptance is eventually followed by a corresponding fulfillment.

Formal definitions. Figure 3 formalizes the three components. The framework is parametric in two type variables: L, the module's event-log type (§4.2), and T, the task type that acceptance and fulfillment range over. For the executor of Example 1, L instantiates to the log of `Tick`/`Pop`/`Poll` events shown above and T to a queued task such as T_1. `ModuleSpec` packages a module's progress invariant and step predicate; `AsyncContract` packages the three contract predicates; and `liveness` ties them together, requiring that (1) each step preserves `progress_inv`, and (2) for every task whose acceptance has fired, if the assumption predicate holds, there exists an n such that fulfillment is guaranteed after any n steps along which the assumption continues to hold. Each module plugs in its own log type L, its own `progress_inv`, and its own `step` definition, making the framework reusable across modules.

4.4 Composing Async Contracts

The full Lion system has multiple async contracts, each covering one part of the path from accepting a task to fulfilling it. Different contracts are chained together to guarantee the end-to-end liveness property: the fulfillment of one contract triggers the acceptance of the next. Chaining rests on the shape every async contract shares: once its acceptance condition holds, its fulfillment condition holds within some bounded number of steps. Contracts with matching conditions therefore compose transitively. If one contract guarantees fulfillment F within n_1 steps of acceptance A, and F establishes the acceptance of a second contract that guarantees G within n_2 steps, then any run of n_1+n_2 steps

```
struct ModuleSpec<L> {
  progress_inv: spec fn(L) -> bool,
  step: spec fn(L, L) -> bool,
}

struct AsyncContract<L, T> {
  acceptance: spec fn(L, T) -> bool,
  fulfillment: spec fn(L, T) -> bool,
  assumption: spec fn(L, T) -> bool,
}

spec fn liveness<T, L>(
  ms: ModuleSpec<L>,
  ac: AsyncContract<L, T>) -> bool {
  // step preserves progress_inv
  forall |l, l'|
    (ms.progress_inv)(l) && (ms.step)(l, l')
    ==> (ms.progress_inv)(l')
  &&
  // eventual fulfillment
  forall |t: T, l: L|
    (ms.progress_inv)(l)
    && (ac.acceptance)(l, t)
    && (ac.assumption)(l, t)
    ==> exists |n: nat|
      forall |l'|
        step_n(ms.step, l, l', n, ac.assumption)
        ==> (ac.fulfillment)(l', t)
}
```

Figure 3. Liveness verification framework: module specification, async contract, and liveness definition.

from A splits at an intermediate state where the first contract has delivered F and the second takes over, so A guarantees G within n_1+n_2 steps. Because acceptance and fulfillment are first-order predicates over the event log, each link in the chain is an ordinary implication plus a trace-splitting lemma, and the whole argument discharges directly in Verus with no temporal-logic machinery.[1]

With async contracts in hand, the end-to-end liveness proof becomes straightforward: we show that on every scheduling path, the task is captured by an async contract. The heavy lifting (proving each contract's liveness) is done per-contract; the composition is just control flow. We illustrate with a scenario that builds on Example 1, where a pending task waits on a reactor timer:

- **Entry point:** T is spawned and enters the executor's input queue. This triggers the `input_queue_poll` contract, which proves that T is eventually polled. T's poll result is either `Ready` or `Pending`.
  - **If Ready:** T is finished. Liveness is satisfied.
  - **Else if Pending:** T registers a timer with the reactor.
    - This triggers the `timer_wakeup` contract, which proves that the timer eventually fires a wakeup signal.
    - The above action then triggers the `reactor_wake_poll` contract, which proves that T is eventually polled again by the executor.
    - The poll result is again `Ready` or `Pending`, returning to the case analysis above. Liveness then follows by induction on the number of polls T still needs.

Why the induction is well-founded. Two facts justify this last step. First, within a single poll the contract chain is *acyclic*: a poll that returns `Pending` registers a *fresh* wake source rather than re-entering the chain, so there is no inner cycle to reason about. The only induction is therefore on the outer back-edge, where T returns `Pending`, becomes runnable, and is polled again. Second, that back-edge carries a ranking function: the number of polls T still needs before reaching `Ready`, bounded by `cap` (§5.2). Each trip around the loop consumes one poll, so the count strictly decreases and is bounded below by zero. A bounded number of such trips, each itself bounded, yields a finite horizon, so the executor and reactor cannot loop forever.

Cross-module composition. The chaining above spans two modules (executor and reactor), each verified independently with its own event log. Since each module's proof only sees its own log, the composition proof must establish cross-module facts explicitly. For example, it must show that when the reactor fires a wakeup for a resource, the owning task actually appears in the executor's reactor-wake queue. These cross-module predicates are natural consequences of how the modules interact at runtime, but must be stated and proved as part of the composition layer. §5 details Lion's full set of cross-module predicates and the composed liveness proof.

Completeness of async contracts. Given that an async contract is a logical branch of the end-to-end liveness property, a natural question is how we know the set of async contracts we have specified is sufficient to establish the end-to-end liveness property. Note that async contracts do not create new behavioral guarantees; they decompose guarantees already encoded in the module's `progress_inv` predicate. When we prove the top-level property (every spawned task eventually reaches `Ready`), the verifier requires every logical branch to be covered. If a contract is missing, for example, if there is another wakeup path we omit, the proof fails because the verifier cannot discharge the branch where a task awaits on that path. Therefore, incomplete contracts result in a failed proof, not unsoundness. One limitation is that the verifier does not pinpoint *which* branch is missing. The proof simply fails, and the developer must inspect the specification to identify the gap.

§5 instantiates the full composition for Lion's runtime, detailing the composed state, cross-module predicates, and the end-to-end liveness proof.

5 Lion Liveness Verification

We now apply the framework from §4 to Lion's full runtime. The property we verify is: *every task spawned into Lion eventually reaches `Ready`.*

5.1 Per-Module Specifications

Each component of Lion's runtime is specified as a `ModuleSpec` (§4.3). The executor and reactor are core

[1] Readers familiar with temporal logic may recognize each async contract as a bounded TLA *leads-to* ($A \rightsquigarrow F$), and chaining as its transitivity.

modules, each with their own async contracts. Utility modules contribute per-task invariants and a cross-task wakeup contract.

Executor. The executor accepts tasks from four queue sources (injection, reactor-wake, task-wake, and deferred) and provides four async contracts, one per source, each guaranteeing that a task entering that queue is eventually polled. A step is one tick cycle, matching Example 1 from §4. The progress invariant is a conjunction of 9 local liveness properties ensuring FIFO task selection, valid polling, and per-tick progress (every tick parks the reactor, drains all wake queues, and polls if the cache is non-empty).

Reactor. The reactor monitors timer and I/O events and provides two async contracts: *timer wakeup* (a registered timer eventually fires) and *I/O wakeup* (a registered I/O resource eventually fires). A step is one park cycle, the action through which the executor drives the reactor. The progress invariant is a conjunction of 19 local liveness properties covering correct registration, local progress (each park cycle advances the clock and polls I/O), waker integrity, and bounded responsiveness (expired timers and ready I/O produce wakeups within the same park cycle).

Utility Modules. Each utility module (e.g., `TcpStream`, `Semaphore`) provides an async contract for inter-task wakeup: the *pass-waker contract*. A typical pattern: a task calls `recv()` on a channel, registers its waker, and returns `Pending`; later, another task calls `send()`, which invokes the stored waker, resuming the waiting task. The pass-waker contract captures this: once a waker is registered (acceptance), driving the utility sufficiently many times guarantees the waker is invoked (fulfillment). The assumptions are utility-specific: a channel needs a matching send, a semaphore needs a permit release, and so on. Violating this contract is a concrete source of liveness bugs: in Issues 3117 and 6751 (Appendix A) a utility fails to invoke a registered waker, leaving the waiting task hung, while Issues 3168 and 4814 break the same wakeup protocol in other ways—a stale notification presented as fresh, and an index that hides closure from a resubscribed receiver.

In addition to this liveness contract, each utility enforces two per-task safety properties that the composition proof relies on: *wakeup guarantee* (every `Pending` return has at least one active wakeup source) and *resource ownership* (reactor operations target only resources belonging to the owning task). The wakeup guarantee ensures the composition proof can always identify which wakeup path a pending task is waiting on; the resource ownership ensures that cross-module predicates can correctly route wakeup signals to the right task. The composition proof depends only on these abstract obligations, which each utility discharges independently.

Supporting invariants. Each contract's leads-to step rests on module safety and ownership invariants. Consider the reactor's *timer wakeup* contract (a registered timer eventually fires). Four safety invariants keep the registration well-formed: each resource id names at most one live timer, a registered timer's deadline lies in the future, every wake traces to a real registration with a valid waker, and each park cycle stamps the clock and polls exactly once. With monotonic time, the deadline is then provably reached, and an expired timer produces a wake. An *ownership* invariant closes the branch: a task owns exactly the timers it registered, so no other task can retire the timer first. In all, 30 such invariants (9 executor, 19 reactor, 2 utility) uphold the end-to-end leads-to chain.

5.2 Assumptions and Scope

Lion's proof establishes that *the runtime never forgets a task*: once a task's awaited event fires, the runtime polls it again within a bounded number of steps. The guarantee is *bounded* and *filtered*: it holds along runs whose every state meets the assumptions below, none of which is mechanically verified. Two of them constrain the world outside the runtime, the environment and user tasks; a third, the append-only ghost log, is trusted code inside it.

Environment. Facts the runtime cannot force. The clock keeps advancing, so timestamps strictly increase, an idealization of the millisecond-granularity system clock. Every wake source a task registers eventually fires within a bound: a timer deadline is reached, a registered I/O becomes ready within a bounded number of polls, and a waker handed to another task is eventually invoked. Submitted tasks arrive in a fixed, finite order and the runnable queue stays bounded, so an unbounded producer cannot starve the runtime.

User tasks. Constraints on user code. A task returns `Ready` within a bounded number of polls, since one that loops forever legitimately never completes. It keeps a resource it awaits registered until woken, it does not cancel another task's pending wait, and task identities are unique.

Trusted log. The ghost log every theorem quantifies over is *append-only by construction* rather than by proof, so it is part of the trusted base. The log is private, mutated only by a small family of `external_body` wrappers whose `ensures` clauses state only that the log grows by one fixed event, framing all other state as unchanged. A single macro per module generates this whole family, so the trusted "log shape" is audited once and no code can rewrite or truncate the log. The complementary properties—no step is a no-op, and `tick` terminates—are discharged by the verifier (§4.2).

Beyond these three, nothing else is taken on faith. The runtime's own module invariants are not assumed but discharged by the separately verified `lion-executor` and `lion-reactor`: FIFO scheduling and draining, timely timer and I/O wakeup, and per-task resource ownership.

Scope. The guarantee is that the runtime never drops a task, not that the application makes progress: *whole-application liveness is out of scope.* Whether a task completes depends on user logic; task abort mid-wait is unmodeled.

5.3 Composing End-to-End Liveness

The end-to-end proof follows the composition pattern from §4.4, instantiated with Lion's specific modules and contracts.

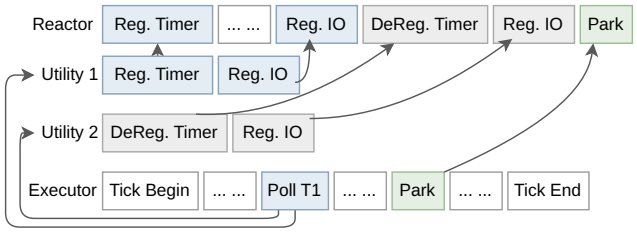


Figure 4. Cross-module log alignment.

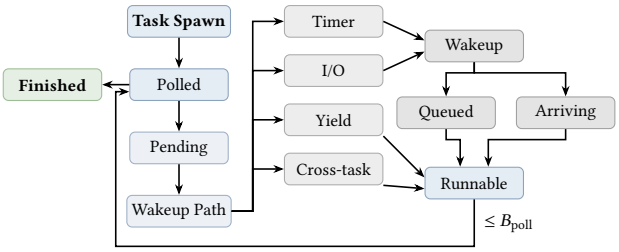


Figure 5. End-to-end proof flow.

Top-level specification. Because the runtime never returns, Lion's guarantee is not a postcondition but a property of the ever-growing event log: from any legal state where a task is waiting, its poll event appears in the log within a bounded number of steps. This is a first-order property of log positions, which Verus proves without temporal operators. Composition is also closed: the executor, reactor, and utility specifications combine into a single composite of the same shape, so the top-level guarantee is that same specification applied to the whole runtime.

Cross-module predicates. As discussed in §4.4, composing independently verified modules requires cross-module predicates that relate their separate logs. Lion uses 21 such predicates, grouped into five families, covering operation correspondence (task operations match reactor events, park counts are consistent between executor and reactor) and wakeup routing (a reactor wakeup for resource *rid* places the owning task in the correct executor queue). Figure 4 illustrates how the executor, reactor, and utility logs correspond within a single tick: each executor action (poll, park, drain) has matching events in the reactor and utility logs, connected by these cross-module predicates.

End-to-end proof. The proof follows the if/else structure from §4.4; a flow chart is shown in Figure 5. A spawned task enters the injection queue; the executor's `input_queue_poll` contract guarantees it is polled. If the task returns `Ready`, it is finished. If it returns `Pending`, the user code assumption (§5.2) ensures at least one wakeup path is active. Lion's runtime has four such paths:

- **Timer**: the reactor's timer wakeup contract fires the wakeup; a cross-module predicate routes the task to the reactor-wake queue.
- **I/O**: same pattern via the reactor's I/O wakeup contract.
- **Yield**: the task defers itself into the deferred queue.
- **Cross-task**: the pass-waker contract (§5.1) guarantees the waker is invoked, placing the task in the task-wake queue.

	Exec (LoC)				
Component	**Verified**	**Trusted**	**Wrapper**	**Adapter**	**Spec+Proof**
Utilities	595	0	3	1,311	2,006
Executor	451	240	230	166	2,844
Reactor	1,651	203	181	60	9,860
Collections	696	64	0	0	3,906
Contracts + Composing	0	0	0	0	21,014
Total	**3,393**	**507**	**414**	**1,537**	**39,630**

Figure 6. LOC breakdown by component.

In all four cases, the corresponding executor contract guarantees the task is polled again. The cycle repeats until the task returns `Ready`.

Verification effort. Figure 6 summarizes the code breakdown across all verified components. Executable code totals 5.9k lines, of which 3.4k are verified by Verus. Of the rest, 0.5k are primitives whose contracts the proofs rely on but whose implementations the verifier accepts unchecked, the trusted base, inventoried item by item in the artifact. The remaining 2.0k lie outside the verified perimeter altogether, mostly the Tokio-compatible I/O surface (TCP/UDP sockets, timeouts, semaphores) wrapped into `Future`/`poll` form. The specification and proof burden is 39.6k lines, roughly 11.7× the verified executable code.

We used an AI assistant (Claude) heavily, but with a human in the driver's seat. The authors designed the specifications, invariants, and proof decomposition, while the assistant filled in mechanical details (boilerplate, quantifier instantiation, syntax) once the structure was fixed; fully autonomous proof repair did not work at this scale. The proof burden reported above thus reflects genuine structural and design effort, not mechanically generated bulk. To gain further confidence that proofs written with machine assistance actually constrain the implementation, we also ran a mutation study (Appendix B), using an LLM to inject liveness-breaking faults into the verified scheduling logic; the verifier rejects every one.

Trusted base. The 0.5k trusted lines divide into three kinds of code. The *shims* give Verus contracts to operations it cannot model: the `mio`/`epoll` I/O and timer syscalls, thread-local storage, and the append-only ghost log every theorem quantifies over (§5.2). The *adapters* wrap each verified kernel with `Arc<Mutex>`/`Future` implementations for Tokio-compatible APIs, delegating all state transitions to the verified inner type. The remaining structural *wrapper* glue is invisible to the proof and outside the trusted base.

6 Discussion

Single- vs. multi-threaded runtimes. Lion's verified core is single-threaded: one executor, one reactor, and one event loop per thread. Tokio, by contrast, offers a multi-threaded work-stealing runtime where idle workers steal tasks from other workers and share a single reactor and timer wheel.

The thread-per-core model, adopted by Monoio [6], Glommio [17], and Seastar [57], pins one event loop to each core with no shared state between cores.

The practical difference between the two models emerges when task sizes are uneven. Under work-stealing, workers share a logical run queue: a heavy task still runs to completion on one worker, but the requests queued behind it are *stealable* onto idle cores. Under thread-per-core, those requests are pinned behind the heavy task and wait until it finishes or reaches its next await point.

Heavy tasks of this kind are discouraged in the first place. Async runtimes ask programmers not to run CPU-heavy work inline in a task [67, 78]. In Oxide's Omicron, for example, a background task parsing TLS certificates held a worker for 10.9 s and stalled the task that hands out database connections, hanging API requests for 8–10 s [51]. Both documented remedies change the application: offload the computation to a separate pool [67], or split it with explicit yield points. In this light, work-stealing is a mitigation: it cannot shorten the offending task, only unpin the requests behind it. Even the mitigation has limits: in the Omicron incident the stalled task sat in Tokio's LIFO slot, the one queue no other worker may steal from [73]. Whether a runtime should provide such a mitigation remains an active debate.

Lion's design aligns with the thread-per-core philosophy. For multi-core deployments, Lion launches independent per-core instances. The liveness guarantees proved for a single instance carry over to every core. A nested local executor of the kind Tokio's `LocalSet` provides is unnecessary in this model, since no task ever migrates, but it lies within the framework's reach: its obligation is an async contract in the sense of Definition 1 (acceptance: a driver registers its waker with the set; fulfillment: the set invokes that waker), and the hang of §7.4 is precisely a fulfillment that never occurs. Extending the verification to cover cross-thread work-stealing, where tasks migrate between cores and shared queues require concurrent reasoning, is a challenging direction for future work. We quantify the cost of forgoing work-stealing in §7.3.

Single-threaded does not mean simple. One might expect that a single-threaded runtime is straightforward enough to not need formal verification. Our bug study (§2.2, Appendix A) shows otherwise: the majority of the confirmed liveness bugs in Tokio occur in single-threaded scheduling logic, not in multi-threaded synchronization. Bugs like Issues 5020 and 3069 (§2.2) involve purely sequential reasoning about the interaction between the executor, reactor, and utility modules. Even without concurrency, the combinatorial complexity of task states, wakeup paths, and queue interactions is sufficient to produce subtle, silent liveness failures that evade testing for years.

While single-threaded execution eliminates concurrent interleavings, conventional runtimes still allow modules to share mutable pointers to tasks and internal structures, making it difficult to reason about each module's behavior in isolation. Lion goes further by giving each module exclusive ownership of its state (§3), so that each can be specified and verified independently. These choices trade some micro-level efficiency for explicit module interfaces, with negligible impact on end-to-end performance (§7).

7 Evaluation

Our evaluation is conducted to answer three questions:

1. What is the performance overhead of Lion's verified design compared to unverified runtimes? (§7.1–§7.3)
2. Does Lion's verified scheduling prevent liveness bugs that affect unverified runtimes under stress, and do the proofs themselves forbid such hangs, or would a broken scheduler pass them just as well? (§7.4)
3. Can Lion improve the performance of existing verified systems by enabling event-driven I/O? (§7.5)

7.1 Micro Benchmarks

Our micro benchmarks cover the core runtime paths where Lion's verified design is most likely to impose overhead:

- *Timer cancel*: each task repeatedly registers a timer and immediately cancels it, generating a high rate of timer insertions and removals handled entirely within the runtime (no system calls), directly stressing the verified timer scheduling path.
- *TCP echo*: each round-trip traverses the full runtime I/O path (epoll readiness notification, waker invocation, and task re-scheduling), exercising the critical I/O hot loop under realistic network concurrency.
- *Filesystem I/O*: tasks perform concurrent file reads and writes, exercising the cross-thread wakeup path where a blocking worker thread notifies the async runtime to resume a waiting task.

For each benchmark we measure both single-thread performance and multi-thread scaling. The evaluation is conducted on a local cluster (EPYC 7702P, 64 cores / 128 threads).

We compare three async runtimes: Tokio and Lion (`epoll`-based) and Monoio (`io_uring`-based). As discussed in §6, Lion and Monoio follow the thread-per-core model with no shared state between cores, while Tokio uses work-stealing with shared state. Monoio is excluded from multi-thread comparisons because its I/O operations cannot migrate across threads. To isolate the scheduling strategy from runtime implementation differences, we also construct *Tokio-Partition*: multiple independent Tokio single-thread runtimes behind the same round-robin interface as Lion, matching Lion's architecture but using Tokio's runtime kernel.

Timer cancel. Each task registers a long timer, immediately cancels it via a short 1 ms timer, and repeats. In the single-thread configuration (Figure 7(a)), Lion leads Tokio by 1.5× at 1K concurrent timers (563K vs. 379K ops/s) and by 1.4× at 5K, benefiting from Lion's more lightweight timer implementation. Only at 10K does Tokio pull ahead, by 8% (2.86M vs. 2.64M ops/s). Tokio's timer wheel is heavily optimized with pointer-based shared state and a fast-path timer reset that bypasses locking; Lion's verification-friendly design avoids

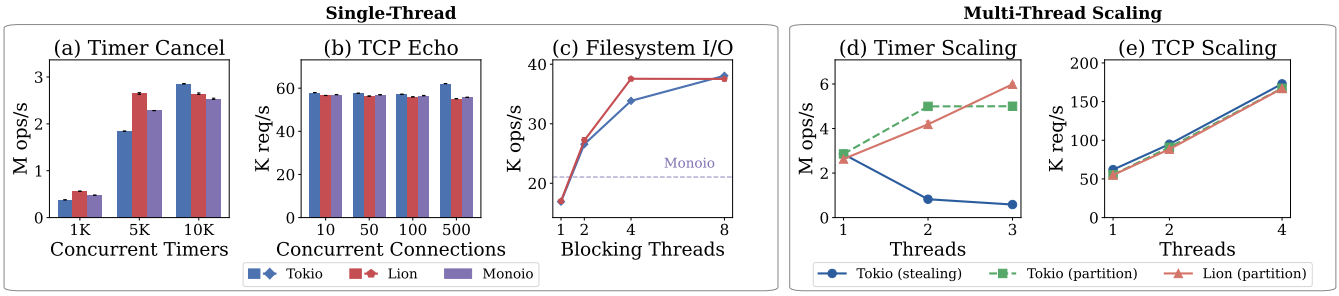


Figure 7. Micro benchmark results (10 runs; trimmed mean ± std shown as error bars, the two highest and two lowest runs dropped). *Single-thread* (left group): (a) Timer cancel throughput under varying concurrency. (b) TCP echo throughput under varying connection counts. (c) Filesystem I/O throughput vs. blocking pool size (50 concurrent tasks). *Multi-thread scaling* (right group): (d) Timer cancel throughput vs. thread count (10K timers total, split across threads). (e) TCP echo throughput vs. thread count (500 connections, SO_REUSEPORT).

shared mutable state, trading this fast path for modular ownership, a cost that surfaces only at the highest timer churn. Monoio sits between the two at 1K and 5K and is last at 10K.

Multi-thread scaling (Figure 7(d)) reveals the architectural divergence: with a fixed load of 10K timers split across the threads, Lion is the only configuration that keeps scaling, reaching 2.3× its single-thread rate on three threads (2.63M to 5.99M ops/s). Tokio-Partition tracks Lion to two threads but then plateaus at 5.00M. Tokio work-stealing *degrades* outright—from 2.85M ops/s on one thread to 0.59M on three—due to cross-thread synchronization overhead on its shared timer wheel (§6).

TCP echo. A server accepts connections and echoes 64-byte messages in a ping-pong loop. In the single-thread configuration (Figure 7(b)), Lion matches Tokio within 2.3% at 10, 50, and 100 connections, and Monoio tracks Lion to within 1.2% throughout—the three runtimes are indistinguishable while the connection count is modest. At 500 connections Tokio pulls ahead of both by 11–13% (62.1K vs. 55.1K req/s). For multi-thread scaling (Figure 7(e)) with 500 connections, all three configurations (Tokio, Tokio-Partition, and Lion) perform within 3.5% of each other at 4 threads. In this I/O-bound scenario the scheduling strategy matters little: Lion's verified runtime adds no measurable overhead.

Filesystem I/O. Since `epoll`/`kqueue` do not support OS-level asynchronous file operations, Tokio and Lion submit file reads and writes to a user-space blocking thread pool, waking the submitting task upon completion. In Figure 7(c), we measure throughput under 50 concurrent file tasks as the blocking pool scales from 1 to 8 threads. Tokio and Lion show similar scaling, from ~17K ops/s with one blocking thread to ~38K with eight; Lion leads by 11% at four threads and the two converge at eight. Monoio achieves 21K ops/s with a single thread using OS-level async file I/O via `io_uring`, requiring no thread pool, but Tokio and Lion surpass this once their blocking pool reaches two threads.

Across all three benchmarks, Lion is competitive with Tokio in single-thread performance and scales comparably or better with multiple threads: formal verification adds no significant overhead.

7.2 Real-World Applications

To evaluate whether Lion can serve as a practical drop-in replacement for Tokio in production software, we port two production open-source Rust applications and build a server on a third open-source framework; together they span the major async runtime usage patterns: network middleware, HTTP serving, and filesystem I/O.

rumqtt (30K LOC) is a production MQTT message broker for IoT deployments, exercising long-lived TCP connections, per-client task spawning, keep-alive timers, and pub/sub dispatch. **Pingora** (75K LOC) is Cloudflare's HTTP proxy replacing nginx, exercising connection pooling, HTTP protocol processing, and high-concurrency TCP forwarding. **Axum** is one of the most popular Rust web frameworks, on which we construct a static file server exercising the full stack including async file reads.

Because Lion implements a Tokio-compatible interface, the changes needed in porting each codebase are small, ranging from 54 to 353 changed lines (added plus removed, over source and manifest files) per port. Both Tokio and Lion are configured with a single-threaded runtime for controlled comparison. For rumqtt, the server runs on one machine and clients on a separate machine connected via 1 Gbps Ethernet (~0.25 ms RTT); Pingora runs in its canonical single-host configuration, with the server and the load generator on the same machine. For Axum, we report both a cross-server configuration (same network setup, ~0.38 ms RTT) and a localhost configuration; cross-server throughput is bandwidth-limited at 1 Gbps for all three workloads, so the localhost numbers better isolate runtime differences.

Figure 8 shows throughput across all configurations. rumqtt's three MQTT workloads (Fanout: 1 publisher to 50 subscribers rate-limited; Fanin: 50 publishers to 10 subscribers; P2P: 50 independent pub/sub pairs) place Lion within 4% of Tokio, 3% ahead on Fanout and inside run-to-run variation on Fanin and P2P. Pingora's HTTP echo workloads (low-concurrency at 50 connections, high-concurrency at 200 connections, and large-payload at 10 KB) show Lion within 3% of Tokio on all three—low-concurrency inside the run-to-run spread, with small but consistent Lion

App	Workload	Tokio	Lion	Ratio
rumqtt (cross)	Fanout (Kmsg/s)	38.1±0.1	39.4±0.6	103%
	Fanin (Kmsg/s)	45.9±0.4	45.6±0.6	99%
	P2P (Kmsg/s)	186.5±3.1	184.3±4.7	99%
Pingora (local)	Low-conc (Kreq/s)	69.2±1.3	69.0±2.3	100%
	High-conc (Kreq/s)	64.3±1.2	66.1±1.8	103%
	Large-10KB (Kreq/s)	12.6±0.1	12.9±0.0	102%
Axum (cross)	API-4KB (Kreq/s)	27.8±0.0	27.8±0.0	100%
	Static-64KB (Kreq/s)	1.8±0.0	1.8±0.0	100%
	Mixed (Kreq/s)	7.1±0.0	7.1±0.0	100%
Axum (local)	API-4KB (Kreq/s)	46.2±0.9	54.9±1.2	119%
	Static-64KB (Kreq/s)	22.9±0.3	24.5±0.4	107%
	Mixed (Kreq/s)	39.7±0.7	43.0±0.8	108%

Figure 8. Real-world application throughput. *cross*: server and client on separate machines over 1 Gbps Ethernet; *local*: server and load generator on the same machine. rumqtt rows are subscriber-side delivered throughput; the MQTT client's publish call returns once the message is queued locally, so its send rate does not measure broker throughput. Cells are trimmed means ± sample standard deviation over 10 runs (the two lowest and two highest runs are dropped).

advantages on the other two. Axum's cross-server results are bandwidth-limited at 1 Gbps, yielding indistinguishable throughput for both runtimes. The localhost configuration removes this bottleneck: Lion leads on all three workloads, with the API workload 19% faster, the mixed workload 8% faster, and the static workload 7% faster.

Across all nine workloads (using Axum localhost), Lion achieves 99–119% of Tokio's throughput. These results demonstrate that a formally verified async runtime can serve as a practical drop-in replacement for Tokio in production Rust applications, with negligible performance cost.

7.3 Work-Stealing vs. Thread-per-Core

Lion adopts the thread-per-core model: each connection is pinned to the core that accepts it, and tasks never migrate across cores (§6). The goal of this section is to quantify the cost of that choice: when does work-stealing outperform thread-per-core, and how much of the gap does Lion recover with its verified `yield_now` primitive?

The two models differ most when task sizes are highly uneven: a heavy task holds its worker either way, but under work-stealing the requests queued behind it are stealable onto idle cores, whereas under thread-per-core they wait for it to finish or reach its next await point. Such variance is ordinary in practice: in a production search service the largest requests exceed the smallest by over 40× [40].

We reproduce this mechanism with an Axum endpoint that runs a log-normal number of SHA-256 rounds per request (mean 4.3 ms, coefficient of variation 5.3), so that a rare heavy request occupies a worker while light ones queue behind it. We compare three arms—inline, inline with yield-chunking, and Tokio's work-stealing baseline—under load offered open-loop from 20% to 90% of the 8-core capacity, plotted against

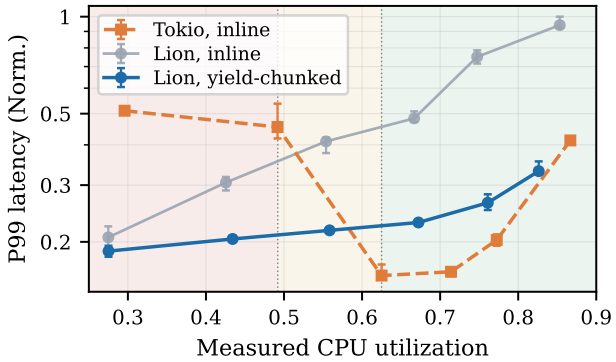


Figure 9. Work-stealing vs. thread-per-core under high-variance CPU load (Axum, CV=5.3, 8 cores, 10 runs; median with interquartile range; p99 normalized to the peak). Run inline, Lion pays >3× Tokio's p99 at high utilization; cooperative yield-chunking, using Lion's verified `yield_now`, reduces this to ≤1.5× and wins at low-to-moderate load. Throughput is within 1% across all three (not shown).

the measured utilization (Figure 9). Sweeping load is what makes the comparison meaningful: neither model dominates at every load [40], so a single operating point would let either side claim the win. Throughput stays within 1% across all runtimes, so the difference is confined to the tail.

The three shaded regimes separate two distinct costs. In the first, Tokio's p99 is 2.5–2.7× that of either Lion arm—not because stealing fails, but because it is delayed. Tokio parks idle workers: at 20% offered load they are parked 70% of the time, 14.9 ms per park, so a request queued behind a heavy one waits for a worker to be unparked before it can be stolen (steals still occur, ~0.5 per request). Lion's per-core resident reactor has no unpark step; pinning a busy loop to every Tokio worker so that none ever parks collapses the probe p99 from 509 ms to 5.2 ms. This is a tunable implementation choice, not a property of work-stealing. In the second regime parking shrinks with load—9% of the time and 0.8 ms per park at 90%—the gap closes, and the curves cross. In the third, the shared run queue earns its cost. Pinning a heavy request to one core leaves Lion's per-core utilization an order of magnitude more uneven than Tokio's (standard deviation 0.055–0.207 vs. 0.003–0.028), and the tail is set by the busiest core: at 65% mean utilization Lion's busiest core runs at 89%. Run inline (the grey curve), Lion therefore pays over 3× Tokio's p99; chunking with Lion's *verified* `yield_now` (blue) restores interleaving and bounds the loss to ≤1.5×. Toward saturation queueing dominates both models and the gap narrows again. What remains is a capacity deficit only cross-core migration can close—the boundary of the thread-per-core model.

7.4 Correctness Under Stress

To evaluate whether Lion's verified liveness guarantees hold under extreme scheduling pressure, we design a combined stress test whose workload is derived from the bug patterns identified in §2.2 and Appendix A. Each task class targets

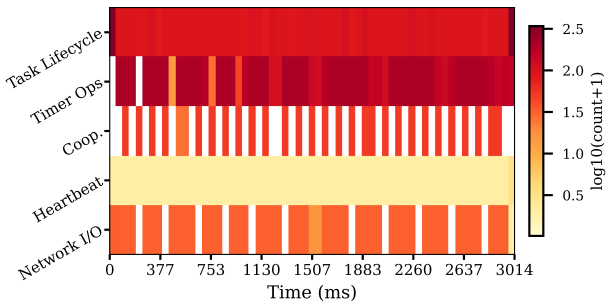


Figure 10. Runtime event heatmap during stress test.

Field	Tokio	libevent	libuv	Lion
Version(s)	1.21, 1.42, 1.44, 1.52.3	2.1.5-beta, 2.1.11	1.43.0	—
Issue(s)	#5020, #7209/#7210	#237, #984	#3503	—
Result	Hang	Hang	Hang	**Pass**

Figure 11. Correctness stress test results across Tokio (Rust), libevent (C), libuv (C), and Lion.

a specific failure mode observed in prior bugs. 200 request tasks each wrap a 1 ms sleep in a 5 s timeout, so every iteration registers a timer and cancels it once the sleep wins the race (~510,000 registration/cancellation pairs over the run), stressing small-duration timer registration and cancellation (Issue 3069 [70]). 50 cooperative-compute tasks interleave `yield_now` with short sleeps (~43,500 yields), exercising the deferred scheduling path. 10 scatter/gather coordinators repeatedly spawn a wave of 10 children and join them, creating 54,750 child tasks and exercising the spawn and wakeup-notification paths. A heartbeat task ticks every 50 ms to check liveness under sustained load, and a TCP echo server serves 48 waves of 15 concurrent clients (720 connections) to exercise I/O event handling; the multi-threaded configuration additionally runs a blocking-offload worker, combining multiple subsystems simultaneously. In total the workload keeps ~400 tasks concurrently live, spawns over 56,000 tasks, and drives more than 550,000 timer and yield operations in 3 seconds. We implement this workload in both Rust (for Tokio and Lion) and C (for libevent and libuv), adapting the concurrency primitives to each runtime's API. The C harnesses preserve the same five-class stress pattern at a reduced scale (roughly one-tenth the operation count), and include the library-specific API sequences that the listed issues document.

Figure 10 shows the event density across five categories over time. All event types remain active throughout the test, confirming that Lion's verified scheduler makes progress on every task under sustained load. Each cell is run three times under a 15 s timeout, comfortably above the workload's longest honest critical path of 8 s (a 3 s deadline plus the 5 s backstop timeout), so a run that does not terminate within it can only mean a permanent stall, which we record as a hang. Figure 11 summarizes the results. Lion passes 3/3, consistent with its verified liveness guarantees; the listed versions of Tokio, libevent, and libuv hang 3/3 on a confirmed liveness bug triggered by the stress workload. These bugs reside in different subsystems across different runtimes, illustrating that liveness failures are not specific to any one implementation but are an inherent challenge of async runtime design.

Tokio's most recent version, release 1.52.3, exhibits a hang as well. We trace it to `LocalSet`: it violates the pass-waker contract of §5.1. The set takes custody of its driver's waker, a task on the set becomes runnable, and the waker is never invoked, so that task never runs again. The hang reproduces deterministically on a single-threaded runtime, with no interleaving to search for, so it is a logic defect rather than a race. It has the same lost-notification shape as Issue 5020 [74], in the same file, and the relevant code was already present when that issue was patched [76]. Applying that repair to 1.52.3 makes the hang disappear, which attributes it to the defect rather than to how we exercise the API.

Unlike the more slowly evolving C runtimes above, Tokio improves continuously and at a rapid pace. That pace is valuable, but it also leaves room for residual liveness risk: Issue 5020 was itself introduced by an optimization that suppresses a notification, and confirmed hangs continue to surface, two with fixes open but as yet unmerged [85, 86],[2] and two with no proposed fix yet [82, 84], one of them after more than a year. A verified liveness approach is aimed at exactly this setting: it lets a runtime keep evolving while still ruling out silent scheduling hangs.

7.5 Improving Verified Systems with Async I/O

To study the potential benefit Lion has over existing verified systems, we integrate Lion into a verified Multi-Paxos implementation provided by IronFleet [25]. IronFleet's Paxos protocol is verified in Dafny and compiled to C#. We preserve this verified Paxos core unmodified (adjusting only two protocol constants, identically in both configurations) and build a Lion-based async I/O layer around it via a Rust shared library loaded through FFI, replacing the original C# IoFramework. The Paxos core logic is identical in both configurations; only the I/O layer differs.

The C# IoFramework creates two dedicated threads per TCP connection (a receiver and a sender), plus per-replica dispatch and listener threads, totaling 7+ threads per replica in a 3-node cluster. End to end, every message crosses 2–3 thread boundaries, each a queue handoff; those on the send path additionally incur a kernel thread wake. The Paxos main loop uses a busy `while(true)` spin, consuming 100% of one core even when idle. Lion replaces this with an async I/O layer where readers, writers, and connection management run as lightweight async tasks on a single background I/O thread. The Paxos main loop remains a C# busy spin (since it is Dafny-generated and cannot be restructured), but the Lion I/O layer implements idle-aware blocking: after 64 consecutive empty receives, the receive call blocks for up to 50 ms on the inbound channel, freeing CPU for the I/O thread sharing the same core.

[2] #8320 was opened on 25 July 2026 and #8331 on 29 July 2026, shortly before this revision, and both were still unmerged when we checked.

	Lion		C# IO	
Metric	**unpin**	**1-core**	**unpin**	**1-core**
Throughput (req/s)	3,330	2,005	1,635	329
Avg latency (ms)	0.50	0.85	1.05	5.28
Peak replica CPU (%)	138	86	505	102

Figure 12. IronFleet Paxos: C# threaded I/O vs. Lion

We deploy a 3-replica cluster driven by a remote client with 2 concurrent connections and no request batching. Figure 12 shows results under two configurations: unpinned (all cores available) and pinned to a single core per replica.

In the unpinned configuration, Lion achieves 2.0× the throughput of the C# backend (3,330 vs. 1,635 req/s) while using only 27% CPU of the C# backend (138% vs. 505%). The C# backend's 505% CPU usage reflects the overhead of 7+ threads per replica. We attribute Lion's 138% chiefly to the Dafny-generated Paxos busy loop on the .NET main thread rather than to the I/O layer: the core is compiled C# that cannot be restructured into async tasks, so the spin lies outside Lion's control and a native async implementation would remove it. We sample CPU per process, not per thread, so we do not split the figure further.

To observe performance under the same limited CPU budget, we pin each replica to a single dedicated core. Lion achieves 2,005 req/s while the C# backend achieves 329 req/s, a 6.1× gap. The difference illustrates the benefit of cooperative async scheduling over preemptive kernel threading: with multiple kernel threads competing for a single core, the OS scheduler must interleave the Paxos spin loop with I/O threads, and the spin loop consumes most of the time slice. Lion's async tasks yield cooperatively, allowing I/O processing to proceed without contending for kernel scheduling.

These results suggest that replacing the simple threaded I/O layer with Lion in existing verified systems could significantly improve performance and reduce resource usage without introducing liveness bugs.

8 Related Work

Liveness guarantees. Liveness [2] is notoriously harder to guarantee than safety. Killian et al. [29] find liveness bugs via state-space exploration but only cover explored executions. Converos [64] applies TLA+/TLC model checking to Rust OS kernel modules, systematically covering more behaviors but remaining limited to finite state spaces.

Deductive verification provides stronger guarantees by proving that all execution paths satisfy liveness. IronFleet [25] and Anvil [63] verify liveness of replicated state machines and cluster controllers using TLA-style temporal operators [32, 33] encoded in Dafny [37] or Verus [35], but this requires a temporal-logic layer atop the verifier and a state-machine encoding of the implementation. At a finer granularity, Kim et al. [30] verify liveness of the MCS lock, Padon et al. [52] reduce liveness to safety in first-order logic, and Performal [94] verifies latency bounds of distributed protocols. Lion also uses deductive reasoning but takes a different approach: it models each module's execution as an append-only logical event log, reducing temporal relationships to first-order assertions over log indices. This requires no temporal-logic infrastructure, no state-machine encoding, and no extension to the verification language. Lion is the first to verify end-to-end liveness of a complete async runtime.

Rust verification with Verus. Verus [34, 35] is an SMT-based verification tool for Rust that has enabled a growing ecosystem of verified Rust systems: Anvil [63] verified Kubernetes controllers, VeriSMo [97] verified a security module for confidential VMs, PoWER [36] verified crash-consistent storage, CortenMM [91] verified memory management, and Atmosphere [15] verified a full-featured microkernel, all in Rust using Verus. These systems demonstrate that SMT-based verification of Rust can scale to practical systems while preserving good performance. Lion continues this trend by applying Verus to verify an async runtime, a system layer that is both performance-critical and, as our bug study shows (§2.2), prone to subtle liveness failures.

Verified systems software. More broadly, formal verification has been applied to building correct systems across many domains. A rich line of work has produced verified kernels [19, 21, 31, 41, 48, 49], file systems [8–11, 13, 14, 60, 98, 99], distributed systems [18, 24, 39, 58, 59, 62, 87, 88, 95, 96], storage and concurrency systems [4, 7, 12, 23, 43], confidential computing [42], and network systems [1, 54, 56, 92]. Complementary efforts automate parts of the verification process, including invariant inference [20, 22, 47, 53, 89, 90] and proof construction [46, 93]. Despite this breadth, these works verify application-level logic (functional correctness, crash consistency, linearizability, information flow [26, 61]) and assume that the underlying I/O and scheduling layer operates correctly. Lion verifies precisely this layer, guaranteeing that the I/O scheduling mechanism eventually makes progress on every task. As our IronFleet case study demonstrates (§7.5), replacing a synchronous I/O layer with Lion enables verified systems to achieve better resource efficiency while preserving their formal guarantees.

9 Conclusion

We presented Lion, the first Rust async runtime with mechanically verified scheduling liveness. By abstracting execution as append-only event logs and decomposing liveness into per-module async contracts, Lion enables modular end-to-end liveness verification in Verus despite its lack of native temporal-logic support. On real-world applications, Lion achieves at least 95% of Tokio's throughput while retaining Tokio API compatibility and eliminating an entire class of silent scheduling bugs. These results show that verified liveness coexists with modularity and low overhead.

Acknowledgments

We thank the anonymous reviewers for their thorough and insightful feedback. This project was supported in part by NSF awards CNS-2321725, CNS-2238768, and CNS-2130590, and by the State University of New York's Empire Innovation Program.

References

[1] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. Tiramisu: Fast multilayer network verification. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[2] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4), 1985.

[3] Jens Axboe. Efficient IO with io_uring. https://web.archive.org/web/20260624135046/https://kernel.dk/io_uring.pdf, 2019.

[4] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. Using lightweight formal methods to validate a key-value storage node in Amazon S3. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[5] Bytebeam. rumqtt: MQTT client and broker in Rust. https://github.com/bytebeamio/rumqtt. Accessed 2026.

[6] ByteDance. Monoio – a thread-per-core Rust async runtime based on `io_uring`. Accessed 2026.

[7] Tej Chajed, M. Frans Kaashoek, Butler W. Lampson, and Nickolai Zeldovich. Verifying concurrent software using movers in CSPEC. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[8] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Argosy: Verifying layered storage systems with recovery refinement. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2019.

[9] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[10] Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. GoJournal: A verified, concurrent, crash-safe journaling system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.

[11] Tej Chajed, Joseph Tassarotti, Mark Theng, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying the DaisyNFS concurrent and crash-safe file system with sequential reasoning. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[12] Yun-Sheng Chang, Ralf Jung, Upamanyu Sharma, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying vMVCC, a high-performance transaction library using multi-version concurrency control. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2023.

[13] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay Mert Ileri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[14] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using Crash Hoare Logic for certifying the FSCQ file system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[15] Xiangdong Chen, Zhaofeng Li, Lukas Mesicek, Vikram Narayanan, and Anton Burtsev. Atmosphere: Towards practical verified kernels in Rust. In *Proceedings of the Workshop on Kernel Isolation, Safety and Verification (KISV)*, 2023.

[16] Cloudflare. Pingora: A library for building fast, reliable and evolvable network services. https://github.com/cloudflare/pingora. Accessed 2026.

[17] Glauber Costa and Datadog. Glommio – a thread-per-core crate for Rust and Linux. Accessed 2026.

[18] Cezara Drăgoi, Thomas A. Henzinger, and Damien Zufferey. PSync: A partially synchronous language for fault-tolerant distributed algorithms. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2016.

[19] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[20] Stewart Grant, Hendrik Cech, and Ivan Beschastnikh. Inferring and asserting distributed system invariants. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2018.

[21] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[22] Travis Hance, Marijn Heule, Ruben Martins, and Bryan Parno. Finding invariants of distributed systems: It's a small (enough) world after all. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021.

[23] Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. Storage systems are distributed systems (so verify them that way!). In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[24] Travis Hance, Yi Zhou, Andrea Lattuada, Reto Achermann, Alex Conway, Ryan Stutsman, Gerd Zellweger, Chris Hawblitzel, Jon Howell, and Bryan Parno. Sharding the state machine: Automated modular reasoning for complex concurrent systems. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2023.

[25] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving practical distributed systems correct. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[26] Atalay Mert Ileri, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Proving confidentiality in a file system using DiskSec. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[27] Hans Kamp. *Tense Logic and the Theory of Linear Order*. PhD thesis, University of California, Los Angeles, 1968.

[28] Dan Kegel. The C10K problem. http://www.kegel.com/c10k.html, 1999. Accessed 2026.

[29] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[30] Jieung Kim, Vilhelm Sjöberg, Ronghui Gu, and Zhong Shao. Safety and liveness of MCS lock—layer by layer. In *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS)*, 2017.

[31] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[32] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3), 1994.

[33] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[34] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed,

Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Oded Padon, and Bryan Parno. Verus: A practical foundation for systems verification. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2024.

[35] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying Rust programs using linear ghost types. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2023.

[36] Hayley LeBlanc, Jacob R. Lorch, Chris Hawblitzel, Cheng Huang, Yiheng Tao, Nickolai Zeldovich, and Vijay Chidambaram. PoWER never corrupts: Tool-agnostic verification of crash consistency and corruption detection. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2025.

[37] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, 2010.

[38] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7), 2009.

[39] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. Chapar: Certified causally consistent distributed key-value stores. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2016.

[40] Jing Li, Kunal Agrawal, Sameh Elnikety, Yuxiong He, I-Ting Angelina Lee, Chenyang Lu, and Kathryn S. McKinley. Work stealing for interactive services to meet target latency. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2016.

[41] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. A secure and formally verified Linux KVM hypervisor. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2021.

[42] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. Design and verification of the Arm confidential compute architecture. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[43] Xupeng Li, Xuheng Li, Wei Qiang, Ronghui Gu, and Jason Nieh. Spoq: Scaling machine-checkable systems verification in Coq. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2023.

[44] libevent contributors. PR #190: evhttp: Fix failure to send all output data for POST/PUT requests, 2015.

[45] libuv contributors. libuv – multi-platform support library with a focus on asynchronous I/O. Accessed 2026.

[46] Haojun Ma, Hammad Ahmad, Aman Goel, Eli Goldweber, Jean-Baptiste Jeannin, Manos Kapritsos, and Baris Kasikci. Sift: Using refinement-guided automation to verify complex distributed systems. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2022.

[47] Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A. Sakallah. I4: Incremental inference of inductive invariants for verification of distributed protocols. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[48] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling symbolic evaluation for automated verification of systems code with Serval. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[49] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. Hyperkernel: Push-button verification of an OS kernel. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[50] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services uses formal methods. *Communications of the ACM*, 58(4), 2015.

[51] Oxide Computer Company. Issue #8334: Large number of stored certificates correlated with hanging API requests, 2025.

[52] Oded Padon, Jochen Hoenicke, Giuliano Losa, Andreas Podelski, Mooly Sagiv, and Sharon Shoham. Reducing liveness to safety in first-order logic. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2018.

[53] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: Safety verification by interactive generalization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2016.

[54] Solal Pirelli, Akvilė Valentukonytė, Katerina J. Argyraki, and George Candea. Automated verification of network function binaries. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2022.

[55] Niels Provos and Nick Mathewson. libevent – an event notification library. Accessed 2026.

[56] Leonid Ryzhyk, Nikolaj S. Bjørner, Marco Canini, Jean-Baptiste Jeannin, Cole Schlesinger, Douglas B. Terry, and George Varghese. Correct by construction networks using stepwise refinement. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[57] ScyllaDB. Seastar – high-performance server-side application framework. Accessed 2026.

[58] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. Programming and proving with distributed protocols. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2018.

[59] Upamanyu Sharma, Ralf Jung, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Grove: A separation-logic library for verifying distributed systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2023.

[60] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button verification of file systems via crash refinement. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[61] Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. Nickel: A framework for design and verification of information flow control systems. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[62] Christoph Sprenger, Tobias Klenze, Marco Eilers, Felix A. Wolf, Peter Müller, Martin Clochard, and David A. Basin. Igloo: Soundly linking compositional refinement and separation logic for distributed system verification. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA), 2020.

[63] Xudong Sun, Wenjie Ma, Jiawei Tyler Gu, Zicheng Ma, Tej Chajed, Jon Howell, Andrea Lattuada, Oded Padon, Lalith Suresh, Adriana Szekeres, and Tianyin Xu. Anvil: Verifying liveness of cluster management controllers. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024.

[64] Ruize Tang, Minghua Wang, Xudong Sun, Lin Huang, Yu Huang, and Xiaoxing Ma. Converos: Practical model checking for verifying Rust OS kernel concurrency. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2025.

[65] Tokio contributors. axum: Ergonomic and modular web framework built with Tokio, Tower, and Hyper. https://github.com/tokio-rs/axum. Accessed 2026.

[66] Tokio contributors. Tokio – an asynchronous runtime for Rust. Accessed 2026.

[67] Tokio contributors. Tokio API documentation: Blocking and Yielding. Accessed 2026.

[68] Tokio contributors. Issue #703: Weird non-blocking block when using two nested tokio::spawn statements, 2018.

[69] Tokio contributors. Issue #1092: Memory leak, 2019.

[70] Tokio contributors. Issue #3069: Tokio 0.3 timer will hang when duration is small, 2020.

[71] Tokio contributors. PR #2887: Fix readiness future eagerly consuming entire socket readiness, 2020.

[72] Tokio contributors. PR #3080: time: use intrusive lists for timer tracking, 2020.

[73] Tokio contributors. Issue #4941: rt: make the LIFO slot in the multi-threaded scheduler stealable, 2022.

[74] Tokio contributors. Issue #5020: LocalEnterGuard would cause current thread runtime to become stuck, 2022.

[75] Tokio contributors. Issue #5894: Sleep is not firing when macOS goes to sleep and then awakens, 2023.

[76] Tokio contributors. PR #6016: tokio: distinguish `LocalSet::enter()` with being polled, 2023.

[77] Tokio contributors. PR #6134: io: fix possible I/O resource hang, 2023.

[78] Tokio contributors. Issue #6315: rt: Tolerate slow task polls, 2024.

[79] Tokio contributors. PR #6534: Using sharded locks instead of a global lock for timers, 2024.

[80] Tokio contributors. PR #6683: time: update `wake_up` while holding all the locks of sharded time wheels, 2024.

[81] Tokio contributors. Issue #7210: `yield_now` should not defer inside `block_in_place`, 2025.

[82] Tokio contributors. Issue #7478: [bug] `handle.dump()` causes multi_thread issues (partially working or all sleeping), 2025.

[83] Tokio contributors. PR #7216: rt: skip defer queue in `block_in_place` context, 2025.

[84] Tokio contributors. Issue #8198: alternative timer compatibility hazard, 2026.

[85] Tokio contributors. PR #8320: tokio-util: wake `DelayQueue` when cleared, 2026.

[86] Tokio contributors. PR #8331: rt: preserve the full `ScheduledIo` tick in readiness events, 2026.

[87] Klaus v. Gleissenthall, Rami Gökhan Kıcı, Alexander Bakst, Deian Stefan, and Ranjit Jhala. Pretend synchrony: Synchronous verification of asynchronous distributed programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2019.

[88] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2015.

[89] Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. DuoAI: Fast, automated inference of inductive invariants for verifying distributed protocols. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[90] Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. DistAI: Data-driven automated invariant learning for distributed protocols. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.

[91] Junyang Zhang, Xiangcan Xu, Yonghao Zou, Zhe Tang, Xinyi Wan, Kang Hu, Siyuan Wang, Wenbo Xu, Di Wang, Hao Chen, Lin Huang, Shoumeng Yan, Yuval Tamir, Yingwei Luo, Xiaolin Wang, Huashan Yu, Zhenlin Wang, Hongliang Tian, and Diyu Zhou. CortenMM: Efficient memory management with strong correctness guarantees. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2025.

[92] Kaiyuan Zhang, Danyang Zhuo, Aditya Akella, Arvind Krishnamurthy, and Xi Wang. Automated verification of customizable middlebox properties with Gravel. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[93] Tony Nuda Zhang, Travis Hance, Manos Kapritsos, Tej Chajed, and Bryan Parno. Inductive invariants that spark joy: Using invariant taxonomies to streamline distributed protocol proofs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024.

[94] Tony Nuda Zhang, Upamanyu Sharma, and Manos Kapritsos. Performal: Formal verification of latency properties for distributed systems. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2023.

[95] Tony Nuda Zhang, Keshav Singh, Tej Chajed, Manos Kapritsos, and Bryan Parno. Basilisk: Using provenance invariants to automate proofs of undecidable protocols. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2025.

[96] Zihao Zhang, Ti Zhou, Christa Jenkins, Omar Chowdhury, and Shuai Mu. AutoMan: Facilitating verified distributed systems development through automatic code generation and manual optimizations. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2025.

[97] Ziqiao Zhou, Anjali, Weiteng Chen, Sishuai Gong, Chris Hawblitzel, and Weidong Cui. VeriSMo: A verified security module for confidential VMs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024.

[98] Mo Zou, Haoran Ding, Dong Du, Ming Fu, Ronghui Gu, and Haibo Chen. Using concurrent relational logic with helpers for verifying the AtomFS file system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[99] Mo Zou, Dong Du, Mingkai Dong, and Haibo Chen. Using dynamically layered definite releases for verifying the RefFS file system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024.

A Detailed Issue Analysis

Issue	Module	Symptom	Root Cause
2460	Utility (task::LocalSet)	Application hangs with 100% CPU; joining on spawned tasks never completes	LocalSet runs inside block_on, which has already enabled a cooperative budget; enabling it again is a no-op, so every future the LocalSet polls draws on the parent scheduler's single budget rather than an independent per-task one. run_until polls its argument before any spawned task, so a JoinAll over 128 or more handles drains the shared budget on every pass; with the budget at zero, the JoinHandles return Pending without checking whether their task has in fact completed
2886	Reactor/Utility	Concurrent send/receive over a shared socket hangs forever	Two tasks share a socket via Arc: one sends, one receives. The readiness future eagerly returns the full socket readiness (read+write) instead of only the requested interest. When one task's I/O attempt returns WouldBlock and it calls clear_readiness, it clears all readiness flags—including the one the other task is waiting on; that task's readiness is wiped out and it is never woken, even though the socket is ready for it [71]
3069	Reactor/Timer	A zero-duration `sleep` in a loop occasionally never fires	Never localized. A registered timer intermittently fails to fire, with no error returned at registration; the report attracted no diagnosis, and the maintainers folded the case into an in-flight redesign of the time driver rather than tracking down the failing path. The bug disappeared when that rewrite replaced the driver's timer-registration mechanism with intrusive lists; its author states that the issue "was fixed as a side-effect of the rewrite" and added the reported loop as a regression test [72]
3117	Utility (task::LocalSet)	Spawned tasks never execute	LocalSet returns Pending when its task queue is empty, registering a waker to be notified of new tasks. When a task is later spawned into the queue via spawn_local, the waker is not called; LocalSet remains asleep and the newly spawned task is never polled
3168	Utility (sync::watch)	Receiving task panics on a spurious notification and dies, leaving the channel unconsumed	A receiver calls changed() to wait for new values: it registers a notification, checks the version, and awaits. When woken, it assumes a new version exists, but the notification can be stale—delivered for an update it has already observed. Under high-frequency sends the receiver wakes, finds the version unchanged, and, having no retry loop around the register–check–await sequence, panics on an internal assertion instead of re-checking
4814	Utility (sync::broadcast)	Resubscribed broadcast receiver hangs on recv after channel closed	Index initialization incorrect: resubscribe() creates a new receiver with next set to the tail position without accounting for the close message slot; the index adjustment logic fails to recognize the close message at that position, so recv() never detects the closed state and hangs forever
5020	Executor/ Utility	Spawned task never completes; the program hangs forever	A later API overloads an existing flag: the scheduler skips the wake whenever the LocalSet's context is set, assuming this implies the set is currently polling its queue. A later API, enter(), sets the same context merely to route spawn_local, so the assumption no longer holds; a task woken while the set is entered but not polled is pushed onto the local queue with no wake, and the parked LocalSet never learns it is there [76]
6133	Reactor	I/O resource can hang if the driver's tick counter wraps (maintainer-identified race)	A single global driver tick, truncated to u8, is used to stamp per-resource readiness so that clear_readiness only clears what the task actually observed. If a task holds readiness stamped at tick t and the driver then completes 256 cycles before the task resumes, a fresh event stamped $t + 256$ aliases to the same byte; the clear treats the new event as the one already observed and discards it, and the resource is never woken again. Widening the counter only lowers the probability—the fix instead replaced the global tick with a per-resource one [77]
6682	Reactor/Timer	timeout() never returns after timeout expires	Lock-granularity refinement incomplete: an optimization split the timer wheel across sharded locks [79], but the driver updates next_wake only after releasing them. In that window next_wake still holds the previous round's stale, non-empty value, so a thread registering a new Sleep finds its deadline later than next_wake and skips the unpark; the driver then parks with nothing scheduled to wake it [80]

Issue	Module	Symptom	Root Cause
6751	Utility (time::DelayQueue)	Task waiting on DelayQueue hangs when the last item is removed	State-change callback missing: poll_expired() stores a Waker when it returns Pending, but remove() and try_remove() never invoke it. A task polling the queue concurrently with a removal that empties it therefore stays pending forever, waiting for an expiration that can no longer occur
7209	Executor	Tasks hang after 128 budgeted operations inside block_in_place	Context isolation broken: the runtime's defer queue is not drained while a thread sits in block_in_place. When a task there exhausts its cooperative budget—here on the 129th Mutex::lock()—the budget-exhaustion path defers its waker onto the outer multi-threaded runtime's defer queue, the same mechanism yield_now() uses. Nothing drains that queue while the thread is blocked, so the waker is never delivered and the task is permanently lost [81, 83]

libevent. The following bugs were identified in libevent, a widely used C-based event notification library. Unlike Tokio, libevent does not separate its functionality into distinct modules; its event loop, I/O buffering, and protocol handling are tightly coupled, making it difficult to attribute bugs to a single component. The "Module" labels below indicate the nearest subsystem but do not reflect clean architectural boundaries.

Issue	Module	Symptom	Root Cause
232	evhttp	Large POST request stops sending midway; both ends wait until the client times out	The write-completion callback fires once the headers are flushed rather than when the output buffer actually drains, so a body too large to write in one pass is left unsent: evhttp treats the request as complete and waits for a response, while the server waits for the rest of the body [44]
237	Bufferevent filter	HTTP callbacks never fire after installing a bufferevent filter via evhttp_set_bevcb	The filter's control path (be_filter_ctrl) does not implement the set-file-descriptor operation that the plain socket path (be_socket_ctrl) uses to assign the read and write events to the descriptor. With the filter installed, no socket event is ever registered with the event loop, so no I/O activity reaches the filter or the HTTP callbacks above it
984	Event dispatch	Application subscribed to close notification never observes the close; the callback instead fires carrying no event	On remote close, epoll reports EPOLLHUP, which the backend tests before EPOLLRDHUP and therefore maps to read/write events rather than a close event. Dispatch then masks these against the caller's registration: read and write are filtered out (only close was requested), but the internal edge-trigger flag survives and marks the event active. The callback fires carrying only that flag, the close notification is lost, and close-detection logic never makes progress

libuv. The following bugs were identified in libuv, the cross-platform async I/O library underlying Node.js. libuv has clearer layering than libevent—with distinct handle types and a well-defined event loop lifecycle—but still lacks Tokio's explicit executor/reactor/utility module separation. The module labels below reflect the handle or subsystem involved.

Issue	Module	Symptom	Root Cause
3503	Handle lifecycle	Binding and listening on a handle that is being closed succeeds silently; application hangs or crashes	The bind and listen entry points do not check whether the handle has already been marked for closure; operations proceed on a half-torn-down handle, leaving the event loop in an inconsistent state
4738	Pipe read (Windows)	Named-pipe read stalls when the read callback restarts the read, as Node.js does	A regression from the rewrite of the Windows pipe path in 1.49.0. The read-completion path optimistically keeps issuing further reads while the handle is in reading state, without checking whether a read was already re-armed from inside the callback via uv_read_stop() followed by uv_read_start(). The read-pending state recorded by that re-armed read is overwritten once the callback returns, so its completion is never picked up by the event loop and the read callback is never invoked again

B Proof Sensitivity to Broken Logic

Passing the stress test (§7.4) shows Lion does not hang, but not that the *proofs* are what forbid hanging: a vacuous or over-permissive specification could admit a broken scheduler just as well. To measure how tightly the proofs constrain the implementation, we use a large language model to inject liveness-breaking mutations into the runtime's execution logic, each breaking one link of the scheduling chain (timer fire → drain → FIFO order → poll), and check whether the verifier rejects them. Every mutant must still pass a plain `cargo build`, so a rejection reflects a violated proof obligation rather than a type

ID	Link	Mutation	Verifier	Stress
M01	timer fire	`advance_to` never updates the wheel's elapsed clock, freezing the wheel's notion of time.	rejected	passes
M02	timer fire	The cascade drops not-yet-due timers instead of re-inserting them; they never fire.	rejected	passes
M03	timer fire	Timer removal no longer invalidates the cached minimum deadline, which then goes stale.	rejected	passes
M04	timer fire	The park-timeout scan reads only the lowest wheel level, missing nearer upper-level deadlines.	rejected	passes
M05	poll	An unconditional `return` at the head of the poll loop ends the tick without polling any task.	rejected	hangs
M06	FIFO	`next_task` discards the popped task and reports a nonempty run queue as empty.	rejected	hangs
M07	drain	The wakeup drain takes woken tasks out of thread-local storage but never enqueues them.	rejected	hangs
M08	FIFO	A spawned task popped from the injection queue is never enqueued, so it is never polled.	rejected	hangs
M09	FIFO	The poll loop polls a task other than the FIFO head it popped, starving the head task.	rejected	hangs
M10	drain	Popping an injected task skips the wake-ledger mark; later wakeups are filtered as duplicates.	rejected	hangs
C1	trusted	The trusted thread-local shim hands the drain an empty batch, discarding every wakeup.	green	hangs
C2	trusted	A trusted wrapper skips the park-begin ghost-log append; only proof-visible state is touched.	rejected	passes
C3	trusted	The trusted `mio` wrapper makes I/O registration a no-op; readiness events are never delivered.	green	hangs

Figure 16. Liveness mutants: the scheduling-chain link each severs, the mutation, the verifier's verdict, and the stress-suite outcome (three repetitions each). M01–M10 mutate verified code; C1–C3 are trusted-region controls.

error. Each mutant is applied, built, verified, and reverted in isolation; the list was fixed before any of them was run. Figure 16 lists all mutants and their outcomes.

The ten verified-region mutants cover all four links: four in timer fire, two in the wakeup drain, three in FIFO order, and one in the poll loop. The verifier rejects all ten, each by the invariant governing the link it breaks; none verifies green. Taking the poll-loop mutant (M05) as an example: an early `return` at the head of the loop finishes a scheduling tick without polling any task even while runnable tasks sit in the queue. The rejection lands on the poll loop's postcondition, which requires a non-empty run queue to imply that at least one task is polled ($queue.\text{len}() > 0 \Rightarrow \exists\, \text{poll}$); the mutated body can no longer establish it. A fault that would stall production is thus caught statically, before the code runs.

All mutants are additionally run under the stress suite of §7.4. These runs are a validity check on the mutants themselves: a liveness mutant ought to hang a real workload, so a static rejection reflects a genuine fault rather than a specification artifact. The six executor-side mutants (M05–M10) all hang as expected. Interestingly, the entire timer-fire link (M01–M04) escapes. M04 is illustrative: it targets the timer wheel's deadline scan, the pass that computes the next park timeout from the wheel's pending deadlines. The mutant consults only the wheel's lowest level, so a nearer deadline stored at an upper level is overlooked and the reactor parks past a due timer. The stress run nonetheless succeeds, because the executor's 100 ms park cap turns the late fires into bounded extra latency. The other three escapes have distinct causes. M03's stale cache can only shorten a park (a spurious early wake). M01's frozen clock is shadowed by the fire path re-checking each deadline against the current time. M02's dropped cascade, which hangs an uncancelled long sleep in a spawned task, goes unobserved because the workload cancels its only long timers within milliseconds. For the timer link, the verifier's rejection is the only signal.

The mutants above all target verified code, where a broken link faces the verifier. Trusted code faces no such guard, and a subset of it is *liveness-critical* (§5.3): a task can be left un-woken if the OS shim misreports readiness or timer expiry, if the timer-wheel model fires wrongly, or if a waker is lost in the thread-local wake plumbing—the very failure the proof excludes in the model but cannot enforce below the shim. The FIFO-container contract is likewise critical, since all scheduling relies on it, and every theorem relies on the ghost log. The adapters carry a comparable risk: since we prove no refinement from the compiled runtime to the model, an adapter that breaks its delegation could let the real runtime diverge from the verified one.

As controls, we therefore mutate three of these sites, which the proofs are not expected to catch. Two of the three behave as declared: an empty thread-local taker (C1) and a no-op `mio` registration (C3) both verify green and hang the stress suite in all three runs, placing the trust boundary where §5 draws it. The third is more informative. Dropping a ghost-log event (C2) is *caught*, because downstream proofs pin that event's position in the log (its stress run passes trivially: only ghost state is touched, leaving the compiled binary unchanged): part of the call protocol we declare as trusted is enforced by callers after all, so the trusted surface is narrower than the declaration rather than wider.