

SITAS

SYSTEM AND RUNTIME DOCUMENTATION

Frode Randers

2026-06-29

Contents

1	Introduction	6
1.1	Purpose	6
1.2	Prerequisites	6
1.3	How to read this project	6
1.4	Why the project has this shape	7
1.5	Scope of this manual	7
1.6	What this project is not	8
1.7	Further reading	8
2	Architecture	9
2.1	Invariants	9
2.2	The shared-nothing rule	9
2.3	Boundaries and lifetimes	10
2.4	Why owned snapshots?	10
2.5	Layered model	10
2.6	Primary modules	11
2.7	How to trace a feature through the layers	12
3	Executor	13
3.1	Role	13
3.2	Why a custom executor?	13
3.3	Responsibilities	13
3.4	Task lifecycle	13
3.5	Wakers and the ready queue	14
3.6	Scheduling groups	15
3.6.1	Scheduling group API shape	15

3.7	Structured child tasks	16
3.8	Wait ordering	16
3.9	Observability	16
4	Sharding	18
4.1	Shard-per-core direction	18
4.2	Cross-shard submission	18
4.3	Mailbox routing	19
4.4	Cache behavior of shard movement	20
4.4.1	Payload size and batching	21
4.4.2	False sharing	21
4.4.3	Many producers and one receiver	22
4.4.4	NUMA and placement	22
4.4.5	What the model buys	22
4.5	Shard-local values	23
4.6	Sharded scheduling groups	23
4.7	CPU placement	24
5	I/O Runtime	25
5.1	Readiness	25
5.2	TCP helpers	25
5.3	Sharded TCP socket options	26
5.4	<code>io_uring</code>	26
5.5	Completion lifecycle	27
5.6	Current limitation	28
5.7	Running statistics	28
6	Examples	30
6.1	Learning examples	30
6.2	Suggested reading path	30
6.3	Observability examples	31

6.4	Sharded index build	31
6.4.1	Mailbox-transfer index build	32
6.5	Representative commands	33
7	Mechanism Glossary	34
7.1	Why this chapter exists	34
7.2	Runtime terms	34
7.3	Scheduling terms	35
7.4	I/O terms	35
7.5	Observability terms	35
7.6	Ownership terms	36
7.7	Mechanism maps	36
7.7.1	Wake path	36
7.7.2	Timer path	36
7.7.3	Readiness path	37
7.7.4	Completion path	37
7.8	Common confusions	37
8	Guided Walkthroughs	38
8.1	Purpose	38
8.2	Walkthrough: spawning one task	38
8.3	Walkthrough: sleeping and timers	38
8.4	Walkthrough: readiness-driven TCP	39
8.5	Walkthrough: scheduling groups	39
8.6	Walkthrough: cross-shard submission	40
8.7	Walkthrough: shard-local state	40
8.8	Walkthrough: <code>io_uring</code> lifecycle	41
8.9	Walkthrough: reading observability output	41
8.10	Walkthrough: sharded index build	42
8.10.1	Mailbox-transfer index path	43

8.11	Debugging checklist	43
9	Hands-on Labs	45
9.1	Purpose	45
9.2	Lab 1: Timers and cooperative polling <i>Beginner</i>	45
9.3	Lab 2: Wake coalescing <i>Beginner</i>	45
9.4	Lab 3: Readiness-driven TCP <i>Intermediate</i>	46
9.5	Lab 4: Scheduling groups <i>Intermediate</i>	46
9.6	Lab 5: Observability snapshots <i>Beginner</i>	47
9.7	Lab 6: Cross-shard submission <i>Intermediate</i>	47
9.8	Lab 7: Shard-local state <i>Intermediate</i>	48
9.9	Lab 8: CPU placement <i>Intermediate</i>	48
9.10	Lab 9: Linux <code>io_uring</code> <i>Advanced</i>	49
9.11	Lab 10: Sharded index build <i>Advanced</i>	49
9.12	After the labs	50
10	Operations	51
10.1	Building this manual	51
10.2	Cleaning generated files	51
10.3	Linux validation	51
10.4	Continuous integration	52
10.5	Documentation conventions	52
10.6	Validation strategy	52
10.7	Interpreting unsupported features	53

1 Introduction

1.1 Purpose

`sitas` is an experimental Rust runtime and service model inspired by Seastar's shard-per-core architecture. The project is about understanding the runtime internals: how work is owned by shards, how cross-shard messages move, how a small executor drives futures, and how OS completion and readiness mechanisms can be exposed without depending on a third-party async runtime.

The code base started with a standard-library-only sharded key-value store. The current branch extends that baseline with a custom executor, Unix readiness primitives, TCP helpers, sharded async execution, shard-local values, observability snapshots, CPU placement, and experimental Linux `io_uring` support.

1.2 Prerequisites

This manual assumes familiarity with:

- Rust ownership, borrowing, and lifetimes;
- `Future`, `Poll`, `Waker`, and `Pin`;
- basic concurrency: threads, channels, mutexes;
- Unix file descriptors and non-blocking I/O (for the I/O chapters).

1.3 How to read this project

Application code sees typed, owned boundaries. A caller asks a shard to do work. The owning shard mutates its local state and returns an owned reply. The implementation avoids hiding service state behind a global mutex.

The runtime keeps that model honest:

1. futures are owned by executor tasks;
2. tasks are assigned to one executor shard;
3. wakers only requeue tasks on that executor;
4. cross-shard work must be submitted explicitly;
5. shard-local values can be borrowed only inside synchronous closures;
6. observability returns copies, not borrowed internals.

This manual follows that path. chapter 2 explains the ownership rules. chapter 3 explains how futures become runnable work. chapter 4 explains how one executor becomes many independent

shards. chapter 5 explains why readiness and completion are different mechanisms and how both appear in the runtime.

Learning goal If you can explain where a future is polled, who owns the state it mutates, what wakes it, and what happens when it is dropped, you understand the most important parts of this code base.

1.4 Why the project has this shape

The runtime is built from small visible pieces. A mature runtime hides task allocation, reactor registration, cross-thread wakeups, cancellation, and kernel ownership behind polished APIs. That works in production but obscures the machinery.

`sitas` therefore accepts some explicitness that a general-purpose runtime might smooth over:

Explicit shards make ownership visible. If work runs on shard 2, shard 2 owns the mutation.

Explicit submitters make cross-shard movement visible. A future does not silently migrate just because another shard is less busy.

Explicit snapshots make observation visible. Monitoring is a read of owned runtime facts, not a borrowed view into live internals.

Explicit OS boundaries make platform behavior visible. A readiness wait, an `io_uring` completion, and a CPU affinity request are different contracts with the operating system.

This is the main design tradeoff. The project prefers APIs that teach the boundary they cross over APIs that hide it.

1.5 Scope of this manual

This manual is a LaTeX companion to the Markdown architecture notes. It is intended for longer-form system documentation and for keeping a printable view of the evolving runtime design.

It focuses on:

- architectural invariants;
- executor and scheduling behavior;
- shard ownership and cross-shard submission;
- readiness, `epoll`, and `io_uring` integration;
- example programs that exercise the runtime;
- build and validation operations.

Note The project is still deliberately experimental. APIs are allowed to move when that exposes cleaner runtime internals or a better teaching experience.

1.6 What this project is not

`sitas` is not trying to become a production replacement for mature Rust runtimes. It is also not trying to copy Seastar. The point is to keep the mechanisms small enough that they can be inspected:

- no Tokio, `async-std`, `Glommio`, `Monoio`, or actor framework is used as the core runtime;
- blocking standard-library service APIs remain distinct from awaitable executor APIs;
- unsafe and FFI code is kept near the operating-system boundary;
- mechanisms are introduced only after their semantics can be tested.

This makes the project useful as a learning resource. It has enough machinery to demonstrate timers, wakers, readiness, `io_uring`, shard-local values, scheduling groups, and CPU placement while keeping those mechanisms visible in ordinary Rust modules.

1.7 Further reading

Seastar The C++ shard-per-core framework that inspired the ownership model.

<https://seastar.io>

ScyllaDB shard-per-core architecture Practical discussion of shared-nothing database design on top of Seastar.

<https://www.scylladb.com/product/technology/shard-per-core-architecture/>

Glommio A Rust thread-per-core runtime using `io_uring`, useful for comparing design choices.

<https://github.com/DataDog/glommio>

2 Architecture

2.1 Invariants

The project is guided by a small set of invariants:

1. Each piece of application service state is owned by one shard.
2. Only the owning shard may mutate its service state.
3. Application services should not be modeled as shared `Arc<Mutex<Service>>` state.
4. Cross-shard interaction happens through typed messages, typed reply handles, or typed submitter calls.
5. Values crossing shard boundaries are owned.
6. References to shard-local state must not escape synchronous access closures and must not cross `.await`.
7. Async work remains on its assigned shard unless explicitly submitted elsewhere.
8. Observability snapshots are owned values, not borrowed runtime internals.
9. Unsafe code is isolated behind small safe APIs.

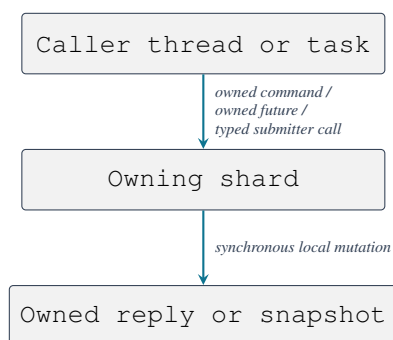
2.2 The shared-nothing rule

The central design rule: application state has one owner. In the standard-library services that owner is a shard thread receiving typed commands. In the async runtime it is an executor shard polling futures.

This is why the project avoids `Arc<Mutex<Service>>`. A mutex makes mutation mechanically safe but hides the question the project studies: which shard owns this state, where does work run, and when does the ownership boundary get crossed?

Shared-nothing does not mean synchronization-free. While wakers, reply state, lifecycle flags, counters, and submission queues use synchronization, application service state does not. It stays local to its shard.

Why avoid a service mutex? A service mutex answers only one question: can two threads mutate this value at the same time? The runtime is asking a stronger question: which shard is responsible for this value, which work is allowed to touch it, and how does another shard request a change? Keeping service state shard-owned preserves that answer in the program structure.



2.3 Boundaries and lifetimes

Most runtime bugs are boundary bugs: a value crosses to another thread without being owned, a reference to local state escapes a closure, a future keeps a borrowed value across `.await`, or a completion operation drops a buffer before the kernel has finished with it.

`sitas` tries to make those boundaries visible:

Cross-shard transfer uses owned values and explicit `Send + 'static` requirements where a future can move to another thread.

Shard-local access gives a closure synchronous access to `&mut T`; the closure returns an owned result and cannot suspend.

Observability returns owned snapshots. A monitoring thread can keep a snapshot after the runtime has moved on.

Kernel I/O stores owned buffers until completions are observed or until abandoned operation cleanup proves they can be discarded.

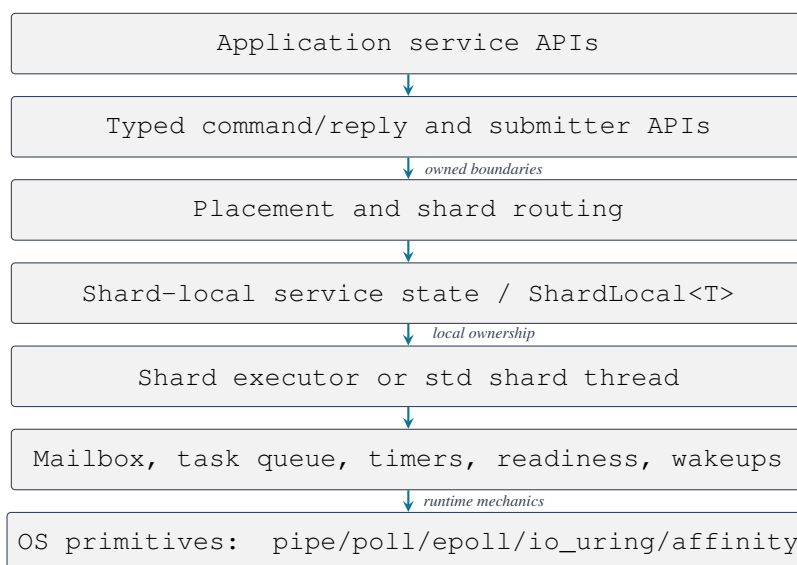
2.4 Why owned snapshots?

Observability could expose references into executor internals, but that would make a diagnostic API participate in the same lifetime and locking problems as the runtime. The project treats observation as a copy boundary. A snapshot is a point-in-time value that can be sorted, printed, diffed, or sent to another thread without keeping the executor borrowed.

This matters because long-running work needs inspection while it is still changing. Owned snapshots let monitoring code be ordinary application code. They also prevent a debugging tool from accidentally extending the lifetime of task, timer, or shard-local internals.

2.5 Layered model

The current runtime can be read as a layered system:



The standard-library baseline exercises the upper part of the model. The custom executor and OS runtime layers fill in the lower layers while preserving the same shared-nothing service direction.

2.6 Primary modules

runtime The reusable standard-library kernel for bounded shard mailboxes, reply handles, shard startup, and shutdown.

kv The reference sharded key-value service.

counter A smaller second service proving that the runtime primitives are not key-value-specific.

sharded A generic `ShardService` trait and `Sharded<S>` runtime providing common life-cycle (start, shutdown, snapshot) for user-defined shard-local services. Shutdown is automatic via `stop_command`.

placement Key-to-shard routing.

stream_reply Streaming reply channels. A `StreamProducer<T>` delivers a sequence of owned values. Two async patterns: `StreamBatch<'a, T>` borrows the reply for `next_batch()` loops, `StreamFuture<T>` owns the reply with `collect()` and `fold()` convenience methods. Senders use an atomic refcount so intermediate clone drops do not prematurely close the stream.

async_service Async wrappers bridging std-layer services (`ShardedKv`, `ShardedCounter`) into the executor via `submit_* + reply.wait_async().await`.

os Unix FFI, wake pipes, readiness backends, affinity, and Linux `io_uring` experiments. Readiness backends now cover `epoll` (Linux), `kqueue` (macOS/iOS/FreeBSD/NetBSD/OpenBSD), and `poll` (other Unix).

executor The single-threaded async kernel. Sub-modules include `tcp`, `udp`, `backpressure`, and Linux `io_uring` dispatch. `Spawner::with_backpressure` integrates a counting semaphore into the spawn path.

sharded_executor One executor per shard thread, plus submission and observation APIs.

shard_local One owned value per executor shard with checked local access. Provides `with_current`, `with_current_result`, `get_current`, `map_all`, and `worker/scope` APIs.

shard_mailbox Typed owned-message transfer between shards. Provides uniform key-to-shard routing through `ShardMailboxSet<M>` and non-uniform logical work-unit routing through `WorkUnitMailboxSet<N, M>`.

sharded_tcp Network-facing sharded TCP server with `SO_REUSEPORT` on Linux, optional Linux `SO_INCOMING_CPU` listener placement, an explicit kTLS ULP hook for accepted connections, a private socket-options helper for `setsockopt` details, and single-accept distribution on other platforms.

metrics Thread-safe atomic-counter metrics collector with owned `MetricsSnapshot` for observability.

2.7 How to trace a feature through the layers

When learning or extending the runtime, trace one feature down the stack. For example, a named task in a scheduling group passes through several layers:

1. user code calls a typed spawn API;
2. the spawner validates that the scheduling group belongs to this executor;
3. a task is allocated with a name and group id;
4. the scheduler places the task into the group's ready queue;
5. the executor polls the selected task;
6. the task records status, wait interest, poll count, and timestamps;
7. a snapshot later resolves the group id back to an owned group name.

This kind of trace is more useful than reading modules in isolation. It shows which layer owns validation, scheduling, and observability, and where the public API remains safe.

3 Executor

3.1 Role

The executor is a minimal single-threaded async kernel. It deliberately avoids Tokio or similar third-party runtimes to keep the internals visible: wakers, task queues, timers, readiness registration, cancellation, and completion dispatch all live in this code base.

3.2 Why a custom executor?

The custom executor exists because the project needs to control the meaning of “where work runs.” In a shard-per-core runtime, a future is not just an async computation; it is also part of a placement decision. The executor must be able to say that this task belongs to this shard, this scheduling group, this ready queue, and this local reactor state.

A third-party runtime would solve many engineering problems but would hide the scheduler internals. The custom executor makes it possible to inspect how a waker requeues a task, how a dropped timeout removes timer state, how a TCP future records fd interest, and how a completion future keeps buffers alive.

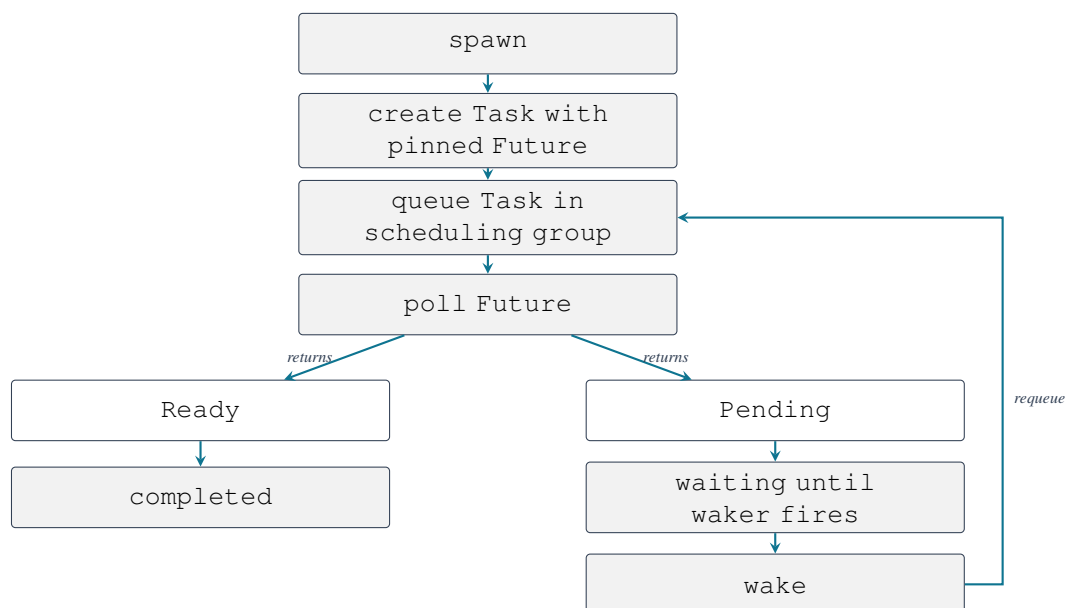
3.3 Responsibilities

- Poll ready tasks with a budget so self-waking tasks cannot starve timers and I/O forever.
- Implement custom task wakers that enqueue tasks onto the local ready queue.
- Catch task panics at executor boundaries.
- Preserve panic payloads where join handles and `block_on` make them observable.
- Provide `sleep`, `timeout`, `race`, and `yield_now`.
- Provide typed join handles and abort support.
- Provide cooperative stop tokens, `Notify`, and `TaskScope`.
- Drive readiness futures for read and write interests.
- On Linux, drive executor-owned `io_uring` read/write-at futures.

3.4 Task lifecycle

A spawned future is stored inside a task. The task has a stable executor-local `TaskId`, an optional name, a scheduling group id, and internal state describing whether it is queued, polling, waiting, completed, or cancelled.

The lifecycle is explicit:



A repeated wake does not create repeated queue entries. The task records that it is already queued, so repeated wakeups coalesce. A chatty future does not multiply itself in the ready queue.

Cancellation is normal future dropping. Aborting a join handle or dropping an executor drops the owned future. Timer and readiness futures must clean up their registrations in `Drop`; otherwise the reactor keeps stale interests for nonexistent tasks.

Why cancellation is treated as ordinary Rust futures are cancelled by being dropped. Cleanup must be owned by the future, task, join handle, or dispatcher being dropped. This is why tests exercise timeout, abort, and dropped-future paths.

3.5 Wakers and the ready queue

The custom waker does one job: make the task runnable again on the same executor. It does not move the task to another shard, poll it directly, or borrow application state. It marks the task queued and hands it to the scheduler.

The executor loop then decides when to poll ready work, in batches bounded by `READY_POLL_BUDGET`. That budget is a fairness valve. A self-waking task can keep itself ready but cannot prevent the executor from checking timers, readiness, and completions.

The ready queue is executor-local. Cross-thread wakeups may notify an executor that work became ready, but the task is still polled by its owning executor. This keeps local task state, scheduling group accounting, and reactor registrations aligned.

3.6 Scheduling groups

Scheduling groups are the first resource-class mechanism. Each group owns a ready queue and has a relative share count. Ordinary spawn calls use the default group. Explicit group APIs classify work such as foreground request handling and background maintenance.

Resource classification is a local scheduling problem. A shard can divide its polling time among foreground and background tasks without moving application state to another shard. That makes scheduling groups a natural fit for the shared-nothing model.

The selection rule is simple. The scheduler chooses a non-empty group with the lowest weighted virtual runtime, pops one task from that group's FIFO queue, polls it, and charges the group for the observed wall-clock poll duration. The charge is scaled by the group's shares.

```

1 for each poll:
2     group = non_empty_group_with_lowest_virtual_runtime()
3     task = group.pop_front()
4     duration = poll(task)
5     group.virtual_runtime += duration * default_shares / group.shares

```

Listing 3.1: Weighted group selection

This is not preemption, though. A long poll still blocks the executor thread until the future returns `Poll::Pending` or `Poll::Ready`. The mechanism is cooperative: futures must yield, await I/O, await timers, or otherwise return control to the executor.

Scheduling group handles are executor-local. A group created by one executor is rejected by another. The default group handle is special because it names the built-in default queue in any executor. The same idea appears in the sharded runtime as a `ShardedSchedulingGroup`, which is one executor-local group per shard plus a runtime identity check.

3.6.1 Scheduling group API shape

`Spawner::create_scheduling_group` creates an executor-local group with a name and a positive share count. The normal spawn methods remain the shorthand for the default group. The group-aware variants classify work before it enters the ready queues:

```

1 spawn_in_group
2 spawn_named_in_group
3 spawn_with_handle_in_group
4 spawn_with_handle_named_in_group

```

Listing 3.2: Local group-aware spawn methods

```

1 let foreground = spawner.create_scheduling_group("foreground", 200)?;
2 let background = spawner.create_scheduling_group("background", 25)?;
3
4 spawner.spawn_named_in_group(&foreground, "request", request_future)?;
5 spawner.spawn_named_in_group(&background, "maintenance", maintenance_future)?;

```

Listing 3.3: Classifying local tasks

`TaskScope` has matching group-aware spawn methods, so lifetime ownership and scheduling classification can be combined. In the sharded runtime, the matching creation helper builds one local group per shard:

```
1 create_scheduling_group_on_all
```

Listing 3.4: Sharded group creation

The returned `ShardedSchedulingGroup` can be used with sharded spawn and submitter methods that accept an explicit group.

Shares are cooperative Shares affect which ready group is selected after tasks return control to the executor. They do not interrupt a long poll. CPU-bound futures still need to call `yield_now`, await timers or I/O, or split work into bounded chunks.

3.7 Structured child tasks

`TaskScope` groups child tasks under one owner. Dropping the scope requests cooperative stop and aborts still-owned children. Calling `shutdown` asks children to stop and waits for them. Calling `shutdown_timeout` bounds that wait and aborts uncooperative children.

Scopes can spawn children into scheduling groups. This matters when a server accepts work that should be both lifetime-scoped and resource-classified. The scope does not implement scheduling; it delegates group validation and task creation to the underlying `Spawner`.

3.8 Wait ordering

Each loop iteration polls ready work, wakes expired timers, dispatches locally available `io_uring` completions, and then decides how to wait.

On Linux, pending `io_uring` submissions are first submitted to the kernel without blocking. The executor then includes the ring descriptor in the same reactor wait as read interests, write interests, reactor wakes, and the next timer deadline. When the ring descriptor becomes readable, the dispatcher harvests completions and wakes the registered task wakers.

This gives the executor one idle wait shape instead of a separate `io_uring`-first wait path. `io_uring` has not become readiness I/O; completion ownership and buffer lifetime are still handled by the dispatcher. The reactor only tells the executor that completion queue progress is available.

3.9 Observability

Executor snapshots are owned values. They include task counts, named task state, polling counters, timer counts, readiness counts, and Linux `io_uring` dispatcher counters when the dispatcher is

installed. Scheduling group snapshots include ready queue length, total charged poll count, total charged poll time, weighted virtual runtime, average poll time, and helper methods for computing each group's share of executor poll time.

Task snapshots include the task name, scheduling group id and name, lifecycle state, wait interest, poll count, accumulated poll time, and key timestamps. Helper methods derive task age and current state duration from a caller-supplied `Instant`. This keeps the snapshot itself factual while making progress views easy to write.

```
1 task_name:           useful for humans
2 status:             queued, polling, waiting, completed, cancelled
3 waiting_for:        timer, readable fd, writable fd, unknown, none
4 poll_count:         how many times the future was polled
5 total_poll_time:    how much executor time the task consumed
6 state_duration_at(now): how long it has been in the current state
```

Listing 3.5: Reading a task snapshot

4 Sharding

4.1 Shard-per-core direction

`ShardedExecutor` connects the single-threaded executor to the shard-per-core model. It starts one executor per shard thread and provides APIs for submitting work to one shard, all shards, or a map/reduce over shards.

Work does not migrate implicitly. A future runs on its assigned shard until code deliberately submits work elsewhere.

This mirrors the most important Seastar idea: parallelism comes from many independent shards, not from many threads mutating the same service object. Each shard has its own executor, task queues, timers, readiness registrations, and shard-local values.

Cache locality and ownership clarity drive this shape. Each shard owns its state and works on its own queue, so no global lock is needed. Cross-shard communication is an explicit design cost, not an accidental side effect of shared references.

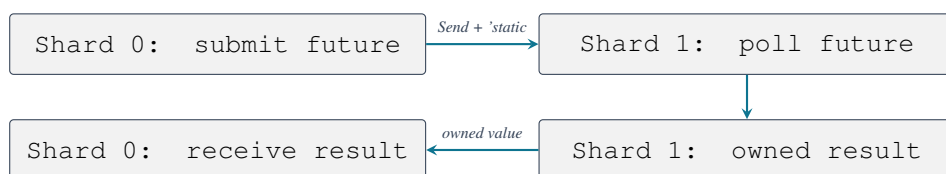


4.2 Cross-shard submission

`ShardedSubmitter` is the explicit cross-shard async mechanism. A task on one shard can submit a future to another shard and await the returned join handle. The remote future is polled on the target shard. The awaiting task resumes on its original shard after the remote work completes.

Submitters own spawner clones. They are lifetime capabilities and must be dropped before a runtime can fully drain.

The important detail is the return path. A task on shard 0 submits work to shard 1 and awaits the join handle. The submitted future is polled on shard 1. On completion, the awaiting future on shard 0 wakes and continues on shard 0. The result crosses the boundary as an owned value.



This is why cross-shard APIs require `Send + 'static'` where a future or value can move to another thread. It is also why references into shard-local state cannot be submitted.

The submitter API is more explicit than work stealing. Work stealing would blur the core lesson: placement is part of the program's semantics. Submitting to shard 1 means shard 1 is the correct owner for that work.

4.3 Mailbox routing

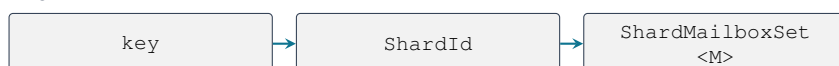
`ShardedSubmitter` moves futures. `shard_mailbox` moves owned messages. A mailbox does not make application state shared; it gives producers a typed way to transfer ownership to a receiver that remains local to its assigned shard.

There are two routing modes.

Uniform shard routing A key maps directly to a physical `ShardId`. The target is one inbound mailbox per shard: `ShardMailboxSet<M>`.

Non-uniform work-unit routing A key maps to a logical work-unit name. A placement table maps that name to the shard that owns its receiver: `WorkUnitMailboxSet<N, M>`. Multiple work units can be assigned to the same shard, and some shards can have no work unit for a particular protocol.

Uniform



Non-uniform



`UniformShardRouter` makes the first path explicit. It routes a hashable key to a `ShardId`. `WorkUnitRouter<N>` makes the second path explicit. It routes a hashable key to a logical name. In both cases the key is still the discriminator; what changes is the target type.

The receiving side remains shard-local. A named work unit has exactly one receiver, and `receiver_for_current_shard` checks that the receiver is taken by a task on the work unit's assigned shard. For non-uniform placement, the logical name alone is not the owner; the name plus its placement determines which executor may mutate the associated state.

Mailbox routing is not load balancing The mailbox layer does not steal work and does not rebalance a hot work unit, but it makes routing explicit. Changing the cluster shape means changing the router targets or the work-unit placement table, then observing the new behavior through owned snapshots and example output.

4.4 Cache behavior of shard movement

The shared-nothing model is often described as a way to improve cache locality. That is true, but it is important to be precise about what improves and what still costs real hardware traffic.

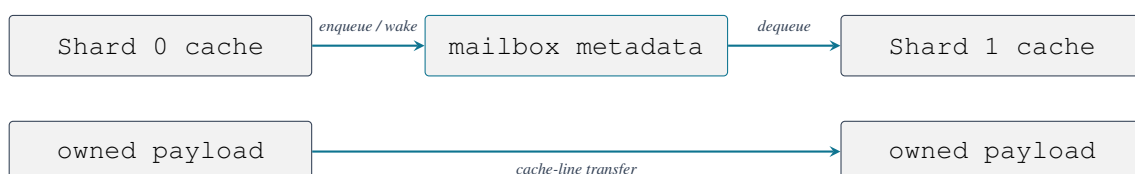
A shard owns its mutable application state. If a key-value shard repeatedly updates its local map, the working set for that map tends to stay in the cache hierarchy of the core running that shard. Other shards do not acquire a mutex around the map, read its buckets, or update its counters. This avoids a common source of coherence traffic: several cores repeatedly invalidating the same cache lines while trying to mutate one logical service object.

Cross-shard communication is different. Sending a message transfers ownership at the Rust and runtime boundary, but the bytes still move through the machine. Suppose shard 0 builds an owned message and sends it to shard 1:

1. Shard 0 writes the message payload and queue slot.
2. Shard 0 updates producer-side queue metadata.
3. Shard 1 observes that the queue is no longer empty.
4. Shard 1 reads the message payload and mutates its own local state.

The first and fourth steps touch the payload. The cache lines holding the payload are likely warm on shard 0 after construction. When shard 1 reads the message, those cache lines must become available to shard 1. On a coherent processor this can mean transferring a modified line from another core's cache, invalidating an old copy, or refetching data through lower cache levels or main memory. The type system proves that shard 0 no longer has a borrowed reference into the message after transfer, but it does not make the physical cache-line movement disappear.

The second and third steps touch mailbox metadata. A bounded mailbox typically has queue slots plus small pieces of coordination state: producer and consumer positions, capacity accounting, closed flags, waiter lists or wakers, rejection counters, and snapshot counters. Those fields are runtime synchronization, not shared application state, but they can still be written by different cores. A single hot metadata cache line can bounce between cores even when every message payload is an owned value.



This is the distinction the architecture relies on:

Application state remains local and is mutated only by the owning shard. This is where the model avoids uncontrolled shared-cache-line contention.

Transport state is allowed to synchronize. Mailboxes, reply handles, wakers, lifecycle flags, and counters may use atomics or locks internally. Their cache behavior matters, but it is confined to the runtime boundary.

Transferred values are owned and type-safe, but they still occupy cache lines and memory bandwidth when consumed on another shard.

4.4.1 Payload size and batching

Small commands are usually cheap to move, but they can create high metadata traffic if sent at very high rates. Large messages reduce per-message overhead but can evict useful local data from the receiver's caches. The right shape is workload-dependent.

For hot paths, prefer commands that describe the operation compactly and let the destination shard perform the mutation locally. When many tiny records share the same route, batching them into a `Vec<T>` can amortize queue metadata updates and wakeups. Batching should still preserve ownership: the batch is an owned value, and after receipt only the destination shard mutates the state associated with it.

Large buffers need a different tradeoff. Sending an owned buffer is semantically clean, but the receiving shard will pull the buffer's cache lines into its own cache hierarchy. If the producing shard also needs the same data again, the program may be better served by routing the work to the shard that already owns or will next use the data, rather than bouncing a large working set across cores.

4.4.2 False sharing

False sharing occurs when independent fields used by different cores reside on the same cache line. For example, a producer-updated counter and a consumer-updated counter may be logically separate but physically adjacent. If they share a cache line, updates by one core can invalidate the other core's copy of the line.

This can affect:

- mailbox producer and consumer indices;
- per-shard progress counters;
- wake flags and closed flags;
- high-frequency observability counters;
- arrays of per-shard statistics where adjacent shards update adjacent entries.

Padding or cache-line alignment is sometimes justified for fields that are known to be hot and written by different cores. It should be applied selectively: padding every structure makes memory use worse and can reduce cache density for data that is mostly read or only touched by one shard.

4.4.3 Many producers and one receiver

An inbound mailbox per shard is clear and predictable, but it can still become a coherence hotspot. If many shards send to one target at high frequency, the target's inbound queue metadata becomes shared by many producers. The application state behind the receiver is still protected by the ownership model, but the transport path may limit throughput.

The usual remedies are architectural, not magical queue tricks:

- route data so work is more evenly partitioned;
- combine small messages before sending;
- use logical work units to split one protocol across multiple receivers when the state model allows it;
- avoid request patterns where every shard repeatedly asks one shard to perform tiny mutations;
- return owned summaries rather than repeatedly querying remote state.

4.4.4 NUMA and placement

On multi-socket systems, cache and memory movement may cross a NUMA boundary. The cost of sending a message between two cores on the same socket can differ from sending between sockets. CPU placement does not by itself solve NUMA placement, but it makes the question visible: which shard runs where, and what data does it mostly touch?

The project's current CPU placement support is experimental and Linux-focused. It can request where shard threads run, but it does not yet provide a complete memory-placement policy by default. On Linux, applied CPU placement also records the NUMA node for the pinned CPU when `sysfs` exposes that topology. Callers that want a memory policy can opt in explicitly. The runtime can apply a shard-thread default policy with `set_mempolicy`: bind to a specific node, prefer a node, interleave across nodes, or bind to the node local to the pinned CPU. Required memory placement turns failure into startup failure; otherwise the result is recorded in snapshots. The conservative assumption is that cross-shard messages have nonzero cost even when placement succeeds, and that cross-socket movement may be materially more expensive.

This is not complete allocator control. The policy affects future allocations made by the shard thread according to kernel rules. The runtime does not yet expose `mbind` for existing address ranges, page migration, or allocator-specific arenas.

4.4.5 What the model buys

The model does not promise zero-copy communication between arbitrary shards and does not make a mailbox free. Its promise is narrower and stronger:

- application state has one mutating owner;
- cross-shard movement is explicit in the API;

- values crossing the boundary are owned;
- runtime synchronization is localized to queues, replies, wakeups, and lifecycle bookkeeping;
- cache-coherence costs are easier to find because they occur at named transfer points rather than hidden shared-state access.

That is why the project treats mailboxes as a transport boundary. They are a necessary shared runtime mechanism, but they do not weaken the rule that application service state belongs to the receiving shard.

4.5 Shard-local values

`ShardLocal<T>` owns one value per executor shard. Access is routed to the owning shard and the closure receives `&mut T` synchronously. The implementation uses internal unchecked storage plus runtime owner checks, not a mutex.

References to local state cannot escape the closure and cannot cross `.await`. Ownership stays local even when the runtime uses synchronization for task bookkeeping.

Two ways to read `ShardLocal<T>`:

- as a safe teaching model for shared-nothing state, because each shard owns exactly one `T`;
- as a runtime boundary test, because the code must prevent references to `T` from becoming async state.

The API prefers closures returning owned results. If a computation needs to wait, copy or compute an owned value from the shard-local state first, then `await` using that owned value outside the access closure.

Why not async closures for local access? An async closure can suspend while holding references captured from its environment. That is exactly what shard-local state must avoid. The synchronous closure form is less flexible, but it makes the safe pattern obvious: inspect or mutate local state now, return an owned value, and only then await.

4.6 Sharded scheduling groups

A sharded scheduling group is not a global queue. It is a convenience handle containing one executor-local scheduling group per shard. Creating a group called `foreground` on a four-shard runtime creates four independent local `foreground` groups, all with the same shares.

Each shard still schedules only its own tasks. The sharded group handle lets callers use one typed API to spawn or submit matching classes of work across shards.

Per-shard snapshots keep the same boundary: each shard's local group poll count, charged poll time, average poll duration, and poll-time share. A monitoring tool can compare those owned

values across shards, but the runtime does not introduce a global scheduling queue to produce them.

Runtime identity matters. A sharded group created by one `ShardedExecutor` is rejected by another runtime, even if both runtimes have a group with the same name. This prevents accidental targeting of same-numbered executor-local group ids in the wrong runtime.

4.7 CPU placement

CPU placement is an experimental Linux-supported runtime request. Linux applies hard affinity with `sched_setaffinity` and observes container cpuset restrictions through `sched_getaffinity`. Non-Linux platforms report unsupported.

By default, placement failures are recorded in shard snapshots and startup succeeds. Required placement turns that into fail-fast behavior.

On Linux, the runtime asks the kernel to pin shard threads to particular CPUs. In containers, the available CPU set may already be restricted, so the runtime first queries what the kernel allows. On macOS and other platforms, placement reports unsupported rather than pretending that pinning happened.

The feature is useful even in experimental form because it forces the runtime to separate desired placement from observed placement. A shard can report which CPU it asked for, which CPU set the host allowed, and whether startup treated failure as advisory or fatal. On Linux, a successfully pinned shard can also report the NUMA node associated with its CPU when the host exposes `/sys/devices/system/cpu/cpuN/node*`. That makes CPU and memory-locality questions visible to callers choosing an explicit memory-placement policy.

Memory placement follows the same explicit-request model. The default is no memory policy change. When requested, Linux applies a thread default memory policy after CPU placement and before the shard executor runs. Snapshot fields separate CPU placement from memory placement so callers can see whether each request was applied, unsupported, or rejected.

5 I/O Runtime

5.1 Readiness

The readiness path is small. A future registers interest in a file descriptor, the scheduler records the waker, the Unix reactor sleeps using Linux `epoll`, macOS/iOS `kqueue`, or fallback `poll` plus a wake pipe, and readiness wakes the interested task.

This path is readiness-based, not completion-based. It is used by the current TCP helpers and remains separate from the experimental `io_uring` completion path.

Readiness is the first I/O layer because it matches Unix non-blocking sockets and is portable. Ownership stays simple: user code owns the stream or listener; the runtime only tracks which task to wake when the descriptor may progress.

Readiness means the OS reports an operation is likely to make progress, not that bytes are available. A readable stream may return fewer bytes than requested. A writable stream may accept only part of a buffer. The future must attempt the operation, handle `WouldBlock`, and re-register interest.

```
1 poll future:
2     try non-blocking operation
3     if operation succeeds:
4         return Ready(result)
5     if operation would block:
6         register fd interest and waker
7         return Pending
8
9 reactor wait:
10     OS reports fd readiness
11     scheduler wakes interested task
```

Listing 5.1: Readiness future shape

Readiness futures do not own file descriptors. The caller owns the stream or listener; the future borrows it for the async operation and registers interest while pending.

A connection handler owns a stream and calls helpers against it in sequence. The reactor is the waiting mechanism, not the connection owner.

Readiness lesson Readiness is a notification that retrying may be useful. Completion is a notification that the operation has already finished. The code keeps those concepts separate because the ownership and cancellation rules are different.

5.2 TCP helpers

The TCP helpers are built on top of readiness futures rather than hiding a separate runtime. They demonstrate the normal async server shape:

1. put the listener or stream in non-blocking mode;
2. accept or connect using readiness futures;
3. spawn one handler task per accepted connection;
4. use `TaskScope` or join handles to wait for handlers;
5. propagate handler errors as ordinary `io::Result` values.

Grouped TCP helper variants classify accepted handler tasks into scheduling groups. The accept loop remains in the caller's task; only per-connection handler work enters the requested group. This separates foreground request handling from background maintenance or low-priority connections.

Accepting connections and serving connections have different lifecycle and scheduling needs. The accept loop is one long-lived task. Handlers are many shorter-lived tasks. Keeping those roles separate makes shutdown, observability, and scheduling group assignment straightforward.

5.3 Sharded TCP socket options

`ShardedTcpServer` keeps Linux socket placement explicit. On Linux, `SO_REUSEPORT` lets each shard bind its own listener so the kernel can distribute incoming connections across shard-local accept loops. The server can also opt into `SO_INCOMING_CPU` with `ShardedTcpIncomingCpu`. This option is disabled by default; callers must request either sequential placement over `available_cpu_ids()` or an explicit CPU list.

Accepted sharded TCP connections expose a Linux `kTLS` ULP helper. That helper attaches the kernel `tls` upper-layer protocol to the socket. It is not a complete TLS implementation: the handler or a TLS stack must still complete the handshake and install the required kernel TX/RX crypto state. The server does not own certificates, handshake transcripts, traffic secrets, or TLS record policy, so it deliberately leaves the full `kTLS` handoff outside the generic accept loop.

5.4 `io_uring`

The Linux `io_uring` backend is experimental. It currently supports a narrow executor integration for owned-buffer `read_at`, `read_exact_at`, and `write_at` style file I/O.

`io_uring` is included because it exercises a harder runtime problem than readiness. The kernel can own an operation after the Rust future that submitted it has been dropped. That forces the runtime to model operation ids, owned buffers, late completions, and abandonment—exactly the internals this project wants to make visible.

Host policy is part of the model. Many Linux deployments, especially containers and hardened hosts, restrict or disable `io_uring` because it exposes a broad kernel/user boundary. Newer kernels are adding more fine-grained restriction mechanisms, including cBPF-style filtering of allowed `io_uring` operations, so a host may allow a small opcode set rather than enabling every operation. This runtime does not install or manage such policy. It keeps the operations narrow and

reports setup or submission failure as unavailable or as an ordinary I/O error, depending on where the host rejects the request.

Each Linux executor run loop installs a thread-local dispatcher when the host allows ring setup. Executor-backed `io_uring` futures store operation ids and owned buffers. When polled on a shard, they queue an operation against the thread-local dispatcher and register their task waker.

When the executor becomes idle, it submits pending ring entries and polls the ring descriptor through the same reactor wait used for fd readiness and timer deadlines. Ring descriptor readiness is only a notification that completion queue entries may be harvested. The completed operation still flows through the dispatcher, which matches operation ids, returns owned buffers, and wakes waiting futures.

The dispatcher tracks:

- queued submissions;
- locally buffered completions;
- registered wakers;
- abandoned operations;
- deferred owned buffers;
- cumulative dispatched, buffered, woken, and discarded operation counters.

5.5 Completion lifecycle

The `io_uring` path is completion-based. A future submits an operation and later receives a completion. That changes the ownership problem. With readiness, the future borrows a stream and retries ordinary system calls. With `io_uring`, the kernel may hold a pointer to an owned buffer until the completion arrives.

The runtime tracks operation state explicitly:

```
1 future first poll
2     allocate operation id
3     move owned buffer into dispatcher tracking
4     submit operation to ring
5     register task waker
6     return Pending
7
8 completion arrives
9     record result and completed buffer
10    wake registered task
11
12 future next poll
13     take completion
14     return Ready(result, buffer)
```

Listing 5.2: Simplified completion lifecycle

If the future is dropped before completion, the operation may still be in the kernel. The dispatcher marks it abandoned and defers buffer cleanup until a completion or safe discard point. This is

one of the most important differences between readiness and completion: cancellation involves kernel-owned in-flight work, not just waker removal.

The dispatcher is a lifecycle object, not just a syscall wrapper. A thin wrapper around `io_uring_enter` would not suffice for async Rust integration. The runtime needs somewhere to hold buffers, match completions to futures, wake tasks, and account for futures that vanished before the kernel answered.

5.6 Current limitation

Timers, readiness waits, and `io_uring` completions now share one executor idle wait shape on Linux. This is still a staged integration rather than a production event loop.

Completion dispatch has a fixed internal per-tick budget that is exposed in executor snapshots. Those snapshots also count non-empty completion dispatch batches, dispatched completions, and completion budget exhaustion events.

They also expose an executor-owned `io_uring` lifecycle status, so callers can distinguish not-started, unavailable, installed, and shutdown states.

Strict availability requests via `SITAS_REQUIRE_IO_URING=1` fail executor setup instead of silently falling back to unavailable.

Executor teardown makes a bounded attempt to drain abandoned completion work before discarding the thread-local dispatcher when no live task wakers remain registered. The final dispatcher snapshot records whether that drain completed, timed out, or was skipped because live wakers remained.

When a root `run_until` future completes with spawned `io_uring` work still pending, the dispatcher remains installed for that executor on the current thread so later executor calls can continue the same operations.

The runtime still does not expose a portable completion source, tunable completion policy, or a deeper shutdown policy for mixed readiness and completion workloads.

5.7 Running statistics

[Welford's algorithm](#) computes mean, variance, and standard deviation in a single pass without storing individual samples. The `running_stats` module provides `RunningStatistics` as a small online accumulator useful for runtime diagnostics and benchmark comparisons.

```
1 let mut stats = RunningStatistics::new();
2 stats.add_sample_duration(elapsed);           // record as seconds (f64)
3 stats.add_sample_interval(start, end);         // convenience for Instant pairs
4 stats.merge(&other);                           // Chan et al. parallel merge
5
6 // Accessors return Option<f64> (None until enough samples):
```

```
7 stats.count()           // u64
8 stats.min() / max()     // Option<f64>
9 stats.mean()           // Option<f64>
10 stats.variance()       // Option<f64>, sample variance (n-1), n >= 2
11 stats.std_dev()        // Option<f64>
12 stats.cv()             // Option<f64>, coefficient of variation in %
13 stats.total()          // u64 integral total
```

Listing 5.3: RunningStatistics use

Duration convenience methods (`mean_duration()`, `min_duration()`, `max_duration()`, `std_dev_duration()`) return `Option<Duration>` instead of raw seconds.

The `merge` method uses Chan et al.'s parallel algorithm so partial accumulators from different threads or shards can be combined without reprocessing the original samples. This is useful when each shard collects statistics independently and the caller wants aggregate numbers.

The display format prints count, mean, standard deviation, CV, min, max, and total in a single human-readable line, which is how the index build examples report per-shard timing statistics.

CV is unit-free. It expresses dispersion relative to the mean, making it useful for comparing the stability of metrics with different scales (e.g. partition scan times versus merge round times).

6 Examples

6.1 Learning examples

The `examples/` directory is part of the teaching material. The examples show the runtime pieces in isolation before combining them:

- basic executor futures and timers;
- readiness-driven TCP helpers;
- sharded executor startup and submission;
- shard-local values;
- CPU placement reporting;
- observability snapshots;
- standard-file, mailbox-transfer, and `io_uring` index-building demos.

The examples are not just smoke tests. Each one is a small experiment with a visible runtime question: what wakes this future, which shard polls it, what state does the snapshot show, or what happens when cancelled? Several examples print more internal state than a normal library example would.

6.2 Suggested reading path

The examples are easiest to learn in layers:

1. Start with `executor_sleep.rs`, `executor_timeout.rs`, and `executor_race.rs`. These show futures, timers, and combinators before sharding enters the picture.
2. Move to `async_readable.rs`, `async_write.rs`, and `async_tcp_pair.rs`. These show readiness and non-blocking file descriptor behavior.
3. Read `async_tcp_server.rs`, `async_tcp_scoped_server.rs`, and `async_tcp_scheduling_groups.rs`. These show handler spawning, scoped shutdown, and scheduling groups.
4. Then read `sharded_executor.rs`, `sharded_submit.rs`, and `sharded_map_reduce.rs`. These show explicit placement of async work on shards.
5. Finally read `shard_local.rs` and `shard_local_worker_observability.rs`. These show local state access and owned runtime snapshots.

Treat examples as executable documentation. Run them, then inspect the modules they use. Most examples print runtime counters or snapshots so the mechanism is visible rather than implied.

If an example feels verbose, that is usually intentional. The output connects a user-visible event to an internal mechanism. A compact demo that hides executor state would be less useful.

6.3 Observability examples

`sharded_observability.rs` samples task state while named shard tasks sleep and wake. The interesting columns are:

status the coarse task lifecycle state;
age_ms how long the task has existed;
state_ms how long it has been in the current state;
polls how many times the future has been polled;
wait the last known wait interest.

`async_tcp_scheduling_groups.rs` shows a different form of observability. It starts two TCP accept loops, parks accepted handlers, takes a snapshot, and prints which scheduling group each live task belongs to. The snapshot is owned data, so any tool can render it.

`scheduling_group_demo.rs` and `sharded_scheduling_groups.rs` expose group-level progress as well: charged poll count, charged poll time, average poll duration, and poll-time share. These fields make long-running cooperative work easier to inspect without adding a tracing subsystem or global scheduler.

6.4 Sharded index build

The sharded index example uses two files: a fixed-size record data file and a sorted index of offsets into that data file. Each shard scans one partition, sorts its partition-local index entries, writes a materialized run file, and then participates in merge rounds until one sorted index remains.

The example demonstrates several runtime APIs in combination:

- `ShardLocal<ShardProgress>` tracks per-shard phase, record count, and byte counters. Each shard task calls `with_current_result` to update only its own slot. No shared mutex protects the progress counters.
- `runtime.map_all` dispatches partition work to all shards with a single call, replacing manual handle collection.
- Chunked record reads (256 records per read) replace one-syscall- per-record scanning.
- `RunningStatistics` collects per-shard scan durations and prints mean, standard deviation, CV, min, and max across shards.
- Explicit merge rounds submit pairs of sorted run files onto shards, stream-merge them into new run files, and carry odd runs forward.
- Wall-clock timing is reported at the end: `partition=34.1ms merge=82.2ms total=175.7ms`.

The `io_uring` variant performs partition reads, run writes, and merge run reads/writes through the executor-backed Linux completion path. It uses the same `ShardLocal` and `map_all` shape

as the `std` variant. Final verification still uses ordinary file I/O because it checks the externally visible result rather than the runtime path that produced it.

The index examples are more involved than the small teaching examples. They exercise the Seastar-like direction: split work by shard, keep partition-local state local while sorting, materialize intermediate runs, merge owned outputs, and collect per-shard timing statistics. They are useful for studying where the project is heading, but not the first examples to read.

The index build is a good larger example because it is naturally partitionable. Each shard sorts a portion of the data without sharing a mutable index structure. The merge phase makes coordination cost explicit by exchanging owned run files and summaries rather than borrowing another shard's working set.

6.4.1 Mailbox-transfer index build

`sharded_index_mailbox.rs` shows a different assembly path than `sharded_index_build.rs`. Scan tasks still run on executor shards, but they send owned `IndexMessage` batches through mailbox senders instead of using intermediate run files as the cross-shard exchange medium.

The demo separates logical ownership from physical shard placement:

- `WorkUnitRouter<IndexWorkUnit>` maps each record key to a logical assembler work unit.
- `WorkUnitMailboxSet<IndexWorkUnit, IndexMessage>` maps each assembler name to the shard that owns its receiver.
- The default run starts one more assembler than shards, so one shard owns two logical assembler work units. This makes non-uniform placement visible in the printed placement table.
- Each assembler receives owned batches, sorts its own entries, writes one final partition file, and returns owned metadata.

The communication paths are mixed. Scanner tasks send `Vec<IndexEntry>` batches to assembler work units through mailboxes. Assemblers do not send mailbox messages to the coordinator. Instead, each assembler returns small owned `PartitionRun` metadata through its task handle, and the coordinator reads the assembler run files during the final k-way merge.

The final index is still a file because it is the externally visible result. The point of the example is that intermediate cross-shard exchange is typed message transfer, not shared mutable state and not borrowed data.

6.4.1.1 Performance aside: mailbox transfer vs file-backed exchange

The two index variants produce identical results but differ in how intermediate data crosses shard boundaries. The build example writes per-shard sorted run files and performs multi-round pairwise

merge I/O. The mailbox example routes entries by key through owned messages to assembler work units, which sort and write pre-partitioned files before a single k-way merge.

Measured on macOS (Apple Silicon, 8 CPUs), release mode, deterministic input:

Records	Shards	Build total	Mailbox total	Build merge	Mailbox scan+xfer
1 M	1	6.1 s	8.9 s	≈0	20 ms
1 M	2	8.6 s	8.9 s	2.9 s	13 ms
1 M	4	10.2 s	8.2 s	5.3 s	9 ms
1 M	8	13.3 s	9.3 s	6.9 s	12 ms
5 M	2	47.2 s	46.3 s	15.0 s	63 ms
5 M	4	57.4 s	45.4 s	27.4 s	39 ms
5 M	8	69.9 s	51.0 s	34.9 s	43 ms

The build example anti-scales with shard count: more shards produce more run files, requiring more merge rounds of full-dataset file I/O. At 5 M records, going from 2 to 8 shards increases the build total by 48%. The mailbox example stays roughly flat because in-memory transfer replaces intermediate file exchange and the final k-way merge reads each partition file once.

Both totals are dominated by unbuffered I/O overhead. `write_index_entry` performs two 8-byte `write_all` calls per entry. At 5 M records that is 10 M write syscalls for the index alone. Data file creation and verification suffer the same pattern. The mailbox’s reported work phases (scan+transfer in tens of milliseconds, assembler join in microseconds) reflect concurrent shard execution where scanning, mailbox transfer, assembler sort, and partition writes overlap on executor threads.

The build example wins at 1 shard because it has no merge phase and copies one file. At 4+ shards the mailbox approach pulls ahead because it avoids $O(N \log S)$ merge I/O. Neither example uses buffered writers, so both pay per-entry syscall costs that dwarf the architectural difference at small scale.

6.5 Representative commands

```

1 cargo fmt --check
2 cargo clippy --all-targets -- -D warnings
3 cargo test
4 cargo doc --no-deps
5 cargo run --example sharded_index_mailbox -- --records 512 --shards 4

```

Listing 6.1: Local validation

```

1 SITAS_DOCKER_IO_URING=1 tools/linux-docker.sh cargo test
2 SITAS_DOCKER_IO_URING=1 tools/linux-docker.sh cargo run --example
  sharded_index_build_uring -- --records 512 --shards 4 --seed 0xbeef --
  cleanup

```

Listing 6.2: Linux validation through Docker

7 Mechanism Glossary

7.1 Why this chapter exists

The same words appear repeatedly in the runtime: task, shard, waker, readiness, completion, submitter, snapshot, and scheduling group. This chapter is a short field guide for those words.

7.2 Runtime terms

Shard A single ownership domain. In the standard-library services it is a worker thread with a mailbox. In `ShardedExecutor` it is a thread running one single-threaded executor.

Executor A single-threaded async kernel. It owns tasks, timers, readiness registrations, optional `io_uring` dispatcher state, and scheduler counters.

Task The executor-owned wrapper around a pinned future. A task has an id, optional name, scheduling group id, lifecycle state, poll counters, timestamps, and a waker path back to its executor.

Root future The future passed to `run_until`. The executor drives it while also polling spawned tasks. When the root future completes, `run_until` returns its output.

Spawner A handle that submits new tasks to one executor. Cloned spawners keep the executor accepting work until they are dropped.

Submitter A sharded capability for explicit cross-shard async submission. It owns the ability to place work on shard executors.

Shard mailbox A bounded typed queue for transferring owned messages to one shard-local receiver. `ShardMailboxSet<M>` is the uniform form: one inbound mailbox per shard.

Work unit A logical receiver role, such as an index assembler. A work unit has a name and an assigned shard. Multiple work units may be assigned to the same shard.

Work-unit mailbox A named mailbox for non-uniform work assignment. The work-unit mailbox set routes by logical name and checks that the receiver is taken on the assigned shard.

Router The key discriminator for message placement. `UniformShardRouter` maps a key to `ShardId`; `WorkUnitRouter<N>` maps a key to a logical work-unit name.

Join handle An awaitable handle for a spawned task result. It is also the cancellation handle: aborting it requests that the task be dropped if it has not completed. Dropping a join handle without awaiting it automatically aborts the underlying task.

Task scope A structured owner for child tasks. It gives a group of children one cooperative stop source and one bounded shutdown path. Dropping a scope requests stop and aborts remaining children.

7.3 Scheduling terms

Ready queue A queue of tasks that should be polled. In the current executor, each scheduling group owns its own ready queue.

Waker The object a future uses to ask the executor to poll it again. In `sitas`, waking a task requeues it on the same executor.

Poll One call into a future. A future either returns `Ready(output)` or `Pending`. Long polls block the whole executor thread, so cooperative code must yield or await.

Scheduling group An executor-local resource class with a name and relative shares. Tasks in a group are charged for observed poll time.

Virtual runtime A weighted counter used to compare scheduling groups. More shares make the same poll duration charge less virtual runtime.

Ready-poll budget The maximum number of ready tasks polled before the executor checks timers, readiness, and completions again.

7.4 I/O terms

Readiness The operating system reports that a descriptor may be ready for reading or writing. The runtime must still retry the operation and handle partial progress or `WouldBlock`.

Completion The operating system reports that a submitted operation has finished. Linux `io_uring` is completion-based.

Interest A registered wish to be woken when a descriptor becomes readable or writable. Interests are task/waker bookkeeping, not file descriptor ownership.

Dispatcher The executor-local `io_uring` state machine that owns submission tracking, completion buffering, waker registration, and abandoned operation cleanup.

Abandoned operation A completion operation whose future was dropped before the kernel completion was observed. The dispatcher must keep enough state to dispose of buffers safely later.

7.5 Observability terms

Running statistics An online accumulator using Welford's algorithm that computes count, mean, variance, standard deviation, coefficient of variation, min, max, and total from streaming `f64` samples without storing the values. Two accumulators can be merged. Duration convenience methods accept `Duration` and `Instant` pairs.

Snapshot An owned point-in-time copy of runtime state. A snapshot is not live and does not keep the runtime alive.

Coefficient of variation (CV) Standard deviation divided by mean, expressed as a percentage. Unit-free, useful for comparing dispersion across metrics with different scales.

Runtime identity A private identity used to reject handles from a different runtime, such as a sharded scheduling group created by another `ShardedExecutor`.

7.6 Ownership terms

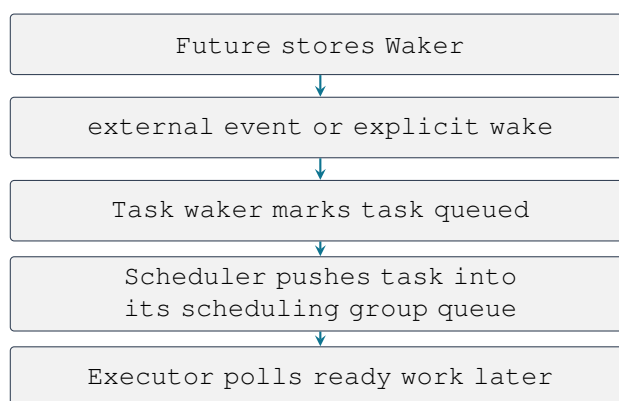
Owned value A value that can safely cross a boundary because no borrowed reference into local runtime or service state escapes with it.

Shard-local value One owned value per shard, accessed only on the owning shard and only through synchronous closures. `with_current_result` skips the `ShardId` parameter. `get_current` returns a clone.

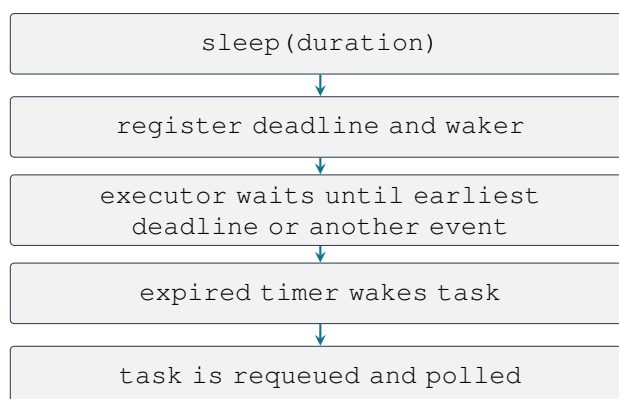
Stoppable shard-local workers A set of shard-local workers sharing a cooperative stop token. Dropping the handle requests stop and aborts remaining workers—the explicit `stop_and_join` path is preferred for clean shutdown.

7.7 Mechanism maps

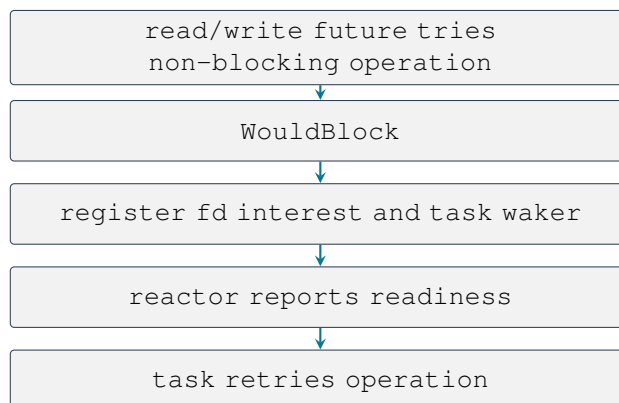
7.7.1 Wake path



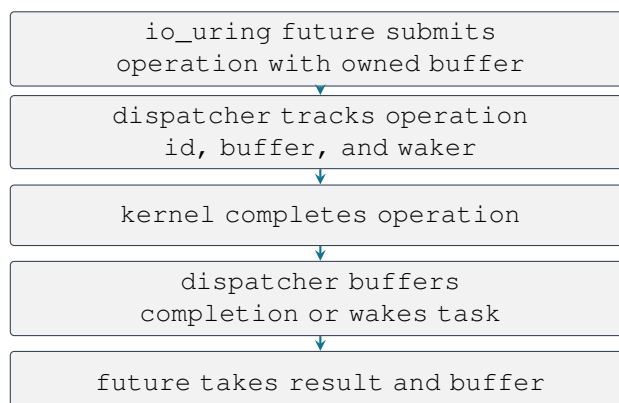
7.7.2 Timer path



7.7.3 Readiness path



7.7.4 Completion path



7.8 Common confusions

Readiness is not completion Readiness says retrying may work. Completion says submitted work has finished.

Shared-nothing is not synchronization-free The runtime uses locks and atomics internally. The application-state model remains shared-nothing.

A scheduling group is not a thread It is a queue and accounting class inside one executor. On the sharded runtime, a sharded group is one matching local group per shard.

A snapshot is not a subscription It is an owned copy. To observe progress over time, take repeated snapshots and compare them.

CPU placement is not load balancing Placement asks the operating system where shard threads should run. It does not redistribute work between shards.

8 Guided Walkthroughs

8.1 Purpose

This chapter gives concrete source-reading paths. Each walkthrough starts from a visible API or example and follows the mechanism through the runtime.

How to use these walkthroughs Keep the source tree open while reading. Read the public function first, then follow the listed modules in order.

8.2 Walkthrough: spawning one task

Start with the smallest async unit: a task spawned onto one executor.

```
1 examples/executor_sleep.rs
2 src/executor.rs
3 src/executor/spawner.rs
4 src/executor/task.rs
5 src/executor/task_state.rs
6 src/executor/scheduler.rs
7 src/executor/task_set.rs
```

Listing 8.1: Task spawn reading path

The public API is `executor_and_spawner`. It returns one executor and one spawner. The spawner creates tasks; the executor polls them.

The useful questions to ask while reading are:

1. Where is the future boxed and pinned?
2. Where does the task get its id and optional name?
3. What prevents repeated wakes from putting the same task into the ready queue many times?
4. What state is recorded before and after each poll?
5. What happens if the executor is dropped before the task completes?

The mechanism is local. A task spawned by a `Spawner` is owned by that spawner's executor. Its waker requeues it there. Nothing in this path migrates the task to another thread.

8.3 Walkthrough: sleeping and timers

The timer path starts with examples such as `executor_sleep.rs` and `executor_timeout.rs`. Then read:

```
1 src/executor/future.rs
2 src/executor/timer.rs
3 src/executor/scheduler.rs
4 src/executor/driver.rs
```

Listing 8.2: Timer reading path

`sleep` creates a future that registers a deadline. When polled before the deadline, the future stores the task waker and returns `Poll::Pending`. The scheduler keeps timer state separate from the ready queues. On each loop iteration the executor checks expired timers and wakes the registered tasks.

`timeout` demonstrates cancellation by drop. If the deadline wins, the inner future is dropped. Any timer or readiness registration owned by that future must clean itself up.

8.4 Walkthrough: readiness-driven TCP

Start from the server helpers rather than the raw reactor:

```
1 examples/async_tcp_server.rs
2 examples/async_tcp_scoped_server.rs
3 src/executor/tcp.rs
4 src/executor/unix_io.rs
5 src/executor/io_interest.rs
6 src/os/reactor.rs
```

Listing 8.3: TCP readiness reading path

The TCP helper shows the application shape: accept, spawn a handler, wait for handlers, propagate errors. The lower layers show the runtime shape: try a non-blocking operation, register interest on `WouldBlock`, sleep in the OS reactor, wake the task on readiness.

When reading `src/executor/tcp.rs`, notice that the accept loop and handler tasks are distinct. Grouped TCP helpers use this: the accept loop stays in the caller's task while accepted handler tasks are spawned into a scheduling group.

8.5 Walkthrough: scheduling groups

The fastest way to see scheduling groups:

```
1 cargo run --example scheduling_group_demo -- 1
2 cargo run --example async_tcp_scheduling_groups
3 cargo run --example sharded_scheduling_groups
```

Listing 8.4: Scheduling group examples

Then read:

```
1 src/executor/scheduling_group.rs
2 src/executor/spawner.rs
3 src/executor/task_set.rs
4 src/executor/snapshot.rs
5 src/sharded_executor.rs
```

Listing 8.5: Scheduling group reading path

`scheduling_group.rs` defines the public handle and its executor ownership. `spawner.rs` validates that a group belongs to the executor before creating a task in that group. `task_set.rs` owns the queues and weighted virtual runtime accounting. `snapshot.rs` resolves group names into task snapshots for observability.

On the sharded runtime, the extra step is runtime identity. A `ShardedSchedulingGroup` is a bundle of executor-local groups, one per shard. It must not be accepted by a different runtime even if names and ids look similar.

8.6 Walkthrough: cross-shard submission

Start with:

```
1 examples/sharded_submit.rs
2 examples/sharded_map_reduce.rs
3 src/sharded_executor.rs
4 src/sharded_executor/join.rs
5 src/executor/spawner.rs
```

Listing 8.6: Cross-shard reading path

A cross-shard submit is not a function call into borrowed remote state. It is owned work sent to another executor shard. The submitted future is polled on the target shard. Its owned output completes a join handle. The original task wakes on its original shard.

While reading the submitter APIs, look for the `Send + 'static` boundaries that mark where work can cross threads.

8.7 Walkthrough: shard-local state

Read `examples/shard_local.rs` first, then:

```
1 src/shard_local.rs
2 src/sharded_executor.rs
3 src/runtime.rs
```

Listing 8.7: Shard-local reading path

`ShardLocal<T>` is the clearest example of the shared-nothing idea in the async runtime. Each shard owns one `T`. Access is routed to the owning shard. The closure receives `&mut T` and returns an owned result.

The key rule is negative: a reference to `T` must not become part of a future that later awaits. The API enforces this through synchronous closures rather than async callbacks.

8.8 Walkthrough: `io_uring` lifecycle

This path is Linux-specific. When available, start with:

```
1 SITAS_DOCKER_IO_URING=1 tools/linux-docker.sh cargo run --example os_uring
2 SITAS_DOCKER_IO_URING=1 tools/linux-docker.sh cargo run --example
  os_uring_abandon
```

Listing 8.8: `io_uring` examples

Then read:

```
1 src/os/uring.rs
2 src/executor/uring.rs
3 src/executor/driver.rs
4 src/executor/tests.rs
```

Listing 8.9: `io_uring` reading path

The key distinction is ownership. A readiness future borrows a descriptor and retries ordinary system calls. An `io_uring` future submits owned buffers to the kernel and must keep them alive until completion or safe abandonment.

`os_uring_abandon.rs` is useful because success paths can hide lifetime complexity. Abandonment forces the dispatcher to keep enough state to clean up operations whose futures no longer exist.

8.9 Walkthrough: reading observability output

Run:

```
1 cargo run --example sharded_observability
2 cargo run --example shard_local_worker_observability
```

Listing 8.10: Observability examples

The columns are a compact runtime story:

ready how many tasks are queued to be polled;
tasks how many tasks are still unfinished;
timers how many timer registrations exist;
polls how many task polls the executor has performed;
done how many tasks completed or were cancelled;
age_ms how long the task has existed;
state_ms how long it has been in the current state;

wait the last known wait interest.

The output is not tracing. There is no event stream. It is repeated sampling of owned snapshots—less powerful than a full console, but simple, portable, and keeps runtime internals inspectable.

8.10 Walkthrough: sharded index build

The index build is the most application-like example. Start with a small run:

```
1 cargo run --example sharded_index_build -- --records 512 --shards 4 --seed 0
  xbeef --cleanup
```

Listing 8.11: Index build example

Then read:

```
1 examples/sharded_index_build.rs
2 src/shard_local.rs
3 src/sharded_executor.rs
4 src/running_stats.rs
```

Listing 8.12: Index build reading path

The example demonstrates several runtime mechanisms working together:

1. `ShardLocal<ShardProgress>` holds per-shard progress counters. Each shard task calls `with_current_result` to update only its own slot without a shared mutex.
2. `runtime.map_all` dispatches partition scanning to all shards in one call, replacing the older pattern of collecting join handles one at a time.
3. Chunked reads (256 records per `read_exact`) replace one-syscall-per-record scanning.
4. After all partitions complete, `RunningStatistics` collects per-shard scan durations and prints mean, standard deviation, CV, min, and max.
5. The merge phase submits pairs of sorted run files back onto shards. Each merge task reads two sorted run files, merges them, and writes a new run file. The round continues until one sorted index remains.
6. Verification reads every index entry and confirms global sort order and correct record offsets, but does so outside the sharded runtime.

The runtime question is *where* work runs. Partition tasks run on assigned shards. Merge tasks are submitted to specific shards (the left input run's shard). Progress reads happen through `ShardLocal::map_all`, which submits synchronous closures to every shard and collects owned results. At no point does a shard borrow another shard's progress counter.

The `io_uring` variant (`examples/sharded_index_build_uring.rs`) replaces partition reads, run writes, and merge run I/O with `read_exact_at_uring` and `write_all_at_uring` futures. It keeps the same `ShardLocal`, `map_all`, and `RunningStatistics` shape.

8.10.1 Mailbox-transfer index path

Then run the mailbox variant:

```
1 cargo run --example sharded_index_mailbox -- --records 512 --shards 4 --seed 0
  xbeef
```

Listing 8.13: Mailbox index example

Read:

```
1 examples/sharded_index_mailbox.rs
2 src/shard_mailbox.rs
3 src/placement.rs
4 src/sharded_executor.rs
```

Listing 8.14: Mailbox index reading path

This variant keeps scan work sharded, but replaces intermediate file-mediated exchange with typed mailbox messages. The key question changes from “which shard owns this key?” to “which target type does this router produce?”

- `UniformShardRouter` produces a physical `ShardId`. That is the uniform route into `ShardMailboxSet<M>`.
- `WorkUnitRouter<IndexWorkUnit>` produces a logical assembler name. That is the non-uniform route into `WorkUnitMailboxSet<N, M>`.
- `WorkUnitSpec<N>` assigns each logical assembler to an executor shard. The default example assigns two assemblers to shard 0, making non-uniform placement visible.

Keep the roles separate while reading the example. Scanner tasks message assembler work units through mailboxes. Assemblers report completion metadata to `main` through task handles, not through another mailbox, and their sorted data reaches coordination as run files consumed by the final k-way merge.

While reading `src/shard_mailbox.rs`, distinguish the transport from the ownership model. The mailbox queue uses synchronization internally, but the message is an owned value and the receiver task is still the only code that mutates the work unit’s local assembly state.

8.11 Debugging checklist

When something behaves unexpectedly, ask these questions in order:

1. Which shard owns the state being mutated?
2. Which executor polls the future?
3. Is the task queued, polling, waiting, completed, or cancelled?
4. If waiting, is it waiting for a timer, fd readiness, completion, or an unknown waker?
5. Did a handle from another executor or runtime cross a boundary?
6. If using `io_uring`, who owns the buffer until completion?

7. If shutdown hangs, which spawner, submitter, join handle, or scoped child is still alive?

These questions match the runtime's invariants. They are usually more productive than starting with performance guesses.

9 Hands-on Labs

9.1 Purpose

This chapter turns the manual into a workbook. Each lab has a command to run, something to observe, and a small modification to try. The goal is to connect visible behavior to the mechanisms described earlier.

The labs are organized around “why” questions. Why does a timer need a waker? Why does readiness require retrying? Why does cross-shard submission return an owned result? Why does `io_uring` need abandonment tracking? Running the examples first makes those design choices concrete.

Lab style Run one lab at a time. After each command, inspect the example source and the runtime module it exercises. The examples are small enough that modifying them is part of learning them.

9.2 Lab 1: Timers and cooperative polling

Beginner

Run:

```
1 cargo run --example executor_sleep
2 cargo run --example executor_timeout
3 cargo run --example executor_race
```

Listing 9.1: Timer examples

Observe that the executor can drive futures that suspend and resume without using a third-party runtime. Then read `src/executor/future.rs` and `src/executor/timer.rs`.

Try this:

1. Change the sleep duration in `executor_sleep.rs`.
2. Add a second spawned task that sleeps for a different duration.
3. Print which task completes first.

The lesson: timers do not run in their own thread. The executor records deadlines, waits until the next event, and wakes tasks whose deadlines expire.

9.3 Lab 2: Wake coalescing

Beginner

Run the unit tests that mention repeated wakes:

```
1 cargo test repeated_wakes_share_one_ready_queue_entry -- --nocapture
```

Listing 9.2: Wake tests

Then read the task queueing code in `src/executor/task.rs` and `src/executor/task_set.rs`.

Try this:

1. Add a tiny future that calls its waker multiple times before returning `Pending`.
2. Spawn it and inspect the executor snapshot's ready queue length.
3. Confirm that repeated wake calls do not create repeated ready queue entries for the same task.

The lesson: a waker requests another poll; it does not flood the scheduler with duplicate queue entries.

9.4 Lab 3: Readiness-driven TCP

Intermediate

Run:

```
1 cargo run --example async_tcp_pair
2 cargo run --example async_tcp_server
3 cargo run --example async_tcp_scoped_server
```

Listing 9.3: TCP readiness examples

Observe the server shape: an accept loop receives a stream, a handler future owns that stream, and shutdown waits for handlers.

Try this:

1. Change a handler to write two chunks instead of one.
2. Add a short sleep between chunks.
3. Watch how the client still observes ordered bytes even though the server task suspends between writes.

The lesson: readiness is retry-oriented. A writable notification does not mean every byte has been written; the helper retries until the operation completes.

9.5 Lab 4: Scheduling groups

Intermediate

Run:

```
1 cargo run --example scheduling_group_demo -- 1
2 cargo run --example async_tcp_scheduling_groups
```

```
3 cargo run --example sharded_scheduling_groups
```

Listing 9.4: Scheduling group demos

Observe the difference between task placement and scheduling accounting. The TCP scheduling example shows which group live handler tasks belong to. The local scheduling-group demo shows virtual runtime and charged poll time changing over time.

Try this:

1. Change the shares in `scheduling_group_demo.rs`.
2. Increase the runtime from one second to three seconds.
3. Compare the `executed`, `charged`, and `vruntime` columns.

The lesson: scheduling groups are cooperative accounting classes, not preemptive priorities. Long-running futures must yield.

9.6 Lab 5: Observability snapshots

Beginner

Run:

```
1 cargo run --example sharded_observability
2 cargo run --example shard_local_worker_observability
```

Listing 9.5: Observability examples

Observe the task rows. Focus on `status`, `age_ms`, `state_ms`, `polls`, and `wait`.

Try this:

1. Increase the sleep duration inside the worker tasks.
2. Add one more worker per shard.
3. Observe how task count, timer count, and state duration change.

The lesson: snapshots are owned samples, not tracing events. To build a console-like view, sample repeatedly and compute differences.

9.7 Lab 6: Cross-shard submission

Intermediate

Run:

```
1 cargo run --example sharded_submit
2 cargo run --example sharded_map_reduce
3 cargo run --example sharded_broadcast
```

Listing 9.6: Cross-shard examples

Observe that each example explicitly names where work should run. Nothing migrates implicitly.

Try this:

1. Submit work from shard 0 to shard 1.
2. Have the remote future return `current_executor_shard()`.
3. Assert that the returned shard id is the target shard, not the caller's shard.

The lesson: the remote future is polled remotely, and the result crosses back as an owned value.

9.8 Lab 7: Shard-local state

Intermediate

Run:

```
1 cargo run --example shard_local
2 cargo run --example shard_local_current
3 cargo run --example shard_local_stoppable_workers
```

Listing 9.7: Shard-local examples

Observe that local values are accessed on owning shards and that workers can be stopped cooperatively.

Try this:

1. Add a second field to the shard-local value.
2. Update it inside `with_current`.
3. Return an owned summary instead of a reference.

The lesson: shard-local access is synchronous. To await, first produce an owned value and await outside the local-state closure.

9.9 Lab 8: CPU placement

Intermediate

Run:

```
1 cargo run --example sharded_cpu_placement
```

Listing 9.8: CPU placement example

On Linux, also try the Docker helper:

```
1 tools/linux-docker.sh cargo run --example sharded_cpu_placement
```

Listing 9.9: Linux CPU placement through Docker

Observe whether placement is applied, unsupported, or rejected. The exact result depends on platform and container CPU permissions.

Try this:

1. Request more shards than the container allows CPUs.
2. Compare best-effort placement with required placement.
3. Inspect the shard snapshots printed by the example.

The lesson: CPU placement is an explicit request to the OS. It is not a load balancer.

9.10 Lab 9: Linux `io_uring`

Advanced

This lab requires a Linux host or container where `io_uring` setup is allowed.

```
1 SITAS_DOCKER_IO_URING=1 tools/linux-docker.sh cargo run --example os_uring
2 SITAS_DOCKER_IO_URING=1 tools/linux-docker.sh cargo run --example
  os_uring_batch
3 SITAS_DOCKER_IO_URING=1 tools/linux-docker.sh cargo run --example
  os_uring_abandon
```

Listing 9.10: `io_uring` labs

Observe the dispatcher snapshots. Focus on pending submissions, tracked operations, buffered completions, registered wakers, and abandoned operations.

Try this:

1. Increase the number of submitted operations in `os_uring_batch.rs`.
2. Drop a future before completion in a small variant of the abandon example.
3. Watch how abandonment and deferred buffer counters change.

The lesson: completion-based I/O has a different cancellation problem than readiness-based I/O. The kernel may still own an operation after the future is gone.

9.11 Lab 10: Sharded index build

Advanced

Run:

```
1 cargo run --example sharded_index_build -- --records 512 --shards 4 --seed 0
  xbeef --cleanup
```

Listing 9.11: Sharded index build

On Linux with `io_uring`:

```
1 SITAS_DOCKER_IO_URING=1 tools/linux-docker.sh cargo run --example
  sharded_index_build_uring -- --records 512 --shards 4 --seed 0xbeef --
  cleanup
```

Listing 9.12: `io_uring` index build

Observe how partition work is spread across shards, then merged into one final sorted index.

For the mailbox-transfer variant:

```
1 cargo run --example sharded_index_mailbox -- --records 512 --shards 4 --seed 0
  xbeef
```

Listing 9.13: Mailbox index build

Observe the printed logical assembler placement. The default mailbox variant uses one more assembler work unit than shards, so one shard owns two logical receivers. Record keys route to assembler names first, and the placement table then resolves those names to shards.

Try this:

1. Change the record count.
2. Change the shard count.
3. Change `-assemblers` in the mailbox variant.
4. Compare per-shard partition summaries and final verification output.

The lesson: the runtime's value appears when a problem splits into owned shard-local work combined through explicit owned results or messages.

9.12 After the labs

After running these labs, return to chapter 8 and re-read the modules. The source is easier to follow after seeing observable behavior. If a result is surprising, use the debugging checklist (section 8.11) before changing code.

10 Operations

10.1 Building this manual

The manual lives in `docs/system`. Build it with:

```
1 cd docs/system
2 make
```

The output is `docs/system/sysdoc.pdf`. The build uses `pdflatex` twice so the table of contents and references settle.

If the build fails, first check that the LaTeX distribution has the standard packages used by `sitasdoc.cls`: `geometry`, `fancyhdr`, `titlesec`, `titletoc`, `listings`, `xcolor`, and `hyperref`. The manual avoids project-specific generated assets so it can be rebuilt from text alone.

Keeping the manual build simple is intentional. The documentation does not depend on the runtime being runnable, Docker being available, or generated diagrams. A plain text-to-PDF path makes it cheap to update design explanations alongside code changes.

10.2 Cleaning generated files

```
1 cd docs/system
2 make clean
3 make distclean
```

`make clean` removes LaTeX auxiliary files. `make distclean` also removes the generated PDF.

10.3 Linux validation

Some runtime behavior is Linux-specific. The Docker helper mounts the project directory into a Linux container and can request an `io_uring`-capable configuration:

```
1 SITAS_DOCKER_IO_URING=1 tools/linux-docker.sh cargo test os::uring::tests --
  --nocapture
```

When the host or container configuration cannot provide `io_uring`, the Linux-only examples report that the backend is unavailable rather than faking a successful run.

The Docker path exists because macOS is a good development environment for most of the code, while Linux is the only honest target for Linux-specific primitives like hard CPU affinity and

`io_uring`. The helper keeps that platform check repeatable without pretending the project is platform-neutral where it is not.

10.4 Continuous integration

The project includes a GitHub Actions workflow (`.github/workflows/ci.yml`) that runs on every push and pull request:

- `rustfmt` and `clippy` checks on Ubuntu.
- Full test suite (`cargo test --all-targets`) on both Ubuntu and macOS.
- Documentation build (`cargo doc --no-deps`) with `#![warn(missing_docs)]` enforcement.
- All example programs are compiled and run on Ubuntu.

This provides cross-platform validation for the unsafe FFI and platform-gated features (`epoll/kqueue/poll`, CPU affinity, `io_uring`).

10.5 Documentation conventions

Every source file carries a `//!` module-level comment describing its purpose and the types it contains. The crate root declares `#![warn(missing_docs)]`, which makes undocumented public items compilation errors.

- **Public API** (`///`): required on all `pub` items. Enforced by the lint.
- **Module docs** (`//!`): present on every `.rs` file, both top-level public modules and internal sub-files.
- **Unsafe blocks** (`#![unsafe]`): every unsafe block has a preceding comment explaining why the operation is sound.
- **FFI blocks**: all use `unsafe extern "C" { ... }` per Rust Edition 2024.
- **Test names**: follow a descriptive `subject_action_expected_outcome` convention.

The standard commands for checking these are part of the normal validation workflow:

```
1 cargo fmt --check
2 cargo clippy --all-targets -- -D warnings
3 cargo test
4 cargo doc --no-deps
```

10.6 Validation strategy

Use narrow tests first, then broaden. This keeps iteration fast while protecting runtime invariants.

```
1 cargo test task_snapshot -- --nocapture
2 cargo test scheduling_group -- --nocapture
3 cargo test serve_tcp -- --nocapture
4 cargo fmt --check
5 cargo test
6 cargo clippy --all-targets -- -D warnings
7 cargo doc --no-deps
```

Listing 10.1: Typical focused-to-broad workflow

For changes touching Linux-only code, use the Docker helper after local macOS validation:

```
1 SITAS_DOCKER_IO_URING=1 tools/linux-docker.sh cargo test
2 SITAS_DOCKER_IO_URING=1 tools/linux-docker.sh cargo clippy --all-targets -- -D
  warnings
```

Listing 10.2: Linux platform pass

The order matters. Focused tests make it easier to understand the failure. Broad tests protect against regressions in adjacent runtime paths. This is especially important in executor code, where a small lifecycle change can affect timers, readiness, joins, and shutdown simultaneously.

10.7 Interpreting unsupported features

Unsupported is a valid result when the platform cannot provide a feature. CPU placement is unsupported on non-Linux platforms. `io_uring` may be unavailable if the kernel, container runtime, or security profile blocks ring setup. The project reports those cases honestly instead of hiding them behind fake success.

A runtime experiment should make platform boundaries visible: readiness works on several Unix backends, hard CPU affinity is Linux-specific, and `io_uring` is both Linux-specific and dependent on host/container permissions.