Serene Language Report

A modern, dependently typed Lisp

Sameer Rahmani · March 21, 2025

Contents

[1] Quark would've been a better name. But let's stay true to our roots.

Preface

> What I cannot create, I do not understand
>
> — Richard Feynman

The Journey to Serene

In 2018, I tried several times to contribute to a certain programming language. I proposed improvements that I believed would help the community. None of them succeeded. The reason was not technical merit. It was that the people who governed the language did not want to improve it and were happy with things as they were. That experience left a bad taste, but it also gave me an idea. With my wife's steady encouragement, I decided to renew my love of computer science and programming languages by building my own.

At that time, as a software engineer, I liked dynamically typed languages with very dynamic runtimes, such as Ruby, Clojure, and Scheme. So it made sense to start by reading their codebases, studying their designs, and looking at how they solved problems. I tried to follow the evidence and the logic like a careful scientist. But I could not find good reasons for many of the design decisions in my favourite languages. The more I looked, the more I saw that I had to start from the beginning: by studying logic, mathematics, and the history of computer science.

Now, after seven years of studying mathematics and logic, trying out different ideas, implementing imaginary languages in various host languages, and building many experimental interpreters and compilers, I can finally say I know what I want. My vision has changed a lot since those early days. It now rests on a deep understanding of the mathematics rather than on surface preferences.

This document is the specification of the language I set out to build—a language called **Serene**.

What is Serene?

Serene is the result of that seven-year journey from intuition to mathematical understanding. It is a programming language for the next generation of software systems, where correctness is not optional but essential. Today a software failure can spread across connected systems and cause anything from financial loss to danger to life. We need languages that help us build reliable software from the start, not as an afterthought.

Serene combines the mathematical rigour of dependent type theory with the practical needs of systems programming. It is both a programming language and a theorem prover. It follows the Curry–Howard correspondence, where programs are proofs and proofs are programs. This is not just academic elegance. It is a practical foundation for building software we can reason about with mathematical certainty.

The language grew out of a simple realization. The problems I had with existing languages were not surface issues that better syntax or more features could fix. They were fundamental design problems that came from a lack of a mathematical foundation. Phillip Wadler[2] puts it well: there are two kinds of programming languages, those that were invented and those that were discovered. Serene starts from solid mathematical principles and builds upward, instead of piling on features and hoping they fit together.

First I set up a mathematical system by defining the principles and axioms that govern computation. Then I discover the language that follows naturally from those foundations. This is how mathematicians work: they do not make up theorems at random, but discover what must follow from their axioms.

The foundation I chose is recursive functions of symbolic expressions together with dependent type theory and the Curry–Howard correspondence, where types are propositions and programs are proofs. Guided by my values and principles, I derive from this foundation, step by step:

- What constitutes a valid expression (the syntax emerges from proof terms)
- How expressions should be evaluated (operational semantics follow from proof normalization)
- What guarantees the language can provide (safety properties are theorems about the type system)
- How effects should be tracked (effect systems emerge from modal logic)
- How modules should compose (following categorical principles of composition)

Because of this approach, every feature in Serene exists not because it seemed like a good idea, but because it follows from the mathematical foundation. There are no arbitrary design decisions—only discoveries about what the mathematics implies.

Why Serene Matters

The software industry has a deep problem with complexity. Our systems have grown too large for us to understand fully, which leads to subtle bugs, security holes, and unpredictable failures. The usual tools for software quality—testing, code review, static analysis—are needed but not enough. They can find bugs, but they cannot prove that no bugs remain.

Working across different programming paradigms taught me that this problem is not inevitable. It comes from languages that put convenience over correctness, that hide complexity instead of managing it, and that push problems to runtime instead of solving them at compile time.

Serene takes a different path: make incorrect programs impossible to write. Through its dependent type system, totality requirements, and explicit effect tracking, Serene ensures that well-typed programs cannot crash, cannot access memory unsafely, and cannot have undefined behaviour. It does this not through runtime checks or garbage collection alone, but through compile-time verification that guarantees correctness.

What This Specification Covers

This specification is a complete technical description of the Serene language, from its mathematical foundations to practical matters of implementation. It includes:

- Guiding Principles: The philosophical foundation that drives every design decision
- Core Language: Syntax, semantics, and type system with formal mathematical treatment
- Type System: Dependent types, universes, and bidirectional type checking
- Operational Semantics: How programs execute at both compile-time and runtime
- Namespaces and Compilation: Modular programming and separate compilation
- Macro System: Hygienic, typed macros with first-class support

[2] https://homepages.inf.ed.ac.uk/wadler/

- Effects and I/O: Safe interaction with the external world
- Foreign Function Interface: Honest interoperability with existing C libraries

Each chapter builds on earlier concepts while keeping mathematical rigour. The specification aims to be precise enough for implementers and still clear enough for language users who want to understand the theory behind it.

The Future of Programming

This specification is more than another programming language. It carries a different view of what programming languages should be. Years of study and experiment have convinced me that programming languages must move past their current limits. Software has become too important to accept a world where bugs, crashes, and security holes are treated as normal.

Serene points to one possible future: a world where program correctness is checked mechanically, where software failure is as rare as an error in a mathematical proof, and where "it compiles, therefore it works" actually means something.

This specification both describes that future and offers a plan for building it. It is not just a set of technical design decisions. It is a belief that we can do better—that we already have the mathematical tools to build software we can trust completely, not because we tested it a lot, but because we proved it correct.

As Feynman said, "What I cannot create, I do not understand." This specification is my attempt to understand programming languages by creating one built on the mathematical principles I have learned. I invite you to join me in this understanding, and in building software we can trust completely.

Sameer Rahmani
March 21, 2025

Part 1

The Language

Guiding Principles and Values

The design of Serene is guided by the following core principles[3], explicitly listed in order of priority. Earlier principles take precedence when resolving design conflicts.

1.1. Correctness Over Performance

Serene puts correctness and robustness first. It should make writing correct programs easy and writing incorrect ones hard. Performance optimisations must never break correctness.

- Type checking is decidable and terminating, explicitly enforced.
- Memory safety is guaranteed without unsafe escape mechanisms.
- All functions must be total: they must terminate for all well-typed inputs.
- Runtime errors must be impossible for well-typed programs.

1.2. Mathematical Simplicity Over Surface Simplicity

The language stays simple by using a small set of orthogonal concepts. It favours deep mathematical simplicity over surface convenience. Serene prefers designs with lower information entropy.

- Single, powerful dependent function type.
- Unified types and values.
- No distinction between statements and expressions.
- Consistent evaluation across contexts.

1.3. Small, Composable Rule Set

Serene is minimal. It is defined by a small set of rules that interact clearly.

- Minimal core language.
- Each rule serves one clear purpose.
- No special cases or ad-hoc rules.
- Understandable fully by comprehending core constructs.

1.4. Proofs as Programs, Programs as Proofs

Serene uses the Curry–Howard correspondence to combine programming with theorem proving. It stays a general-purpose language.

- Specifications as types, implementations as terms.

[3] Principles and not guarantees

- Unified language for proofs and programs.
- Computable proofs, verifiable programs.
- No separation of “value-level” and “type-level” programming.

1.5. Explicit Over Implicit

Explicit notation is preferred to avoid ambiguity. Implicit notation is allowed only when it is clear.

- Predictable, decidable type inference.
- Implicit arguments resolved by clear, decidable unification.
- Explicit evaluation order.
- No hidden allocations or side effects.

1.6. Phase Distinction Clarity

Serene explicitly distinguishes compile-time and runtime phases.

- Types exist exclusively at compile time.
- Macros and staging explicitly marked.
- No runtime type information by default.
- Clear compile-time evaluation semantics.

1.7. Composition Over Inheritance

Serene favours composing small, reusable components over inheritance hierarchies.

- First-class types, namespaces, functions, macros, etc.
- Structural typing where suitable.

1.8. Declarative Instead of Imperative

Serene promotes declarative programming. It expresses logic clearly, without explicit control flow.

- Clear and direct logical expression.
- No free mutable state and side effects.
- Facilitates reasoning, refactoring, and verification.

1.9. Transparency and Predictability

Program behaviour in Serene is transparent and predictable.

- No implicit side effects or hidden states.
- Defined semantics for intuitive reasoning.
- Supports debugging, testing, and verification.

1.10. Local Reasoning

One should be able to reason about expressions locally.

1.11. Secure by Construction

Serene builds in security through explicit constraints and clear encapsulation.

- Clearly delineated unsafe operations.

- Explicit management of IO and unsafe operations.
- Safe and secure program design by default.

1.12. Human-Friendly Tooling and Documentation

Serene aims for tooling and documentation that are easy to use and complete.

- Built-in documentation.
- Intuitive refactoring and navigation tools.
- Complements instructive error messages.

1.13. Error Messages as Teaching Tools

Error messages teach the user. They explain clearly why an issue happened.

- Detailed explanations of errors.
- Type errors include derivation details.
- Suggestions for resolutions.
- References to relevant language principles.

1.14. Stability Through Simplicity

Serene stays stable by staying simple. It avoids frequent changes to the core.

- Small and stable core language.
- Extensions via libraries rather than language features.
- Guaranteed backward compatibility.
- Features expressible through existing constructs.

1.15. Interoperability as Necessity

Interoperability with existing systems is essential for Serene.

- Clean FFI to C and other system libraries.
- Predictable memory layout for integration.
- No mandatory runtime or garbage collection.
- Generates interoperable libraries for other languages.

1.16. Non-Goals

Explicit non-priorities for Serene include:

- Compatibility with existing Lisp languages.
- Type-level Turing completeness.
- Maximum performance at correctness's expense.
- Feature parity with extensive languages (Haskell, Agda, Coq).
- Implicit coercions; all conversions are explicit.
- Full fledge theorem prover.

Description of the Language

This chapter describes the grammar of the implementation-agnostic **Serene language**.

2.1. Mathematical Foundation

Serene is designed based on intensional dependent type theory with quantitative type theory (QTT) as an extension.

The mathematical foundation is developed in the type theory chapter. The core constructs Serene provides in terms of a dependent type theory are:

- A hierarchy of Universes
- Π-Type
- Σ-Type
- Identity Type
- Inductive Type

2.2. Serene Programs

A program consists of an expression and an environment.

2.3. Expressions

Everything in Serene is an expression, and expressions are either types, or are described by some types. Expressions have evaluation (normalization) rules based on their types.

2.4. Values

Expressions in their normal form are Values. Values always evaluate to themselves.

2.5. Atoms[4]

Atoms are the basic building blocks of Serene expressions. They include:

2.5.1. Numbers

Numbers are values[5].

[4] Quark would've been a better name. But let's stay true to our roots.

[5] Thus, evaluate to themselves

```
42        ; Integer
3.14      ; Float
-17       ; Negative integer
```
Serene

2.5.2. Strings

Strings are values.

```
"hello"           ; String literal
"line one\ntwo"   ; With escape sequences
```
Serene

2.5.3. Symbols

Symbols are not values. Every symbol is a data constructor of type `Symbol`. A symbol normalises to the value it is bound to. The environment assigns a value to each symbol, and the symbol evaluates to that value. They are like variables in other languages, but immutable.

```
x
Int
+
```
Serene

Built-in symbolic constants include:

```
    λ          ; Lambda symbol
    ∀          ; Universal quantifier (Π-Type)
    ∃          ; Existential quantifier (Σ-Type)
    ->         ; Function type constructor
    ≡          ; Identity type
    universe   ; Universe function (Y is an alias)
    quote      ; Prevent evaluation (also written as ')
    quasiquote ; Enable interpolated quoting (also written as `)
    unquote    ; Evaluate inside quasiquote
    splice     ; Splice multiple elements
    force      ; Evaluate a quoted form
    refl       ; Equality proof constructor
    :          ; Type annotation
    the        ; Another way type annotation
```
Serene

2.6. Compound Forms

2.6.1. Vectors

Vectors are values.

```
[x y z]
[1 2 3]
```
Serene

2.6.2. Lists

Lists are not values. They normalise to a function application.

```
(f x)
(f x y z)
```
Serene

2.7. Functions and Application

The first element of a list is evaluated and called based on its type. The rest of the elements are passed to it as arguments.

```
(+ 1 2)   ; => 3
((+ 1) 2) ; => 3
```
Serene

2.8. Core Built-in Forms

2.8.1. Type Annotation

```
(: x Int)           ; x of type Int
(: x Int y String)  ; x of type Int and y of type String

;; Or as an alternative to the above
(the Int x)
(the Int x String y)
```
Serene

Both `:` and `the` are functions.

2.8.2. Function Type Constructor (->)

```
 (-> Int Bool) ; A function that take an Int and returns a Bool
 ;; A function that taken and Int and returns a function
 ;; that take an Int and returns an Int
 (-> Int (-> Int Int))
```
Serene

2.8.3. Lambda (λ)

Lambda is a function, and functions are values. λ takes a **vector** of parameters as its first argument and the body expression as its second argument.

```
(λ [x] x)
(the (-> Nat Nat) (λ [x] (+ 1 x)))
(: (λ [x] (+ 1 x)) (-> Nat Nat))
(λ [(: x Int)] (+ x 1))
(λ [(: x Int) (: y Int)] (+ x y))
```
Serene

2.8.4. Universal Quantifier (∀)

Universal quantifier or a Π-type.

```
(∀ [(: x A)] B)
```
Serene

2.8.5. Existential Quantifier (∃)

Existential quantifier or a Σ-type.

```
(∃ [(: x A)] B)
(: example (∃ [(: n Nat)] (Vec n Int)))
```
Serene

2.8.6. Identity Type (≡)

```
(≡ 2 2)
```
Serene

2.8.7. Quoting and Laziness

Quotes are values.

```Serene
(quote x)
'x
(force (quote (+ 1 2)))
(quasiquote (list ,x ,y))
(unquote x)
(splice xs)
```

2.9. Binding Constructs

2.9.1. def

Short for *definition*, binds the first argument to the second argument globally.

```Serene
(def x 5)
(def (: x Int) 5)
```

2.9.2. let

```Serene
    (let [(: x Int) 10
          y         30]
      (+ x y))
```

2.9.3. match

```Serene
(match expr [(pattern result) ...])
```

2.10. Defining Data Types

2.10.1. deftype

```Serene
(deftype (: Bool (universe 0)) (: true Bool) (: false Bool))
(deftype (: Nat (universe 0)) (: zero Nat) (: succ (-> Nat Nat)))
(deftype (: List (-> (universe 0) (universe 0)))
  (: a (universe 0))
  (: nil (List a))
  (: cons (-> a (List a) (List a))))
(deftype (: Vec (-> Nat (universe 0) (universe 0)))
  (: n Nat)
  (: a (universe 0))
  (: nil (Vec zero a))
  (: cons (∀ [(: m Nat)] (-> a (Vec m a) (Vec (succ m) a)))))
```

2.11. Evaluation Order

2.11.1. Compile Time

- Type expressions are normalized during compilation.
- Normalization follows a strict call-by-value strategy.

2.11.2. Runtime

- Types are erased.
- Runtime uses call-by-value for evaluation.

2.12. Type Universes

```
(: (universe 0) (universe 1))
(: Int (universe 0))
(: (List Int) (universe 0))
(: (-> (universe 0) (universe 0)) (universe 1))
(: (μ 0) (μ 1))
```
Serene

2.13. Complete Example

```
(deftype (: Nat (universe 0))
  (: zero Nat)
  (: succ (-> Nat Nat)))

(def (: plus (-> Nat Nat Nat))
  (λ [(: m Nat) (: n Nat)]
    (match m
      [zero n]
      [(succ m′) (succ (plus m′ n))])))

(def (: head (∀ [(: n Nat) (: a (universe 0))]
                (-> (Vec (succ n) a) a)))
  (λ [(: n Nat) (: a (universe 0)) (: xs (Vec (succ n) a))]
    (match xs [(cons m x xs′) x])))
```
Serene

2.14. Type Inference

- All unannotated definitions infer the most general type.
- Inference is conservative: either succeeds uniquely or fails.
- Annotations do not affect behaviour, only document intent.

2.15. Protocols

```
(defprotocol Num [a]
  (: zero a)
  (: plus (-> a a a)))

(definstance (: Num Nat)
  (: zero zero)
  (: plus (λ [(x Nat) (y Nat)] ...)))
(Vec (plus 2 2) Int)
```
Serene

2.16. Scope and Closures

Serene uses **lexical scoping**: variables are bound by the structure of the code, not by the call site. The same scoping rules apply to all binding forms, such as λ, `let`, `match`, and `def`.

- **Bindings are local** to their enclosing expression.
- **Shadowing** is permitted.
- **Closures** capture their environment.
- **Name resolution** is lexical.

```
(def (: make-adder (-> Int (-> Int Int)))
  (λ [(: x Int)]
    (λ [(: y Int)] (+ x y))))

(def add-five (make-adder 5))
(add-five 2) ; → 7
```
Serene

2.17. Totality and Recursion

Serene enforces totality for all functions: every function must terminate for all well-typed inputs. This gives strong guarantees such as memory safety, predictable evaluation, and decidable type checking. As a result, Serene does not allow general recursion or partial functions.

2.17.1. Structural Recursion (Required)

All recursive functions must be structurally recursive. That is, each recursive call must use an argument that is syntactically smaller. The compiler checks this at compile time.

```
(def length [(: xs (List a))]
  (match xs
    [nil 0]
    [(cons _ tail) (succ (length tail))]))
```
Serene

This form of recursion is predictable, simple to implement, and makes termination checking clear to the user.

2.17.2. Fuel-Based Recursion

Some computations are not structurally recursive but still terminate. For these, Serene encourages an encoding based on **fuel**: an explicit counter argument that bounds the recursion.

```
(def retry [(: fuel Nat) (: try (-> Result))]
  (match fuel
    [zero fail]
    [(succ f) (match (try)
                [success x x]
                [fail (retry f try)])]))
```
Serene

This technique models bounded retries, search loops, and backtracking while keeping functions total. Because the fuel argument strictly decreases, termination is clear and enforced.

2.17.3. Excluded: Partial Functions

Serene does not provide partial functions via a `Partial` type qualifier or annotation, for the following reasons:

- Violates Serene's principle of "correctness over convenience"
- Introduces implicit unsoundness at the boundary of total and partial code
- Complicates type checking and user understanding
- Requires a whole system of purity tracking, totality propagation, and unsafe coercions

Instead, all functions in Serene must be total. To express failure or non-returning computations, use explicit `Maybe` or `Either` types.

2.17.4. Excluded: General Well-Founded Recursion

Serene does not provide general well-founded recursion via user-defined measures (such as lexicographic orderings) or sized types. This would allow definitions such as the Ackermann function, but it is excluded for the following reasons:

- Substantially increases complexity of the termination checker
- Introduces new syntax and user obligations for defining and verifying measures
- Makes error messages harder to understand and reason about
- Violates Serene's design principle of having a small, composable rule set

Serene is limited to structural recursion. To encode well-founded or non-structural behaviour, use fuel arguments or coinductive techniques.

2.17.5. Example: Fuel-Based Ackermann Approximation

The Ackermann function is total but not structurally recursive. In Serene, you must encode it using fuel:

```
(def ack-fuel [(: fuel Nat) (: m Nat) (: n Nat)]
  (match fuel
    [zero n] ; give up
    [(succ f)
     (match m
       [zero (succ n)]
       [(succ m1)
         (match n
           [zero (ack-fuel f m1 1)]
           [(succ n1)
            (ack-fuel f m1 (ack-fuel f m n1))])])])))
```
Serene

This encoding ensures termination by bounding the recursion depth, even though the original function cannot be written structurally.

2.17.6. Summary

Serene enforces totality through structural recursion. Computations that are not structurally recursive can be encoded with explicit constructs such as fuel. Its priorities are simplicity, predictability, and mathematical integrity.

Part 2

Theory

Type System

This chapter covers the type theory behind the language. It is about the type theoretic concepts, not the implementation. The syntax stays as close as possible to the common literature of the field, and in particular to the style of [1].

Serene's type checker is a bidirectional type checker. It draws heavily on [2].

3.1. Contexts

A **context** Γ is a sequence of variable bindings:

$$\Gamma ::= \cdot \mid \Gamma, x : \tau$$

3.2. Syntax

3.2.1. Core Terms

$t, s ::=$	c		constant
	$\mid x$		variable
	$\mid (t\ s)$		(application)
	$\mid (\lambda\ x\ t)$		abstraction
	$\mid (\forall\ x\ B)$		dependent function type
	$\mid (:\ t\ A)$		(type annotation)
	$\mid$ let $[(:\ x\ A)\ s]\ t)$		(let binding)
	$\mid \mu_\ell$		(universe at level ℓ)
	$\mid ?\,\alpha$		(metavariable)

3.2.2. Constants

Constants unify literals and primitive types into one sort:

$c ::=$	n	(integer literal)
	$\mid r$	(double literal)
	$\mid s$	(string literal)
	$\mid$ ch	(character literal)
	$\mid T_{\text{prim}}$	(primitive type as value)
	$\mid$ world	(IO token)

$$T_{\text{prim}} ::= \mathsf{Int} \mid \mathsf{I8} \mid \mathsf{I16} \mid \mathsf{I32} \mid \mathsf{I64} \mid \mathsf{B8} \mid \mathsf{B16} \mid \mathsf{B32} \mid \mathsf{B64} \mid \mathsf{String} \mid \mathsf{Char} \mid \mathsf{Double} \mid \mathsf{World}$$

3.2.3. Semantic Values

Values produced by evaluation (NbE domain):

$$\begin{array}{rll} V ::= & \mathsf{VU}\ \ell & \text{(universe value)} \\ & \mid \mathsf{VConst}\ c & \text{(constant value)} \\ & \mid \mathsf{VClosure}\ \rho\ x\ t & \text{(closure)} \end{array}$$

3.3. Contexts and Judgments

A **context** Γ is a sequence of typed bindings:

$$\Gamma ::= \cdot \mid \Gamma, x : A$$

We use the following judgment forms:

Judgment	Reading
$\Gamma \vdash t : A$	In context Γ, term t has type A
$\Gamma \vdash t \Rightarrow A$	Synthesize: infer that t has type A
$\Gamma \vdash t \Leftarrow A$	Check: verify t against type A
$\Gamma \vdash A \equiv B$	A and B are definitionally equal

Resources

[1] F. Pfenning and R. Davies, “A judgmental reconstruction of modal logic,” *Mathematical Structures in Computer Science*, vol. 11, no. 4, pp. 511–540, Aug. 2001, doi: 10.1017/S0960129501003322.

[2] J. Dunfield and N. R. Krishnaswami, “Complete and easy bidirectional typechecking for higher-rank polymorphism,” in *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, Boston Massachusetts USA: ACM, Sep. 2013, pp. 429–442. doi: 10.1145/2500365.2500582.

Part 3

lxsameer's Serene Compiler

The Memory Manager

Every program the runtime hosts must get its memory from somewhere, and the shape of that somewhere decides more than it first appears. A language reaches for memory in a few very different rhythms. Some values live for the length of a scope and then all die together, some must last as long as the program runs, and a few grow and shrink and are freed one at a time. A single allocator tuned for one of those rhythms serves the other two badly. This chapter describes the part of the runtime that gives all three cheap allocation without pretending they are the same thing — the *memory manager*.

The plan is one mechanism and three lifetimes over it. The mechanism is a large slab of memory that hands out space by moving a single pointer forward. On top of that one primitive sit three ways to own the memory it produces — a scope that frees everything at once, a chain that never frees until the program ends, and a plain allocator for buffers that come and go. The rest of the chapter builds the mechanism first, then the three lifetimes, then the ownership handle that ties a scope to a block chain, and finally an honest account of what the design buys and what it costs.

4.1. The Block, the One Mechanism

Everything starts with the *block*. A block is a large slab of memory taken from the operating system in one request. Its size is two to the power of the `mm.block_size_magnitude` configuration knob, 128 KiB by default (a magnitude of 17). A power of two at least as large as the OS page is a whole number of pages on every platform whose pages are powers of two, which is all of them in practice. The runtime still verifies the multiple at startup, so a platform reporting an unusual page size fails loudly, and the slab always sits on page boundaries with nothing to round. Allocation into a block is the cheapest thing an allocator can do. The block keeps an `offset` into itself that marks where the used region ends, and to allocate you round that offset up to the requested alignment, check that the object fits, place it there, and move the offset past it. There is no free list to search and no header on each object. Handing out memory is a few arithmetic operations and a store.

One slab is finite, so a block also carries a `next` pointer. When a slab fills, the manager allocates a fresh one and links it on, and a run of linked slabs behaves as one large region — the *chain*. The chain is freed all at once by walking the `next` pointers and releasing each slab. No object is ever freed on its own. The whole structure fits in a small header sitting at the front of the slab, with the payload following it.

```c
typedef struct srn_block_t {
  /// Next slab in the chain, allocated when this one runs out of room.
  struct srn_block_t *next;

  /// Total size of the slab, header plus payload.
```

```
    size_t              size;

    /// Bump pointer. Bytes below this offset are handed out; the rest is free.
    size_t              offset;

    /// Start of the payload. Forced to a 16-byte boundary.
    uint8_t             base[];
} srn_block_t;

_Static_assert(offsetof(srn_block_t, base) % 16 == 0,
               "srn_block_t::base must be 16-byte aligned");
```

The `base[]` field is a flexible array, so the payload lives in the same slab right behind the header. A static assertion forces the payload to start on a 16-byte boundary, which is the strictest alignment ordinary allocations ask for. Every object handed out of the block is placed at an offset from a 16-aligned start, so the alignment maths in the allocator only ever has to round up, never to fight the slab's own placement.

The allocation walk is one loop over the chain. Given a request for a size and an alignment, the manager first rejects anything larger than a single slab's payload — since every slab in a chain is the same size, a request that does not fit one cannot fit any, and this is the case the current design does not serve. Otherwise it starts at the root block and, for each block in turn, rounds the block's offset up to the requested alignment and tests whether the object fits in what remains. If it does, the object goes at that aligned offset, the offset advances past it, and the address is returned. If it does not, the walk moves to the `next` block, or, if there is no next block, allocates a fresh slab, links it on, and continues into it.

```c
static inline void *
alloc_in_block(srn_mm_t *mm, srn_block_t *root_block, size_t size, size_t
alignment) {
  // The caller holds the chain's lock for the whole call.
  if (size > block_capacity(root_block)) {
    // A request larger than one slab's payload takes a dedicated slab sized
    // to the request (the large-allocation rule); this walk is the common
    // path.
  }

  srn_block_t *target_block = root_block;
  for (;;) {
    // Align the absolute address, not the offset. The payload base carries
    // only the default 16-byte alignment.
    uintptr_t base        = (uintptr_t)target_block->base;
    uintptr_t aligned     = (base + target_block->offset + (alignment - 1)) &
                          ~((uintptr_t)alignment - 1);
    size_t aligned_offset = (size_t)(aligned - base);

    if (aligned_offset + size > block_capacity(target_block)) {
      if (target_block->next != nullptr) {
```

```
        target_block = target_block->next;
        continue;
      }
      srn_block_t *b = alloc_block_internal(mm);
      target_block->next = b;
      target_block       = b;
      continue;
    }

    target_block->offset = aligned_offset + size;
    return (void *)aligned;
  }
}
```

That single routine is the whole allocator. Every path in the memory manager ends up here, bumping a pointer into some chain. What changes between the paths is which chain is used and who is allowed to free it.

The manager keeps its blocks in a fixed table of `MAX_NUMBER_OF_BLOCKS`, which is 256 entries. A block id is nothing more than the index into that table, and a 256-bit occupancy bitmap records which entries are in use. Allocating a block finds a clear bit, marks it, and stores the new slab there; releasing a block frees the chain and clears the bit. A reserved sentinel, `SRN_BLOCK_NO_ID`, means "no block" and can be stored in a field to say that no chain is attached yet.

4.2. Three Lifetimes Over One Mechanism

The bump-into-a-chain primitive is neutral about how long its memory should live. Three public paths put it to use, and each one makes a different promise.

Lifetime	Freed	Use it for
Arena / block	Whole chain at once, when the owner releases it	Values tied to a scope that all die together
Immortal	Never during a run; only at shutdown	State that must last the whole program
Manual	Individually, by the caller	Dynamic, resizable, separately freed buffers

The first is the *arena*. A caller allocates a block, receives its id, and bumps allocations into that chain through the id. Everything it allocates shares one lifetime. There is no call to free a single object, and when the caller is done it releases the whole chain in one step. This is the common case, and it is reached through a context, described in the next section.

```c
srn_block_id_t srn_mm_allocate_block(srn_mm_t *mm);
void          *srn_mm_allocate_in_block_aligned(srn_mm_t *mm, srn_block_id_t id,
                                                size_t size, size_t alignment);
void           srn_mm_release_block(srn_mm_t *mm, srn_block_id_t id);

#define srn_mm_allocate_in_block(mm, id, T) \
```

```
7    (T *)srn_mm_allocate_in_block_aligned(mm, id, sizeof(T), alignof(T))
```

The second is the *immortal* chain. The manager holds one chain of its own that is never released while the program runs and is torn down only at shutdown. Allocations into it last for the entire life of the engine. This is where the runtime puts the state that has no shorter natural lifetime — the engine itself, the scheduler, the reactor, the JIT, and the interned keywords and strings that must stay valid as long as any code might name them. Reaching for the immortal chain is a deliberate act. A caller uses it precisely when it knows a value must outlive every scope.

```c
void *srn_mm_immortal_allocate_aligned(srn_mm_t *mm, size_t size, size_t
alignment);

#define srn_mm_immortal_allocate(mm, T) \
   (T *)srn_mm_immortal_allocate_aligned(mm, sizeof(T), alignof(T))
```

The third path steps outside the block machinery entirely. Some structures are genuinely dynamic — they grow, they shrink, and they are freed one at a time, so neither an arena nor the immortal chain fits them. For these the manager offers a plain allocator with the familiar three verbs.

```c
void *srn_mm_malloc(srn_mm_t *mm, size_t size);
void *srn_mm_reallocate(srn_mm_t *mm, void *ptr, size_t new_size);
void  srn_mm_free(srn_mm_t *mm, void *ptr);
```

These route through the manager so the backend can change without touching callers; the `mm` argument names the manager whose backend serves the call. Passing a null pointer to `srn_mm_free` is a no-op, matching the C library's own rule. The runtime uses this path for structures that resize over their life, such as the reactor's heap of pending timers and the scheduler's run-scoped arrays of workers and threads.

The choice between the three is left to the caller and is meant to be obvious from the value's lifetime. A value that shares a scope's fate goes in an arena. A value that must never die goes in the immortal chain. A buffer that resizes and is freed on its own goes through the manual allocator. Nothing picks for the caller, and nothing hides the choice.

4.3. The Context as the Ownership Handle

An arena is only useful if something remembers to release it. That something is the *context*. A context owns exactly one block chain and stands for a scope's worth of memory. It is small — a pointer to the engine, an optional parent, and the block id it allocates into.

```c
typedef struct srn_context_t {
   srn_engine_t         *engine;
   struct srn_context_t *parent;
   srn_block_id_t        block_id;
} srn_context_t;
```

Making a context allocates a fresh block and then, neatly, places the context struct itself inside that very block. The handle lives in the memory it owns, so there is nothing extra to free alongside it. Allocating through the context bumps into its chain, and the `ALLOC` and `ALLOCN` macros wrap the common cases of one value and an array.

```c
void *srn_allocate(const srn_context_t *ctx, size_t size, size_t alignment);
```

```c
void  srn_release (const srn_context_t *ctx, void *ptr);

#define ALLOC(ctx, T)     (T *)srn_allocate(ctx, sizeof(T), alignof(T))
#define ALLOCN(ctx, T, N) (T *)srn_allocate(ctx, sizeof(T) * (N), alignof(T))
```

Releasing a context frees its whole chain in one call. Because there is no way to free a single allocation, `srn_release` on one pointer is a deliberate no-op. It exists so that call sites can name the intent to be done with a value, and so that a future design could give it meaning, but today the only real reclamation is the whole chain going at once.

This matches how the language uses memory. A scope in the source has a beginning and an end, and the values it creates share that span. A context is that span made concrete. Open a context when the scope begins, allocate freely inside it while the scope runs, and release it when the scope ends and every value made in it goes at the same moment. There is no bookkeeping to get wrong, because there is nothing to free individually.

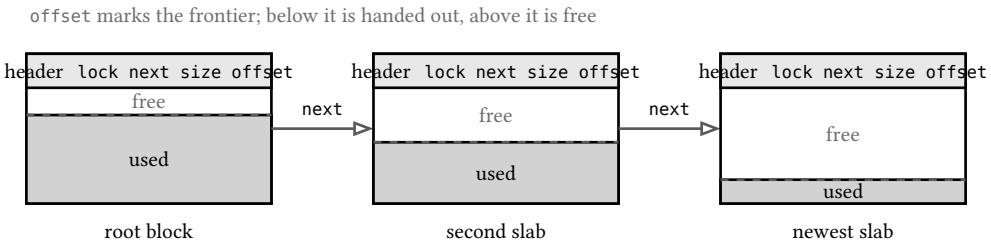

Figure 1: One context's block chain. Allocation bumps the frontier up inside the current slab; when a slab fills, a fresh one is linked on through `next`. Releasing the context walks the chain and frees every slab at once.

4.4. The Page Provider

The manager does not call the operating system for a slab directly. It goes through a small interface, the *provider*, with one function to get raw memory and one to give it back.

```c
typedef struct srn_memory_provider_t {
  void *(*allocate)(size_t size, size_t alignment);
  void  (*release)(void *p);
} srn_memory_provider_t;
```

The manager holds a pointer to one provider and asks it for every slab. The only provider today wraps the C library's aligned allocation — `aligned_alloc` and `free` on POSIX, the matching aligned pair on Windows. That is enough to run, and the provider lets the source of slabs change without the manager noticing. A pool that recycles freed slabs, an allocator tuned to the platform, or a hand-written replacement for `malloc` all fit behind the same two function pointers. This is composition rather than a fixed backend — the manager depends on the capability to get and return a slab, not on where the slab comes from.

4.5. Concurrency

The runtime runs fibers across a pool of worker threads, so the allocator has to be safe under threads. It is made safe by two separate lock scopes that never overlap.

The first is a single lock on the manager as a whole. It guards the block table and the occupancy bitmap — the shared bookkeeping that says which ids are in use and which slab each id points at. Any change to that bookkeeping, allocating a block or releasing a chain, takes the manager lock.

The second is one lock per chain, held for the whole allocation walk. The chain locks live in the manager, keyed by block id (plus one for the immortal chain), not inside the blocks. That placement is load-bearing. A release holds the chain's lock while freeing the chain itself, so an allocation racing a release either finishes first or resolves the released id to nothing and fails loudly.

The rule that keeps these two from tangling is stated plainly in the manager's own notes — the manager lock is for the table and bitmap, the chain lock is for a chain's contents. Allocation takes only the chain lock. Release takes the manager lock and then the chain lock before freeing. That is the one place the two nest, always in that order, so there is a single ordering rule and no cycle to deadlock on.

This split is what fits the M:N scheduler. Independent contexts each own their own chain, so two fibers working in different contexts, even on different worker threads, bump into different blocks and take different locks. They never contend. Contention appears in exactly one place — two fibers that share one context and run at the same time bump into the same chain and serialize on that chain's lock. That is the price of sharing a context, and it is confined to the case where sharing was chosen. The common path, a context per unit of work, has no contention at all.

The lock itself is the simplest thing that works — a spinlock built on an atomic flag, acquired with acquire ordering and released with release ordering, spinning until it wins. It suits the allocator because the region it protects is tiny, just a few arithmetic operations on the offset, so a waiting thread spins for a very short time rather than paying to sleep and wake.

4.6. Pros and Cons

The design earns its keep, and it also gives things up. Both deserve to be stated without spin.

The strengths follow from the one mechanism. Allocation is a constant-time bump of a pointer, with no free list to search and no per-object metadata to store or maintain, so both the time and the space overhead of an allocation are close to nothing. Releasing a whole chain in one step matches the language's scope model exactly — a scope's values are made together and die together, and the allocator frees them the same way, which means a dangling free of one object is not a bug the caller can even write. The two lock scopes suit the scheduler, letting independent work allocate without ever contending and confining contention to the deliberate case of a shared context. The provider interface isolates the source of memory, so the backend can improve without disturbing anything above it. Slabs are large and contiguous, so allocations made together sit together and read well from cache. And the three explicit lifetimes let a caller say plainly how long a value should live rather than leaving the allocator to guess.

The costs are the other face of the same choices.

- A single allocation larger than one slab's payload is not served. The walk rejects it outright rather than stitching slabs together for one object, so a request over roughly 128 KiB has no home in the block paths today.
- The block table is a fixed 256 entries and does not grow. A program that needs more live chains than that at once will run out of ids, and the ceiling is a compile-time constant rather than something that expands on demand.
- A live arena is never reclaimed in part. Because there is no individual free, a long-lived context only ever grows until it is released. A scope that lives a long time and allocates steadily holds every byte it ever asked for until the end.

- Immortal allocations are never reclaimed during a run at all. Anything placed in the immortal chain stays until shutdown by definition, so it must be used only for state that truly lasts the whole program.
- The manual allocator is a thin path beside the block machinery. It exists for structures that resize over their life, and its backend is swappable per manager through the `mm` argument.
- Alignment requests are honoured for any power of two: the walk aligns the absolute address rather than the offset, so the payload's own 16-byte base does not bound what a caller may ask for. An alignment must still fit a slab's payload, like a size must.
- The spinlocks busy-wait. Under contention on a shared chain a waiting thread burns cycles rather than yielding, which is the right trade for the tiny critical section here but is a cost worth naming.

None of these is a defect in the mechanism. Each is a boundary drawn on purpose to keep the common path fast and the model small, and each is a place the design can grow when a real workload asks it to.

4.7. How the Design Reflects Serene's Principles

The memory manager is small enough that the language's values show through it clearly, without having to be pointed at one by one.

Correctness and memory safety come first. Freeing a whole region at once removes an entire class of bugs — there is no individual free to double, to forget, or to call on a stale pointer, because the only free is the chain going all at once. When the manager is misused, it does not limp on. A request too large for a slab, a block id out of range, or a bitmap that disagrees with itself all stop the program loudly rather than corrupting memory quietly.

The design is explicit rather than implicit. The caller chooses a lifetime and passes the manager or the context on every call. Nothing allocates behind the caller's back, and no value acquires a lifetime it was not given. Which chain a value lives in, and therefore when it dies, is written at the allocation site.

The rule set is small and composable. One primitive — bump a pointer into a chain of slabs — underlies all three lifetimes. The arena, the immortal chain, and even the context all reduce to that same walk over the same block structure, with only the ownership differing. There is one idea to understand, and the rest is how that idea is put to use.

And the design favours composition over a fixed backend. The provider interface makes the source of raw memory a part that can be swapped, so the manager is built by composing a bump allocator over an interchangeable page source rather than by wiring one particular allocator into its core.

4.8. Summary

The memory manager comes down to a handful of commitments.

Aspect	**Choice**
Mechanism	One primitive — bump a pointer into a chain of large slabs
Slab size	Two to the `mm.block_size_magnitude` knob, 128 KiB by default
Block table	Fixed 256 entries, indexed by block id, tracked by a 256-bit bitmap
Arena lifetime	Whole chain freed at once; no individual free
Immortal lifetime	One chain that lives until shutdown
Manual lifetime	`malloc` / `realloc` / `free` for resizable buffers

Aspect	**Choice**
Ownership handle	A context owns one chain; release frees it all
Page source	Behind a provider interface; the current one wraps aligned `malloc`
Locking	A manager-wide lock for the table, a per-block lock for the bump pointer, never nested

Fibers

> The art of programming is the art of organising complexity, of mastering multitude and avoiding its bastard chaos as effectively as possible.
>
> — Edsger W. Dijkstra

A language that talks to the world must, sooner or later, learn to wait. A program reads from a socket that has nothing to say yet; it asks a disk for bytes that have not arrived; it sleeps until a timer fires. The real question is what the rest of the program does in the meantime. The answer that runs through this part of the report is that it does other work. Waiting is not a wall the whole process hits, but a single thread of computation stepping aside so its neighbours can run. This chapter describes what makes stepping aside possible — *fibers*.

This is a chapter about the C-level machinery. How fibers eventually surface in the Serene language — the `IO` type, `Future`, `async-race`, and friends — is the subject of the *Effects and I/O* chapter, and we only point at it here. What follows is the engine room — how a stack is created, how the processor is handed from one fiber to another, and how all of this stays safe when the ground beneath it is raw memory and bare registers.

5.1. What is a Fiber exactly?

A fiber is a unit of execution scheduled by the runtime, not the operating system — a *cooperative, lightweight thread* that runs alongside the operating-system thread. It can be paused in the middle of a deep call chain and resumed later exactly where it stopped, with every local variable and every pending return intact, and many fibers live inside a single operating-system thread, taking turns cooperatively. A fiber and a thread are alike, and one difference between them decides most of the rest — *when*, and *by whom*, a running path is paused so that another can run.

A thread is usually *preemptive*. (Whether it really is depends on the operating system, so treat this as the common case, not a rule.) The system can interrupt it at any point, on an instruction it did not choose. Two things follow from that. First, data can be left in a bad state. A thread may be stopped halfway through updating some data, and other threads must be kept from reading it until the update finishes. Second, in return, the kernel can run several threads at once across many CPUs and cores, which gives real parallelism — but it leaves the programmer to guard every piece of shared data.

A fiber is *cooperative*. Its own worker never interrupts it. It runs until it yields (again, depending on what the layer below allows), so it stops and restarts only at points it chose, and two fibers on the same worker never overlap. Switching between them is cheap — it stays in user space, with no trip into the kernel and no full save of processor state. Fibers run on worker threads, and no two fibers can

run on the same woker in an instance. Cooperative scheduling removes preemption within a worker, not concurrency across the pool. Fibers on different workers can run at once, just as threads can.

None of this is new in the abstract — green threads, coroutines, and goroutines are cousins of the same idea — but the shape it takes in Serene is forced by Serene's own commitments, and that is what makes the design worth writing down carefully. We have no garbage collector, no stack maps, and a runtime that must host code the compiler never saw — hand-written C, foreign libraries, and machine code the JIT emits at run time. Every decision in this chapter follows from taking those constraints seriously, and where one of Serene's guiding principles decides the matter, I name it as we go.

5.2. A Fiber Among Its Neighbours

The last section compared the fiber with the thread it most resembles. But the thread is only one of a wider family, and a fiber is defined as much by what it is *not*. It helps to place the others on the same axes — who schedules them, whether they can be interrupted against their will, whether they carry their own stack, and whether they can use more than one processor at once. The thread sits at one end — scheduled by the kernel, preemptive, with its own large stack, and truly parallel — and Serene runs its fibers across a pool of such threads (see the scheduler section), so threads give the parallelism and fibers the cheap, waitable concurrency on top.

A *stackless coroutine* — the async/await of many modern languages — gets the same pause-and-resume behaviour a different way. The compiler rewrites a function that can suspend into a state machine object on the heap, so there is no live stack to keep. This is cheap and precise, but it *colours* functions — only a rewritten function can suspend, and only another rewritten function can call it, so the property spreads through the call graph and stops at any boundary the compiler does not control. A fiber keeps a real stack so that it needs no rewriting and colours nothing — which is what lets it suspend hand-written C, foreign libraries, and JIT-emitted code alike. The next sections make that argument in full.

A *goroutine*, or the older *green thread*, is the closest match — also scheduled in user space, also carrying a stack, also run over operating-system threads. The differences are about policy. A goroutine runs on a stack that can grow and be moved, and it can be preempted — both of which a runtime can afford only if it has a garbage collector and precise stack maps to fix the pointers a moved stack breaks. Serene has neither, so its fibers are strictly cooperative and use a fixed stack. That makes fibers a good candidateeven for use cases in which a garbage collector is envolved.

A plain *callback*, or an *event loop*, gets concurrency with none of these. You register a function to run when an event fires. The cost is that a computation that would read top to bottom is turned inside out into a set of handlers, each having lost the stack that said what it was doing. A fiber gives that stack back — it lets waiting code be written as if it did not wait, one line after another, while the wait still hands the processor to others.

Primitive	**Scheduled by**	**Preemptible**	**Own stack**	**Parallel**
OS thread	the kernel	yes	yes, large	yes
Stackless coroutine	the caller	no	no	no
Goroutine / green thread	a language runtime	partly	yes, growable	yes
Serene fiber	the Serene runtime	no	yes, fixed	yes, M:N over threads

The Serene fiber sits where these three goals meet – to follow the values from the first chapter It is user-scheduled and cooperative, so nothing interrupts it without its consent (explicit over implicit). It is stackful, so it can run foreign and generated code that was never rewritten for it. And its stack has

a fixed size, because without a garbage collector a stack cannot be safely moved or grown. The next sections take each of these in turn.

5.3. Why a Fiber Needs Its Own Stack

To pause a computation and resume it later, you must save everything it was in the middle of doing. On a normal machine, that state lives on the stack — the chain of call frames, each holding the caller's saved registers, its locals, and the address to return to. A fiber that suspends three calls deep into a parser, which is itself ten calls deep into the evaluator, must keep that whole set of frames in place. When it resumes, the processor finds the frames exactly as it left them — same frame pointers, same locals, same return addresses — and carries on.

So a fiber is a call chain frozen in place. A frozen call chain cannot share its memory with anyone. Fiber A's suspended frames sit in a fixed span of addresses, say `[base_A .. sp_A]`. If a second fiber started writing its own frames in the same span, it would overwrite A's saved state, and A could never wake correctly. The conclusion is clear. Each fiber must own its own separate region of memory to use as a stack. This is the single fact that most of the chapter follows from, and it names the model.

We call this the *stackful* model, because each fiber carries a real machine stack. The alternative is the *stackless* model. There the compiler rewrites every function that might suspend into a heap-allocated state machine, so suspending means returning from the function with its progress recorded in an object, not freezing a live stack. Stackless coroutines are cheap when they apply, but they apply only to code the compiler rewrote. They "colour" every function in the call chain — a function that may suspend can only be called by another that may suspend, and the property spreads outward until it reaches a boundary the compiler does not control. Serene's runtime is built entirely of such boundaries. The evaluator will hand control to JIT-produced machine code; that code will call into hand-written C in the runtime; that C will call out through the FFI into a foreign library; and any of those frames might be on the stack when a fiber decides to wait. A stackful fiber suspends all of them the same way, because suspending is a property of the *stack*, not of the *functions* that built it. It needs no help from the compiler, so it works for code the compiler never compiled. The fiber works with foreign code as it finds it, rather than demanding that foreign code be rebuilt in the fiber's image. That is exactly what a language with a JIT and an FFI needs, and it is why Serene is stackful.

The price of a real stack is that switching between fibers means switching stacks, which we cover in the next section but one. First a harder question — how big should that stack be, and what happens when it is not big enough?

5.4. Why the Stack Cannot Grow

The operating system gives a thread a stack that seems to grow on demand. You might hope a fiber's stack could do the same. It cannot, and the reason drives the rest of the design.

There are only two ways to grow a stack, and Serene can use neither.

The first is to *grow by moving*. When the stack fills, allocate a larger region, copy the old contents over, and continue there. The problem is that a stack is full of pointers *into itself*. Frame pointers chain one frame to the next; the address of a local is taken and passed to a callee; an `alloca` buffer lives at a computed offset; return addresses point back into the caller's frame. Move the stack and every one of those pointers is now wrong. To fix them you must know, at the moment of the move, exactly which words on the stack are pointers into the stack and which are just integers that happen to look like addresses. That knowledge is a *precise pointer map*, produced by the compiler and used by a precise garbage collector. Serene has no garbage collector and emits no stack maps, and it runs JIT and FFI code for which no such maps could exist. We cannot find the pointers, so we cannot rewrite them, so

we cannot move the stack. While it might reads like a weakness but in fact, it is a trade-off that allows us to avoid unnecessary logic in exchange for growable stack. Which makes the scheduler lighter, faster, and safer.

The second is to *grow by chaining*, the segmented-stack approach. When a frame would overflow the current chunk, allocate a fresh chunk, link it to the old one, and run there, with a small check in every function's prologue to detect the boundary. Golang took this road and then abandoned it because of the *hot split* problem. Suppose a tight loop makes calls that straddle a segment boundary. Every iteration crosses into a new chunk and then unwinds back, so the runtime allocates and frees a segment on every trip through the loop — a lot of churn right where performance matters most. Worse for us, the scheme needs *every* function prologue to cooperate, including the FFI and JIT code that, as we just saw, cooperates with nothing.

That rules out both growth strategies. What remains is a stack of a *fixed* size, chosen when the fiber is created and never changed. This is not a reluctant compromise. It is the only option that fits the universal reach that made us choose stackful fibers in the first place. A fixed stack asks nothing of the compiler and works for any code. Its cost is specific. A fiber that truly outgrows its stack must *fault* rather than expand. Left alone, that cost would be unacceptable — a silent overflow into a neighbour's frames is exactly the kind of memory corruption Serene exists to abolish. The next section shows how a single mechanism makes that cost acceptable.

5.5. The Guard Page and Lazy Commit

What makes a fixed stack acceptable is the *guard page*. It is the first of four reasons the runtime maps each fiber stack straight from the operating system's virtual memory, rather than taking it from `malloc` or the block arena. The arena is wrong for a separate reason — it is a bump allocator that frees whole chains of allocations at once, whereas fiber stacks are created and reclaimed one at a time. A raw virtual memory mapping gives four properties the arena cannot, and every platform Serene targets can provide them, each under its own name. On Linux, macOS, and the BSDs the call is `mmap`; on Windows it is `VirtualAlloc`; on WebAssembly it is a slice of linear memory.

It helps to picture the layout of a fiber stack. Addresses run from high at the top to low at the bottom. The stack grows *downward* as calls nest, and the guard page sits at the very bottom, at the low end the stack grows toward.

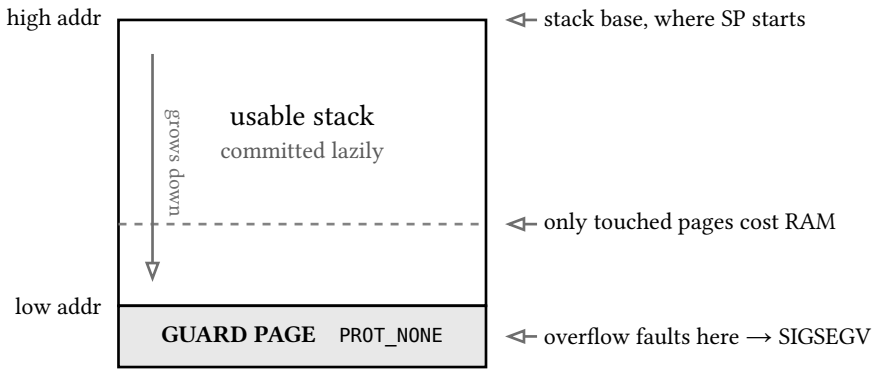

Figure 2: The layout of a fiber stack. Low addresses are at the bottom; the stack grows downward, toward the guard page that terminates it.

Guard page — overflow becomes a deterministic fault. The lowest page is mapped with no access at all (`PROT_NONE` on POSIX, `PAGE_NOACCESS` on Windows). The moment the stack grows far enough to touch it — just before it would start clobbering whatever lies below — the memory-management unit faults, and the process stops at the exact instruction that overran. The failure is local, immediate, and easy to debug — a crash with a clear culprit, not a silent corruption that shows up a thousand

instructions later as a wrong answer somewhere unrelated. This is the most important reason the runtime maps stacks this way, and it is *correctness over performance* almost word for word. We accept a hard fault, which is a fixable engineering problem, to rule out silent memory corruption, which is not. It turns the one real danger of a fixed stack — overflow — from undefined behaviour into a clean, deterministic fault.

Lazy commit — "fixed but roomy" is nearly free. A fresh mapping hands out address space, not physical memory. Pages are backed by real memory only when first touched, and zero-filled at that moment. So a fiber can reserve a large stack and, if it only ever nests a few frames deep, pay for just one or two pages of real memory. This answers the obvious objection to fixed stacks — that reserving room for the worst case wastes memory in the common case. It does not. Reserving is cheap, and only use costs anything.

Whole pages, aligned to a page. Per-page protection can only be set on a page boundary, so the guard page needs a mapping aligned to one. That same granularity gives a clean half-open span `[base, base + size)` that we can hand straight to the sanitiser, which we rely on shortly.

Independent lifetime. Each stack is its own mapping, reserved when a fiber is created and unmapped or returned to a pool when it dies, on its own schedule, with no ties to any other allocation's lifetime.

The trade is made on purpose. Overflow is made *visible and safe* rather than *absorbed automatically*. We reduce the cost by choosing a sensible default size, by letting a caller that knows it will recurse deeply ask for a larger stack, and in time by the evaluator's tail-call handling, which keeps many computations shallow that would otherwise grow without bound.

5.6. The Stack Provider

`mmap` is a POSIX idiom, but the runtime is not a POSIX program; it must also run on Windows and inside WebAssembly. So getting a stack is hidden behind a narrow internal interface, and the platform detail never reaches `fiber.c`. This interface pays off twice. It keeps the fiber core portable, and it leaves room for the *source* of stacks to change later — per-thread caches for a multi-threaded scheduler, a stack pool in the memory manager, a WebAssembly backend — without touching anything above it. It is the small, composable rule set applied to platform differences. The capability is the same everywhere, and only how it is expressed changes from one platform to the next.

A stack is described by the two ends of its usable region and its guard page; the usable size is derived (`srn_fiber_stack_size`). Two functions allocate and free it.

```c
typedef struct srn_fiber_stack_t {
  void *start;  // high end of the usable region; SP starts here
  void *limit;  // low end of the usable region, one page above the guard
  void *guard;  // the guard page: the mapping's base address
} srn_fiber_stack_t;

[[nodiscard]] srn_fiber_stack_t srn_fiber_stack_alloc(size_t size);
void                            srn_fiber_stack_free(srn_fiber_stack_t stack);
```

Reserving address space with lazy commit and arming a guard page at the low end are the two primitives every backend must provide.

Platform	Reserve + lazy commit	Guard page
Linux / macOS / BSD	`mmap` anonymous (`MAP_ANON` / `MAP_ANONYMOUS`)	`mprotect(PROT_NONE)`
Windows	`VirtualAlloc(MEM_RESERVE)` then `MEM_COMMIT`	`VirtualProtect(PAGE_NOACCESS)`
WASM	a slice of linear memory	an explicit check, or none

The POSIX corner hides a few platform details worth recording, because they cost an afternoon each if you find them the hard way. Anonymous mappings on POSIX are demand-zero, so lazy commit is free without any special flag; `MAP_NORESERVE` is mostly a Linux knob for swap accounting, since macOS overcommits anyway and FreeBSD treats the flag as a no-op. `MAP_STACK` is the flag that really matters, and it matters differently across the BSDs. OpenBSD *requires* the stack pointer to always sit in memory mapped with `MAP_STACK`, so fiber stacks there must request it or the kernel faults the first time the switch lands on the new stack. FreeBSD treats it as an advisory grows-down hint; Linux currently ignores it; and macOS does not define it at all. Windows offers native OS fibers through `CreateFiber` and `SwitchToFiber`; they are a fine option on that platform, but we keep the hand-rolled switch everywhere for uniformity, so we debug one mechanism rather than several. Each platform supplies a provider behind the same two-function interface: POSIX through `mmap` plus `mprotect` (with `MAP_STACK` where the platform honours it), Windows and WebAssembly through their own mechanisms.

That leaves the size. A stack is fixed at creation, guarded, and faults on overflow. The size is a per-fiber parameter, so a caller that knows it needs depth can ask for it right at the creation site. This is *explicit over implicit* applied to memory — a fiber that will recurse deeply requests the room it needs, rather than relying on a hidden mechanism to rescue it. When the caller states no preference, the configured default applies — the `fiber.stack_size` knob, 128 KiB unless overridden. Lazy commit makes the address-space cost of a generous default nearly free, while committed memory stays one page until a fiber actually recurses, so the default errs large and the per-fiber parameter covers the outliers.

5.7. Handing Over the Processor — the Context Switch

With each fiber owning a stack, switching from one to another is a small act. The processor state that must survive the switch is not the whole machine — it is only the registers the calling convention requires a function to preserve, plus the stack pointer itself. Save those from the running fiber into its control block, load them from the fiber being resumed, and the processor is now running on the other fiber's stack, in the middle of whatever it was doing when it last paused. One routine does this.

```c
// Save the current context into `from`, restore `to`, and resume on `to`'s
// stack. Returns --- on `from`'s stack --- when some fiber later switches
// back into `from`.
void srn_fiber_swap(srn_fiber_ctx_t *from, srn_fiber_ctx_t *to);
```

The signature is worth reading carefully. A call to `srn_fiber_swap` does not return in the usual sense; control leaves onto `to`'s stack and may not come back for a long time, or ever. When it *does* return — on `from`'s stack, to the instruction after the call — it is because some other fiber has switched back into `from`. From `from`'s point of view the world simply resumes, perhaps much later, perhaps doing something completely different. This single routine is the pivot the whole library turns on.

Because what must be saved is set by the architecture's calling convention, `srn_fiber_swap` is written in assembly for the architectures we care most about, and falls back to the portable `ucontext` facility elsewhere.

Target	Mechanism and saved state
x86-64	Hand-written switch saving `rbx`, `rbp`, `r12–r15`, `rsp`, the x87 control word and `MXCSR`.
aarch64	Hand-written switch saving `x19–x30`, `sp`, `d8–d15` and the floating-point control register.
other native	`ucontext` (`swapcontext`) fallback.
WASM	Asyncify or the stack-switching proposal, behind the same signature.

Two details in the saved set are easy to forget and costly to omit. The floating-point control registers — `MXCSR` and the x87 control word on x86-64, the FP control register on aarch64 — carry rounding modes and exception masks that belong to the fiber as much as any general-purpose register, and a switch that skipped them would let one fiber silently change another's arithmetic. The general-purpose sets are exactly the callee-saved registers of the respective C ABIs — including the frame pointer and, on aarch64, the return-address register `x30` — because the switch is, formally, a function call the C compiler must see as preserving them. This routine is the *only* assembly in the whole library; every other line is plain C. Keeping the machine-dependent surface this small is the small-composable-core principle paying off where it is hardest to honour.

5.7.1. The Bootstrap Fiber

A switch needs two parties — a fiber to switch *to* and a fiber to switch *from*. The very first switch has a problem — the thread that calls it is not running inside any fiber we created; it is running on the operating system's own thread stack. We solve this by adopting that thread as *fiber 0*, the bootstrap fiber. It is a fiber object like any other, except that its stack is the thread's original stack — no region is mapped for it, and its stack is never freed, because the library did not create it. Its only job is to be a valid `from` for the first switch, so that starting up has the same shape as every switch after it — one rule, no special case for the very first time.

5.7.2. Entry and Exit Trampolines

A freshly created fiber poses the opposite problem. It has never run, so its saved context cannot point at "the instruction after some earlier switch" — there is no earlier switch. So its first resume cannot just restore registers and return into frames, because no frames exist yet. The fix is to set the new fiber's saved stack pointer so that the first switch lands not in the middle of some function but at the top of a small C trampoline. The trampoline runs on the fresh stack and calls the fiber's entry function, building the first real frame itself.

```c
// Runs on the new fiber's stack the first time it is resumed. The assembly
// trampoline pops the fiber pointer the frame fabrication planted and calls
// this launcher. The result is type erased and a null result is legal.
[[noreturn]] static void srn_fiber_launcher(void *fiber_ptr) {
  srn_fiber_t *fiber = fiber_ptr;
  fiber->result = fiber->entry(fiber->ctx, fiber->arg);
  fiber->state  = SRN_FIBER_DONE;
  srn_fiber_switch_final(srn_fiber_worker_loop()); // switches away; never back
```

```
9 }
```

The exit is the trampoline's job too, and it is delicate for a matching reason. When the entry function returns, it returns *into the trampoline* — but the trampoline must not return in turn, because there is nothing beneath it to return to; the stack below the trampoline is empty. So instead of returning, the trampoline marks the fiber `DONE`, records its result, and switches away to the scheduler, never to come back. A fiber's life thus begins and ends inside the trampoline, and control never falls off the bottom of a fresh stack into frames that were never there.

The entry function's shape is generic at this tier — it takes an opaque argument — so the fiber library stands on its own, with no dependency on the value model above it. Fibers produced by code generation wrap the Serene ABI, the uniform signature every compiled Serene function presents, behind this same generic entry.

```c
srn_value_t *(srn_context_t *ctx, srn_value_t *argv, uint32_t argc);
```

so a fiber is, in the end, just a Serene function running on a stack of its own. The trampoline is the only code that calls the entry function, so switching from the generic shape to the ABI shape later costs nothing above it.

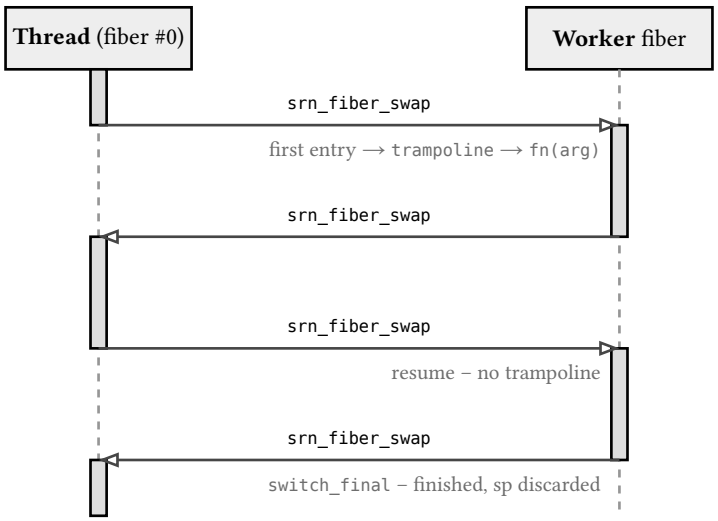

Figure 3: The life of a worker fiber. Every arrow is one `srn_fiber_swap`, which saves the leaving context's stack pointer and loads the target's; the shaded bar shows which side is running. The trampoline is reached only on the first entry — every later resume returns to the point just after the worker's own swap.

5.7.3. Telling the Sanitiser About the Switch

There is a subtle hazard in hand-rolling a stack switch, and Serene hits it because the build enables `-fsanitize=address,undefined` by default. AddressSanitizer keeps its own shadow picture of where the stack is and what is live on it. When we swap stacks behind its back, that picture goes stale, and the sanitiser starts reporting things that are not there — false stack-overflow reports, and spurious use-after-return complaints as a fiber resumes on a stack the sanitiser thinks was abandoned. The fix is to tell the sanitiser about the switch explicitly, wrapping the swap in two calls.

```c
// On the outgoing fiber, immediately before swapping to `to`:
__sanitizer_start_switch_fiber(&from->fake_stack, to->stack.limit,
                               srn_fiber_stack_size(to->stack));
srn_fiber_swap(&from->fiber_ctx, &to->fiber_ctx);
// Back on `from` after it is later resumed:
```

```
6  __sanitizer_finish_switch_fiber(from->fake_stack, NULL, NULL);
```

The `start` call tells the sanitiser that execution is about to move to `to`'s stack, hands it the clean `[limit, start)` span the provider kept, and saves the outgoing fiber's shadow bookkeeping in `from->fake_stack` for when it resumes. The matching `finish` call, run once `from` is scheduled again, restores that bookkeeping. The two calls must sit right around the swap — `start` the last thing before it, `finish` the first thing after. There is one special case. A fiber that is finishing and will never resume passes a null `fake_stack_save` to `start`, which tells the sanitiser to discard rather than save the dying fiber's shadow stack. This is the case the exit trampoline hits when it switches away from a `DONE` fiber. ThreadSanitizer has a matching set of calls — `__tsan_create_fiber`, `__tsan_switch_to_fiber`, `__tsan_destroy_fiber` — which will matter once real threads arrive under a multi-threaded scheduler. Making the switch announce itself to the tooling is not an afterthought; it is the same correctness-over-performance commitment carried into the build configuration, where the cheap thing would be to silence the sanitiser and the correct thing is to teach it.

5.8. The Fiber Object and Its Life

Everything above comes together in one structure. A fiber is created, runs, perhaps yields or parks and resumes any number of times, and eventually finishes. That history is a small state machine of five states.

State	Meaning
`NEW`	Created, stack mapped, never resumed.
`READY`	On the run queue, eligible to run.
`RUNNING`	Currently executing on its OS thread.
`SUSPENDED`	Parked off the run queue; awaits an external `ready`.
`DONE`	Entry function returned; result and error are final.

The transitions are few, and each one says something plain about the fiber.

From	To	Trigger
`NEW`	`READY`	creation enqueues the fiber
`RUNNING`	`READY`	`yield`
`RUNNING`	`SUSPENDED`	`suspend`
`SUSPENDED`	`READY`	`ready`
`READY`	`RUNNING`	scheduled
`RUNNING`	`DONE`	entry returns

Every legal thing a fiber can do is one of these six arrows. `DONE` is terminal, nothing re-enters `NEW`, and anything not on the list cannot happen — a fiber cannot be readied while it runs, cannot run while it is parked, cannot leave `DONE`. The difference between `READY` and `SUSPENDED` is the hinge of the whole design. A `READY` fiber will run again on its own, while a `SUSPENDED` fiber will run again only when some

outside party calls `ready` on it. That difference is what keeps the scheduler from needing to know *why* a fiber waits, a point we return to below.

The structure that carries this is mostly built from parts the earlier sections already justified, which is a good sign the design hangs together.

```c
typedef struct srn_fiber_t {
    srn_fiber_ctx_t           fiber_ctx;   // saved SP + callee-saved registers
    srn_fiber_stack_t         stack;       // mapped stack + guard page
    _Atomic srn_fiber_state_t state;       // wakers CAS SUSPENDED -> READY
    srn_context_t            *ctx;         // GIVEN, not owned (see below)
    srn_fiber_entry_t         entry;       // entry function
    void                     *arg;         // argument handed to the entry
    srn_fiber_result_t        result;      // set on DONE; type erased, may
    be null
    srn_fiber_park_fn         park_commit; // one-shot suspend hand-off
    void                     *park_arg;
    void                     *fake_stack;  // sanitizer bookkeeping
    struct srn_fiber_t       *link;        // intrusive run/wait-queue link
    struct srn_fiber_t       *waiters;     // fibers blocked in wait_for
    struct srn_fiber_t       *reg_prev, *reg_next; // live-fiber registry links
    const char               *name;        // optional, for debugging
} srn_fiber_t;
```

The `fiber_ctx` is the register and stack-pointer snapshot the context switch saves and restores; `stack` is the mapping with its guard page; `fake_stack` is the sanitiser's bookkeeping; `result` is the single, type-erased way an entry ends (failures travel inside it as values). The `link` field is worth a note. A fiber is threaded directly onto whatever queue it belongs to — the run queue, a wait queue — through this one embedded pointer, so enqueueing and dequeueing allocate nothing, and a fiber is on at most one such queue at a time, exactly as the state machine guarantees. One field, `mem`, carries a contract important enough to set apart.

5.8.1. The Memory Contract

A fiber does not create or own a memory context. It is *given* one — usually the `srn_context_t` already in use at the site that spawned it — and everything it allocates goes into whatever block that context points at. Borrowing rather than owning is *explicit over implicit* applied to a fiber's memory, and the decision has consequences worth spelling out, because they are what keeps fibers cheap and their results easy to hand back.

- The fiber never releases the context. Its lifetime belongs to whoever created it, not to the fiber.
- Several fibers may share one context. Their allocations interleave in the same block and all live until the owner releases it, which is exactly what you want for a group of fibers working together on one task.
- A fiber's `result` is allocated in that shared context, so it lives as long as the context does. There is no separate copy-to-escape step when a result outlives its producer; it was never anywhere it needed to escape from.

The contract on the caller is simple, and it is the caller's job to honour it. The context handed to a fiber must outlive the fiber and every consumer of its results. Lend a fiber a context that you free too early and you have a use-after-free. The design stays simple by making that the caller's explicit responsibility. There is one wrinkle that does not bite today but will later. When a multi-threaded scheduler arrives and two fibers that share a context run on two operating-system threads at once,

they will bump-allocate into the same block at the same time. The block already carries a spinlock — the same kind of lock the engine uses to guard its shared tables — so this is *safe*. But it is a point of contention, and it is the one place where the convenience of a shared context will have to be weighed against the cost of fibers fighting over one allocator. We note it now and revisit it then.

5.9. The Scheduler

A scheduler is what gives fibers their turns, and Serene's is deliberately minimal. An `srn_scheduler_t` owns the ready work and the registry of live fibers, and that is nearly all of it. There is *no event loop* inside it. The verb `srn_sched_run` drains the ready work across its worker threads — resuming each ready fiber in turn, letting it run until it yields, parks, or finishes — and returns once the pool goes quiet. What happens when the work runs out is not the scheduler's concern; the embedder, or the reactor, decides when more arrives. This restraint is the point. The scheduler hands out turns; it does not own the program's outer loop and does not assume it knows what the program is for. A scheduler that does exactly one thing is a scheduler you can reason about completely.

5.9.1. Suspend, Wake, and Staying Out of the I/O Business

The core discipline of the scheduler is that it never knows *why* a fiber waits. It knows only three verbs.

Primitive	Behaviour
`srn_fiber_yield`	Cooperative. Re-enqueue self at the tail and pick the next ready fiber; the yielding fiber runs again soon, of its own accord.
`srn_fiber_suspend`	Park self off the run queue. The fiber stays parked until an external `ready` wakes it.
`srn_fiber_ready(f)`	Move a suspended fiber back to `READY`. This is what "wake me when X happens" actually calls.

The useful idiom — *suspend, and wake me when X happens* — is built from these, and that is where the design's restraint pays off. Before it parks, a fiber hands its own handle to whatever party can detect X — a timer that will fire, an I/O reactor watching a descriptor, another fiber that will finish a piece of work. Then it suspends. When X finally happens, that party calls `srn_fiber_ready` on the handle, and the fiber rejoins the run queue. The fiber library provides the mechanism of waiting and waking and nothing more; it has no opinion about what is being waited *on*. This is the clearest expression in the whole design of a small, composable rule set. Rather than list the reasons a fiber might wait and bake each one into the scheduler, we provide one reason-agnostic pair, suspend and ready, and let every concrete reason be built on top of it without the scheduler ever learning the reason's name. It is also why there is no built-in event loop and no I/O in the scheduler itself; the scheduler is, by design, agnostic to the reason any fiber waits.

The whole surface is small enough to read at a glance.

```c
srn_scheduler_t   *srn_sched_init(srn_engine_t *engine);
void               srn_sched_run(srn_scheduler_t *sched, size_t nworkers);

srn_fiber_t       *srn_fiber_make(srn_context_t *ctx, srn_scheduler_t *sched,
                                  const char *name, srn_fiber_entry_t entry,
                                  void *arg, size_t stack_size);
void               srn_fiber_schedule(srn_fiber_t *fiber);
```

```
srn_fiber_t       *srn_fiber_spawn(srn_context_t *ctx,
                                   srn_fiber_entry_t entry, void *arg);
srn_fiber_t       *srn_fiber_spawn_copy(srn_context_t *ctx,
                                        srn_fiber_entry_t entry,
                                        const void *arg, size_t size);
void               srn_fiber_yield(void);
void               srn_fiber_suspend(srn_fiber_park_fn commit, void *arg);
void               srn_fiber_ready(srn_fiber_t *fiber);
srn_fiber_result_t srn_fiber_wait_for(srn_fiber_t *target);
srn_fiber_t       *srn_fiber_current(void);
```

`srn_fiber_make` carries the two contracts of the chapter in its signature. It takes the borrowed `ctx` the new fiber will allocate into, and the explicit `stack_size` the caller chooses. Creation and execution are separate steps. `make` registers the fiber and leaves it `NEW`, so a caller wires up a whole graph of fibers before anything runs, and `srn_fiber_schedule` makes one runnable, exactly once. The spawn pair collapses the two steps with every default chosen, the engine's scheduler, the configured stack, an autogenerated name, and `srn_fiber_spawn_copy` additionally copies the argument into the fiber's context, so the argument's lifetime is the fiber's own rather than the caller's stack frame. `srn_fiber_wait_for` is the one verb not yet seen, and it is no new mechanism but a combination of the two above — the waiting fiber suspends itself until the target reaches `DONE`, and the target, as the last thing it does before finishing, calls `ready` on its waiter and hands back the `result`. `srn_fiber_current` reports the fiber running on the calling worker — a thread-local read.

5.10. The Work-Stealing Deque

Fibers are spread across worker threads without a central bottleneck by the classic method. Each worker owns its own run queue, and an idle worker *steals* from a busy one. The queue is a *Chase–Lev work-stealing deque* — a fixed ring of ready fibers with two ends, where the owning worker works one end and thieves take from the other. Besides the per-worker deques, the scheduler keeps a single shared global queue, which is both the overflow and the place fibers enter from outside any worker.

This rests on one thing the design leaves out. A fiber stores no pointer to the operating-system thread running it. It does not belong to a worker; which worker runs it is decided fresh each time it is scheduled, never baked into the fiber. That is what lets a fiber created on one worker be stolen and run on another. The shared-context contention noted in the memory contract is the one place this reaches back and touches another choice — the block's spinlock keeps concurrent allocation through a shared context correct.

Two signed, monotonically increasing counters index a worker's ring — `bottom`, the owner's end, and `top`, the thieves' end. The live slot for an index is `index & (cap - 1)`, so the capacity is a power of two.

Operation	What it does
`push` (owner, `bottom`)	Add a fiber at the bottom. Reports "full" so the caller can overflow it to the global queue.
`pop` (owner, `bottom`)	Take the newest fiber from the bottom. Lock-free and uncontended, save for the last element.

Operation	What it does
`steal` (thief, `top`)	Take the oldest fiber from the top with a compare-and-swap. Several thieves may race; one wins.

Because the owner alone touches `bottom`, the common path — a busy worker pushing and popping its own work — is lock-free and contends with no one. A thief reaches across to `top`, the opposite end, so owner and thief collide only over the one remaining element when the deque is nearly empty. That single race is settled by a sequentially-consistent fence and a compare-and-swap on `top`. `pop` and `steal` both try the CAS, and exactly one succeeds. The counters are signed on purpose — the owner's `pop` briefly computes `bottom - 1`, which on an empty deque must read as *empty* rather than wrap to a huge value.

Getting that race right on a weakly-ordered processor (ARM, not just x86, whose strong model would hide the bug) is the whole subtlety, and it is not ours to reinvent. The implementation follows the published, machine-checked formulation.

- David Chase and Yossi Lev, *Dynamic Circular Work-Stealing Deque*, SPAA 2005 — the original algorithm. doi.org/10.1145/1073970.1073974
- Nhat Minh Lê, Antoniu Pop, Albert Cohen and Francesco Zappa Nardelli, *Correct and Efficient Work-Stealing for Weak Memory Models*, PPoPP 2013 — the C11-atomics version with the exact fences, which is what the code implements. doi.org/10.1145/2442516.2442524

A worker looks for work in a fixed order — its own deque first, then the global queue, then a steal from each peer in turn. Work produced while a fiber runs — a yield, a spawn, a wake — is pushed onto the running worker's own deque, keeping it close to the data it just touched; work entering from outside a worker, or spilling from a full deque, goes to the global queue. The single shared lock the scheduler still holds guards only that global queue and the parking of idle workers, never the hot path, where each worker runs from its own deque untouched by the others.

5.11. The Reactor

The scheduler stays out of the I/O business on purpose, but few programs never wait on the outside world. Something has to watch descriptors and clocks and turn their events into the `ready` calls the scheduler understands. That something is the *reactor*, and the same discipline that shaped the scheduler shapes it too. The reactor is a wake source and nothing more. It connects the operating system to the fibers by calling `srn_fiber_ready` when an awaited event arrives, and it does so without the scheduler ever learning what was awaited. The scheduler leaves the pumping of its queue to an outside agent; the reactor is that agent — a separate part, not stitched into the scheduler's core.

5.11.1. Completion, Not Readiness

There are two ways an I/O facility can talk. A *readiness* interface — the shape of `epoll` and `kqueue` — tells you *when* a descriptor could be read or written and leaves the syscall to you. A *completion* interface — the shape of `io_uring` and IOCP — takes the whole request, *read these bytes into this buffer*, and tells you when it is *done*. Serene's reactor presents a completion interface, because completion is the model that fits fibers. A fiber says "read this," suspends, and gets back a finished result, with no second step where it must remember to do the read itself.

The model is not the mechanism. Underneath the completion-shaped surface sits a *backend*, chosen for the platform, and the surface is the same whether that backend is really completion-based or only readiness-based. A readiness backend just does the translation the caller would otherwise do. It arms the descriptor, and when the kernel signals readiness it does the transfer itself and posts the finished

result. So the contract is the common subset every platform can support; anything one backend offers and another cannot stays private to that backend and never surfaces. Which kernel facility noticed an event is hidden from the fibers exactly as the reason a fiber waits is hidden from the scheduler — the same restraint, one level down.

5.11.2. A Thread of Its Own

The reactor runs on its own operating-system thread, outside the pool of workers that run fibers. The analogy is `io_uring`'s own — the reactor plays the part the kernel plays there. Workers hand it requests and collect finished results; the reactor does the waiting and the syscalls and produces the results, and — importantly — it never runs a fiber. This costs one thread, but that thread spends nearly all its life blocked in a single wait, so the cost is a *parked* thread, which is to say almost nothing.

The reactor runs on its own between nudges. It sleeps in its wait until one of three things wakes it — an event it was watching for, a timer it was counting down, or a *nudge* from a worker that has just handed it new work. A worker submits by placing a request on its channel and then poking the reactor's own wake descriptor. The reactor cannot see a freshly placed request while asleep, so the poke is what brings it back — to drain the new work, arm the new descriptor or insert the new timer, and return to waiting. It does not spin watching for work, which would trade the parked thread for a busy one and buy nothing.

5.11.3. Channels

A worker and the reactor talk through a *channel* — a pair of queues, one carrying requests toward the reactor and one carrying finished results back. There is one channel per worker, and a channel's identity is just the worker's — the `channel_id` and the worker number are the same number. The reactor owns every channel's queues and is solely responsible for their correctness; a worker holds only the small integer that names its channel and never touches a raw queue. This is the encapsulation the deque got, one part further out — the hard concurrency lives in one place, behind a handle.

Each request carries an opaque datum — the *fiber data* — that the reactor passes through untouched. It rides in on the submission and comes back out on the completion, and the reactor never looks inside it. It is the pointer by which a finished result finds the fiber that was waiting. No separate, stable identifier is needed to survive work stealing, because a fiber waiting on I/O is suspended and so off every queue. It cannot be stolen while its operation is in flight, the completion always returns to the channel that submitted it, and the fiber it belongs to is exactly where it was left. The routing key a shared queue would have needed never comes up.

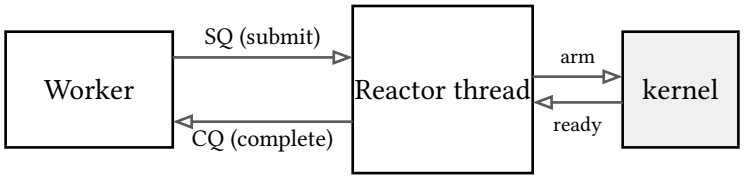

Figure 4: A single worker's channel. Each worker has a submission queue and a completion queue; the reactor takes submissions, drives the kernel, and posts each finished result back to the channel it came from. There is one channel per worker, and the fiber data on a request comes back on its completion.

Operation	What it is
Timer	Wait out a deadline; no descriptor. Completes when the time arrives.
Read / write	Transfer bytes to or from a descriptor. Completes when the transfer finishes or fails.

Operation	What it is
Accept / connect	Establish stream connections: take a pending peer off a listener, or complete an outbound connect once the socket is ready.
Recvfrom / sendto	Datagram transfers with an explicit peer address.
Recvmsg / sendmsg	Scatter/gather and ancillary-data transfers through a `msghdr`.
Poll	Wait for readiness only, no transfer; the gateway for timerfd, signalfd, eventfd and friends.
Cancel	Stop one in-flight operation, named by its fiber data; it completes as cancelled.
No-op	Completes at once; the way the submit-and-complete path is exercised on its own.

5.11.4. Waking Without Knowing

When the reactor finishes an operation it puts the result on the owning channel and then tells the scheduler that *that channel has events* — nothing more. It calls a single hook, `notify(channel_id)`, and the scheduler decides what that means. The reactor knows channels; it does not know workers, parking, or how a sleeping thread is woken. The scheduler owns the map from channel to worker and the wake primitive, and hides both behind the hook. This keeps the reactor as unaware of scheduling as the scheduler is of I/O — neither side knows the other's business, both ways across the boundary.

A result is collected in one of two ways, one for each state a worker can be in. A worker that is *running* drains its own completion queue as a normal step of its loop — a cheap read of a queue it already owns, needing no notification at all. A worker that has gone to sleep with nothing left to run cannot drain anything; for it, the notification is the only thing that wakes it. So the notification exists for exactly one purpose — to wake a *parked* worker. As with every wake in this design it must be free of the lost-wakeup race — a worker, as it parks, commits only after confirming its completion queue is empty and no notification is pending, so a result that lands the instant it falls asleep is never lost.

5.11.5. Time

A timer is just an operation with no descriptor. The reactor keeps the pending timers in a heap ordered by deadline, and each time it is about to wait it sets its timeout to the nearest deadline — or to forever, when no timer is pending. A newly submitted timer whose deadline falls sooner than the current wait allows needs no special handling. The same nudge that announces a new descriptor announces it, the reactor inserts it, recomputes the nearest deadline, and waits again on the shorter interval. "A sooner timer" is not a case to handle; it falls out of recomputing the timeout on every pass.

5.11.6. Order

Completion interfaces raise a worry about ordering. If two writes to a socket can finish in either order, the bytes could interleave. For a single fiber the worry goes away, because the blocking shape of the surface lets a fiber have only one operation in flight at a time — it submits a write, suspends, and cannot submit the next until the first has completed. So the reactor never holds two of one fiber's operations at once, and there is nothing to reorder. Two *different* fibers sharing one descriptor can of course interleave, but that is an ordinary race over a shared resource, to be settled by whatever higher-level object owns the descriptor, not by the reactor covering it up.

5.11.7. Stopping

The reactor changes what it means for the system to be done. The scheduler alone could stop the moment no fiber was ready, but a fiber suspended on I/O is not ready and yet must not be abandoned — its result is still coming. So the quiescence condition gains a clause. The system is finished only when no fiber is runnable *and* no operation is in flight, the latter tracked by a single count the reactor keeps. Three situations follow.

Situation	What happens
Quiescent	Nothing runnable, nothing in flight — the natural end. Everything is torn down.
Graceful	A stop is requested mid-flight (a signal, say). New submissions are refused — a fiber that asks to wait is handed a cancelled result and unwinds — while operations already in flight are allowed to finish and their fibers to drain. When the last one completes, the graceful case has turned into the quiescent one.
Forced	The abrupt stop. The reactor thread may be inside a syscall, where nothing outside can cleanly unwind it, so little is promised beyond letting the process end and the operating system reclaim what remains.

The graceful drain finishes only as its in-flight operations do. An operation that never completes on its own — a read on a connection that stays silent, a long timer — is bounded by *cancelling* it, the subject of the next section.

Two calls drive these endings from outside. `srn_sched_drain` asks for the graceful wind-down. It blocks new submissions and lets the operations already in flight finish, so the pool reaches quiescence on its own. `srn_sched_stop` is blunter — it stops the workers at the next clean boundary between fibers and leaves any still-queued fibers unrun, to be cleaned up when the subsystem is torn down. Neither call blocks, and both are safe to call from another thread or a fiber. Neither is async-signal-safe, and by design. A runtime that owns an event loop folds signals into it rather than promising handler safety call by call. A signal-initiated shutdown routes the signal into ordinary thread context first — a dedicated thread in `sigwait`, or a `signalfd`-style registration with the reactor — and calls `stop` or `drain` from there.

5.11.8. Cancelling an Operation, Versus a Fiber

A line the rest of the design leans on is worth stating plainly. The reactor knows only operations, so the most it can cancel is *one operation* — named by its fiber data, an in-flight operation is stopped and delivered as a cancelled result to whatever waited on it. Cancelling a *fiber* is a larger idea, and not the reactor's to do. A fiber suspended on I/O is cancelled by cancelling the operation it waits on; a fiber busy on the processor is cancelled by setting a flag it reads at its next suspend point. Either way the fiber unwinds itself — cooperatively, since C offers no way to unwind through frozen frames — and the reactor's per-operation cancel is one of the two ways the scheduler cancels a fiber. The reactor cancels operations; the scheduler cancels fibers.

5.12. Fiber I/O

The reactor is the engine; this is the surface a fiber actually calls. It is the third tier of the design — the blocking-shaped wrappers — and the only part that knows both the fiber world and the reactor. Everything above turned kernel events into `ready` calls; this hides that machinery behind a call that reads like ordinary code.

5.12.1. The Call That Looks Like It Blocks

From inside a fiber, I/O reads like plain code, one statement after another. A fiber calls `srn_fiber_read`, and the bytes are there when the call returns. Nothing at the call site shows that the fiber was parked and its worker ran other fibers in the meantime. This is the scheduler's *suspend, and wake me when X happens* idiom, with I/O as the X, wrapped so closely that the suspension is out of sight. There are no callbacks and no second colour of function; a fiber that does I/O looks like one that does not.

The care is in the wrapping. The call builds its request on the fiber's own stack and then suspends. The request is handed to the reactor not before the suspension but *inside* it, in the park-commit the worker runs once the fiber is off its stack. This is the ordering that makes `suspend` race-free. The fiber is fully parked before anything can wake it, so a completion cannot arrive while it is only half parked. The request carries a pointer to a small *waiter* on that parked stack. When the completion comes back on the worker's channel, the worker copies the result into the waiter and calls `srn_fiber_ready`; the fiber then resumes where it suspended, reads its result off its own stack, and returns.

5.12.2. A Result That Carries Its Error

The surface does not copy the system calls beneath it. A system call reports failure out of band, through a shared `errno` the caller has to remember to check. Serene's I/O returns a *fallible value* instead — either the operation's result or the error that replaced it. The error travels with the data it describes, and success is just the absence of an error, checked in one place. A call with nothing to report on success, such as a connect or a sleep, returns only that — the error, or nothing.

The errors are neutral. A failure is named by a tag that means the same on every platform — a refused connection, a reset peer, a broken pipe — not by a raw `errno` number whose value changes between systems. Only the backend touches the platform, so only the backend translates. It maps the operating system's error to one of the neutral tags, so a fiber handles a refused connection the same way whether it came from `epoll` on Linux or a completion backend elsewhere. The same rule that keeps the reactor from knowing about fibers keeps this surface from knowing about the platform. A few of its calls — create a socket, bind, open a file — wait on nothing and never touch the reactor; they are thin wrappers over the plain system call that share only the same way of reporting failure, so a caller sees one convention across the whole surface.

5.12.3. Files the Reactor Cannot Watch

One kind of descriptor defeats a readiness backend. `epoll` and `kqueue` answer the question *is this ready yet*, but a regular file is always ready. There is no moment of readiness to wait for, and arming a file for it is either an error or a no-op. A readiness backend cannot serve a read of a file the way it serves a read of a socket.

So the surface checks the descriptor first. When the backend cannot poll files and the descriptor is a regular file — or a block device, or a directory — the read or write runs synchronously, on the worker itself, and never reaches the reactor. For that moment the worker is just busy, as it is while running a fiber busy on the processor; the operation does not suspend and does not count against the number of operations in flight that quiescence watches. A completion backend such as `io_uring` has no such gap, serves files directly, and skips this detour.

The cost is plain. The worker's thread is blocked for the length of the syscall, and with a single worker that is the whole pool. This is an accepted trade-off for now. The next step — a small pool of threads for blocking work, so that file I/O concurrency no longer depends on the number of workers and one slow read cannot stall the rest — is designed but left until a real file workload needs it.

5.13. Summary

The design comes down to a small set of commitments.

Aspect	**Choice**
Execution model	Stackful, cooperative
Stack size	Fixed; no growth
Overflow	Guard page; deterministic fault
Stack memory	`mmap` + `mprotect` (POSIX), behind a provider interface
Scheduler	M:N work-stealing — per-worker Chase–Lev deques, a shared global queue, idle workers steal
I/O	A separate completion-based reactor on its own thread; the scheduler itself stays reason-agnostic
Sanitiser	`__sanitizer_*_switch_fiber` around every switch
Fiber memory	Given a `srn_context_t`; not owned
Language surface	Out of scope; a layer over spawn, suspend, ready, and wait_for