Serene Language Report

A modern, dependently typed Lisp

Sameer Rahmani · March 21, 2025

Contents

Preface 5
- The Journey to Serene 5
- What is Serene? 5
- Why Serene Matters 6
- What This Specification Covers 6
- The Future of Programming 7

Guiding Principles and Values 9
- Correctness Over Performance 9
- Mathematical Simplicity Over Surface Simplicity 9
- Small, Composable Rule Set 9
- Proofs as Programs, Programs as Proofs 9
- Explicit Over Implicit 10
- Phase Distinction Clarity 10
- Composition Over Inheritance 10
- Declarative Instead of Imperative 10
- Transparency and Predictability 10
- Local Reasoning 10
- Secure by Construction 11
- Human-Friendly Tooling and Documentation 11
- Error Messages as Teaching Tools 11
- Stability Through Simplicity 11
- Interoperability as Necessity 11
- Non-Goals 11

Description of the Language 12
- Mathematical Foundation 12
- Serene Programs 12
- Expressions 12
- Values 12
- Atoms[1] 12
  - Numbers 12
  - Strings 13
  - Symbols 13
- Compound Forms 13
  - Vectors 13
  - Lists 13
- Functions and Application 14

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

- Core Built-in Forms 14
  - Type Annotation 14
  - Function Type Constructor (->) 14
  - Lambda (λ) 14
  - Universal Quantifier (∀) 14
  - Existential Quantifier (∃) 14
  - Identity Type (≡) 15
  - Quoting and Laziness 15
- Binding Constructs 15
  - def 15
  - let 15
  - match 15
- Defining Data Types 15
  - deftype 15
- Evaluation Order 16
  - Compile Time 16
  - Runtime 16
- Type Universes 16
- Complete Example 16
- Type Inference 16
- Protocols (Preview) 16
- Scope and Closures 17
- Totality and Recursion 17
  - Structural Recursion (Required) 17
  - Fuel-Based Recursion 17
  - Rejected Idea: Partial Functions 18
  - Rejected Idea: General Well-Founded Recursion 18
  - Example: Fuel-Based Ackermann Approximation 18
  - Summary 18

Type System 21
- Contexts 21
- Syntax 21
  - Core Terms 21
  - Constants 21
  - Semantic Values 22
- Contexts and Judgments 22

Resources 23

Fibers 25
- Why a Fiber Needs Its Own Stack 25
- Why the Stack Cannot Grow 26
- The Guard Page, and the Role of `mmap` 27
- The Stack Provider Seam 28
- Handing Over the Processor: the Context Switch 29
  - The Bootstrap Fiber 30
  - Entry and Exit Trampolines 30
  - Telling the Sanitiser About the Switch 31
- The Fiber Object and Its Life 32
  - The Memory Contract 33
- The Scheduler 33

Suspend, Wake, and Staying Out of the I/O Business 34
Building 1:N So That M:N Is a Swap, Not a Rewrite 35
The Work-Stealing Deque 35
What the Language Gets, Later 36
Open Questions and Deferred Work 36

Preface

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

The Journey to Serene

In 2018, I made a series of attempts to contribute to a certain programming language, proposing improvements that, I believed, would benefit the community. Regrettably, none of them succeeded—not because of technical merit, but rather because the governing body of that language was unconcerned with improving the language and was content with the current state of affairs. Though that interaction left a bitter taste, it also planted a seed. I resolved to reignite my passion for computer science and programming languages by pursuing an exhilarating endeavor to create my own language, with the unwavering encouragement of my lovely wife.

At that time, I was attracted to dynamically typed languages with highly dynamic runtimes, such as Ruby, Clojure, and Scheme, as a software engineer. It appeared logical to commence by examining their codebases, designs, and problem-solving strategies. However, despite my efforts to emulate a diligent scientist by adhering to evidence and logic, I was unable to identify compelling rationales for specific design decisions in my preferred programming languages. The more I investigated, the more I realized I needed to start from the beginning: by studying logic, mathematics, and the history of computer science.

Fast-forward to the present: after seven years of studying mathematics and logic, experimenting with different ideas, implementing imaginary languages in various host languages, and building countless experimental interpreters and compilers, I can finally say I know what I want. My vision has evolved dramatically from those early days, shaped by deep mathematical understanding rather than surface preferences.

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

What is Serene?

Serene represents the culmination of that seven-year journey from intuition to mathematical understanding. It is a programming language designed for the next generation of software systems where correctness is not optional, but essential. In an era where software failures cascade across interconnected systems, causing everything from financial losses to life-threatening situations, we need programming languages that help us build reliable software by construction, not as an afterthought.

Serene combines the mathematical rigour of dependent type theory with the practical needs of systems programming. It is simultaneously a programming language and a theorem prover, embodying the Curry–Howard correspondence, where programs are proofs and proofs are programs. This isn't just academic elegance—it's a practical foundation for building software that we can reason about with mathematical certainty.

The language emerged from a simple realization: the problems I encountered with existing languages weren't surface-level issues that could be resolved with better syntax or more features. They were fundamental design problems rooted in a lack of mathematical foundation. Phillip Wadler[2] articulates it succinctly: There are two categories of programming languages: those that were invented and those that were discovered. Serene starts from solid mathematical principles and builds upward, rather than accumulating features and hoping they work together.

Initially, I establish a mathematical system by defining the initial principles and axioms that regulate computation, and subsequently identify/discover the language that emerges naturally from those foundations. Similar to the manner in which mathematicians operate—they do not generate theorems at random, but rather discover what follows necessarily from their axioms.

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

- 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)

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

Why Serene Matters

The software industry faces a fundamental crisis of complexity. Our systems have grown beyond our ability to understand them fully, leading to subtle bugs, security vulnerabilities, and unpredictable failures. Traditional approaches to software quality—testing, code review, static analysis—are necessary but insufficient. They can find bugs, but they cannot prove their absence.

My journey through different programming paradigms taught me that this crisis isn't inevitable. It's a consequence of languages that prioritize convenience over correctness, that hide complexity instead of managing it, and that defer issues to runtime instead of solving them at compile time.

Serene takes a different approach: 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. This isn't achieved 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 its practical implementation considerations. It includes:

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

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

Each chapter builds upon previous concepts while maintaining mathematical rigour. The specification is intended to be precise enough for implementers while remaining accessible to language users who want to understand the theoretical foundations.

The Future of Programming

This specification represents more than just another programming language—it embodies a different philosophy about what programming languages should be. Through years of study and experimentation, I've become convinced that programming languages must evolve beyond their current limitations. Software has become too important to accept the current state of affairs where bugs, crashes, and security vulnerabilities are considered normal.

Serene represents one possible future: a world where program correctness is verified mechanically, where software failure becomes as rare as mathematical proof errors, and where the phrase "it compiles; therefore it works" actually means something.

This specification is both a description of that future and a roadmap for building it. It represents not just technical design decisions, but a commitment to the idea that we can do better—that the mathematical tools exist to build software we can trust completely, not because we've tested it extensively, but because we've proven it correct.

As Feynman said, "What I cannot create, I do not understand." This language specification is my attempt to understand programming languages by creating one that embodies the mathematical principles I've learned. I invite you to join me in this understanding, and in creating software that 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.

Correctness Over Performance

Serene prioritises correctness and robustness. It should facilitate writing correct programs and actively discourage incorrect ones. Performance optimisations must never compromise 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.

Mathematical Simplicity Over Surface Simplicity

The language maintains minimal complexity through a small set of orthogonal concepts, favouring deep mathematical simplicity over superficial convenience. Thus, Serene values designs with lower information entropy.

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

Small, Composable Rule Set

Serene is minimalistic, defined by a small, clearly interacting set of rules.

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

Proofs as Programs, Programs as Proofs

Serene embraces the Curry–Howard correspondence, blending programming with theorem proving with being a general purpose language as the goal.

[3] Principles and not guarantees

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

Explicit Over Implicit

Explicit notation is preferred to prevent ambiguity. Implicit notation is allowed only when unambiguously clear.

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

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.

Composition Over Inheritance

Serene encourages composition of smaller, reusable components over inheritance hierarchies.

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

Declarative Instead of Imperative

Serene promotes declarative programming, clearly expressing computational logic without explicit control flows.

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

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.

Local Reasoning

One should be able to reason about expressions locally.

Secure by Construction

Security is inherent in Serene through explicit constraints and clear encapsulation.

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

Human-Friendly Tooling and Documentation

Serene emphasises intuitive, comprehensive tooling and documentation.

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

Error Messages as Teaching Tools

Error messages educate users, clearly explaining the rationale behind issues.

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

Stability Through Simplicity

Serene achieves stability via simplicity, avoiding frequent core changes.

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

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.

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 pertains to the grammar and description of the implementation-agnostic **Serene language**.

Mathematical Foundation

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

We will dive into the mathematical aspects of Serene in the next chapter, but for now, we just need to know the bare minimum constructs that Serene provides in terms of a dependent type theory.

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

Serene Programs

A program is consist of an expression and an environment.

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.

Values

Expressions that are in their normal form are Values. And Values naturally, always evaluate to themselves.

Atoms[4]

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

Numbers

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

Numbers are values[5].

```
1 42        ; Integer
2 3.14      ; Float
3 -17       ; Negative integer
```
Serene

Strings

Strings are values.

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

Symbols

Symbols are not values. Every symbol is a data constructor of type `Symbol`. Symbols normalise to a value that they are bound to. In simpler terms, symbols are assigned a value within the environment in which they evaluate to. Similar to variables in other languages, but immutable.

```
1 x
2 Int
3 +
```
Serene

Built-in symbolic constants include:

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

Compound Forms

Vectors

Vectors are values.

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

Lists

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

[5] Thus, evaluate to themselves

```serene
(f x)
(f x y z)
```

Functions and Application

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

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

Core Built-in Forms

Type Annotation

```serene
  (: 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)
```

Both `:` and `the` are functions.

Function Type Constructor (->)

```serene
   (-> 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))
```

Lambda (λ)

Lambda is a function. And functions are values. λ itself, is a function that expects a **vector** of parameters as the first argument and an expression of the body, as the second argument.

```serene
  (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))
```

Universal Quantifier (∀)

Universal quantifier or a Π-type.

```serene
(∀ [(: x A)] B)
```

Existential Quantifier (∃)

Existential quantifier or a Σ-type.

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

Identity Type (≡)

```
(≡ 2 2)
```
Serene

Quoting and Laziness

Quotes are values.

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

Binding Constructs

def

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

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

let

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

match

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

Defining Data Types

deftype

```
(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)))))
```
Serene

Evaluation Order

Compile Time

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

Runtime

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

Type Universes

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

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])))
```

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.

Protocols (Preview)

```
(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)
```

Scope and Closures

Serene uses **lexical scoping**: variables are bound according to the structure of the code, not the call site. The scoping rules apply uniformly 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.

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

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

Totality and Recursion

Serene enforces totality for all functions: every function must terminate for all well-typed inputs. This commitment enables strong guarantees such as memory safety, predictable evaluation, and decidable type checking. Consequently, Serene prohibits general recursion and partial functions by design.

Structural Recursion (Required)

All recursive functions must be written in a structurally recursive style. That is, recursive calls must be made on arguments that are syntactically smaller. The compiler enforces this constraint statically.

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

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

Fuel-Based Recursion

Some computations are not structurally recursive but are still terminating. In these cases, Serene encourages an encoding style based on **fuel**—an explicit counter argument that bounds the recursion.

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

This technique models bounded retries, search loops, and backtracking, while preserving totality. Since the fuel argument strictly decreases, termination is evident and enforced.

Rejected Idea: Partial Functions
We considered supporting partial functions via a `Partial` type qualifier or metadata annotation. However, we rejected this approach 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. Users can use explicit `Maybe` or `Either` types to express failure or non-returning computations.

Rejected Idea: General Well-Founded Recursion
We also considered supporting well-founded recursion via user-defined measures (e.g., lexicographic orderings) or sized types. While this would allow definitions such as the Ackermann function, it was rejected 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 remains limited to structural recursion in its initial form. However, users may encode well-founded or non-structural behaviour using fuel arguments or coinductive techniques.

Example: Fuel-Based Ackermann Approximation
The Ackermann function is known to be total but not structurally recursive. In Serene, one must encode it using fuel:

```serene
(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))])])]))
```

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

Summary
Serene enforces totality through structural recursion. Computations that are not structurally recursive can be encoded using explicit constructs such as fuel. Rejected alternatives such as partial functions and general well-founded recursion are documented to clarify Serene's priorities: simplicity, predictability, and mathematical integrity.

This concludes the core language description. All features build upon this uniform, total, and type-safe foundation.

Part 2

Theory

Type System

This chapter is dedicated to the type theory behind the language. This chapter is purely about the type theoric concepts and not the implementation. We are trying to use a type theoric syntax as close as possible to the common litrature of the field and in particular the style of [1].

Serene's type checker is a bidirectional type checker heavily influenced by [2].

Contexts

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

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

Syntax

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)
	$\mid \text{let}\ [(:\ x\ A)\ s]\ t)$		(let binding)
	$\mid \mu_\ell$		(universe at level ℓ)
	$\mid ?\,\alpha$		(metavariable)

Constants

Constants unify literals and primitive types into one sort:

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

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

Semantic Values
Values produced by evaluation (NbE domain):

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

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, Sept. 2013, pp. 429–442. doi: 10.1145/2500365.2500582.

Part 3

lxsameer's Serene Compiler

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 intends to talk 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 honest 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 — that waiting is not a wall the whole process slams into, but a single thread of computation stepping aside so its neighbours can run. This chapter describes the substrate that makes stepping aside possible: *fibers*.

A fiber is a unit of execution that the runtime, rather than the operating system, schedules. 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. Many fibers live inside a single operating-system thread and take turns cooperatively. Nothing here is novel in the abstract — green threads, coroutines, and goroutines are cousins of the same idea — but the shape this idea 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 falls out of taking those constraints seriously, and where one of Serene's guiding principles does the deciding, I will name it as we go.

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 will only point at it here. What follows is the engine room: how a stack is conjured, 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.

Why a Fiber Needs Its Own Stack

To pause a computation and resume it later, you must preserve everything it was in the middle of doing. On a conventional machine, "everything it was in the middle of doing" 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 entire tower of frames standing. When it resumes, the processor has to find the tower exactly as it was left — same frame pointers, same locals, same return addresses — and simply carry on.

So a fiber *is*, in a precise sense, a call chain frozen in place. And a frozen call chain cannot share its memory with anyone. Fiber A's suspended frames occupy a definite span of addresses, say `[base_A .. sp_A]`; if a second fiber started laying down its own frames in the same span, it would write straight over A's saved state, and A could never wake correctly. The conclusion is unavoidable: each fiber must own a *distinct* region of memory to use as its stack. This is the single fact from which most of the chapter follows. It is also the fact that names the model.

We call this the *stackful* model, because each fiber carries a real machine stack. Its alternative is the *stackless* model, in which the compiler rewrites every function that might suspend into a heap-allocated state machine, so that suspension means returning from the function with its progress recorded in an object rather than freezing a live stack. Stackless coroutines are elegant and 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 propagates 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 identically, because suspending is a property of the *stack*, not of the *functions* that built it. It asks for zero cooperation from the compiler, and so it works for code the compiler never compiled. This is *composition over inheritance* at its most literal: the fiber composes with foreign code as it finds it, rather than demanding that foreign code be rebuilt in the fiber's image. That universality is exactly what a language with a JIT and an FFI needs, and it is why Serene is stackful.

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

Why the Stack Cannot Grow

The operating system gives a thread a stack that appears to grow on demand; one might hope a fiber's stack could do the same. It cannot, and understanding why is worth the detour, because the impossibility is what 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 across, and continue there. The fatal difficulty is that a stack is saturated with pointers *into itself*. Frame-base 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, precisely which words on the stack are pointers-into-the-stack and which are merely integers that happen to look like addresses. That knowledge is a *precise pointer map*, produced by the compiler and consumed by a precise garbage collector. Serene has no garbage collector and emits no stack maps, and in any case 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. The door is closed.

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 planted in every function's prologue to detect the boundary. This is the road Go walked and then deliberately abandoned, undone by the *hot split* problem. Imagine a tight loop whose calls happen to 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 storm of churn precisely

where performance matters most. Worse for us, the scheme demands that *every* function prologue cooperate, including the FFI and JIT code that, as we have just established, cooperates with nothing. The door is closed here too.

What remains, once both growth strategies are ruled out, 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 consistent with the universality 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 stark and specific: a fiber that genuinely outgrows its stack must *fault* rather than expand. Left untreated, that cost would be intolerable — a silent overflow into a neighbour's frames is exactly the class of memory corruption Serene exists to abolish. The next section shows how a single mechanism turns that intolerable cost into an acceptable one.

The Guard Page, and the Role of `mmap`

The hinge that makes a fixed stack acceptable is the *guard page*, and the guard page is the first of four reasons the runtime maps fiber stacks with `mmap` rather than carving them out of `malloc` or the block arena. The arena is the wrong tool for an independent reason: it is a bump allocator that frees whole chains of allocations at once, whereas fiber stacks are born and reclaimed one at a time. `mmap` gives us four properties the arena structurally cannot.

The layout of a fiber stack is worth picturing. 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.

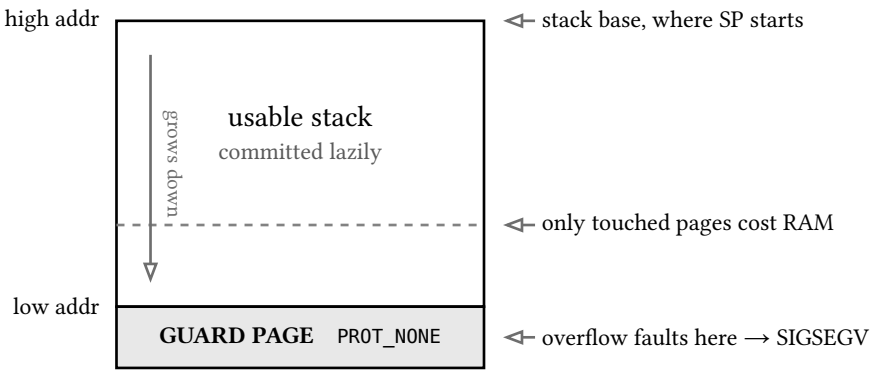

Figure 1: The layout of a fiber stack. Low addresses are at the bottom; the stack grows downward, toward the guard page that terminates it.

Guard page: overflow becomes a deterministic fault. The lowest page is mapped with no access permission at all (`PROT_NONE`). The instant the stack grows far enough to touch it — which is to say, the instant before it would have started 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 debuggable: a crash with a clear culprit, not a silent corruption that surfaces a thousand instructions later as a baffling wrong answer somewhere unrelated. This is the most important reason we use `mmap`, and it is *correctness over performance* stated almost verbatim: we accept a hard fault, which is a recoverable engineering problem, in exchange for ruling out silent memory corruption, which is not. It converts the one genuine danger of a fixed stack — overflow — from undefined behaviour into a clean, deterministic fault.

Lazy commit: "fixed but roomy" is nearly free. An anonymous `mmap` mapping hands out address space, not physical memory. Pages are demand-zero: the kernel commits real RAM only when a page is first touched. A fiber may therefore reserve a generous stack and, if it only ever nests a few frames deep, pay for just one or two pages of actual memory. This dissolves the obvious objection to fixed

stacks — that reserving room for the worst case wastes memory on the common case. It does not; reservation is cheap and only use is dear.

Page-aligned, page-granular regions. Setting a per-page protection is only possible on page boundaries, so the guard page presupposes a page-aligned mapping. The same granularity yields a clean half-open span `[base, base + size)` that we can hand verbatim to the sanitiser, a convenience we will lean on shortly.

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

The trade is made deliberately and with eyes open: overflow is rendered *visible and safe* rather than *automatically absorbed*. It is mitigated 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.

The Stack Provider Seam

`mmap` is a POSIX idiom, but the runtime is not a POSIX program; it must also run on Windows and, eventually, inside WebAssembly. So the act of obtaining a stack is hidden behind a narrow internal interface, and the platform detail never reaches `fiber.c`. This seam earns its keep twice over: 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 disturbing anything above it. It is the small, composable rule set applied to platform variation: the capability is universal, and only its expression differs from one platform to the next.

A stack is described by where it begins, how much of it is usable, and where its guard lies. Two functions allocate and free it:

```c
typedef struct srn_fiber_stack_t {
  void  *base;   // start of the usable region (high end; SP starts here)
  size_t size;   // usable bytes, excluding the guard page
  void  *guard;  // address of the guard page
} 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 subtleties worth recording, because they are the kind of detail that costs an afternoon if discovered the hard way. Anonymous mappings on POSIX are demand-

zero, so lazy commit comes for free without any special flag; `MAP_NORESERVE` is essentially a Linux knob for swap accounting, since macOS overcommits regardless and FreeBSD treats the flag as a no-op. `MAP_STACK` is the flag that genuinely matters, and it matters unevenly across the BSDs: OpenBSD *enforces* that the stack pointer always reside in memory mapped with `MAP_STACK`, so fiber stacks there must request it or the kernel will fault 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, for its part, offers native OS fibers through `CreateFiber` and `SwitchToFiber`; they are a legitimate option on that platform, but we keep the hand-rolled switch everywhere for uniformity, so that one mechanism is debugged rather than several. The first implementation builds the POSIX provider — `mmap` plus `mprotect`, with `MAP_STACK` where the platform honours it — and stubs the Windows and WebAssembly providers behind the same two-function seam.

That leaves the size itself. 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 in plain sight 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 an invisible mechanism to rescue it. What the default should be, when the caller expresses no preference, is a question we have left open. Lazy commit argues for erring on the roomy side — unused space is nearly free — but the right figure wants to be chosen against real workloads rather than guessed, so we name it here as a knob to be set later rather than a number to be enshrined now.

Handing Over the Processor: the Context Switch

With each fiber owning a stack, switching from one to another reduces to a surprisingly small act. The processor's state that must persist across the switch is not the whole machine — it is only the registers the calling convention obliges 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, without any further ceremony, 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 repays a careful reading. A call to `srn_fiber_swap` does not return in the ordinary sense; control departs 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 chosen to switch back into `from`. From `from`'s point of view the world simply resumes, perhaps much later, perhaps on a different theme entirely. This single routine is the pivot on which the whole library turns.

Because what must be saved is defined 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`.

Target	Mechanism and saved state
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 (later).

Two details in the saved set are easy to forget and expensive 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 neglected them would let one fiber silently alter another's arithmetic. The general-purpose sets are precisely 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 that the C compiler must see as preserving them. This routine is the *only* assembly in the entire library; every other line is plain C, which is how we like it, and keeping the machine-dependent surface this small is the small-composable-core principle paying off where it is hardest to honour.

The Bootstrap Fiber

A switch needs two parties: a fiber to switch *to* and a fiber to switch *from*. But 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 purpose is to be a legitimate `from` for the first switch, so that the asymmetry of "start" has the same shape as every switch thereafter — one rule, no special case for "the very first time".

Entry and Exit Trampolines

A freshly created fiber poses the mirror-image problem. It has never run, so its saved context cannot point at "the instruction after some earlier switch" — there is no earlier switch. Its first resume therefore cannot simply restore registers and return into frames, because no frames exist yet. The arrangement 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 with its own hands.

```c
// Runs on the new fiber's stack the first time it is resumed.
[[noreturn]] static void srn_fiber_trampoline(void) {
  srn_fiber_t *self = srn_fiber_current();
  self->result = self->entry(self->arg);  // generic until codegen emits fibers
  self->state  = SRN_FIBER_DONE;
  srn_fiber_return_to_scheduler();         // switches away; never comes back
}
```

The exit is the trampoline's responsibility too, and it is delicate for a symmetric reason. When the entry function returns, it returns *into the trampoline* — but the trampoline must not, in turn, return, 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 kept generic for the moment — it takes an opaque argument — so the fiber library can be built and tested long before there is an evaluator to run inside it. Once code generation begins producing fibers, the entry adopts the Serene ABI, the uniform signature every compiled Serene function presents:

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

so that a fiber is, in the end, nothing more exotic than 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.

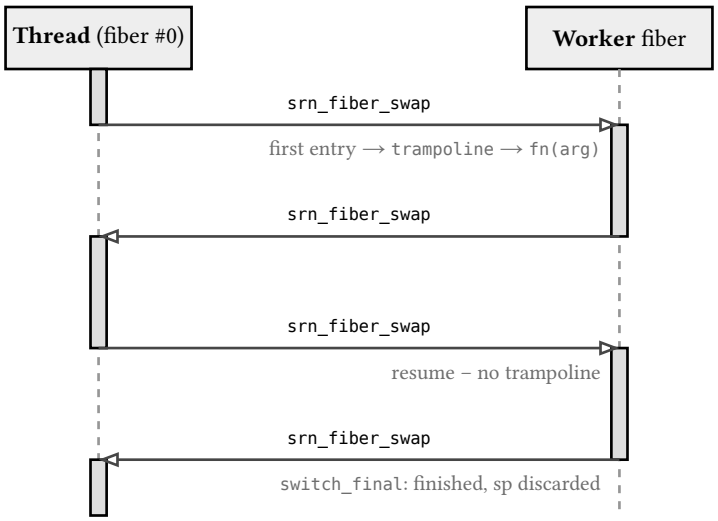

Figure 2: The life of a worker fiber. Every arrow is one `srn_fiber_swap`, which saves the leaving context's stack pointer and loads the target's; the shaded bar shows which side is running. The trampoline is reached only on the first entry — every later resume returns to the point just after the worker's own swap.

Telling the Sanitiser About the Switch

There is a subtle hazard in hand-rolling a stack switch, and Serene runs straight into it because the build enables `-fsanitize=address,undefined` by default. AddressSanitizer maintains 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 begins to hallucinate: false stack-overflow reports, spurious use-after-return complaints as a fiber resumes on a stack the sanitiser thinks was abandoned. The cure is to narrate the switch to the sanitiser explicitly, bracketing the swap with two calls:

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

The `start` call warns the sanitiser that execution is about to move to `to`'s stack, handing it the clean `[base, size)` span the provider seam took care to preserve, and stashing the outgoing fiber's shadow bookkeeping in `from->fake_stack` for when it resumes. The matching `finish` call, executed once `from` is scheduled again, restores that bookkeeping. The two calls must straddle the swap precisely — `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 exposes an analogous family of calls — __tsan_create_fiber, __tsan_switch_to_fiber, __tsan_destroy_fiber — which will matter once real threads enter the picture under a multi-threaded scheduler. That the switch must announce itself to the tooling is not an afterthought bolted on; it is the same correctness-over-performance commitment followed through to the build configuration, where the cheap thing would be to silence the sanitiser and the correct thing is to teach it.

The Fiber Object and Its Life

Everything above converges on one structure. A fiber is born, 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 says something plain about the fiber's fortunes.

From	To	Trigger
NEW	RUNNING	first resume
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 simply cannot happen — a fiber cannot be readied while it runs, cannot run while it is parked, cannot leave DONE. The distinction between READY and SUSPENDED is the hinge of the whole design: a READY fiber will run again of its own accord, whereas a SUSPENDED fiber will run again only when some outside party calls ready on it. That difference is what keeps the scheduler ignorant of *why* a fiber waits, a point we return to below.

The structure that carries this is mostly an assembly of parts the preceding sections have already justified, which is a good sign that the design hangs together.

```c
typedef struct srn_fiber_t {
    srn_fiber_ctx_t     ctx;        // saved SP + callee-saved registers
    srn_fiber_stack_t   stack;      // mapped stack + guard page
    srn_fiber_state_t   state;
```

```
    srn_context_t      *mem;        // GIVEN, not owned (see below)
    srn_fiber_entry_t   entry;      // entry function
    void               *arg;        // argument handed to the entry function
    srn_value_t        *result;     // set on DONE
    srn_error_t        *error;      // set on DONE if the fiber errored
    void               *fake_stack; // AddressSanitizer bookkeeping
    struct srn_fiber_t *link;       // intrusive run-queue / wait-queue link
    const char         *name;       // optional, for debugging
} srn_fiber_t;
```

The `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` and `error` are the two ways an entry can end. The `link` field deserves a word: a fiber is threaded directly onto whatever queue it belongs to — the run queue, a wait queue — through this single 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.

The Memory Contract

A fiber does not create or own a memory context. It is *given* one — typically the `srn_context_t` already in play 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 the behaviour you want for a cluster of fibers cooperating on one task.
- A fiber's `result` is allocated in that shared context, so it survives as naturally and for 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 this imposes on the caller is simple to state and the caller's responsibility to honour: 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 buys its simplicity 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 concurrently. 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 single 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.

The Scheduler

A scheduler is what gives fibers their turns, and Serene's is deliberately spare. An `srn_scheduler_t` owns a queue of `READY` fibers and the bootstrap fiber, and that is nearly the whole of it. There is *no event loop* inside it. The single verb `srn_sched_run` drains the run queue — resuming each ready fiber in turn, letting it run until it yields, parks, or finishes — and returns the moment no fiber is ready. What happens after the queue empties is not the scheduler's concern; the embedder, or one day an I/O reactor, decides when to pump it again. This restraint is the point. The scheduler arbitrates turns;

it does not own the program's outer loop and does not presume to know what the program is for. A scheduler that does exactly one thing is a scheduler that can be reasoned about completely.

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

The central 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 a composition of these, and the composition is where the design's restraint pays off. Before it parks, a fiber hands its own handle to whatever party is in a position to 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 supplies the mechanism of waiting and waking and nothing more; it has no opinion about what is being waited *on*. This is the cleanest expression in the whole design of a small, composable rule set: rather than enumerate the reasons a fiber might wait and bake each 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 library at all — an earlier design leaned on `libuv`, which has since been removed; the scheduler is, by intent, agnostic to the reason any fiber waits.

The whole surface is small enough to read in one breath.

```c
srn_scheduler_t *srn_sched_make(srn_engine_t *engine);
void             srn_sched_run(srn_scheduler_t *sched);

srn_fiber_t *srn_fiber_make(srn_scheduler_t *sched, srn_context_t *mem,
                            srn_fiber_entry_t entry, void *arg,
                            size_t stack_size);
void         srn_fiber_yield(srn_scheduler_t *sched);
void         srn_fiber_suspend(srn_scheduler_t *sched);
void         srn_fiber_ready(srn_fiber_t *fiber);
srn_value_t *srn_fiber_join(srn_scheduler_t *sched, srn_fiber_t *fiber);
srn_fiber_t *srn_fiber_current(void);
```

`srn_fiber_make` carries the two contracts of the chapter in its signature: it takes the borrowed `mem` context the new fiber will allocate into, and the explicit `stack_size` the caller chooses. `srn_fiber_join` is the one verb not yet met, and it is no new mechanism but a composition of the two above: the joining fiber suspends itself until the joinee reaches `DONE`, and the joinee, as the last thing it does before finishing, calls `ready` on its waiter and hands back the `result`. `srn_fiber_current` reports the fiber executing on the calling thread — a thread-local read today, and the foundation of the multi-threaded story below.

Building 1:N So That M:N Is a Swap, Not a Rewrite

The scheduler described above runs every fiber on one operating-system thread: many fibers, one thread, the arrangement usually written *1:N*. That is where Serene begins, because a single thread needs no locks and is far easier to get right. But the destination is an *M:N* work-stealing scheduler, with many fibers spread across several threads, each thread stealing work when its own queue runs dry. The discipline we adopt from the first day is that the single-threaded design must be a strict *subset* of the multi-threaded one — so that anything which would have to differ under M:N already sits behind a function call today, and the eventual move is an implementation change rather than a rewrite of everything that touches scheduling.

Concern	1:N today	M:N later
Current fiber	thread-local, read via `srn_fiber_current()`	unchanged; one slot per worker
Run queue	one queue behind `enqueue_ready` / `next_ready`	per-worker deques + work stealing
Stack source	the provider seam	per-OS-thread stack caches + shared pool
Wake (`ready`)	enqueue to the queue	enqueue to a worker; cross-thread wakeups need a signal
Locking	none needed (single thread)	add behind the same queue functions

The keystone of this discipline is a small abstinence: a fiber stores no pointer to "the current operating-system thread." It does not belong to a worker. Which worker runs it is decided afresh each time it is scheduled, never baked into the fiber. That one omission is what lets a fiber created on one thread be stolen and run on another without rewriting anything about the fiber itself. The shared-context contention noted in the memory contract is the single place where the M:N future reaches back and touches a choice we make now — the block's spinlock makes it correct today, and M:N is when its cost must be measured; everything else genuinely is a later substitution behind a stable seam.

The Work-Stealing Deque

The M:N arrangement is realised with the classic remedy for spreading fibers across threads without a central bottleneck: 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.

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.

Operation	What it does
`pop` (owner, `bottom`)	Take the newest fiber from the bottom. Lock-free and uncontended, save for the last element.
`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 attempt the CAS, and exactly one succeeds. The counters are signed on purpose — the owner's `pop` momentarily 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, and not only x86, whose strong model would mask 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 deque that is full, 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.

What the Language Gets, Later

It is worth saying plainly where all this is headed, if only to keep the C-level machinery from looking like an end in itself. The asynchronous surface of Serene — the `IO` type, `Future`, `async-race`, `async-parallel`, the whole vocabulary the *Effects and I/O* chapter develops — maps onto exactly the primitives built here. An asynchronous operation spawns or suspends a fiber and resolves when its work completes. A `Future` is a fiber's eventual result; racing two computations is suspending on whichever fiber finishes first; running them in parallel is spawning both and joining both. That such rich surface concurrency should reduce to four C verbs — spawn, suspend, ready, and join — is not a coincidence; it is the small-composable-core principle reaching its intended payoff. None of that surface exists yet, and its shape is not settled in this chapter; this is a direction, not a commitment to syntax. The point is only that the language's concurrency, when it arrives, will be a thin layer over these four verbs, and not a second mechanism bolted alongside this one.

Open Questions and Deferred Work

A few decisions are honestly still open, and it is better to name them than to paper over them.

The *default stack size*, when a caller specifies none, is unfixed. Lazy commit argues for generosity, since a roomy reservation costs address space rather than memory, but the figure should be calibrated against real workloads once the evaluator is producing them, rather than guessed in advance.

Where stacks live in the long run is likewise open. The fiber library `mmap`s them today, behind the provider seam; whether they eventually migrate into the memory manager as a stack-pool policy — pooling and reusing mappings rather than mapping and unmapping each time, and feeding the per-thread caches that M:N wants — is a question the seam was built precisely to leave cheap to answer either way.

Cancellation of a suspended fiber, together with the running of whatever cleanup it owes, is deferred and absent from the first cut. C has no stack unwinding to lean on, so a destructor-style "throw through the frozen frames" approach is not available; the likely shape is a cooperative cancel flag that a fiber checks at its suspend points, followed by an ordinary early return that lets the shared context reclaim everything in the normal way. It is a known gap, named here so it is not mistaken for an oversight.

What is *not* open is the spine of the design, and it is worth gathering in one place.

Aspect	**Choice**
Execution model	Stackful, cooperative
Stack size	Fixed; no growth
Overflow	Guard page; deterministic fault
Stack memory	`mmap` + `mprotect` (POSIX), behind a provider seam
Scheduler	Single-threaded cooperative (1:N) first; M:N is the goal
I/O	None in the library; reactor-agnostic suspend and wake
Sanitiser	`__sanitizer_*_switch_fiber` around every switch
Fiber memory	Given a `srn_context_t`; not owned
Language surface	Out of scope; a later layer over spawn / suspend / ready / join

The thread that runs through the open questions is the same one that runs through the firm choices. The hard commitments — stackful execution, fixed and guarded stacks, a reason-agnostic scheduler, a borrowed memory context — are made now and made firmly, because they follow from principles that do not bend: they are what the constraints leave standing once "no GC, no stack maps, must host code the compiler never saw" is taken as given. The soft questions — a default size, a stack pool, a cancellation protocol — are held open behind seams, because the principles do not yet dictate an answer and an honest design says so rather than guessing. That is the shape of building from foundations: decide what the principles decide, and leave the rest visibly, deliberately open. The fiber library is the first place in the runtime where Serene's commitment to correctness over performance has to be cashed out in raw memory and bare registers, and it is, I think, a fair test of whether the commitment was sincere.