# **The Digital Organism: A Treatise on the Philosophy and Architecture of Legacy-Free Unikernels in #[no\_std] Rust**

**Abstract**

This treatise posits a new architectural paradigm for high-assurance, high-performance computing: the Unikernel as a **Governed Symbiotic Network**. We reject the legacy of monolithic, general-purpose operating systems, which impose unacceptable overheads in the form of context switching, privilege transitions, and non-deterministic scheduling, thereby creating performance-destroying jitter.1 Instead, we propose a "digital organism"—a self-contained, single-address-space system built entirely in

#[no\_std] Rust that runs directly on bare metal. Its architecture is inspired by the decentralized, resilient, and resource-efficient properties of mycorrhizal networks in fungal ecology.2 This system is composed of autonomous tasks that communicate and trade resources in a localized, NUMA-aware market, enabling emergent, high-performance behavior. This symbiotic network is not anarchic; it is governed by a "Social Contract," a concept borrowed from political philosophy, that establishes fundamental rights for tasks and mandates formal verification for the use of

unsafe code. The entire system, from kernel primitives to high-level application logic, is unified by Parseltongue, a zero-cost, declarative Domain-Specific Language. This treatise presents the philosophy, architecture, and profound implications of this approach, arguing that it represents the logical endpoint for the next generation of secure, predictable, and ultra-low-latency systems.

## **The Post-Legacy Imperative: Towards Digital Autonomy**

The history of computing is a history of abstraction. We have built layers upon layers—hardware, firmware, kernel, user space—to manage complexity and provide generality. This model, embodied by the modern General-Purpose Operating System (GPOS) like Linux, has been fantastically successful. However, for a critical class of performance-sensitive applications, this success has come at a crippling cost: **unpredictable latency**, or jitter.1

A GPOS is fundamentally designed for throughput and fairness, not determinism. Its core mechanisms are antithetical to predictable performance 1:

* **The Context Switch Tax:** A pre-emptive scheduler constantly interrupts running tasks to give others a turn. Each context switch is a costly operation, requiring the saving and restoring of the entire processor state. On a modern x86 CPU, this can consume thousands of cycles and involves flushing critical processor resources like the Translation Lookaside Buffer (TLB), leading to a cascade of expensive cache misses for the newly scheduled task.4 This process introduces a variable, unpredictable delay measured in microseconds.5
* **The Tyranny of Interrupts:** The CPU is constantly assailed by hardware interrupts (IRQs) from network cards, disk drives, and timers. Each interrupt forces an involuntary context switch into an Interrupt Service Routine (ISR), stealing cycles from the critical application at unpredictable moments.1
* **The User/Kernel Chasm:** Every system call—for file I/O, network communication, or memory allocation—requires a transition from unprivileged user mode (Ring 3) to privileged kernel mode (Ring 0). This is not a simple function call but a complex, hardware-mediated operation that adds significant overhead.
* **Cache Pollution:** When the scheduler switches from Task A to Task B, Task B begins populating the CPU caches (L1, L2, L3) with its own data. When Task A is eventually rescheduled, it finds its previously "hot" cache is now "cold," filled with Task B's data. It must then suffer a storm of slow main-memory accesses to repopulate the cache, a phenomenon known as cache pollution, which is a major indirect cost of context switching.7

The traditional solution, the Real-Time Operating System (RTOS), addresses these issues but often at the cost of the rich ecosystem of a GPOS.1 Kernel bypass techniques like DPDK and AF\_XDP attack the I/O overhead but still leave the application vulnerable to the scheduler and other sources of OS jitter.1

The architectural imperative is clear: to achieve true determinism, we must abandon the GPOS model entirely. We must create a system that is fully responsible for its own existence, running on bare metal, in a single address space, with no user/kernel distinction and no legacy abstractions. This is the Unikernel: a self-contained digital organism.10

## **An Architecture of Symbiosis: The Mycorrhizal Kernel**

Our proposed architecture, the **Mycorrhizal Kernel**, is a radical departure from the hierarchical, centrally-controlled design of traditional operating systems. It is a decentralized, adaptive system where global order emerges from local, symbiotic interactions. The conventional approach of running a unikernel on a hypervisor reintroduces the very overheads of context switching (via VM exits) and host OS scheduling interference that this architecture seeks to eliminate.10 Our model runs directly on the hardware, treating the system's components not as subjects of a central kernel but as autonomous agents in a resource-sharing network, much like plants connected by fungi.2

The following table contextualizes this approach by contrasting it with other architectural philosophies, demonstrating why a synthesis of models is necessary to achieve the desired balance of performance, resilience, and governability.

**Table 1: A Comparative Analysis of Unikernel Architectures**

| Feature | Conventional Unikernel | Mycorrhizal Kernel | Leviathan Kernel | Brutalist Kernel |
| --- | --- | --- | --- | --- |
| **Scheduling Philosophy** | Pre-emptive (Host OS) | Decentralized Resource Market | Sovereign Contract (Pre-emptive/Cooperative) | Exposed FIFO Queue |
| **Memory Management** | Centralized Heap (LibOS) | NUMA-local Nutrient Pools | Centrally Granted Allocations | Direct Physical Addressing |
| **Inter-Task Communication** | N/A (Single Process) or Host Networking | Localized SPSC "Hyphae" Network | Sovereign-Mediated IPC | Exposed Shared Memory |
| **Fault Tolerance Model** | Host-level Restart (VM) | Emergent Network Healing | Sovereign-Enforced Termination | Caller-Managed Faults |
| **API Philosophy** | Abstracted (LibOS API) | Economic/Signaling Metaphors | Constitutional/Rights-Based | Functionally Honest / Unforgiving |

### **The Mycelial Substrate: A NUMA-Aware, Decentralized Memory System**

Modern multi-socket servers are NUMA (Non-Uniform Memory Access) systems. A processor can access its own local memory much faster than memory attached to a different processor socket.15 A traditional, unified heap allocator is oblivious to this physical reality, leading to performance degradation when a thread on one node frequently accesses memory on another.

The Mycorrhizal Kernel embraces this reality. Memory is not a single global resource but a collection of localized "nutrient pools," one for each NUMA node.

* **Memory Management:** Each NUMA node has its own memory manager, responsible for the physical memory pages directly attached to it. When a task is created, it is pinned to a specific core, and its initial memory allocations are satisfied from that core's local NUMA node, guaranteeing the lowest possible access latency.
* **Resource Trading:** A task can request memory from a remote NUMA node, but this is treated as an explicit, high-cost "trade." This is not a transparent allocation but an IPC request to the remote memory manager. The cost of this trade is not just the higher hardware latency but also a "tariff" imposed by the kernel, discouraging non-local access. This incentivizes developers to design their applications with data locality in mind. For tasks that must share data across NUMA boundaries, the kernel provides explicit mechanisms for creating shared memory regions, but the performance implications are made brutally clear by the API.

### **The Hyphal Network: High-Performance Inter-Task Communication**

In our single-address-space Unikernel, IPC is not an afterthought; it is the primary organizing principle. The "hyphae" that connect our tasks are a dense mesh of high-performance, lock-free SPSC (Single-Producer, Single-Consumer) ring buffers implemented in shared memory.

* **Mechanism:** An SPSC ring buffer is a wait-free data structure. The producer and consumer manipulate separate head/tail pointers using atomic operations, eliminating the need for locks, mutexes, or any other blocking synchronization primitive that could introduce jitter.18 Benchmarks show that shared memory IPC is orders of magnitude faster than any other method, with latencies in the sub-microsecond or even nanosecond range.20
* **Topology:** Tasks are not connected arbitrarily. The kernel encourages a NUMA-aware topology. Communication between two tasks on the same core or within the same NUMA node is cheap. Communication across NUMA nodes is possible but incurs a higher scheduling cost, reflecting the physical reality of the hardware interconnect. This creates an economic incentive for tasks to form localized communication clusters.

### **The Rhizomorphic Scheduler: A Market for CPU Time**

The most radical element of the Mycorrhizal Kernel is its rejection of a central, omniscient scheduler. Instead, scheduling is an emergent property of a decentralized resource market.

* **Cooperative Foundation:** The baseline scheduling model is cooperative (Lockean). A task runs until it explicitly yields, typically at an .await point. This is the most efficient model, as it minimizes involuntary context switches.22
* **The Auction for Cycles:** A purely cooperative model is vulnerable to misbehaving or greedy tasks. To solve this, we introduce an auction-based mechanism for allocating CPU time.23
  1. **Time as a Resource:** The kernel divides time into discrete, fine-grained epochs (e.g., 100 microseconds).
  2. **Bidding:** At the beginning of each epoch, tasks that are ready to run submit a "bid" for the next time slice. This bid is not monetary but a representation of the task's urgency, which can be derived from its deadline, priority, or other metrics.
  3. **The Auctioneer:** A lightweight, per-core "auctioneer" runs at the end of each epoch. It uses a computationally efficient auction algorithm, such as a Vickrey (second-price) auction, to determine the winner.25 The highest bidder wins the next time slice. A Vickrey auction is chosen because it incentivizes tasks to bid their true valuation (urgency), leading to a more efficient allocation.26
  4. **Decentralization:** This auction happens independently on each core, operating only on the tasks pinned to that core. This eliminates global scheduler locks and ensures the scheduling mechanism itself is scalable and NUMA-local.

### **Symbiotic Fault Tolerance**

Resilience in the Mycorrhizal Kernel is not achieved through a top-down supervisor but emerges from the network itself.

* **Local Fault Detection:** If a task panics, the kernel catches the panic. The task is terminated, and its resources (memory, open channels) are marked as "decomposed."
* **Network Healing:** Tasks communicating with the failed task will discover the failure when their channel operations fail. This is a "distress signal".3 They can then react to the failure—for example, by rerouting requests to a replica or entering a degraded mode of operation.
* **Resource Recycling:** The memory and other resources held by the failed task are returned to the local NUMA node's resource pool, becoming "nutrients" available for new tasks. The system heals locally without requiring a global state reset.

## **The State of Nature: A Social Contract for unsafe Code**

The Mycorrhizal model describes the *mechanism* of the system, but the Leviathan model provides its *constitution*. The unsafe keyword in Rust is a necessary tool for systems programming, allowing direct interaction with hardware and the implementation of data structures like our SPSC queues.28 However, it represents an entry into a "state of nature" where the compiler's guarantees are suspended. In our high-assurance Unikernel, this cannot be an informal, ad-hoc process.

We propose a **formally-enforced Social Contract for unsafe code**:

* **The Contract:** To use an unsafe block, a task must provide a machine-checkable proof that its usage does not violate the system's core invariants (e.g., memory integrity, pointer validity, temporal safety). This moves the burden of proof from the programmer's comments to a formal artifact.
* **Implementation:** We will leverage a verification-oriented subset of Rust and a tool like Verus.30 The kernel will expose its core data structures and invariants as formal specifications. Any library or application code containing an  
  unsafe block must be accompanied by a Verus proof that it correctly interacts with these specifications.
* **Enforcement:** The build system for the Unikernel will integrate the Verus verifier. Code that contains an unsafe block without a corresponding valid proof will fail to compile. This is the ultimate enforcement of the social contract: an unproven assertion of safety is a treasonous act, and the "citizen" (the code) is rejected by the "sovereign" (the compiler) before it can harm the "state."

This approach transforms unsafe from a dangerous escape hatch into a formally-governed, auditable, and high-assurance feature, creating a system where even the lowest-level code is subject to the rule of law. The following table makes this contract explicit, mapping the fundamental safety guarantees provided by the kernel to the proof obligations required of any code that wishes to temporarily suspend them.

**Table 2: The Social Contract of unsafe Code**

| Citizen's Right (Guaranteed by Kernel) | Citizen's Obligation (When Entering unsafe) | Enforcement Mechanism |
| --- | --- | --- |
| **Right to Memory Integrity** (Spatial Safety) | Must prove all raw pointer dereferences are valid and within allocated bounds. | Hardware Page Tables / IOMMU |
| **Right to Temporal Safety** (No Use-After-Free) | Must prove lifetimes of raw pointers do not exceed the lifetime of the underlying allocation. | Formal Verification (Verus lifetime proofs) |
| **Right to Exclusive Mutable Access** (No Data Races) | Must prove that no mutable aliases exist for a given memory location at any point in time. | Formal Verification (Verus separation logic proofs) |
| **Right to a Fair Hearing** (Freedom from Starvation) | Must not enter an unbounded loop without a yield point. | Decentralized Watchdog Timer / Bidding System |
| **Right to Due Process** (No Arbitrary Termination) | Must uphold all function contracts and invariants when calling unsafe functions. | Formal Verification (Verus pre/post-condition proofs) |

## **The Raw Concrete: A Brutalist API Philosophy**

The Unikernel's Application Programming Interface (API) will not hide the reality of the hardware. Drawing inspiration from Brutalist architecture, the API will be **functionally honest**.32

* **No Hidden Costs:** Abstractions that hide performance costs are forbidden. If a function call might involve a cross-NUMA memory access, its signature will reflect that, perhaps by requiring an explicit RemoteMemoryToken. The cost of operations will be visible in the API's structure.
* **Exposed Primitives:** The API will provide direct, if unsafe, access to hardware primitives. For example, instead of a high-level networking API, the system will provide a safe wrapper around the raw NIC descriptor rings, allowing the application to directly manipulate them if it needs to, after accepting the corresponding unsafe social contract.
* **Honest Naming:** Functions will be named to reflect their true behavior. A function that polls a device in a busy-loop will be named poll\_device\_busy\_wait(), not get\_data(). This forces the programmer to confront the performance implications of their choices.
* **Unforgiving but Predictable:** The API will be unforgiving of misuse but its behavior will be simple and predictable. It will not attempt to guess the user's intent. This reduces complexity and makes it easier to reason about the system's behavior, aligning with the Brutalist emphasis on raw, unadorned functionality.34

## **Parseltongue: A Unifying Vernacular for a Digital Organism**

The complexity of orchestrating this symbiotic network, defining the economic behavior of tasks, and writing formally-verified unsafe code would be overwhelming using raw Rust. The **Parseltongue DSL** is the essential unifying abstraction that makes this architecture tractable.

Parseltongue is not an interpreted language; it is a set of powerful procedural macros that provide a high-level, declarative syntax for describing the entire system.1 It is a zero-cost abstraction, meaning it compiles down to highly optimized

#[no\_std] Rust with no runtime overhead.37

* **Declaring the Organism:** Parseltongue will be used to define:
  + **Tasks (organism):** Declare a new task, its pinned core, its local memory allocation, and its "economic profile" (e.g., its bidding strategy for CPU time).
  + **Communication Channels (hypha):** Declaratively establish an SPSC channel between two organisms, with the macro generating the underlying shared memory and initialization code.
  + **Hardware Interfaces (root):** Define the interface to a piece of hardware passed through via VFIO, with the macro generating the unsafe boilerplate for register access and DMA mapping.
  + **UI Components (venom):** As described in the user query, Parseltongue will provide a declarative, React-inspired syntax for the immediate-mode browser engine, compiling UI descriptions directly into optimized rendering commands.1
* **Example (Conceptual Parseltongue Syntax):**  
  Rust  
  // This is Parseltongue code, processed by a procedural macro  
  parseltongue! {  
   // Define a network packet processor organism  
   organism PacketProcessor on core 3 with memory 16MB {  
   // Define its resource bidding strategy  
   bidding\_strategy: high\_priority(deadline: 10us);  
    
   // Establish a communication channel to the logger  
   hypha log\_channel -> Logger;  
    
   // Root into the hardware NIC  
   root nic0: NetworkCard @ pci("00:03.0");  
    
   // Main loop logic  
   on\_packet(packet: nic0.rx()) {  
   let processed = process(packet);  
   log\_channel.send(f!("Processed packet: {}", processed.id));  
   }  
   }  
    
   organism Logger on core 7 with memory 2MB {  
   bidding\_strategy: background();  
   //... logger implementation  
   }  
  }

The parseltongue! macro would parse this declaration and generate all the necessary Rust code: the task struct, the main loop, the code to initialize the SPSC channel, and the unsafe VFIO boilerplate to map the NIC's registers. This allows the developer to reason at the level of the organism and its interactions, while the macro handles the complex and error-prone low-level implementation. The following table illustrates this principle of zero-cost abstraction.

**Table 3: Parseltongue DSL to Architectural Primitive Mapping**

| Parseltongue Declaration | Generated Rust Code (Conceptual) | Architectural Principle |
| --- | --- | --- |
| organism PacketProcessor on core 3 | struct PacketProcessor {... }; fn main\_loop() {... } kernel::spawn(3, main\_loop); | Task Abstraction & Core Pinning |
| hypha log\_channel -> Logger; | let (p, c) = kernel::ipc::channel::<LogMsg>(); let log\_channel = Producer::new(p); | Zero-Cost, Type-Safe IPC |
| root nic0: NetworkCard @ pci("00:03.0"); | unsafe { let nic0 = vfio::map\_pci\_device("00:03.0"); } let nic0 = safe\_wrapper::NetworkCard::new(nic0); | Hardware Interface Generation & Encapsulation |
| on\_packet(packet: nic0.rx()) {... } | loop { if let Some(packet) = nic0.rx\_burst() {... } else { yield().await; } } | Event-Driven Logic Generation |

## **Implications and Future Horizons**

The architecture of the Governed Symbiotic Network has profound implications:

* **Performance:** By eliminating the GPOS and designing the entire system around the physical reality of the hardware (NUMA, caches), we can achieve deterministic, predictable performance with latencies in the single-digit microsecond or even nanosecond range, as targeted by the Aether project.1
* **Security:** The combination of a minimal attack surface (Unikernel principle), hardware-enforced isolation (IOMMU), and the memory safety of Rust creates a high-assurance environment. Elevating unsafe to a formally-verified contract further strengthens these guarantees, approaching the level of assurance seen in systems like seL4.39
* **Resilience:** The decentralized, networked architecture provides inherent resilience to faults. The system is designed to heal locally rather than fail globally, a property absent in traditional monolithic kernels.
* **Developer Productivity:** The Parseltongue DSL provides a high-level, declarative interface that abstracts the immense complexity of the underlying system, allowing developers to build highly performant and correct applications without needing to be world-class experts in kernel development or formal methods.

This architecture is not an endpoint but a foundation. Future work will explore extending the model to **Composable Secure Partitions**, where the formal verification of the unsafe social contract can be used to create attested, verifiable communication channels between the Unikernel and hardware-enforced Trusted Execution Environments (TEEs) like Intel SGX or AMD SEV.1 This would unify the worlds of high-performance, real-time computing and high-assurance confidential computing, creating a truly next-generation platform for critical systems.

#### Works cited

1. RustHallows Evolving RustWeb Runtime Ecosystem.docx
2. Mycorrhizal network - Wikipedia, accessed on August 13, 2025, <https://en.wikipedia.org/wiki/Mycorrhizal_network>
3. Underground Networking: The Amazing Connections Beneath Your Feet - National Forest Foundation, accessed on August 13, 2025, <https://www.nationalforests.org/blog/underground-mycorrhizal-network>
4. Context switch - Wikipedia, accessed on August 13, 2025, <https://en.wikipedia.org/wiki/Context_switch>
5. What is the overhead of a context-switch? - Stack Overflow, accessed on August 13, 2025, <https://stackoverflow.com/questions/21887797/what-is-the-overhead-of-a-context-switch>
6. A Case Against (Most) Context Switches - acm sigops, accessed on August 13, 2025, <https://sigops.org/s/conferences/hotos/2021/papers/hotos21-s01-humphries.pdf>
7. WRL Technical Note TN-16 The Effect of Context Switches on Cache Performance - Bitsavers.org, accessed on August 13, 2025, <https://bitsavers.org/pdf/dec/tech_reports/WRL-TN-16.pdf>
8. Reducing Cache-Associated Context-Switch Performance Penalty Using Elastic Time Slicing - FAST Lab, accessed on August 13, 2025, <https://memlab.ece.gatech.edu/papers/VMTJ_2015_1.pdf>
9. What cpu context switch and cache pollution are and how do they impact performance, accessed on August 13, 2025, <https://andrewpakhomov.com/posts/what-are-context-switches-memory-cache-pollution-and-what-is-their-impact-on-the-performance/>
10. A Rust-Based, modular Unikernel for MicroVMs - TIB AV-Portal, accessed on August 13, 2025, <https://av.tib.eu/media/61644>
11. Exploring Rust for Unikernel Development, accessed on August 13, 2025, <https://plos-workshop.org/2019/preprint/plos19-lankes.pdf>
12. Introducing Unikraft - Lightweight Virtualization Using Unikernels - KubeSimplify blog, accessed on August 13, 2025, <https://blog.kubesimplify.com/introducing-unikraft-lightweight-virtualization-using-unikernels>
13. The Cost of Virtualization Exits - Yasser Shalabi, accessed on August 13, 2025, <http://yshalabi.github.io/VMExits/>
14. Ventures into Hyper-V - Fuzzing hypercalls | WithSecure™ Labs, accessed on August 13, 2025, <https://labs.withsecure.com/publications/ventures-into-hyper-v-part-1-fuzzing-hypercalls>
15. Non-uniform memory access - Wikipedia, accessed on August 13, 2025, <https://en.wikipedia.org/wiki/Non-uniform_memory_access>
16. Optimizing Applications for NUMA - Intel, accessed on August 13, 2025, <https://www.intel.com/content/dam/develop/external/us/en/documents/3-5-memmgt-optimizing-applications-for-numa-184398.pdf>
17. STARNUMA: Mitigating NUMA Challenges with Memory Pooling, accessed on August 13, 2025, <https://faculty.cc.gatech.edu/~adaglis3/files/papers/starnuma_micro24.pdf>
18. Circular lock-free buffer - c++ - Stack Overflow, accessed on August 13, 2025, <https://stackoverflow.com/questions/871234/circular-lock-free-buffer>
19. Top performing SPSC queue - faster than moodycamel and rigtorp : r/cpp - Reddit, accessed on August 13, 2025, <https://www.reddit.com/r/cpp/comments/1g6xfrd/top_performing_spsc_queue_faster_than_moodycamel/>
20. IPC speed and compare - Stack Overflow, accessed on August 13, 2025, <https://stackoverflow.com/questions/2854004/ipc-speed-and-compare>
21. goldsborough/ipc-bench: :racehorse: Benchmarks for Inter-Process-Communication Techniques - GitHub, accessed on August 13, 2025, <https://github.com/goldsborough/ipc-bench>
22. Cooperative vs. Preemptive: a quest to maximize concurrency power | by Bobby Priam | Traveloka Engineering Blog | Medium, accessed on August 13, 2025, <https://medium.com/traveloka-engineering/cooperative-vs-preemptive-a-quest-to-maximize-concurrency-power-3b10c5a920fe>
23. Mastering Auction Algorithms - Number Analytics, accessed on August 13, 2025, <https://www.numberanalytics.com/blog/mastering-auction-algorithms>
24. Auction Algorithms - MIT, accessed on August 13, 2025, <https://web.mit.edu/dimitrib/www/Auction_Encycl.pdf>
25. Mastering Vickrey Auctions in Algorithmic Game Theory - Number Analytics, accessed on August 13, 2025, <https://www.numberanalytics.com/blog/mastering-vickrey-auctions-algorithmic-game-theory>
26. Vickrey–Clarke–Groves auction - Wikipedia, accessed on August 13, 2025, <https://en.wikipedia.org/wiki/Vickrey%E2%80%93Clarke%E2%80%93Groves_auction>
27. The Lovely but Lonely Vickrey Auction - Paul Milgrom, accessed on August 13, 2025, <https://milgrom.people.stanford.edu/wp-content/uploads/2005/12/Lovely-but-Lonely-Vickrey-Auction-072404a.pdf>
28. Unsafe Rust - The Rust Programming Language - Brown Rust Book, accessed on August 13, 2025, <https://rust-book.cs.brown.edu/ch20-01-unsafe-rust.html>
29. How Do Programmers Use Unsafe Rust? - Programming Methodology Group, accessed on August 13, 2025, <https://pm.inf.ethz.ch/publications/AstrauskasMathejaMuellerPoliSummers20.pdf>
30. verus-lang/verified-memory-allocator - GitHub, accessed on August 13, 2025, <https://github.com/verus-lang/verified-memory-allocator>
31. Towards Practical Formal Verification for a General-Purpose OS in Rust - Asterinas, accessed on August 13, 2025, <https://asterinas.github.io/2025/02/13/towards-practical-formal-verification-for-a-general-purpose-os-in-rust.html>
32. howtorhino.com, accessed on August 13, 2025, <https://howtorhino.com/blog/architecture-styles/brutalist-architecture/#:~:text=This%20style%20embraces%20massive%20forms,function%20and%20purpose%20in%20society.>
33. Brutalist Architecture: Origins, Characteristics, and Examples - Dans Le Gris, accessed on August 13, 2025, <https://danslegris.com/blogs/journal/brutalist-architecture>
34. Brutalist Architecture Explained - Spreaker, accessed on August 13, 2025, <https://www.spreaker.com/podcast/brutalist-architecture-explained--6099098>
35. atomic\_queue | C++14 lock-free queue. - GitHub Pages, accessed on August 13, 2025, <https://max0x7ba.github.io/atomic_queue/>
36. Domain-Specific Languages - CWI, accessed on August 13, 2025, <https://homepages.cwi.nl/~paulk/publications/Sigplan00.pdf>
37. Procedural Macros - The Rust Reference, accessed on August 13, 2025, <https://doc.rust-lang.org/reference/procedural-macros.html>
38. Compile-Time Polymorphism: A Comparison of C++ and Rust Metaprogramming - Medium, accessed on August 13, 2025, <https://medium.com/@martin00001313/compile-time-polymorphism-a-comparison-of-c-and-rust-metaprogramming-7be921cd01f8>
39. The seL4 Microkernel | seL4, accessed on August 13, 2025, <https://sel4.systems/>
40. SeL4 Whitepaper [pdf], accessed on August 13, 2025, <https://sel4.systems/About/seL4-whitepaper.pdf>