https://doi.org/10.31436/iiumej.v27i2.4361

# TSGL-Rust: Teaching Concurrency Through Compile-Time Enforcement via Rust-Based Visualization of Classical Synchronization Problems

**TEDDY SURYA GUNAWAN***

*Electrical and Computer Engineering Dept., International Islamic University Malaysia, Malaysia*

**Corresponding author: tsgunawan@iium.edu.my*

*(Received: 10 March 2026; Accepted: 30 April 2026; Published online: 10 May 2026)*

***ABSTRACT:*** Concurrent programming remains a persistent challenge in undergraduate computer science and engineering education, largely because its failure modes are difficult to observe and reason about. Race conditions and deadlocks often emerge only under specific thread interleavings, leaving students to reconcile correct-looking code with unpredictable runtime behavior. Prior instructional approaches, such as the Thread-Safe Graphics Library (TSGL), address this issue by enabling real-time visualization, but they rely on languages that do not enforce safe access to shared state. This separation between observation and enforcement leaves a gap between how concurrency errors are detected and how they are prevented. This paper presents TSGL-Rust, a Rust-based reimplementation of classical TSGL visualizations that integrates compile-time safety guarantees with runtime visualization. Implemented using the egui/eframe framework, the system reproduces three canonical synchronization problems, namely Producer–Consumer, Reader–Writer, and Dining Philosophers, while shifting certain classes of errors from runtime to compilation through Rust's ownership model and borrow checker. The architecture deliberately separates concurrency logic from rendering through a crossbeam channel, decoupling worker threads from the egui main-thread render loop. This paper is presented as a system and tool description: it focuses on architectural design, the mapping from C++ TSGL primitives to Rust constructs, the channel-based communication pattern, and the instructional affordances that follow from these design choices. The contribution lies in the system itself and in the boundary it makes visible between data-race prevention and logical errors, such as deadlock, rather than in any claim about measured learning gains.

***ABSTRAK:*** Pengaturcaraan serentak kekal sebagai cabaran utama dalam pendidikan sarjana muda sains komputer dan kejuruteraan, kerana mod kegagalannya sukar dicari dan difahami secara sistematik. Keadaan perlumbaan dan kebuntuan lazimnya hanya muncul di bawah jalinan pelaksanaan jalur tertentu, sekaligus menyebabkan pelajar perlu menyelaras kod yang kelihatan betul dengan tingkah laku masa jalan yang tidak menentu. Pendekatan pengajaran terdahulu, seperti Koleksi Grafik Jalur-Selamat (TSGL), menangani isu ini melalui visualisasi masa nyata, tetapi bergantung pada bahasa yang tidak menguatkuasa capaian selamat pada keadaan berkongsi. Pemisahan antara pemerhatian dan penguatkuasaan ini mewujudkan jurang antara bagaimana ralat serentak dikesan dan bagaimana ia dicegah. Kajian ini membentangkan TSGL-Rust, iaitu pelaksanaan semula TSGL berasaskan Rust yang mengintegrasi jaminan keselamatan masa kompilasi dan visualisasi masa jalan. Melalui rangka kerja e-GUI/e-rangka, sistem ini menjana semula tiga masalah menyegerakan kanonik, iaitu Pengeluar-Pengguna, Penulis-Pembaca, dan Dining Philosophers, sambil mengalihkan sebahagian kelas ralat daripada masa jalan kepada peringkat kompilasi melalui model pemilikan dan pemeriksa pinjaman Rust. Seni bina sistem ini secara sengaja memisahkan logik serentak daripada paparan alur silang, dengan jalur pekerja dipisahkan daripada gelung peparan utama eGUI. Kajian ini dibentangkan sebagai penerangan sistem dan alat: tumpuan

https://doi.org/10.31436/iiumej.v27i2.4361

diberi pada reka bentuk seni bina, pemetaan primitif TSGL C++ pada konstruk Rust, corak komunikasi berasaskan saluran, dan keupayaan instruksional yang berpunca daripada reka bentuk ini. Sumbangan utama terletak pada sistem ini sendiri dan pada sempadan yang dijelaskan antara pencegahan keadaan perlumbaan dan ralat logik seperti kebuntuan, dan bukan pada sebarang dakwaan terhadap pencapaian pembelajaran yang diukur.

***KEYWORDS:*** *Concurrent programming, Rust, compile-time safety, visualization, operating systems education*

## 1. INTRODUCTION

Multicore processors now define the baseline computing environment, and concurrency has correspondingly become a required competency in modern computer science curricula [1]. Within operating systems and systems programming courses, students are expected to reason about interleavings, synchronization mechanisms, and subtle failure modes that arise under parallel execution. Despite this expectation, conceptual mastery often lags behind exposure. Concurrency resists intuition.

Among these difficulties, the invisibility of failure remains central. Race conditions emerge only under specific scheduling sequences, often fleeting and irreproducible. Deadlocks surface under particular resource acquisition patterns that may not appear during routine testing. Students frequently encounter programs that compile cleanly and appear correct, yet behave inconsistently at runtime. The resulting gap between static code and dynamic behavior complicates reasoning and may partly reflect the absence of structural feedback in conventional languages [2].

Prior work attempted to address this gap through visualization. The Thread-Safe Graphics Library (TSGL) was introduced in [3] to enable real-time rendering under multithreaded execution, allowing students to observe concurrency as it unfolds. Empirical results suggested improved comprehension of parallel constructs. Even so, TSGL operates within C++, where data races are undefined behavior with no compile-time guarantees. Visualization exposes symptoms but does not prevent their cause. Students still confront misbehavior after execution, often without clear guidance on its origin.

Rust introduces a different dynamic. Its ownership model and borrow checker enforce memory safety and eliminate data races at compile time [4]. Access to shared mutable state is restricted unless properly synchronized, and violations are rejected during compilation with precise diagnostics. The location of failure shifts. Instead of debugging unpredictable runtime outcomes, programmers encounter structured compiler feedback that links unsafe code patterns directly to concurrency violations. This shift narrows the conceptual distance between program structure and correctness, though its accessibility likely depends on prior familiarity with ownership semantics [5].

Building on this premise, TSGL-Rust reinterprets visualization through compile-time enforcement. Implemented in Rust using the egui immediate-mode GUI framework, the system reproduces three canonical synchronization problems, namely Producer-Consumer, Reader-Writer, and Dining Philosophers. Each visualization preserves the temporal dynamics emphasized in TSGL while embedding them within a language that prevents data races before execution. The tool was developed for an undergraduate Operating Systems course at IIUM, where Rust is a required component, and is presented here as a system and tool design rather than as an empirical educational study.

This work makes three contributions. First, a system architecture that integrates compile-time safety with runtime visualization for the three classical synchronization problems within a single Cargo workspace, separating concurrency logic from rendering via a crossbeam channel. Second, an explicit mapping from C++ TSGL primitives to their Rust equivalents. Third, an instructional affordance: compile-time diagnostics and runtime visualization together expose the boundary between data-race prevention and logical errors, such as deadlock—a boundary that is structural rather than empirical and follows directly from the choice of language and architecture.

## 2. Background and Related Works

Situating TSGL-Rust requires drawing together several strands of prior work: concurrency visualization tools, Rust's safety model, recent advances in Rust pedagogy, and dynamic analysis or model-checking systems for concurrency. The discussion below positions TSGL-Rust against each other.

### 2.1. Original TSGL

Adams et al. (2018) introduced the Thread-Safe Graphics Library (TSGL) [3], a C++11 framework that enables multiple threads to render to a shared canvas in near-real time. The design goal was pragmatic. Instructors could insert drawing calls into existing multithreaded programs with minimal restructuring, thereby making execution traces visible as graphical artifacts. Students no longer relied solely on textual output or static reasoning. They observed concurrency unfold. The original TSGL is available at https://github.com/Calvin-CS/TSGL/tree/visExpUpdates.

Empirical evaluation suggested measurable benefits. Students exposed to visualization demonstrated improved understanding of parallel constructs compared to those using traditional methods. More recent work in computing education has reinforced this general finding, indicating that dynamic visual representations can reduce cognitive load when reasoning about temporal processes [6, 7]. Despite this, TSGL inherits fundamental limitations from C++. Synchronization remains a matter of discipline rather than enforcement. A program containing a data race compiles successfully and may execute without immediate failure, leaving inconsistent behavior as the only evidence of the underlying fault. In practice, visualization reveals symptoms but offers little structural guidance toward correctness.

### 2.2. Rust's Safety and Concurrency Model

Rust approaches concurrency from a different premise. Memory safety and data race freedom are enforced through the ownership model and borrow checker, both evaluated at compile time [4]. A value is either uniquely owned, immutably borrowed by multiple references, or mutably borrowed by exactly one reference. These constraints are not advisory. The compiler rejects violations outright.

For concurrent programming, this design encodes synchronization into the type system itself. Shared ownership is implemented using constructs such as `Arc<T>`, while controlled mutation is mediated by `Mutex<T>` or `RwLock<T>`. Access to protected data is only possible through guard objects returned by lock acquisition, ensuring that unsynchronized access is structurally impossible. Research on Rust's formal semantics indicates that these guarantees extend even under relaxed memory models, providing strong safety properties without runtime overhead [5]. Evidence from large-scale systems further indicates that adopting memory-safe languages significantly reduces vulnerability density, particularly for concurrency-related

defects [8, 9]. Industry and government guidance have reinforced this position, with recent recommendations from the U.S. Office of the National Cyber Director advocating memory-safe languages for critical software [10].

These guarantees are bounded. Deadlock, starvation, and other liveness errors remain outside the scope of the type system. That boundary is pedagogically useful. It distinguishes clearly between what the language enforces and what must still be reasoned about explicitly.

### 2.3. egui and eframe

`egui` is an immediate-mode graphical user interface library for Rust that constructs the entire interface each frame from the current application state [11]. Unlike retained-mode frameworks, where UI elements persist across frames, `egui` treats the interface as a function of state, recalculated continuously. This design aligns closely with the requirements of concurrency visualization. State changes frequently. Visual feedback must follow immediately.

The accompanying `eframe` crate provides the application scaffold, including window management, event handling, and the render loop, which executes on the main thread. As a result, worker threads cannot directly manipulate the interface and must instead communicate state updates to the render loop. That constraint introduces a clear separation between concurrency logic and rendering. While initially restrictive, it may reduce incidental complexity by preventing shared mutable access to UI state across threads.

Immediate-mode rendering shifts the developer's mental model. Rather than updating widgets imperatively, the program describes what the interface should look like given the current state at each frame. `egui` also compiles to native binaries across major platforms and to WebAssembly, allowing deployment without external graphics dependencies. For undergraduate settings, this reduces setup overhead and broadens accessibility.

### 2.4. Recent Rust Pedagogy and Ownership Visualization

Beyond the language itself, recent work has examined how Rust is learned. Crichton et al. [12] developed a grounded conceptual model of Rust's ownership types, implemented as the Aquascope visualizer, and integrated it into a forked online edition of The Rust Programming Language. Their controlled study reported a measurable improvement in scores on an Ownership Inventory after exposure to permission-based visualizations of borrowing checking, relative to the baseline pedagogy. Subsequent work has extended these ideas into broader curricular guidance for systems courses delivered in Rust [13]. While these contributions target the static side of the language, i.e., the ownership and borrowing rules themselves, they share with TSGL-Rust an emphasis on visual representation as a means of access to otherwise opaque static analyses.

TSGL-Rust complements this body of work along a different axis. Rather than visualizing the borrow checker's internal reasoning, it accepts the compile-time guarantees as given and visualizes the runtime behavior of programs that survive them. The two perspectives are not in competition. A course that uses Aquascope-style visualizations for ownership during early lectures and TSGL-Rust during the synchronization unit would address both sides of the static–dynamic divide that Rust learners are reported to find difficult [12].

### 2.5. Visualization, Dynamic Analysis, and Model Checking

Several established systems analyze concurrent behavior, though their assumptions and instructional roles differ in important ways. Helgrind and ThreadSanitizer instrument program execution to detect data races dynamically. Java PathFinder explores execution paths through

model checking. TLA+ supports formal specification and verification of concurrent algorithms. These tools intervene before, during, or after execution. Violations are discovered retrospectively or through exhaustive search.

**Table 1**. Comparison of concurrency education and debugging tools.

| Tool / System | Primary purpose | Timing of feedback | Type of feedback | Instructional role | Vis. + Safety |
|---|---|---|---|---|---|
| Helgrind / ThreadSanitizer | Dynamic data-race detection | Runtime | Warnings, stack traces | Post-hoc debugging; non-deterministic triggering | — |
| Java PathFinder / TLA+ | Model checking and formal verification | Pre-execution (state exploration) | Counter-example traces | Formal-methods exposure; steep learning curve | — |
| Progvis [14-16] | Object-graph visualization with model checking | Runtime; verifier-supported | Object-graph rendering, feedback | Concurrency intuition with shared-memory focus | Vis. |
| TSGL [3] | Real-time concurrency visualization | Runtime only | Graphical thread-state rendering (C++ / OpenGL) | Conceptual insight into parallel behavior | Vis. |
| Aquascope / The Rust Book fork [12] | Ownership and borrow-checker visualization | Compile-time (static) | Permission diagrams alongside the source | Rust ownership pedagogy | Safety |
| TSGL-Rust (this work) | Visualization plus compile-time safety | Compile-time AND runtime | Compiler diagnostics + graphical thread-state rendering | Conceptual insight AND structural safety | Vis. + Safety |

**Note:** "Vis." denotes real-time graphical visualization of program state. "Safety" denotes compile-time data-race prevention enforced by the type system. TSGL-Rust is the only tool in this comparison that provides both within a single environment.

Educational visualization operates under a different constraint. Rather than locating errors, it renders the temporal structure of concurrent execution visible as it unfolds. Progvis, developed by Strömbäck and colleagues [14], visualizes concurrent C and C++ programs as object graphs and has been extended with a weak memory model and integrated model checking [15] and with model-checking-driven feedback combined with peer grading [16]. These extensions move Progvis closer to the dynamic-analysis end of the spectrum while preserving its visualization-first character. TSGL [3] follows the visualization-first approach by exposing runtime behavior. TSGL-Rust extends this approach by introducing compile-time enforcement as an additional feedback channel. Table 1 contrasts these systems along dimensions relevant to undergraduate instruction.

TSGL-Rust occupies a hybrid position. It does not replace existing tools but shifts part of the feedback earlier: compile-time rejection prevents unsafe programs from executing, while runtime visualization remains essential for liveness errors such as deadlock and starvation, which lie beyond the reach of the type system. The two channels are complementary, i.e., one constrains what can be expressed; the other reveals how correct programs behave over time.

https://doi.org/10.31436/iiumej.v27i2.4361

Within this landscape, TSGL-Rust differs from Aquascope along the static–dynamic axis (static permissions versus runtime thread states), from Progvis along the language and safety axis (Rust with compile-time data-race prevention versus C/C++ with runtime model checking), and from the original TSGL along the enforcement axis (a memory-safe language versus an unsafe one). The novelty, therefore, lies less in any single capability than in the combination, and in the architectural choices that make this combination tractable for an undergraduate operating systems setting.

## 3. Classical Synchronization Problems

Classical synchronization problems provide structured entry points through which students can connect abstract concurrency concepts to concrete operating system behavior, while also exposing the boundary between compile-time enforcement and runtime correctness. Fig. 1 illustrates three classical synchronization problems, including Producer–Consumer, Reader–Writer, and Dining Philosophers. In practice, these problems arise in both process-based and thread-based concurrency, depending on how execution is scheduled and how memory is shared [17-19]. Processes may coordinate through pipes, sockets, or shared memory regions, while threads typically interact within a shared address space using synchronization primitives such as mutexes or condition variables. Despite these differences, the underlying challenge remains consistent. Multiple execution contexts must coordinate access to shared state without violating correctness.

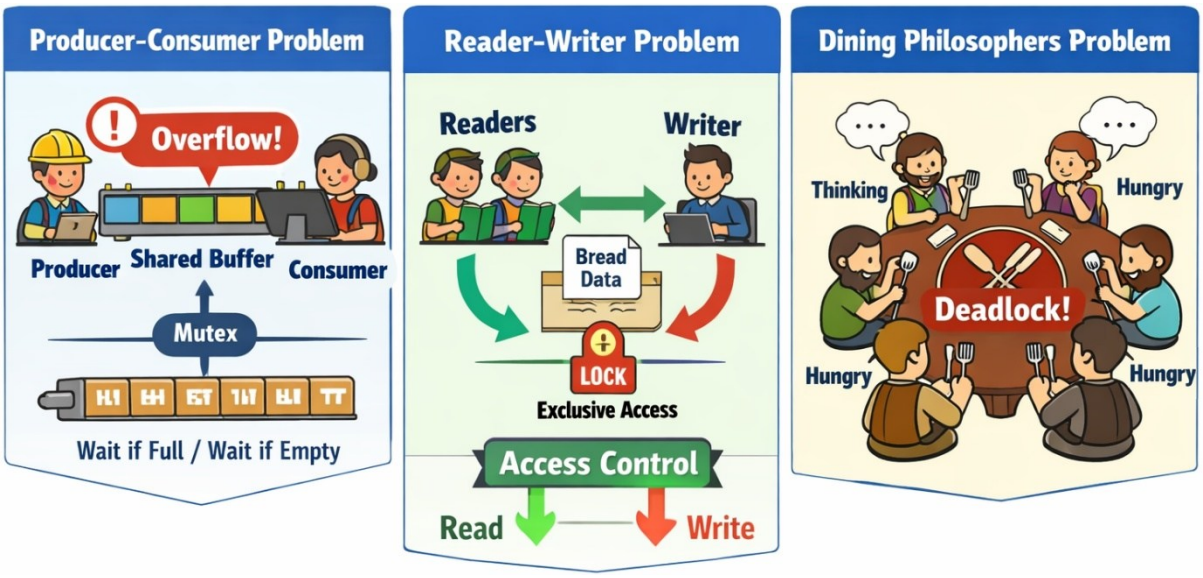



**Figure 1**. Classical synchronization problems addressed by TSGL-Rust: Producer–Consumer, Reader–Writer, and Dining Philosophers.

Resources involved in these interactions can be categorized as consumable or reusable, and this distinction shapes how synchronization is designed. Consumable resources are produced and then removed once used, as in a Linux pipe or Windows message queue, where data flows from producer to consumer and does not persist. Reusable resources, by contrast, can be acquired and released repeatedly, such as a mutex protecting a kernel structure or a file lock shared among processes. Consumable resources emphasize availability and ordering, while reusable resources emphasize controlled access and ownership. These patterns appear frequently in operating systems [20-22], though they may not be immediately visible to system users. Grounding the classical problems in such familiar mechanisms makes the abstractions

feel less artificial, and the role of synchronization clearer as a recurring constraint in system design rather than an isolated curricular topic.

### 3.1. Producer–Consumer Problem

The Producer–Consumer problem models coordination over a bounded shared buffer between producers that generate data and consumers that process it. Correct behavior depends on mutual exclusion for buffer access and condition-based blocking when the buffer becomes full or empty. Under incorrect synchronization, subtle failures emerge. Data may be overwritten, duplicated, or lost entirely. These errors often appear intermittently, which makes them particularly difficult to diagnose.

In practice, this pattern appears throughout operating systems. A Linux shell pipeline such as `ls | grep` implements a bounded pipe buffer managed by the kernel. In Windows, anonymous pipes and producer–consumer queues in the Win32 API follow the same structure, with blocking semantics enforced by the system. Print spoolers in both environments exhibit identical behavior: applications produce jobs, and a background service consumes them. The abstract buffer is not merely theoretical. It exists in tools that are routinely used. Table 2 summarizes representative operating system instances of this pattern.

**Table 2**. Real Operating System Examples of the Producer-Consumer Pattern.

| Real-World Instance | Producer | Consumer | Buffer |
|---|---|---|---|
| Linux pipe (`ls \| grep`) | ls process (`stdout`) | grep process (`stdin`) | OS pipe buffer |
| Windows anonymous pipe | Writing process/thread | Reading process/thread | OS-managed pipe buffer |
| Print spooler | User applications | Print service/daemon | Spool queue (disk/memory) |
| Network socket | Application (`send()`) | OS network stack | Socket send buffer |
| Video streaming | Decoder thread | Renderer thread | Frame ring buffer |

The shell pipeline is the most accessible example. It is routinely used without explicit recognition that blocking and synchronization are being enforced beneath the surface.

### 3.2. Reader–Writer Problem

The Reader–Writer problem addresses controlled access to shared data that can be read concurrently but must be written exclusively. Multiple readers may proceed in parallel. Writers require exclusive access. The central challenge lies in balancing fairness and throughput, since prioritizing one class of operations may delay the other.

This pattern aligns closely with language-level abstractions. Rust's `RwLock<T>` encodes the distinction between shared and exclusive access directly in the type system. A thread may hold multiple read guards or a single write guard, but never both simultaneously. This constraint is enforced at compile time, shaping how access must be structured before execution begins. It shifts part of the reasoning earlier, though understanding the implications of fairness policies still requires runtime observation.

Operating systems rely heavily on this pattern. Linux employs reader–writer locks (`rwlock_t`) in subsystems such as virtual memory management. Windows provides Slim Reader/Writer (SRW) locks for efficient shared access. File metadata, DNS caches, and server configuration structures all depend on this form of synchronization. Table 3 summarizes representative instances.

https://doi.org/10.31436/iiumej.v27i2.4361

**Table 3**. Real Operating System Examples of the Reader-Writer Pattern.

| Real-World Instance | Readers | Writers | Shared Resource |
|---|---|---|---|
| Linux `rwlock_t` | CPU cores (page reads) | Core updating page table | Virtual memory page table |
| Windows SRW lock | Multiple threads (read access) | Single thread (write access) | Shared memory region |
| File system inode | Processes calling `stat()` | Process calling `chmod()` | `Inode` metadata |
| DNS resolver cache | DNS lookup threads | Cache refresh process | Cached DNS entries |
| Web server configuration | Worker threads (HTTP requests) | Admin reload operation | Configuration structure |

This problem often appears straightforward until contention increases. Under load, the interaction between readers and writers becomes visible, and fairness policies take on greater importance.

### 3.3. Dining Philosophers Problem

The Dining Philosophers problem models resource contention in the presence of circular dependencies. Each participant requires multiple shared resources, and if all attempt to acquire them in the same order, a circular wait can arise. No participant can proceed. The system halts.

Unlike the previous problems, this one exposes a limitation of compile-time guarantees. Proper use of mutexes ensures memory safety, yet deadlock remains possible. The failure arises not from unsafe access but from the ordering of resource acquisition. This distinction often becomes clearer through observation than through abstract reasoning alone.

Operating system examples are widespread. In Linux, kernel subsystems acquiring multiple mutexes may deadlock if lock ordering is inconsistent, prompting the use of runtime tools such as lock dependency checkers. In Windows, file and handle locking across threads can produce similar circular waits. Database systems provide a particularly concrete example. Transactions acquiring row-level locks may form a cycle, leading to deadlock detection and forced rollback. Table 4 summarizes representative instances.

**Table 4**. Real Operating System Examples of the Dining Philosophers (Deadlock) Pattern.

| Real-World Instance | Actors | Resources | Deadlock Scenario |
|---|---|---|---|
| Database transactions | Concurrent transactions | Row-level locks | Circular wait detected (e.g., transaction rollback) |
| Multi-threaded file ops | Processing threads | File locks | Thread A waits for file2, Thread B waits for file1 |
| Linux kernel mutexes | Kernel subsystems | Kernel mutexes | Lock ordering violation detected at runtime |
| Windows handle locking | Threads/processes | File/handle locks | Circular dependency across resource acquisition |
| GPU shader execution | Shader threads | Memory banks | Cyclic resource dependency |

Database deadlocks are often the most visible. Errors such as transaction rollback occur in practice, yet the underlying pattern is rarely recognized without explicit instruction.

### 3.4. Comparative Synthesis and Rust Correctness Boundaries

Table 5 summarizes the three problems with respect to their primary hazard, synchronization mechanism, corresponding Rust abstraction, representative operating system example, and the extent to which Rust prevents the associated error.

**Table 5**. Comparative Summary of the Three Classical Synchronization Problems.

| **Property** | **Producer–Consumer** | **Reader–Writer** | **Dining Philosophers** |
|---|---|---|---|
| Primary hazard | Buffer race | Read–write conflict | Deadlock/starvation |
| Key primitive | Mutex + Condvar | RwLock | Mutex per resource |
| Rust abstraction | `Arc<Mutex<VecDeque>>` | `Arc<RwLock<T>>` | `Vec<Arc<Mutex<()>>>` |
| OS example | Linux/Windows pipe | Kernel/SRW locks | DB/file locking |
| Prevented by Rust | Yes (data race) | Yes (data race) | No (logical error) |

The final row carries particular weight. Rust eliminates data races in the first two problems through compile-time enforcement. Deadlock remains. The result is a layered model of correctness, in which some classes of errors are prevented structurally while others must still be reasoned about through execution and design.

### 3.5. Extension to Deep Learning Training and Inference Workflows

The correctness boundaries identified in Section 3.4 are not specific to operating systems. They recur at the application level in modern artificial intelligence workloads, where they govern the throughput and correctness of training and inference systems. The Producer–Consumer pattern appears most directly in the data loading pipeline. PyTorch's DataLoader with `num_workers > 0` spawns worker processes that produce preprocessed and augmented mini-batches into a bounded queue, while the training loop on the GPU consumes them. The `prefetch_factor` parameter sets the buffer capacity, and the same starvation and saturation conditions illustrated in Section 3.1 govern whether the GPU stalls or the CPU workers block. Frame ingestion pipelines for video tasks, i.e., capture, preprocessing, inference, rendering, chain several producer–consumer stages with bounded buffers between them. Asynchronous logging of training metrics and checkpoint writes follows the same pattern, but at a lower volume.

The Reader–Writer pattern surfaces in inference serving. Many concurrent request handlers read the model weights to perform forward passes, while a control thread occasionally swaps the weights when a new checkpoint is hot-loaded. Shared lookup tables for physics priors, for example, precomputed turbulence kernels or atmospheric coefficients indexed by parameters such as the refractive index structure constant, exhibit the same readers-many, writer-rare structure. The Dining Philosophers pattern is less direct in this domain but can arise in distributed training, where lock-ordering errors across gradient synchronization primitives can cause hangs in multi-GPU settings. Within this broader landscape, the structural distinction that TSGL-Rust makes visible, between data-race prevention and liveness, has currency beyond the systems curriculum in which it is currently deployed, and the same enforcement boundary that separates Sections 3.1 and 3.2 from 3.3 reappears in AI pipelines that combine high-throughput data movement with shared mutable state.

## 4. SYSTEM ARCHITECTURE

This section describes the architecture of TSGL-Rust, covering the workspace organization, the mapping of C++ concurrency primitives to equivalent Rust constructs, and the channel-

https://doi.org/10.31436/iiumej.v27i2.4361

based communication pattern that separates worker thread logic from the egui rendering pipeline.

### 4.1. Design Overview

TSGL-Rust is organized as a Cargo workspace containing three independent binary targets: `reader_writer`, `producer_consumer`, and `dining_philosophers`. Each binary is self-contained, with its own concurrency logic and visualization code. The workspace uses a shared `Cargo.toml` declaring common dependencies: `eframe` and `egui` for visualization, `parking_lot` for efficient mutex and `RwLock` implementations, and `crossbeam` for multi-producer, multi-consumer channels. The overall system architecture is illustrated in Figure 2.

The central architectural decision is the separation of concurrency logic from rendering. In the original TSGL, drawing calls are made directly from worker threads into the shared Canvas, which is internally synchronized. In TSGL-Rust, worker threads never touch the GUI. Instead, each thread computes its state and transmits it to the main thread via a `crossbeam` channel. The `egui` render loop, running on the main thread at each frame, reads the latest state snapshot and renders the visualization. This design is required by `egui`'s threading model, but it also yields a cleaner separation of concerns: unit tests can verify synchronization behavior without instantiating any GUI, and changes to the visualization style require no changes to thread code, i.e., a structural guarantee that TSGL's shared-canvas approach cannot provide.

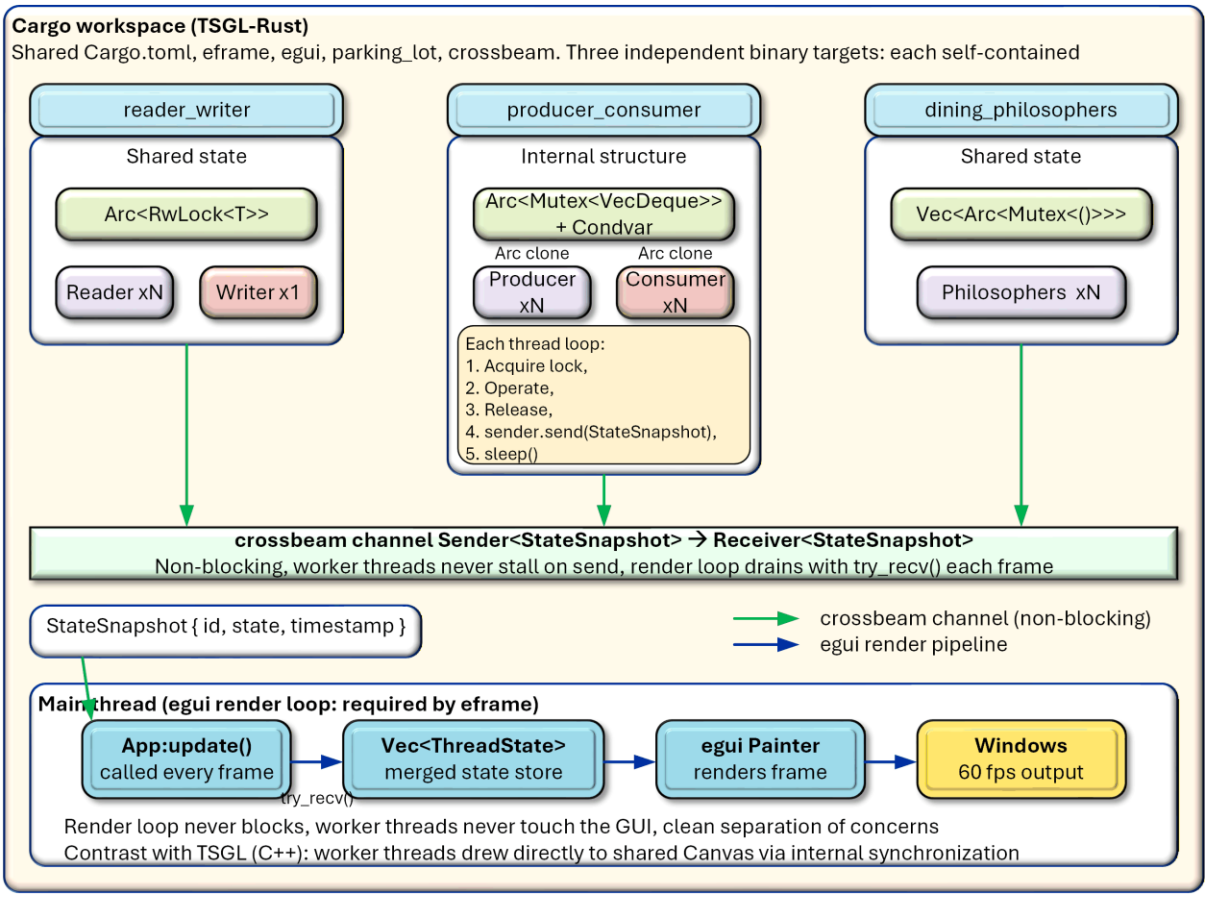



**Figure 2**. Overall system architecture of TSGL-Rust illustrating the Cargo workspace, worker thread concurrency logic, crossbeam channel, and egui render loop separation

### 4.2. C++ to Rust Primitive Mapping

Table 6 shows the direct mapping between C++ concurrency primitives used in TSGL and the corresponding Rust constructs used in TSGL-Rust. The most consequential difference is not syntactic but semantic. In C++, `std::mutex` provides mutual exclusion, but nothing in the type system prevents a thread from accessing the protected data without acquiring the lock. In Rust, `Arc<Mutex<T>>` wraps the data inside the lock: the protected value is only accessible through the lock guard returned by `lock()`, and the compiler rejects any attempt to access it otherwise. The synchronization requirement is encoded in the type, not enforced by programmer discipline. This distinction applies equally to `Arc<RwLock<T>>` in the `reader_writer` binary and to `Vec<Arc<Mutex<()>>>` in `dining_philosophers`: in each case, the relationship between the synchronization primitive and the data it protects is structurally enforced at compile time rather than assumed at runtime.

**Table 6**. Mapping of C++ TSGL concurrency primitives to Rust TSGL-Rust equivalents

| C++ (TSGL) | Rust (TSGL-Rust) | Notes |
|---|---|---|
| `std::mutex` | `Arc<Mutex<T>>` | Compile-time lock enforcement |
| `std::condition_variable` | `Arc<Condvar>` | Paired with `Mutex` |
| `std::shared_mutex` | `Arc<RwLock<T>>` | `ReaderWriter` only |
| `std::thread` | `std::thread::spawn` | Returns `JoinHandle` |
| `shared_ptr<T>` | `Arc<T>` | Atomic reference counting |
| TSGL Canvas draw calls | egui Painter + channels | State sent to render thread |
| `std::this_thread::sleep_for` | `std::thread::sleep` | Direct equivalent |

### 4.3. Thread Communication Pattern

All three programs follow a common communication pattern. Each worker thread is spawned with an `Arc` clone of a `crossbeam` `Sender`, moved into the thread closure at spawn time. At each state transition, a consumer removes an item; a philosopher changes from thinking to hungry; a writer acquires the database; the thread constructs a lightweight `StateSnapshot` struct containing the thread ID, current logical state, and a timestamp, and transmits it via `try_send()`. Using `try_send` rather than the blocking send ensures that worker threads are never stalled by rendering latency. If the channel buffer is momentarily full, the snapshot is silently discarded, and the thread proceeds immediately. Under normal operating conditions, the render loop drains the channel faster than threads produce updates, so snapshot loss is rare and inconsequential for an educational visualization.

The render loop calls `try_recv()` each frame to drain all available updates, merging each snapshot into a `Vec<ThreadState>` indexed by thread ID. This `Vec` always reflects the most recent known state of every thread, regardless of how many snapshots were transmitted since the last frame, and it is this `Vec` that the egui `Painter` renders at each update cycle. The pattern guarantees that the render loop never blocks waiting for thread updates and that the concurrency logic is never delayed by the rendering pipeline: a clean, deterministic boundary that reflects the ownership discipline Rust enforces throughout the codebase.

## 5. IMPLEMENTATION

Each of the three TSGL-Rust binaries follows the same channel-based architecture described in Section 4, but differs in the shared state type, synchronization primitives, and visual layout that are appropriate to its specific concurrency problem.

### 5.1. Reader-Writer

The Reader-Writer implementation uses `Arc<RwLock<i32>>` as the shared database. Multiple reader threads call `read()` on the `RwLock` in a loop, holding the read guard for a randomized duration representing reading time, then sleeping before the next read. A single writer thread calls `write()`, holds the exclusive guard for a shorter duration, and increments the protected value. Figure 3 shows the internal block structure of this binary.

The key insight is the compile-time box in Fig. 3. In Rust, a thread holding a read guard cannot call any method that mutates the protected value, in which the type system makes mutation via `&RwLockReadGuard` a compile-time error rather than a runtime failure. First-reader preference emerges naturally from `parking_lot`'s `RwLock` implementation with no explicit counter management by the programmer.

The visualization renders a left panel showing active reader count and a right panel showing writer state. A central rectangle representing the shared database pulses when a write is in progress. Thread state badges, including Reading, Waiting, and Writing, use blue for active, orange for waiting, and green for completed states. A speed control slider adjusts sleep durations, allowing the simulation to be slowed to a rate suitable for classroom demonstration.

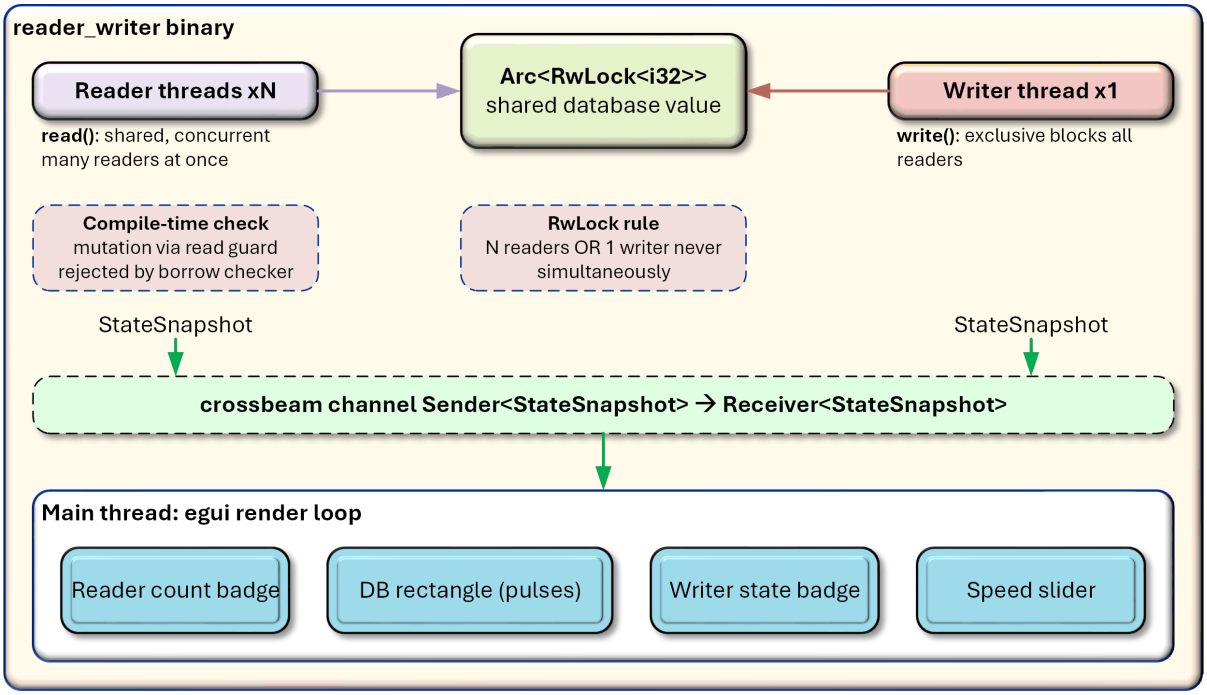



**Figure 3**. Reader-Writer Internal Block Structure

### 5.2. Producer-Consumer

The Producer-Consumer implementation uses `Arc<Mutex<VecDeque<u32>>>` as the bounded buffer, with a capacity constant configurable at compile time. `Condvar` pairs are used for blocking: producers call `wait()` on a full condition variable when the buffer reaches capacity, and consumers call `wait()` on an empty condition variable when the buffer is empty. Figure 4 shows the internal block structure of this binary.

The `Condvar::wait()` semantics box in Fig. 4 is worth noting. The method atomically releases the Mutex, suspends the calling thread, and then re-acquires the lock when signaled. This prevents the missed-wakeup bug that arises when a thread checks the condition and then releases the lock as separate operations, which is a common error in manual C implementations.

In Rust, `Condvar::wait()` requires a `MutexGuard` argument, making the lock relationship explicit in the type signature and matching the semantics of `std::condition_variable::wait()` in C++ while preventing access to the protected data without the guard.

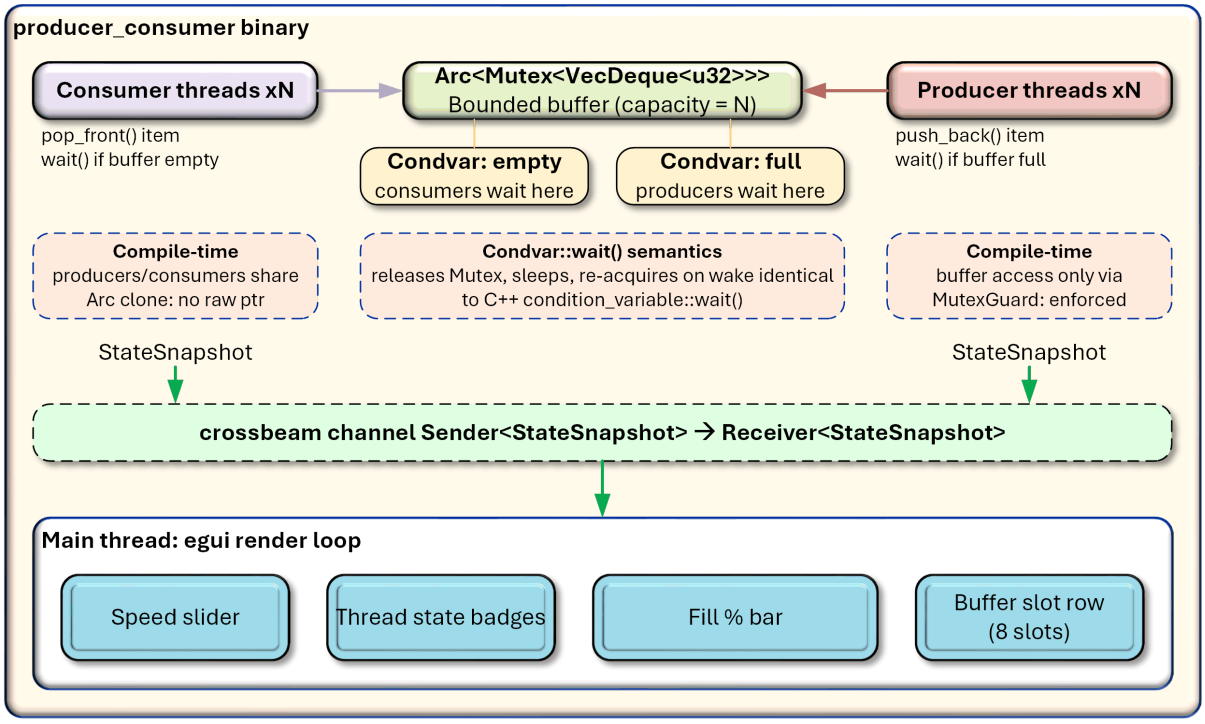



**Figure 4**. Producer-Consumer Internal Block Structure

The visualization renders a horizontal row of buffer slots, each colored blue when occupied and light gray when empty, with a fill percentage bar above. Producer threads appear on the left with directional indicators pointing into the buffer; consumer threads appear on the right pulling from it. When all slots are gray, consumers block; when all slots are blue, producers block. Saturation and starvation conditions are immediately visible without reading any code.

### 5.3. Dining Philosophers

The Dining Philosophers implementation uses a `Vec<Arc<Mutex<()>>>` with one mutex per fork, following the resource ordering deadlock avoidance strategy: each philosopher acquires the lower-numbered fork before the higher-numbered fork, breaking the circular wait condition identified by Dijkstra [17, 18]. Figure 5 shows the internal block structure of this binary.

The red dashed box in Fig. 5 is the most important teaching point of the entire section. Rust's borrow checker prevents data races, i.e., concurrent unsynchronised access to shared memory, but it cannot prevent deadlock, because deadlock is a logical error arising from lock acquisition order rather than from unsafe memory access. If the resource ordering strategy is removed and all philosophers acquire forks in the same ascending order, the program compiles without error and deadlocks at runtime. This behavior is intentional in TSGL-Rust. The Dining Philosophers visualization illustrates that Rust's compile-time guarantees are powerful yet bounded, and that correct synchronization logic remains the programmer's responsibility.

The visualization renders a circular table using the `egui::Painter` API with philosopher nodes distributed evenly around the circumference, coloured grey for thinking, orange for hungry, and green for eating. Fork segments between adjacent nodes are coloured to indicate availability. The circular layout makes the potential for circular waiting geometrically apparent, allowing us to trace the dependency ring that would form if all philosophers simultaneously held their left fork and waited for their right to be traced visually.

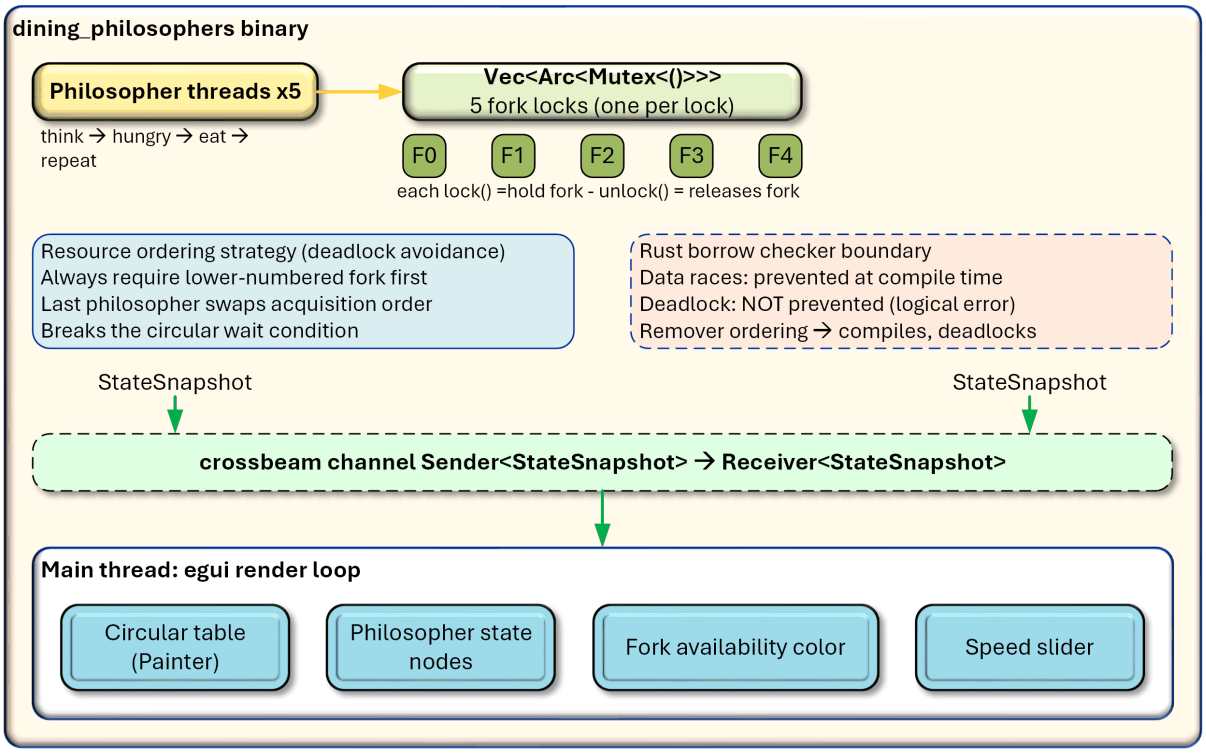



**Figure 5**. Dining-Philosophers Internal Block Structure

The complete source code for TSGL-Rust, including all three binary targets (`reader_writer`, `producer_consumer`, and `dining_philosophers`), the shared `Cargo.toml` workspace configuration, and the `README.md` project documentation, is publicly available at https://github.com/tsgunawan/TSGL_rust. The repository is structured to allow immediate execution on Windows, Linux, and macOS via a single `cargo run --bin <target>` command without any external graphics library installation, lowering the barrier to adoption for instructors who wish to integrate the tool into an existing operating systems laboratory session. Each binary is accompanied by inline code comments that map Rust synchronization constructs to their C++ equivalents, providing a side-by-side reference to support students transitioning from a C++-based concurrency curriculum. Instructors can fork the repository and modify the thread count, buffer capacity, and sleep duration constants, all of which are declared as named compile-time constants at the top of each source file, to produce variants suitable for specific classroom demonstrations or laboratory assessments.

## 6. Comparative Analysis

Comparing TSGL and TSGL-Rust reveals how compile-time enforcement and runtime visualization interact to shape both correctness guarantees and the architectural separation between concurrency logic and rendering.

### 6.1. Safety Guarantees

Table 7 compares the two systems with respect to core concurrency safety properties. The first row carries the most immediate pedagogical consequence. In C++ TSGL, a data race is undefined behavior. A program may compile, execute, and produce incorrect results without any warning. The presence of a race must be inferred from inconsistent output. In Rust TSGL-Rust, the same unsafe access pattern is rejected at compile time. The compiler reports the exact location and nature of the violation, requiring the structural cause to be addressed before execution begins.

**Table 7**. Safety property comparison between C++ TSGL and Rust TSGL-Rust.

| **Property** | **C++ TSGL** | **Rust TSGL-Rust** |
|---|---|---|
| Data race prevention | Runtime (undefined behavior) | Compile-time (borrow checker) |
| Deadlock prevention | None | None (logical error) |
| Memory leak prevention | Manual / `shared_ptr` | Automatic (ownership) |
| Null pointer safety | Runtime crash | Compile-time (`Option<T>`) |
| Thread safety of canvas | Library-enforced (TSGL) | Type-enforced (`Arc`) |

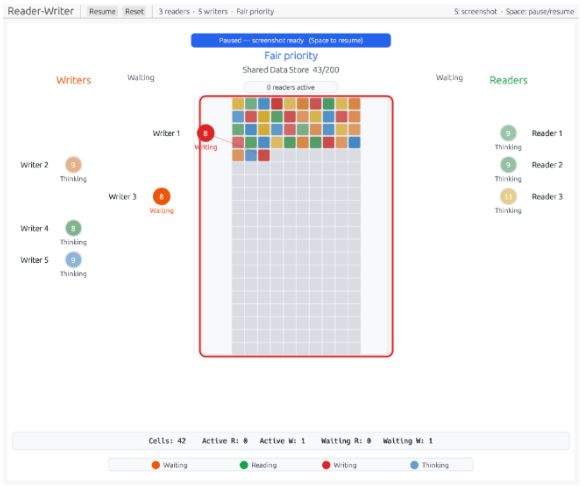
Not all classes of errors are eliminated. Deadlock remains possible in both systems. This separation matters because it distinguishes memory safety from logical correctness. Some errors are prevented by the language; others persist as design problems that require reasoning about execution order.

### 6.2. Visualization and Portability

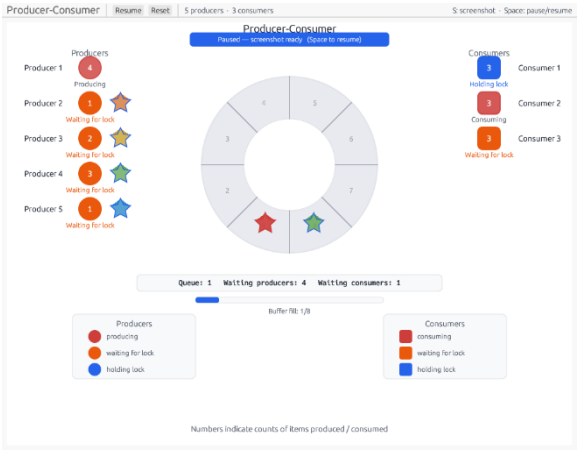
Both TSGL and TSGL-Rust provide real-time visualization of thread states, yet they differ in how easily those visualizations can be deployed and extended. TSGL relies on a custom OpenGL-based canvas, which introduces non-trivial installation requirements, particularly on Windows systems where additional graphics dependencies must be configured manually. TSGL-Rust, by contrast, builds using Cargo and the eframe framework without requiring external graphics libraries. Cloning the repository and executing a binary with a single command reduces setup overhead in laboratory environments.

Fig. 6 presents representative screenshots of the three classical synchronization problems implemented in TSGL-Rust. The visualizations expose thread states, resource availability, and blocking conditions directly, removing the need for textual debugging output to follow execution. The immediate-mode rendering model of `egui` ensures that each frame reflects the current system state without retaining outdated UI elements, which produces a direct mapping between program state and visual output.

Portability extends beyond installation. `egui` supports compilation to WebAssembly, enabling browser-based execution. A user without a local Rust toolchain could interact with the visualizations through a web interface. Such deployment is not available in TSGL's OpenGL-based design. While this capability is not yet fully exercised in the current implementation, it suggests a pathway toward broader accessibility in future iterations.




(a) reader_writer 3 5




(b) producer_consumer 5 3


(c) dining_philosophers 5

**Figure 6**. Sample Screenshots of TSGL-Rust: (a) Reader–Writer with 3 readers and 5 writers under fair priority. (b) Producer–Consumer with 5 producers and 3 consumers sharing the bounded buffer. (c) Dining Philosophers with 5 philosophers under the resource-ordering deadlock-avoidance strategy.

### 6.3. Architectural Complementarity

TSGL-Rust is not intended to replace TSGL. The two systems emphasize different points in the development cycle. TSGL exposes runtime behavior, allowing incorrect synchronization to be observed during execution. TSGL-Rust shifts part of that experience earlier by preventing certain classes of errors before the program runs.

Used in sequence, the two systems illustrate the same problem from complementary angles. A race condition that appears in C++ TSGL only as inconsistent output is rejected at compile time in Rust, with diagnostics that identify the unsafe access pattern. One system shows what goes wrong; the other constrains what can be written. Yet the combination highlights a limit rather than removing it: deadlock remains visible only at runtime in both systems, since liveness errors lie beyond the scope of static analysis. The resulting instructional affordance is a single environment in which the boundary between data-race prevention and liveness is made structurally explicit; whether and how instructors exercise that affordance lies outside the scope of the present system description.

## 7. DISCUSSION

This section discusses the design constraints that follow from the choices documented above, the practical context of the IIUM deployment, and the limitations of the current system.

### 7.1. Design Constraints and Trade-offs

Several design constraints emerge from situating TSGL-Rust within an undergraduate operating systems setting. Rust's ownership model introduces a conceptual overhead that must be in place before concurrency primitives can be used productively. When ownership and synchronization are introduced in close succession, the resulting cognitive load is substantial [23]. Borrow checker diagnostics, while precise, can initially appear opaque without prior grounding in the underlying permissions model [12]. This is not a property of TSGL-Rust as such, but a constraint inherited from the language; it shapes how and when the tool can be introduced in a course.

Deadlock remains unaffected by Rust's guarantees. The Dining Philosophers implementation makes this boundary explicit. Programs that satisfy all compile-time constraints may still fail at runtime due to incorrect lock acquisition order. The separation between memory safety and liveness properties is intrinsic to the language design, and TSGL-Rust surfaces it rather than resolves it.

On the rendering side, the immediate-mode model of egui required a redesign rather than a direct translation of TSGL visualizations. The resulting interface presents thread states and resource availability through a different visual idiom than the original OpenGL canvas. This is a design consequence of the framework choice, traded against the elimination of external graphics dependencies and the resulting portability gains.

### 7.2. Deployment Context

TSGL-Rust was developed for and deployed in the EECE 4342 Operating Systems course at IIUM, in which Rust proficiency is a stated outcome. Within that course, the tool is positioned after an ownership-focused introduction to Rust and is used during the synchronization unit alongside conventional lecture material on the three classical problems. The placement reflects the conceptual prerequisite identified above: that exposure to ownership semantics in isolation makes subsequent borrow checker diagnostics in concurrent code more interpretable. This sequencing is described here as a deployment choice rather than as a tested intervention; no controlled comparison was conducted.

**Table 8**. Comparative Characteristics of TSGL and TSGL-Rust

| **Dimension** | **TSGL (C++)** | **TSGL-Rust** |
|---|---|---|
| Error detection | Runtime only | Compile-time and runtime |
| Feedback form | Indirect (output discrepancy) | Direct (compiler diagnostic) |
| Determinism of rejection | Low (race-dependent) | Higher (compile-time rejection) |
| Reasoning supported | Temporal | Structural and temporal |

The Cargo-based build, the absence of external graphics dependencies, and the single-command execution model were all driven by deployment pragmatics in this setting. Laboratory machines run a mix of Windows, Linux, and macOS, and prior experience with TSGL on Windows highlighted the cost of platform-specific OpenGL setup. Removing that cost was a primary motivation for choosing egui/eframe.

Table 8 summarizes the comparative characteristics of TSGL and TSGL-Rust along dimensions that follow directly from these design and deployment choices. The dimensions are technical: error-detection timing, the form of feedback, the determinism of rejection, and the type of reasoning the tool supports. Claims about the effect of these characteristics on student learning lie outside the scope of this paper and would require a separate, controlled study to establish.

### 7.3. Limitations

The contribution of this paper is a system and a design. Several limitations follow from that scope and should be stated plainly. No controlled experiment was conducted. No quantitative learning outcomes were measured. Comparisons with TSGL are technical and structural, not empirical, and any references to instructional context describe deployment circumstances rather than tested effects.

The visualization style differs from the original TSGL due to a change in rendering framework, and a side-by-side perceptual comparison was not conducted. The WebAssembly deployment path is supported by the egui toolchain in principle, but has not been exercised end-to-end in the current implementation. Coverage is restricted to the three canonical problems addressed; other classical problems, such as the Sleeping Barber or the Cigarette Smokers, would require additional binaries following the same architectural pattern.

Finally, the boundary that TSGL-Rust makes visible, i.e., the line between data-race prevention and liveness, is a property of Rust and of the chosen synchronization primitives. The system surfaces this boundary; it does not extend it. Tools such as Progvis [14], which integrate model checking, sit on the other side of that line and would complement TSGL-Rust in a curriculum that aims to address liveness systematically. Combining the two would be an interesting direction for future work, but it is not attempted here.

## 8. CONCLUSIONS

TSGL-Rust is a Rust-based reimplementation of the Producer–Consumer, Reader–Writer, and Dining Philosophers visualizations. It preserves the temporal visibility of the original TSGL and combines it with compile-time data-race prevention, yielding a single environment in which compiler diagnostics and runtime visualization operate side by side.

The technical contribution is threefold. First, an architecture that decouples concurrency logic from rendering through a channel between worker threads and the immediate-mode render loop, i.e., a separation that the original shared-canvas design cannot provide. Second, an explicit mapping from C++ synchronization primitives to their Rust equivalents, with the synchronization requirement encoded in the type rather than enforced by discipline. Third, a build path that requires no external graphics dependencies, enabling consistent execution across Windows, Linux, and macOS without platform-specific OpenGL setup.

From these properties follows an instructional affordance rather than a measured effect. By rejecting unsafe access patterns at compile time and visualizing the runtime behavior of programs that survive the borrow checker, TSGL-Rust makes the boundary between data-race prevention and liveness structurally visible, with deadlock deliberately exposed in the Dining Philosophers binary as a logical error beyond the reach of the type system.

Whether this affordance translates into learning outcomes is an empirical question this paper does not answer. A controlled study comparing comprehension and debugging behavior under matched conditions is the principal direction for future work, alongside end-to-end

WebAssembly deployment, integration with model-checking tools that address liveness, and extension toward concurrency primitives in deep learning training and inference pipelines, where the same producer–consumer and reader–writer structures recur at a different layer of the stack.

## ACKNOWLEDGMENT

This work is partially supported by and also conceptually aligned with ongoing research under the AOARD project FA2386-25-1-4061 and SPI25-295-0295, "Adaptive Physics-Guided Transformer with Explainable Attention for Atmospheric Turbulence Mitigation," particularly in relation to concurrent data acquisition, inference, and visualization pipelines. The author thanks the students of the EECE 4342 Operating Systems course at IIUM for their engagement during the development and classroom deployment of TSGL-Rust. The author used generative AI tools (ChatGPT and Claude) solely for language refinement and structural editing during manuscript preparation; all AI-assisted text was reviewed and verified by the authors, and no AI was used to generate research data, results, or substantive technical content.

## REFERENCES

[1] A. N. Kumar *et al.*, "Computer science curricula 2023," ed: ACM, 2024.
[2] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: A case study at Google," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, 2018, pp. 181-190.
[3] J. C. Adams *et al.*, "TSGL: A tool for visualizing multithreaded behavior," *Journal of Parallel and Distributed Computing,* vol. 118, pp. 233-246, 2018.
[4] S. Klabnik and C. Nichols, *The Rust programming language*. No Starch Press, 2023.
[5] H.-H. Dang, J.-H. Jourdan, J.-O. Kaiser, and D. Dreyer, "RustBelt meets relaxed memory," *Proceedings of the ACM on Programming Languages,* vol. 4, no. POPL, pp. 1-29, 2019.
[6] J. C. Castro-Alonso, P. Ayres, and J. Sweller, "Instructional visualizations, cognitive load theory, and visuospatial processing," in *Visuospatial processing for education in health and natural sciences*: Springer, 2019, pp. 111-143.
[7] R. Duran, A. Zavgorodniaia, and J. Sorva, "Cognitive load theory in computing education research: A review," *ACM Transactions on Computing Education (TOCE),* vol. 22, no. 4, pp. 1-27, 2022.
[8] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "Safe systems programming in Rust," *Communications of the ACM,* vol. 64, no. 4, pp. 144-152, 2021.
[9] S. Panter and N. Eisty, "Rusty Linux: Advances in Rust for Linux kernel development," in *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2024, pp. 496-502.
[10] W. House, "Back to the building blocks: A path toward secure and measurable software," Office of the National Cyber Director, Executive Office of the President, 2024.
[11] E. Ernerfeldt. "egui: an easy-to-use GUI in pure Rust." https://github.com/emilk/egui (accessed 18 April 2026).
[12] W. Crichton, G. Gray, and S. Krishnamurthi, "A grounded conceptual model for ownership types in rust," *Proceedings of the ACM on Programming Languages,* vol. 7, no. OOPSLA2, pp. 1224-1252, 2023.
[13] K. R. C. Siek, "An Undergraduate Computer Systems Curriculum Using Rust," Virginia Tech, 2025.
[14] F. Strömbäck, L. Mannila, and M. Kamkar, "Pilot study of progvis: A visualization tool for object graphs and concurrency via shared memory," in *Proceedings of the 24th Australasian Computing Education Conference*, 2022, pp. 123-132.
[15] F. Strömbäck, L. Mannila, and M. Kamkar, "A weak memory model in Progvis: Verification and improved accuracy of visualizations of concurrent programs to aid student learning," in

*Proceedings of the 22nd Koli Calling International Conference on Computing Education Research*, 2022, pp. 1-12.
[16] F. Strömbäck, L. Mannila, and M. Kamkar, "Using Model-Checking and Peer-Grading to Provide Automated Feedback to Concurrency Exercises in Progvis," in *Proceedings of the 25th Australasian Computing Education Conference*, 2023, pp. 11-20.
[17] W. Stallings, *Operating Systems: Internals and Design Principles*. Pearson Education Limited, 2018.
[18] A. S. Tanenbaum, "Modern Operating Systems, 4th Edition," ed: Pearson, 2015.
[19] J. C. Adams, E. R. Koning, and C. D. Hazlett, "Visualizing classic synchronization problems: Dining philosophers, producers-consumers, and readers-writers," in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 2019, pp. 934-940.
[20] P. Yosifovich, M. E. Russinovich, A. Ionescu, and D. A. Solomon, Windows Internals, Part 1: System architecture, processes, threads, memory management, and more, 7th Edition. Microsoft Press, Pearson, 2017.
[21] A. Allievi, M. E. Russinovich, A. Ionescu, and D. A. Solomon, *Windows Internals, Part 2, 7th Edition*. Microsoft Press, Pearson, 2021.
[22] K. N. Billimoria, Linux Kernel Programming: A comprehensive and practical guide to kernel internals, writing modules, and kernel synchronization. Packt Publishing Ltd, 2024.
[23] J. Sweller, "Cognitive load theory and educational technology," *Educational technology research and development,* vol. 68, no. 1, pp. 1-16, 2020.