Rust

- Synchronization Primitives Guide
- Rust Declarative Macros

Synchronization Primitives Guide

Rust & Tokio Synchronization Primitives Guide

When building concurrent or asynchronous applications in Rust, choosing the right synchronization primitive is critical for performance and safety. Below is a comprehensive guide to the standard library (`std`) and Tokio synchronization tools.

Quick Comparison Table

Primitive	Ecosystem	Concurrency Pattern	Best Use Case
`std::sync::Arc`	Std	Shared Ownership	Sharing memory safely across multiple threads. Almost always wraps another primitive (e.g., `Arc<Mutex<T>>`).
`std::sync::Mutex`	Std	Exclusive Access	Brief, synchronous locks (pushing to a `Vec`, updating a number). **Never hold across an `.await`.**
`tokio::sync::Mutex`	Tokio	Async Exclusive Access	When you absolutely *must* hold a lock while waiting for an asynchronous operation (across an `.await`).
`std::sync::RwLock`	Std	Multi-Read / Single-Write	High-Read / Low-Write patterns. Allows dozens of threads to read simultaneously without blocking.

Primitive	Ecosystem	Concurrency Pattern	Best Use Case
`tokio::sync::watch`	Tokio	SPMC State Broadcast	Broadcasting the *latest* configuration or status. Only the newest value is kept; slow receivers drop old data.
`tokio::sync::broadcast`	Tokio	MPMC Event Stream	Pub/Sub systems like chat rooms where multiple listeners must receive *every* message in order.

---

1. The Standard `Arc<Mutex<T>>` (Synchronous Mutability)

Use `std::sync::Mutex` when you need to safely mutate data across threads, and the mutation is fast and synchronous.

Rule of Thumb: It is significantly faster than Tokio's Mutex. Use it as long as you do not call `.await` while the lock is locked.

```
use std::sync::{Arc, Mutex};
use std::thread;

// Wrap our data in a Mutex (for safe mutation) and Arc (for safe sharing)
let counter = Arc::new(Mutex::new(0));
let mut handles = vec![];

for _ in 0..10 {
    let counter_clone = Arc::clone(&counter);

    let handle = thread::spawn(move || {
        // 1. Lock the mutex
        let mut num = counter_clone.lock().unwrap();
        // 2. Modify the value quickly
        *num += 1;
        // 3. Lock is automatically released here when `num` goes out of scope
    });

    handles.push(handle);
}
```

```
for handle in handles {
    handle.join().unwrap();
}

println!("Final count: {}", *counter.lock().unwrap());
```

---

2. The `tokio::sync::Mutex` (Async Mutability)

Use Tokio's Mutex **only** when your lock must remain active while the thread yields to the async runtime (i.e., across an `.await` boundary). It is heavier because it requires internal bookkeeping to allow other tasks to run on the thread while the current task sleeps.

```
use std::sync::Arc;
use tokio::sync::Mutex;
use tokio::time::{sleep, Duration};

#[tokio::main]
async fn main() {
    let shared_state = Arc::new(Mutex::new(String::from("Idle")));

    let state_clone = Arc::clone(&shared_state);
    tokio::spawn(async move {
        // Acquire the async lock
        let mut state = state_clone.lock().await;
        *state = String::from("Fetching Data...");

        // SAFE: Holding a Tokio Mutex across an .await point
        sleep(Duration::from_secs(2)).await;

        *state = String::from("Complete");
    });
}
```

---

3. The `Arc<RwLock<T>>` (High-Read / Low-Write)

If you have a configuration struct or global state that is read constantly by many UI components or threads, but only updated occasionally, `RwLock` maximizes performance by allowing parallel reads.

```
use std::sync::{Arc, RwLock};
use std::thread;

let config = Arc::new(RwLock::new(vec!["dark_mode", "auto_save"]));

// READER THREAD 1
let config_read1 = Arc::clone(&config);
thread::spawn(move || {
    let read_guard = config_read1.read().unwrap();
    println!("Thread 1 sees: {:?}", *read_guard);
});

// READER THREAD 2 (Can run at the exact same time as Thread 1)
let config_read2 = Arc::clone(&config);
thread::spawn(move || {
    let read_guard = config_read2.read().unwrap();
    println!("Thread 2 sees: {:?}", *read_guard);
});

// WRITER THREAD (Will block until all readers are finished)
let config_write = Arc::clone(&config);
thread::spawn(move || {
    let mut write_guard = config_write.write().unwrap();
    write_guard.push("telemetry_enabled");
});
```

---

4. Tokio Watch Channel (`tokio::sync::watch`)

A highly optimized Single-Producer, Multi-Consumer (SPMC) channel. Perfect for broadcasting state where receivers only care about the *latest* data.

```
use tokio::sync::watch;

#[tokio::main]
async fn main() {
    // Initialize with a default state
    let (tx, mut rx1) = watch::channel("Offline");

    // Receivers can be cloned incredibly cheaply
    let mut rx2 = rx1.clone();

    tokio::spawn(async move {
        // Wait for the state to change
        while rx1.changed().await.is_ok() {
            println!("UI Component 1 Update: {}", *rx1.borrow());
        }
    });

    tokio::spawn(async move {
        while rx2.changed().await.is_ok() {
            println!("UI Component 2 Update: {}", *rx2.borrow());
        }
    });

    // Sender updates the state instantly
    tx.send("Connecting...").unwrap();
    tx.send("Online").unwrap();
}
```

Rust Declarative Macros

Rust Declarative Macros: A Guide to `macro_rules!`

Declarative macros, defined using `macro_rules!`, are a powerful feature in Rust that allow you to write code that generates other code. They work by pattern matching on Rust token streams and replacing match arms with expanded code templates.

---

1. Defining a Simple Macro

A basic declarative macro uses pattern matching (similar to a `match` statement) to inspect the code tokens passed to it:

```
macro_rules! say_hello {
    () => {
        println!("Hello, World!");
    };
    ($name:expr) => {
        println!("Hello, {}!", $name);
    };
}
```

---

2. Exporting a Macro

If you want to define a macro inside a module and use it in a different module within the same crate, standard use statements will not work for local macro_rules! unless you use one of two patterns:

Option A: The #[macro_use] Pattern (Textual Scope)

This is the classic way to make a module's macros visible to the rest of your crate. You annotate the module declaration with #[macro_use]:

```
#[macro_use]
mod my_module; // This makes all macros inside `my_module` visible downstream
```

The Catch: This relies on textual order. The macro will only be available in files or modules declared after mod my_module in your main.rs or lib.rs file.

Option B: The Modern Path Import (Rust 2018+)

In modern Rust editions, if you tag a local macro with #[macro_export], it is hoisted to your own crate root. You can then import it anywhere in the same crate using a standard path-based import:

```
use crate::my_macro; // Works anywhere in the crate
```

Key Behavior: Root Namespace Placement

When a macro is tagged with `#[macro_export]`, Rust automatically places it at the **root namespace** of your crate [22, 38]. It does not matter how deeply nested within submodules the macro file is originally defined; it will always be exposed at the crate root level [22, 38].

---

3. Crucial Best Practice: Macro Hygiene (`$crate`)

When you export a macro, you lose control over what dependencies, imports, or local items exist in the downstream scope where the macro is eventually invoked [22, 38]. If your macro relies on standard library types, external crates, or internal library items using plain, unqualified names, it will fail to compile downstream unless the consumer has explicitly imported them [11, 22, 38].

To make your macro robust and hygienic, you must use **fully-qualified absolute paths** [11, 22, 38].

Standard Library and External Crates

Always use absolute paths starting with `::` for external crates or standard library items [11, 22]:

- ❌ `Vec::new()` (will fail if the caller hasn't imported or has shadowed `Vec`)
- ✅ `::std::vec::Vec::new()` (hygienic and bulletproof)

Referencing Internal Library Items

If your macro needs to reference items, structs, helper functions, or traits from your own library crate (such as a custom `Result` type), you **cannot** use standard relative paths or the invalid `::crate` syntax [22, 29]. Instead, you must use the **`$crate`** meta-variable [22, 29, 38].

The `$crate` meta-variable automatically expands to the absolute path of your library, ensuring it resolves successfully from external targets [22, 38].

❌ Fragile & Broken Approach

```
// In your library
#[macro_export]
macro_rules! serde_binary {
    ($id:ident) => {
        impl $id {
            // This will fail downstream because 'Result' and 'BinarySerde'
            // are referenced as unqualified or invalid '::crate' paths.
            pub fn to_binary(&self) -> Result<Vec<u8>> {
                ::crate::BinarySerde::serialize(self)
            }
        }
    };
}
```

✅ Robust & Hygienic Approach

```
// In your library
#[macro_export]
macro_rules! serde_binary {
    ($id:ident) => {
        impl $id {
            // Using $crate ensures these resolve directly back to your library crate
namespace,
            // and using ::std::vec::Vec ensures the standard vector is used hygienically.
            pub fn to_binary(&self) -> $crate::error::Result<::std::vec::Vec<u8>> {
```

```
                $crate::BinarySerde::serialize(self)
            }
        }
    };
}
```

---

4. How to Consume (Import) the Macro

Once your library has properly exported the macro, downstream crates or other targets in your workspace can import and consume it.

Method A: Direct Item Import (Recommended)

You can bring the macro into scope using a standard `use` statement targeting the crate root, just like any other item [22, 38]:

```
// In a downstream crate or binary
use my_library::serde_binary;

#[derive(serde::Serialize)]
struct UserSession {
    username: String,
}

// Invoke the macro
serde_binary!(UserSession);
```

Method B: Global Prelude Pattern (Legacy)

Alternatively, if you are using the legacy or workspace-wide prelude approach, you can annotate the external crate import with `#[macro_use]` to pull all exported macros from that crate into the global scope [22, 38]:

```
#[macro_use]
extern crate my_library;

// Now `serde_binary!` is globally available in this crate without an explicit import.
serde_binary!(UserSession);
```

---

5. Alternative Architectural Design: Traits

While generating arbitrary inherent `impl` blocks via macros (as shown in the `serde_binary!` example above) is functional, it has a few architectural drawbacks:

1. It clutters auto-generated documentation [22, 38].
2. It can cause naming conflicts if another module or trait tries to implement a method with the same name on the same type [22, 38].

Instead, a highly recommended alternative is to define a **Trait** and use your macro strictly to implement that trait quickly [22, 38]. This provides cleaner architecture, is more self-documenting, and allows you to enforce generic bounds down the road [22, 38].

Defining the Trait and Macro Implementation

```
pub trait BinarySerializable {
    fn to_binary(&self) -> $crate::error::Result<::std::vec::Vec<u8>>;
}

#[macro_export]
macro_rules! impl_binary_serializable {
    ($id:ident) => {
        impl $crate::BinarySerializable for $id {
            fn to_binary(&self) -> $crate::error::Result<::std::vec::Vec<u8>> {
                $crate::BinarySerde::serialize(self)
            }
        }
```

```
    };
}
```

Using this approach, you can now write generic helper functions like `fn save_to_disk<T: BinarySerializable>(obj: T)` while keeping your APIs highly structured and type-safe [22, 38].