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`).

Primitive	Ecosystem	Concurrency Pattern	Best Use Case
`std::sync::RwLock`	Std	Multi-Read / Single-Write	High-Read / Low-Write patterns. Allows dozens of threads to read simultaneously without blocking.
`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();
}
```

Revision #1
Created 2026-07-17 15:51:59 UTC by David Pires
Updated 2026-07-17 16:11:48 UTC by David Pires