Parallel Programming in Rust

Lukáš Hozda

2025-04-10

Braiins Systems s.r.o

Language Model & Fundamentals

Data Race Definition

- Multiple threads accessing shared data
- At least one thread is writing
- No synchronization between accesses
- Undefined behavior in unsafe code
- Impossible in safe Rust

Race Conditions vs Data Races

- Race conditions: Logic-level timing issues
- Safe Rust allows race conditions
- Race conditions cannot violate memory safety alone
- Only unsafe code + race condition can break safety

Example: Safe Race Condition

```rust
use std::thread;
use std::sync::atomic::{AtomicUsize, Ordering};
use std::sync::Arc;

let data = vec![1, 2, 3, 4];
let idx = Arc::new(AtomicUsize::new(0));
let other_idx = idx.clone();

thread::spawn(move || {
    other_idx.fetch_add(10, Ordering::SeqCst);
});

// Race condition: May panic, but memory safe
println!("{}", data[idx.load(Ordering::SeqCst)]);
```

Example: Unsafe Race Condition

```rust
use std::thread;
use std::sync::atomic::{AtomicUsize, Ordering};
use std::sync::Arc;

let data = vec![1, 2, 3, 4];
let idx = Arc::new(AtomicUsize::new(0));
let other_idx = idx.clone();

thread::spawn(move || {
    other_idx.fetch_add(10, Ordering::SeqCst);
});

if idx.load(Ordering::SeqCst) < data.len() {
    unsafe { // UNSAFE: idx could change after bounds check
        println!("{}", data.get_unchecked(idx.load(Ordering::SeqCst)));
    }
}
```

Rust's Memory Model

- Single writer XOR multiple readers
- Enforced at compile time
- Extends to concurrent code
- Built around preventing data races

Send and Sync Traits

- Send: Safe to transfer between threads
- Sync: Safe to share reference between threads
- Automatically derived for most types
- Unsafe traits - incorrect impl causes UB
- T is Sync if and only if &T is Send

Notable Send/Sync Types

Not Send or Sync:

- Raw pointers
- Rc (refcount not threadsafe)
- Non-threadsafe cells (UnsafeCell is Send, but not Sync)

Send and Sync:

- Most standard collections
- Arc<T> where T: Send + Sync
- Atomic types

Thread Basics

- OS-level threads
- Guaranteed preemptive multitasking
- Heavier than async tasks
- Can use all CPU cores

```rust
use std::thread;

let handle = thread::spawn(|| {
   // New thread
   println!("Hello from thread!");
});

handle.join().unwrap(); // Wait for completion
```

Moving Values Into Threads

- Closure must be ‘static
- Use move keyword
- Values are transferred between threads

```
let v = vec![1, 2, 3];

let handle = thread::spawn(move || {
   println!("Vector: {:?}", v);
}); // v is moved into thread

// Cannot use v here anymore
handle.join().unwrap();
```

Scoped Threads

- Allows borrowing from parent scope
- Guarantees threads finish before scope ends
- More flexible than regular threads

```rust
let mut v = vec![1, 2, 3];
thread::scope(|s| {
    s.spawn(|| {
        // Can borrow v here!
        println!("First: {:?}", &v);
    });
    s.spawn(|| {
        v.push(4); // But we cannot mutate here!
    });
});
v.push(5); // We can mutate here :)
```

Sharing Immutable Data

- Arc = Atomic Reference Counting
- Thread-safe shared ownership
- Clone = increment reference count
- Drop = decrement reference count

```
use std::sync::Arc;

let data = Arc::new(vec![1, 2, 3]);

let data_clone = Arc::clone(&data);
thread::spawn(move || {
    println!("Thread sees: {:?}", *data_clone);
});
```

Arc vs Rc

- Rc = single-threaded reference counting
- Arc = atomic (thread-safe) reference counting
- Arc has small performance overhead
- Cannot convert between them

```
use std::rc::Rc;       // Wrong!
use std::sync::Arc;    // Correct!

let bad = Rc::new(42);
// thread::spawn(move || {  // Compile error!
//     println!("{}", bad);
// });

let good = Arc::new(42);
thread::spawn(move || {     // Works!
    println!("{}", good);
});
```

Shared Mutable State

Mutex<T>

- Mutual exclusion
- Only one thread can access data at once
- Others must wait
- RAII-style locking

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

let counter = Arc::new(Mutex::new(0));
let counter2 = Arc::clone(&counter);

thread::spawn(move || {
    let mut num = counter2.lock().unwrap();
    *num += 1;
}); // lock automatically released here
```

parking_lot::Mutex

- Smaller and faster than std::sync::Mutex
- No poisoning on panic
- More efficient spinning
- Platform-specific optimizations

```rust
use parking_lot::Mutex;

let mutex = Mutex::new(0);
{
    let mut guard = mutex.lock(); // No unwrap needed!
    *guard += 1;
} // lock released here
```

RwLock - Reader-Writer Lock

- Multiple readers OR single writer
- Good for read-heavy workloads
- Higher overhead than Mutex

```rust
use std::sync::RwLock;

let data = RwLock::new(vec![1, 2, 3]);

// Multiple readers
let r1 = data.read().unwrap();
let r2 = data.read().unwrap();

// Only one writer
let mut w = data.write().unwrap();
w.push(4);
```

Deadlock Prevention

- Lock ordering matters
- Try to acquire locks in same order
- Use scope-based RAII locks, shortest possible lifetime
- Nested lock problems (use ReentrantMutex from parking_lot)

```rust
let mutex1 = Mutex::new(0);
let mutex2 = Mutex::new(0);

// Potential deadlock!
let _guard1 = mutex1.lock().unwrap();
let _guard2 = mutex2.lock().unwrap();

// Better: always lock in same order
// across all code paths
```

Condition Variables

- Wait for condition to become true
- Avoids busy waiting
- Always paired with mutex
- Allows thread park/wake

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

let pair = Arc::new((Mutex::new(false), Condvar::new()));
let pair2 = Arc::clone(&pair);

thread::spawn(move || {
    let (lock, cvar) = &*pair2;
    let mut started = lock.lock().unwrap();
    *started = true;
    cvar.notify_one();
});
let (lock, cvar) = &*pair;
let mut started = lock.lock().unwrap();
while !*started {
    started = cvar.wait(started).unwrap();
}
```

Memory Models and Atomics

C++ Memory Model Inheritance

- Rust uses C++20 memory model
- Pragmatic choice for tooling
- Based on happens-before relationships
- Bridges hardware and software needs

Compiler Reordering

- Compilers optimize aggressively
- May change execution order
- Example transformation:

```
// Original code
x = 1;
y = 3;
x = 2;

// What compiler might do
x = 2;
y = 3;
```

Hardware Memory Models

- x86/64: Strongly ordered
  - Most operations implicitly synchronized
  - Cheaper to provide strong guarantees
  - Might hide incorrect sync code
- ARM/RISC: Weakly ordered
  - Operations freely reordered
  - Explicit barriers needed
  - Better tests concurrent code

Memory Access Types

- Data accesses
  - Unsynchronized
  - Can be reordered
  - Cause data races
  - Cannot be used for synchronization
- Atomic accesses
  - Thread-aware
  - Ordering guarantees
  - Safe for synchronization
  - Various strength levels

Memory Ordering Levels

- SeqCst (Strongest)
  - Global operation order
  - All threads agree
  - Most expensive
  - Safe default choice
- Release-Acquire
  - Paired operations
  - Good for locks
  - Prior writes visible
  - Cheaper than SeqCst
- Relaxed (Weakest)
  - Only atomicity
  - No synchronization
  - Good for counters
  - Best performance

Sequentially Consistent (SeqCst)

```rust
use std::sync::atomic::{AtomicBool, Ordering};

static FLAG: AtomicBool = AtomicBool::new(false);

// All threads will agree on operation order
FLAG.store(true, Ordering::SeqCst);
let x = FLAG.load(Ordering::SeqCst);
```

Release-Acquire Example

```rust
use std::sync::atomic::{AtomicBool, Ordering};

static LOCK: AtomicBool = AtomicBool::new(false);

// Thread 1: Release ensures all writes visible
LOCK.store(false, Ordering::Release);

// Thread 2: Acquire sees released writes
while LOCK.compare_exchange(
    false, true,
    Ordering::Acquire,
    Ordering::Relaxed
).is_err() {}
```

Relaxed Atomics

```rust
use std::sync::atomic::{AtomicUsize, Ordering};

static COUNTER: AtomicUsize = AtomicUsize::new(0);

// Simple counter, no synchronization needed
thread::spawn(move || {
    COUNTER.fetch_add(1, Ordering::Relaxed);
});
```

Memory Fences

- CPU instruction enforcing memory order
- Prevents reordering across fence
- Types:
  - LoadLoad: Orders loads before/after
  - StoreStore: Orders stores before/after
  - LoadStore: Orders loads before stores
  - StoreLoad: Strongest, orders all access

```rust
use std::sync::atomic::fence;
use std::sync::atomic::Ordering;

// Ensure all previous stores visible
fence(Ordering::Release);

// Strongest fence, ensures all ordering
fence(Ordering::SeqCst);
```

Message Passing

Channel Basics

- MPSC = Multiple Producer, Single Consumer
- Thread-safe message queue
- Send values between threads
- Built into standard library

```
use std::sync::mpsc::channel;

let (tx, rx) = channel();

thread::spawn(move || {
    tx.send(42).unwrap();
});

println!("Got: {}", rx.recv().unwrap());
```

Channel Types

- channel() - Unbounded sync
- sync_channel(n) - Bounded sync
- Bounded channels block when full
- Choose based on backpressure needs
- Many types of channels (spsc, mpsc, etc.)

```rust
use std::sync::mpsc::{sync_channel, channel};

// Unbounded
let (tx1, rx1) = channel();

// Bounded to 5 messages
let (tx2, rx2) = sync_channel(5);

// Will block if buffer full
tx2.send(42).unwrap();
```

Flume

- Drop-in replacement for std::sync::mpsc
- Better performance
- Bounded/unbounded variants
- Select operation support

```rust
use flume::unbounded;

let (tx, rx) = unbounded();

thread::spawn(move || {
    tx.send("Hello from flume!").unwrap();
});

assert_eq!(rx.recv().unwrap(), "Hello from flume!");
```

Real World Patterns

Producer-Consumer

- Common concurrent pattern
- Multiple producers feed queue
- Single consumer processes items

```rust
let (tx, rx) = flume::unbounded();
// Spawn multiple producers
for i in 0..3 {
    let tx = tx.clone();
    thread::spawn(move || {
        tx.send(format!("msg from {}", i)).unwrap();
    });
}
// Single consumer
for msg in rx.iter() {
    println!("Got: {}", msg);
}
```

Work Stealing

- Dynamic task distribution
- Threads steal work when idle
- Good for variable workloads

```rust
use crossbeam_deque::{Worker, Stealer};
let w = Worker::new_fifo();
let s = w.stealer();
// Main thread produces work
w.push(42);
// Other thread steals work
thread::spawn(move || {
    while let Some(work) = s.steal() {
        println!("Stolen: {}", work);
    }
});
```

Thread Pools

- Reuse threads instead of creating new
- Better performance
- Control system resource usage
- Common in real applications

```rust
use rayon::prelude::*;

let data: Vec<_> = (0..100).collect();

// Parallel iterator using thread pool
let sum: i32 = data.par_iter()
    .map(|x| x * x)
    .sum();
```

Advanced Patterns

Memory Ordering

- Atomic operations have ordering guarantees
- Different levels of synchronization
- Performance vs. strictness tradeoff

```rust
use std::sync::atomic::{AtomicBool, Ordering};

static STOP: AtomicBool = AtomicBool::new(false);

// Relaxed - fastest, weakest guarantees
STOP.store(true, Ordering::Relaxed);

// Release-Acquire - synchronizes data
STOP.store(true, Ordering::Release);
let must_stop = STOP.load(Ordering::Acquire);

// SeqCst - strongest, slowest
STOP.store(true, Ordering::SeqCst);
```

Barrier Synchronization

- Coordinate multiple threads
- Wait for all threads to reach point
- Useful for phased computations

```rust
use std::sync::{Arc, Barrier};

let barrier = Arc::new(Barrier::new(3));

for _ in 0..3 {
    let b = Arc::clone(&barrier);
    thread::spawn(move || {
        println!("before barrier");
        b.wait();  // Wait for others
        println!("after barrier");
    });
}
```

Lock-Free Data Structures

- No mutex required
- Uses atomic operations
- Higher performance potential
- More complex to implement

```rust
use crossbeam_queue::ArrayQueue;

let queue = ArrayQueue::new(100);

// Producer
queue.push(42).unwrap();

// Consumer
while let Some(item) = queue.pop() {
    println!("Got: {}", item);
}
```

Custom Sync Types

- Implement Send and Sync traits
- Make your types thread-safe
- Careful with unsafe code

```rust
struct MyThreadSafeType {
    data: Arc<Mutex<Vec<i32>>>,
}

// Automatically Send + Sync because
// all fields are Send + Sync
impl MyThreadSafeType {
    fn push(&self, value: i32) {
        self.data.lock().unwrap().push(value);
    }
}
```

Best Practices

Choose Right Tool

- Channels for message passing
- Mutex for shared state
- Atomic for simple values
- Thread pools for CPU work

```
// Bad: Mutex for single counter
let counter = Arc::new(Mutex::new(0));

// Better: Atomic for counter
let counter = Arc::new(AtomicUsize::new(0));
```

Performance Considerations

- Measure before optimizing
- Consider contention
- Right granularity of locks
- Cache coherency effects

```rust
// Bad: Too fine-grained
let numbers: Vec<_> = (0..1000)
    .map(|i| Arc::new(Mutex::new(i)))
    .collect();

// Better: Coarser granularity
let numbers = Arc::new(Mutex::new(
    (0..1000).collect::<Vec<_>>()
));
```

Questions?