Advanced Rust (2026): Parallel Programming, Atomics

Lecture 1

Lukáš Hozda

Spring 2026

MFF CUNI

Foundations

Data Race vs Race Condition

- Data race: conflicting memory access without synchronization.
- At least one access must be a write.
- Race condition: timing-dependent logic behavior.
- Safe Rust prevents the first, not the second.

Safe Race Condition

- This is still a bug, but it stays memory-safe.
- Correct fix is at the program logic level, not `unsafe`.

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

let data = vec![10, 20, 30, 40];
let idx = Arc::new(AtomicUsize::new(0));
let idx2 = Arc::clone(&idx);

thread::spawn(move || idx2.store(99, Ordering::SeqCst));

if let Some(value) = data.get(idx.load(Ordering::SeqCst)) {
    println!("{value}");
} else {
```

Safe Race Condition

```
    println!("out-of-range index");
}
```

Unsafe TOCTOU

- TOCTOU: check and use are separated in time.
- `unsafe` may convert a logic race into UB.

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

let data = vec![10, 20, 30, 40];
let idx = Arc::new(AtomicUsize::new(0));
let idx2 = Arc::clone(&idx);

thread::spawn(move || idx2.store(99, Ordering::SeqCst));

if idx.load(Ordering::SeqCst) < data.len() {
    unsafe {
        println!("{}", data.get_unchecked(idx.load(Ordering::SeqCst)));
```

Unsafe TOCTOU

```
        }
}
```

Send + Sync

- Thread-safety is expressed through trait bounds.
- The compiler rejects non-thread-safe ownership patterns.

```rust
use std::rc::Rc;
use std::sync::Arc;

fn assert_send_sync<T: Send + Sync>() {}

assert_send_sync::<Arc<Vec<u8>>>();
// assert_send_sync::<Rc<Vec<u8>>>(); // does not compile
```

`move` to Thread

- `move` transfers ownership into the closure.
- Closure environment must satisfy thread constraints.

```rust
use std::thread;

let values = vec![1, 2, 3, 4];

let handle = thread::spawn(move || values.iter().sum::<i32>());
println!("sum = {}", handle.join().unwrap());
```

Scoped Threads

- `thread::scope` enables borrowing stack data safely.
- No need for `'static` capture in this pattern.

```rust
use std::thread;

let values = vec![1_i64, 2, 3, 4, 5, 6, 7, 8];
let mid = values.len() / 2;
let (left, right) = values.split_at(mid);

let total = thread::scope(|s| {
    let a = s.spawn(|| left.iter().sum::<i64>());
    let b = s.spawn(|| right.iter().sum::<i64>());
    a.join().unwrap() + b.join().unwrap()
});

println!("total = {total}");
```

Shared State

Mutex Counter I

- Canonical shared mutable state pattern.
- First step before trying lock-free alternatives.

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

let counter = Arc::new(Mutex::new(0_i64));
let mut handles = Vec::new();

for _ in 0..8 {
    let c = Arc::clone(&counter);
    handles.push(thread::spawn(move || {
        for _ in 0..100_000 {
            *c.lock().unwrap() += 1;
        }
    }));
}
```

Mutex Counter II

- Join all workers before final observation.
- Final read remains synchronized.

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

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

Poisoning

- Panic while locked poisons `std::sync::Mutex`.
- Poisoning protects invariant boundaries by default.

```rust
use std::sync::Mutex;

let value = Mutex::new(1_i32);

let _ = std::thread::scope(|s| {
    s.spawn(|| {
        let mut g = value.lock().unwrap();
        *g = 10;
        panic!("failed while holding lock");
    });
});

let mut g = value.lock().unwrap_or_else(|e| e.into_inner());
```

Poisoning

```rust
*g += 1;
println!("{}", *g);
```

RwLock

- Multiple readers, single writer.
- Throughput depends on read/write ratio.

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

let state = Arc::new(RwLock::new(vec![10, 20, 30]));
let s1 = Arc::clone(&state);
let s2 = Arc::clone(&state);

let r1 = thread::spawn(move || s1.read().unwrap().iter().sum::<i32>());
let r2 = thread::spawn(move || s2.read().unwrap().iter().sum::<i32>());

state.write().unwrap().push(40);
println!("{}, {}", r1.join().unwrap(), r2.join().unwrap());
```

Condvar

- Condition variables model state transitions.
- They avoid polling and busy-wait loops.

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

let shared = Arc::new((Mutex::new(false), Condvar::new()));
let shared2 = Arc::clone(&shared);

thread::spawn(move || {
    let (lock, cv) = &*shared2;
    *lock.lock().unwrap() = true;
    cv.notify_one();
});

let (lock, cv) = &*shared;
let mut ready = lock.lock().unwrap();
```

Condvar

```rust
while !*ready {
    ready = cv.wait(ready).unwrap();
}
```

Deadlock Example

- Opposite lock order can deadlock.
- The error is structural, not random bad luck.

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

let a = Arc::new(Mutex::new(0));
let b = Arc::new(Mutex::new(0));

let (a1, b1) = (Arc::clone(&a), Arc::clone(&b));
let _t1 = thread::spawn(move || {
    let _ga = a1.lock().unwrap();
    let _gb = b1.lock().unwrap();
});

let (a2, b2) = (Arc::clone(&a), Arc::clone(&b));
let _t2 = thread::spawn(move || {
```

Deadlock Example

```
        let _gb = b2.lock().unwrap();
        let _ga = a2.lock().unwrap();
    });
```

Deadlock Rule

- Keep a single global lock order.
- Never invert that order in another path.
- Keep lock lifetimes short.
- Reduce nested lock depth where possible.

Memory Model

C++20 Memory Model Inheritance

- Rust atomics follow the C++20 model.
- Core relation is happens-before ("A must be visible before B").
- Reasoning is graph-like: operations are nodes, ordering rules are edges.
- Hardware and compiler optimizations must obey this model.

Compiler Reordering

- Compilers may legally reorder non-atomic operations.
- Source order is not execution order.
- This is one reason plain reads/writes are unsafe for thread synchronization.

```
// Source
x = 1;
y = 3;
x = 2;

// Possible optimized form
x = 2;
y = 3;
```

Hardware Memory Models

- x86-64: relatively strong ordering.
- ARM/RISC-V: weaker ordering, more explicit barriers.
- Correct algorithm must be architecture-independent.
- Testing only on x86 can hide bugs that appear on weaker models.

Access Classes

- Plain data access:
  - no inter-thread ordering guarantees
  - cannot safely coordinate threads
- Atomic access:
  - atomicity plus ordering guarantees
  - valid building block for synchronization

Ordering Lattice

- `SeqCst`: global total order of SeqCst operations.
- `Release/Acquire`: pairwise synchronization edge.
- `Relaxed`: atomicity only, no ordering edge.
- Stronger ordering is easier to reason about, but can cost performance.

Sequentially Consistent

- Easiest to reason about globally.
- Good baseline for initial correct implementation.
- You can often weaken later once correctness is proven.

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

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

FLAG.store(true, Ordering::SeqCst);
let seen = FLAG.load(Ordering::SeqCst);
println!("{seen}");
```

Release/Acquire Publication

- Release publishes prior writes.
- Acquire on the same atomic, when it sees that release, makes those writes visible.
- Direction mnemonic: release keeps "before" before; acquire keeps "after" after.

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

static READY: AtomicBool = AtomicBool::new(false);
static DATA: AtomicUsize = AtomicUsize::new(0);

DATA.store(42, Ordering::Relaxed);
READY.store(true, Ordering::Release);

while !READY.load(Ordering::Acquire) {}
println!("{}", DATA.load(Ordering::Relaxed));
```

Relaxed Atomics

- Suitable for counters/statistics only.
- Not a substitute for synchronization.
- Useful when you need atomicity but not cross-thread visibility guarantees.

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

static COUNTER: AtomicUsize = AtomicUsize::new(0);
COUNTER.fetch_add(1, Ordering::Relaxed);
```

Compare-Exchange Loop

- Primitive transition from one state to another.
- It is in a loop because another thread may win the race and change the value first.
- Success and failure orderings are separate choices.

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

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

while LOCK
    .compare_exchange(false, true, Ordering::Acquire, Ordering::Relaxed)
    .is_err()
{}

LOCK.store(false, Ordering::Release);
```

Compare-Exchange Ordering Rules

- On failure, compare-exchange behaves like a load.
- Failure ordering cannot be stronger than success ordering.
- Failure ordering cannot be `Release` or `AcqRel`.
- Typical pattern: success `Acquire`/`AcqRel`, failure `Relaxed`.
- Incorrect order pairings do not compile.

Fences

- `fence(Ordering::X)`: hardware + compiler barrier.
- `compiler_fence(Ordering::X)`: compiler barrier only.
- `compiler_fence` does not synchronize threads by itself.
- Use fences when operation-local ordering on atomics is not enough.

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

compiler_fence(Ordering::Release);
fence(Ordering::SeqCst);
```

Atomic TOCTOU

- Two loads are two observations.
- Second read may disagree with first.
- Think "two photos taken at different moments".

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

let idx = AtomicUsize::new(0);
let values = vec![10, 20, 30];

if idx.load(Ordering::Relaxed) < values.len() {
    let _ = values[idx.load(Ordering::Relaxed)];
}
```

Correctness Models

Happens-Before Edges

- Layman: if you can draw arrows from A to B, B must see A’s effects.
- Program order gives arrows inside one thread.
- Synchronization operations add arrows across threads.
- If no arrow path exists, visibility is not guaranteed.
- In Rust, common edges come from mutex lock/unlock, thread join, channels, and release/acquire atomics.

Linearizability

- Layman: behavior looks as if operations happened one-by-one in some order.
- Each operation has a single conceptual "instant" where it takes effect.
- This is the usual target for concurrent maps, queues, stacks.
- In Rust lock-based APIs are often linearizable by design; lock-free structures need explicit proof.

Progress Guarantees

- Layman: this tells us who is guaranteed to finish.
- Blocking: a thread may wait forever.
- Lock-free: at least one thread keeps making progress.
- Wait-free: every operation finishes in bounded steps.
- Obstruction-free: progress is guaranteed only when running without contention.
- Rust gives primitives for all of these styles; which one you get depends on algorithm design.

Weak-Memory Examples

Purpose

- Same code can produce different outcomes under different memory orderings.
- Same code can produce different outcomes on strong vs weakly ordered CPUs.
- These examples are for correctness reasoning, not benchmarking.

Store Buffering (SB)

- Classic weak-memory example.
- Outcome `(r1 = 0, r2 = 0)` is forbidden under SeqCst.
- Under weaker orderings this outcome may appear.

```rust
// Thread 1
x.store(1, Ordering::Relaxed);
r1 = y.load(Ordering::Relaxed);

// Thread 2
y.store(1, Ordering::Relaxed);
r2 = x.load(Ordering::Relaxed);
```

Load Buffering (LB)

- Reads may happen before writes become visible globally.
- Architecture and ordering level determine allowed outcomes.
- This example demonstrates why "looks obvious" is not a proof in concurrent code.

```rust
// Thread 1
r1 = y.load(Ordering::Relaxed);
x.store(1, Ordering::Relaxed);

// Thread 2
r2 = x.load(Ordering::Relaxed);
y.store(1, Ordering::Relaxed);
```

Lock-Free Pitfalls

ABA Problem

- Compare-exchange only checks current value equality.
- State may change A -> B -> A and still pass compare-exchange.
- Typical in stacks/freelists with pointer reuse.
- Requires additional versioning or memory-reclamation scheme.
- Rust ownership helps, but lock-free reclamation still requires careful design.

ABA Mitigations

- Tagged pointers/version counters.
- Epoch-based reclamation.
- Hazard pointers.
- Avoid premature node reuse.

False Sharing

- Independent counters on one cache line still contend.
- Cache-line ping-pong can dominate runtime.
- Logical independence does not imply hardware independence.

```rust
use std::sync::atomic::AtomicU64;

#[repr(align(64))]
struct PaddedCounter(AtomicU64);
```