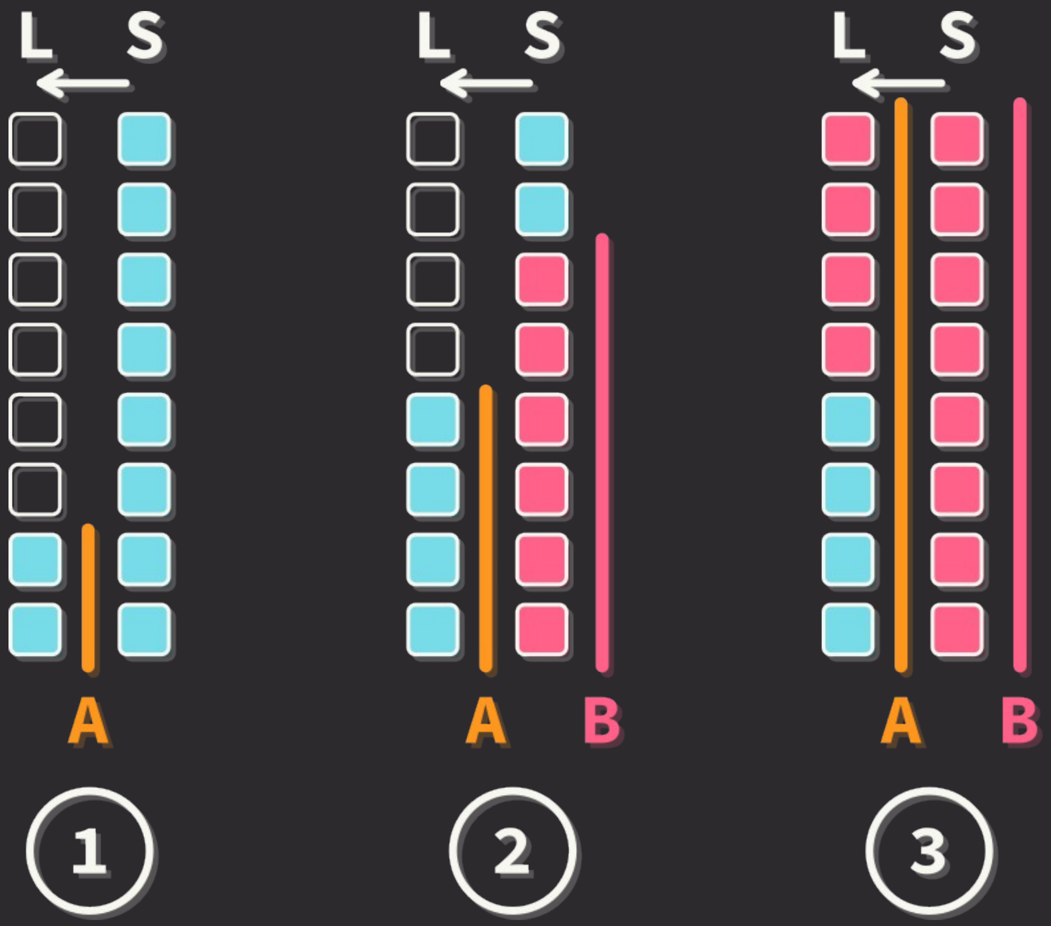
Shared Ownership | Threads & Mutex

Shared Data w/Threading

- Threads execute non-deterministically
  - Can read/write at random times
- Multiple threads can work with the same data
  - Data can become corrupted easily
    - Difficult to work with threads

Data Corruption

L = Thread-Local
S = Shared



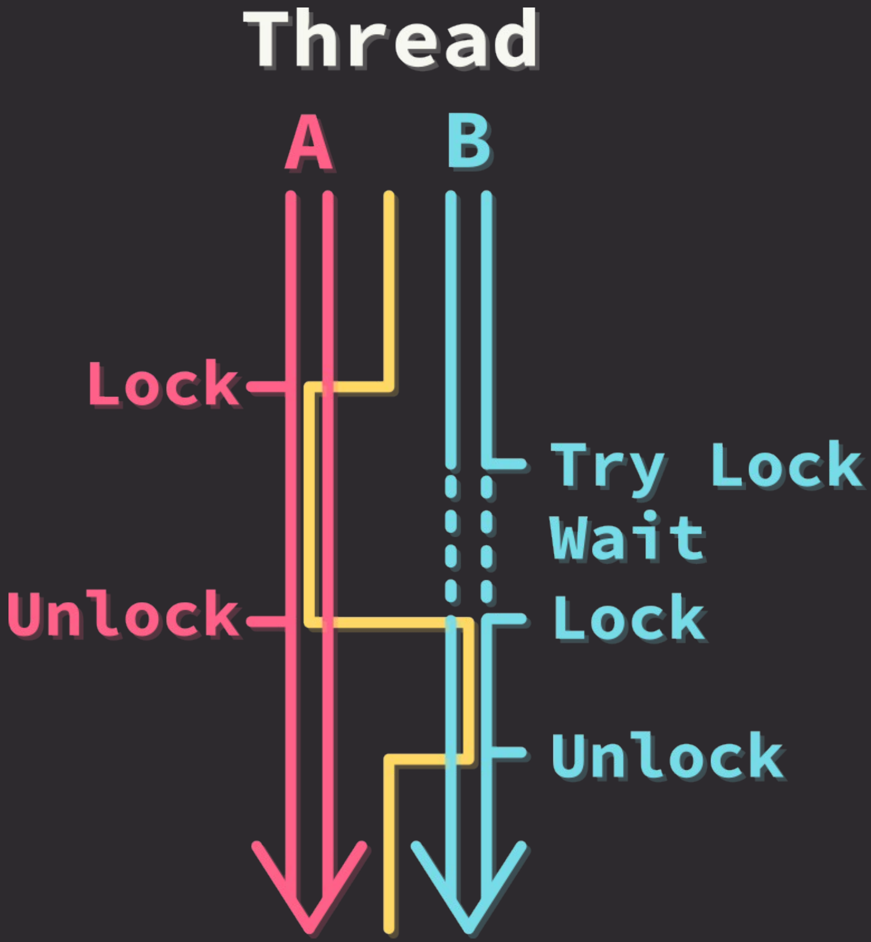
Synchronization

- Data needs to be synchronized for safe access
- Common synchronization primitive is a ***Mutex***
  - Mutually Exclusive lock
- Uses atomic operations to ensure that data is only accessed by one thread at a time
  - Atomic operations are “all or nothing” operations, enforced by the CPU
    - Data stays consistent

Mutex

- ***Mutexes*** wrap data, making data mutually exclusive
  - Only one thread can access at a time
  - All other threads will wait until finished
- ***Mutexes*** <u>cannot</u> be shared among threads
  - Wrap with a smart pointer (***Arc***)
  - Share the ***Arc*** among threads
- Use ***parking_lot*** crate for a ***Mutex***
  - Better API & performance than stdlib

How Mutex Works: Locks



Example

```
use parking_lot::Mutex;
use std::sync::Arc;
use std::thread;

struct Counter(usize);

let counter = Counter(0);
let shared_counter = Arc::new(Mutex::new(counter));

let thread_1_counter = Arc::clone(&shared_counter);
let thread_2_counter = shared_counter.clone();
```

$\boxed{\texttt{Arc<Mutex<Counter>>}}$

Example

```rust
let thread_1 = thread::spawn(move || {
    let mut counter = thread_1_counter.lock();
    counter.0 += 1;
});

let thread_2 = thread::spawn(move || {
    let mut counter = thread_2_counter.lock();
    counter.0 += 1;
});

thread_1.join().and_then(|_| thread_2.join());
println!("{}", shared_counter.lock().0);
```

Recap

- Data access from threads must be synchronized
  - Wrap data in a ***Mutex***
  - Use ***.lock()*** to acquire a lock
  - Unlocking occurs when the lock is dropped
- ***Mutexes*** cannot be shared
  - Wrap in ***Arc*** to share between threads
- Lock a minimum amount of time by performing computations <u>before</u> taking a lock