
# Threads and Ownership in Rust

Lecture 13

Slides: Neil Kaushikkar

# Goals For Today


- Answering Your Questions
- Thread Joins & Tracing Through Multi-Threaded Code Execution
- MPSC & the Drop Trait
- More on Thread Execution Order

# Reminders


- HW9 due 10/13 at 11:59pm
- HW10 releasing today, due 10/18 at 11:59pm
- MP3 releasing today, due 10/21 at 11:59pm
- Social/Office Hours tomorrow 10/12 from 12pm-2pm

# Answering Your Questions!


- "I still don't get much about such things as clone, reference and pointer."
  - Clone: deep copy of some data
    - Deep Copy: a piece of data with identical fields, but does not share the same references with the original piece of data
    - We are forced to call **.clone()** because deep copies can be expensive (aka time consuming, resource intensive, etc...)
  - Pointer: Some memory address where there is data we are interested in
    - This data might be invalid/deleted/corrupted if it is no longer in use
    - Rust guarantees memory safety, so you will never interact with these in safe Rust code (unsafe Rust code exists, but should be avoided AT ALL COSTS)
  - Reference: A pointer that is guaranteed to be valid/in use

# Answering Your Questions!


- "Can you go over why we need to use .as_ref() for the linked list a little more please? Thanks :)"
  - **.as_ref()**: Converts from **&Option<T>** to **Option<&T>**
    - We want a reference to the data inside the Option since we want to check on the next link, and links are the data inside the Option
    - This is an **&Option<T>** specific method!
    - (other types may also have it but only if they implement it)
  - **.as_mut()**: Converts from **&mut Option<T>** to **Option<&mut T>**
    - Same principle, just with mutable references
    - (borrowing rules apply)

Reference:

- https://doc.rust-lang.org/std/option/enum.Option.html#method.as_ref

# Answering Your Questions!


- “For [HW9], if you were to **.join()** each thread in the vector, why does it do every thread in parallel? I thought that the **.join()** call on each thread would force the main method to wait until that thread is finished, after which it may move on to the next thread join.”
  - Join blocks the current thread…
  - BUT all threads have been created before you call join…
  - This means our threads are running in parallel…
  - BUT the order threads are waited on is NOT connected with how they run as long as threads are created before any joins happen

# Thread Joining Experiment - Estimate Join Wait Times

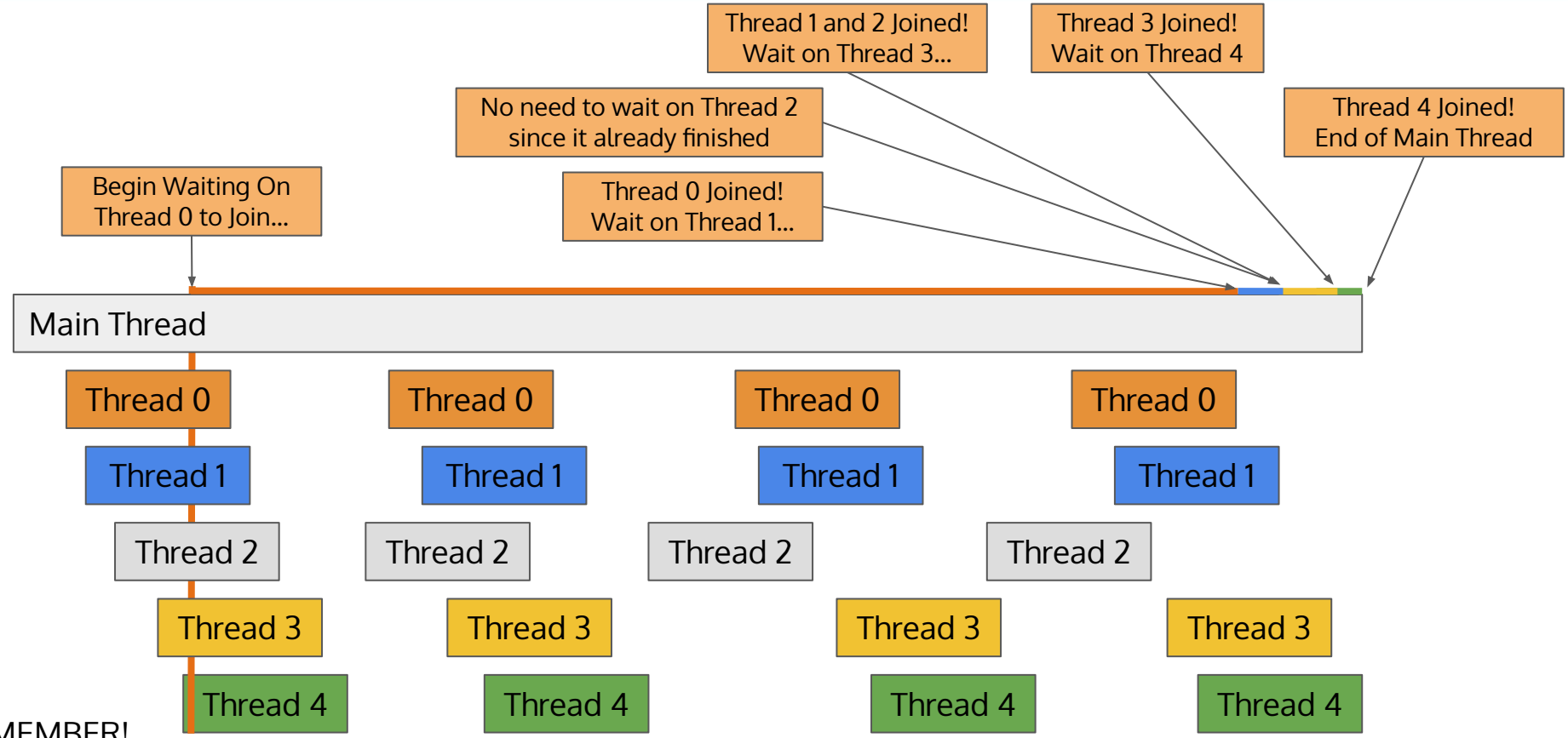

```rust
use std::time::{Duration, Instant};
use std::thread;

fn main() {
    let mut handles = Vec::new();
    for tid in 0..5 {
        let h = thread::spawn(move || {
            println!("Hello from thread {tid}!");
            for _ in 0..3 {
                thread::sleep(Duration::from_millis(1000));
                println!("Hello from thread {tid}!");
            }
        });

        handles.push(h);
    }

    for (idx, h) in handles.into_iter().enumerate() {
        let start = Instant::now();
        h.join().unwrap();
        let duration = Instant::now() - start;
        println!("Joined thread with id={idx} after waiting {} µs", duration.as_micros());
    }
}
```

# Thread Joining Hypothesis - Estimate Join Wait Times




REMEMBER!

- There is no guarantee of thread execution order!

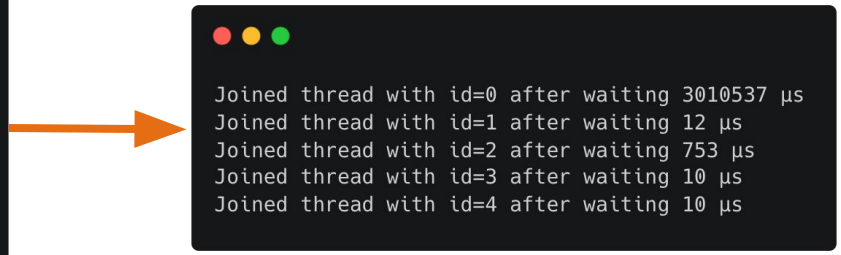
# Thread Joining Experiment Results - Actual Wait Times

```rust
use std::time::{Duration, Instant};
use std::thread;

fn main() {
    let mut handles = Vec::new();
    for tid in 0..5 {
        let h = thread::spawn(move || {
            println!("Hello from thread {tid}!");
            for _ in 0..3 {
                thread::sleep(Duration::from_millis(1000));
                println!("Hello from thread {tid}!");
            }
        });

        handles.push(h);
    }

    for (idx, h) in handles.into_iter().enumerate() {
        let start = Instant::now();
        h.join().unwrap();
        let duration = Instant::now() - start;
        println!("Joined thread with id={idx} after waiting {} µs", duration.as_micros());
    }
}
```



```
Joined thread with id=0 after waiting 3010537 µs
Joined thread with id=1 after waiting 12 µs
Joined thread with id=2 after waiting 753 µs
Joined thread with id=3 after waiting 10 µs
Joined thread with id=4 after waiting 10 µs
```

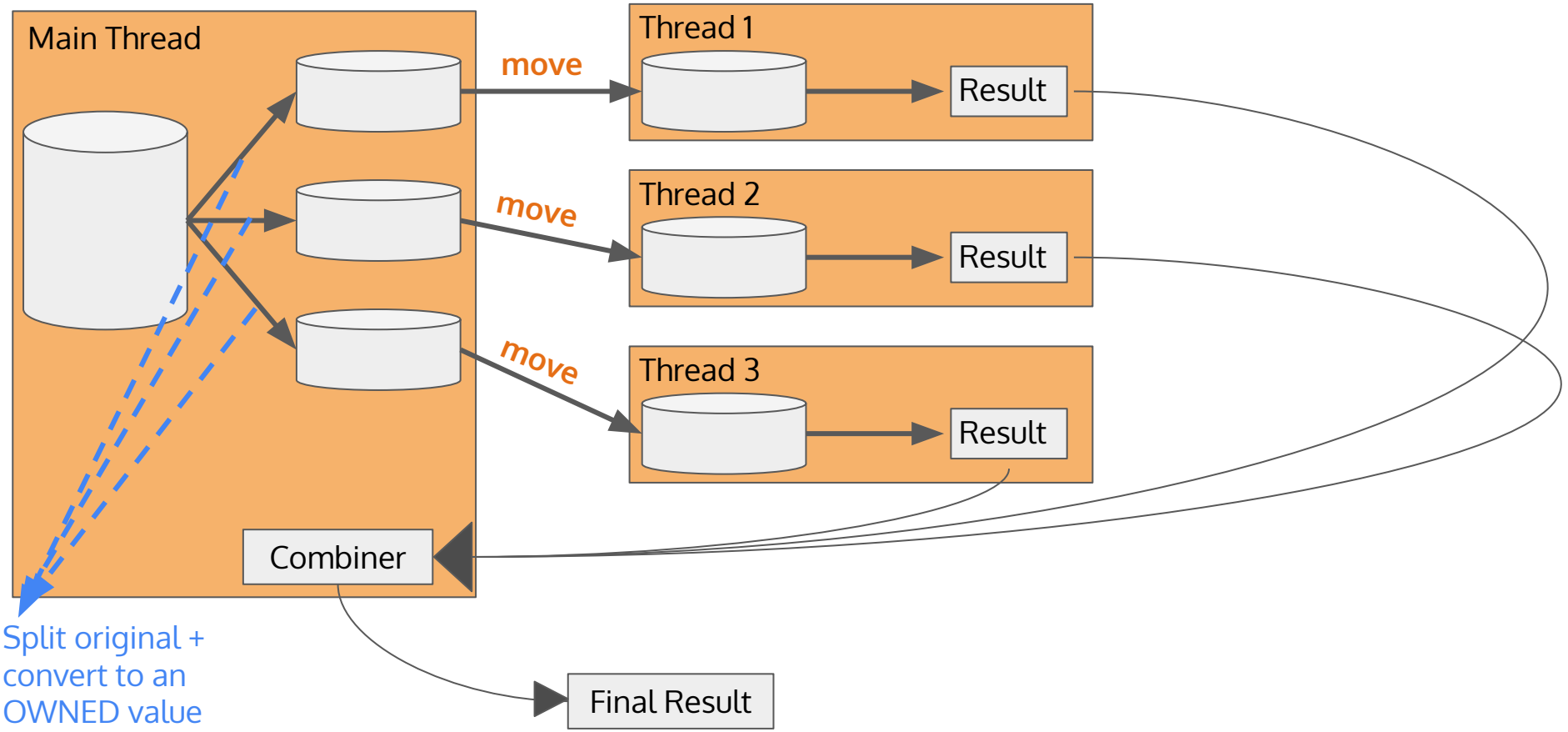
# Review: MPSC


- MPSC == Multiple Producer Single Consumer
- Senders are **clone**-able (multi-producer) such that many threads can send simultaneously to one receiver (single-consumer)
    - **Clone** the sender/transmitter (tx) and **move** the cloned tx to a thread
    - Send messages within your threads using the cloned sender (tx)
    - Receive messages on the main thread with your receiver (rx)

Reference:

- https://doc.rust-lang.org/std/sync/mpsc/

# Solving Problems with Parallelism



# Review: MPSC + Drop

```rust
use std::sync::mpsc;
use std::thread;

fn main() {
    let chunk_size = 10_000;
    let num_threads = 10;
    let max_data = chunk_size * num_threads;
    let data = (0..max_data).collect::<Vec<usize>>();
    let (tx, rx) = mpsc::channel();

    for i in 0..num_threads {
        let start = i * chunk_size;
        let end_excl = start + chunk_size;
        let owned_subvec = data[start..end_excl].to_vec();
        let tx_clone = tx.clone();

        thread::spawn(move || {
            let sub_vec_sum: usize = owned_subvec.into_iter().sum();
            tx_clone.send(sub_vec_sum).unwrap();
        });
    }

    drop(tx);
    let mut total = 0;
    while let Ok(value) = rx.recv() {
        println!("Receiver got {value}!");
        total += value;
    }
}
```

Important!!!

# Transmitters and Drop (Ownership Rules!)


- **rx.recv()** returns **Err()** when
  - All transmitters have been dropped and ...
  - All messages have been received
- Transmitters are be dropped when...
  - They go out of scope
  - You call **drop()** on the transmitter
- Note: Rust automatically calls **drop()** when a variable goes out of scope, but you can do it ahead of time if you wish

# MPSC + IMPLICIT Drop (Hint for MP3)

```rust
fn split_work_and_create_threads(data: &Vec<usize>,
        num_threads: usize, chunk_size: usize) -> Receiver<usize> {
    let (tx, rx) = mpsc::channel();

    for i in 0..num_threads {
        // GET SUB-VECTORS OF SIZE 10,000
        let start = i * chunk_size;
        let end_excl = start + chunk_size;
        let owned_subvec = data[start..end_excl].to_vec();

        let tx_clone = tx.clone();

        thread::spawn(move || {
            let min = owned_subvec[0];
            let max = owned_subvec[owned_subvec.len() - 1];

            let sub_vec_sum: usize = owned_subvec.into_iter().sum();
            println!("Subvec sum from {min} to {max} is {sub_vec_sum}");
            tx_clone.send(sub_vec_sum).unwrap();
        });
    }

    rx
}
```

`tx` goes out of scope here and is dropped from memory (Rust calls `drop(tx)` for you!

Each clone of `tx` goes out of scope when each thread ends!!!

```rust
fn receive(rx: Receiver<usize>) -> usize {
    let mut total = 0;
    while let Ok(value) = rx.recv() {
        println!("Receiver got {value}!");
        total += value;
    }

    total
}
```

`recv()` will block until all messages are received

```rust
fn main() {
    let chunk_size = 10_000;
    let num_threads = 10;

    let max_data = chunk_size * num_threads;
    let data = (0..max_data).collect::<Vec<usize>>();

    let rx = split_work_and_create_threads(&data, num_threads, chunk_size);
    let result = receive(rx);

    println!("Total sum is: {}", result);
```

There is a blocking call in `receive()`, so it will block!!!

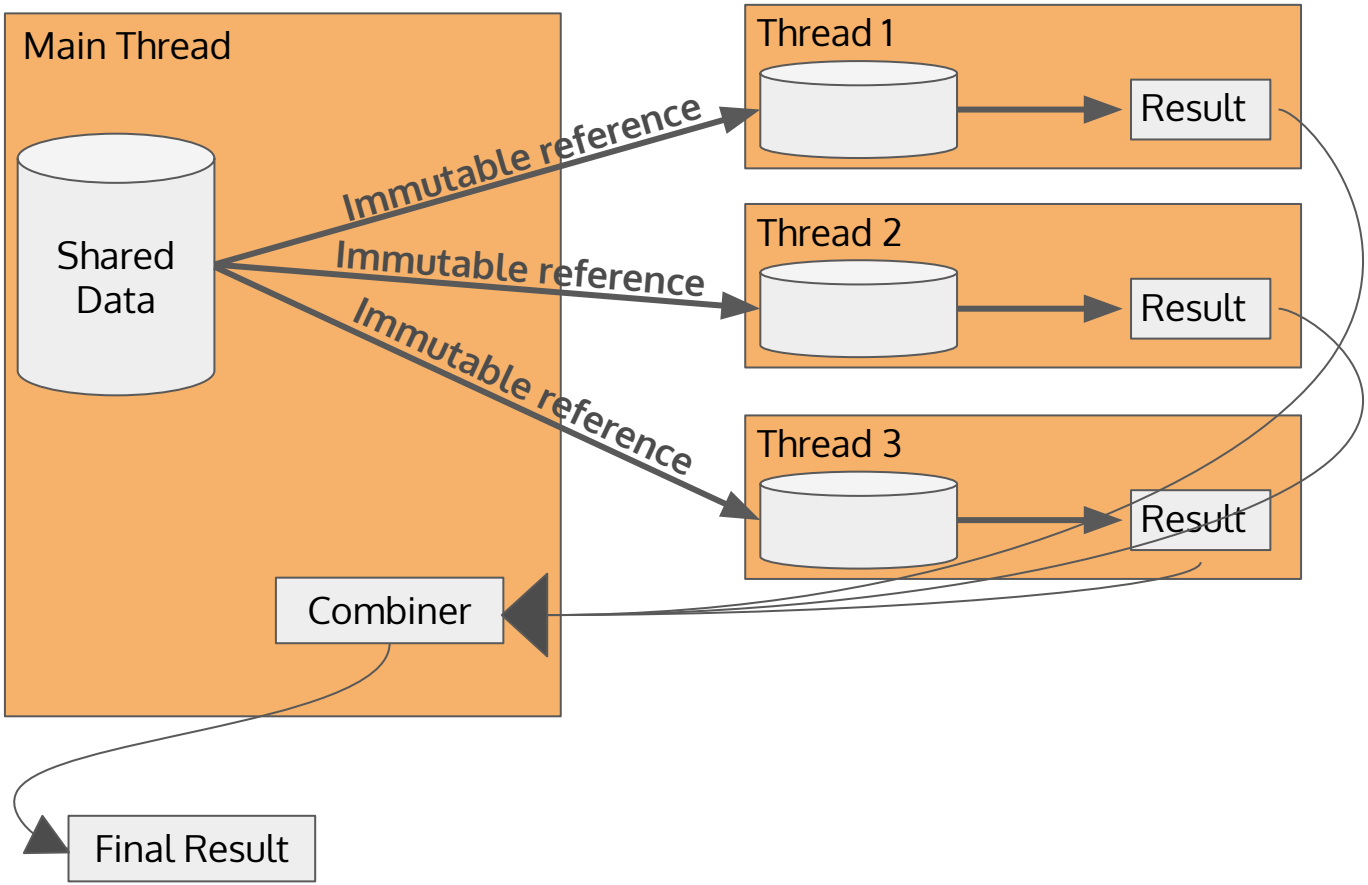
REMEMBER!

- Transmitters are be dropped when ... they go out of scope!

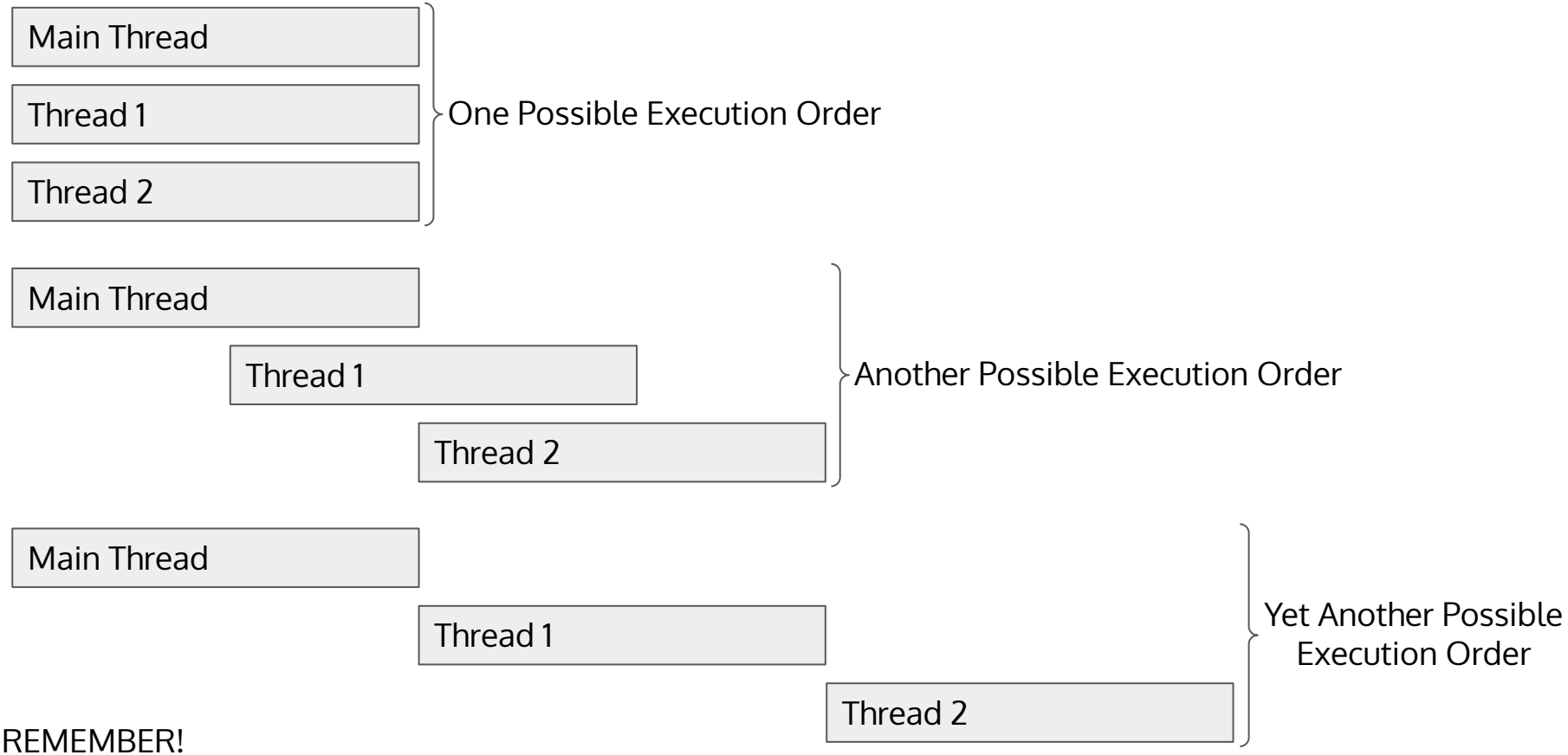
# Threads and Ownership


- A new thread owns all variables moved into it from the parent thread
- Variables, values, etc get dropped when they go out of scope
  - Data owned by threads gets dropped when the thread finishes
- Takeaway
  - We want threads to own their own data to guarantee memory safety
  - WHY???

# Thread Ownership (BUGGY MODEL)



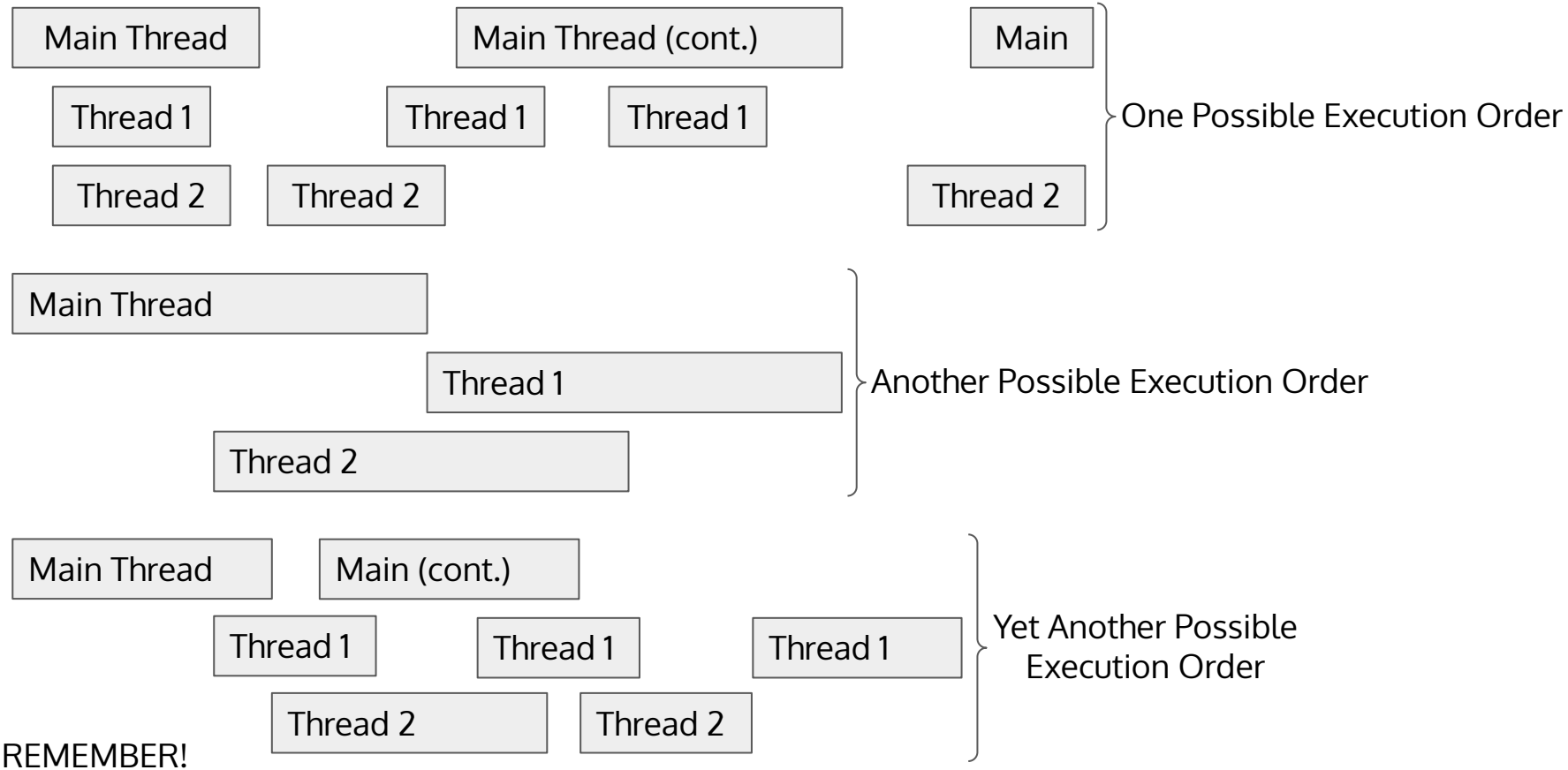
# Thread Execution Order (Spot the Safety Issue?)




REMEMBER!

- There is no guarantee of thread execution order!

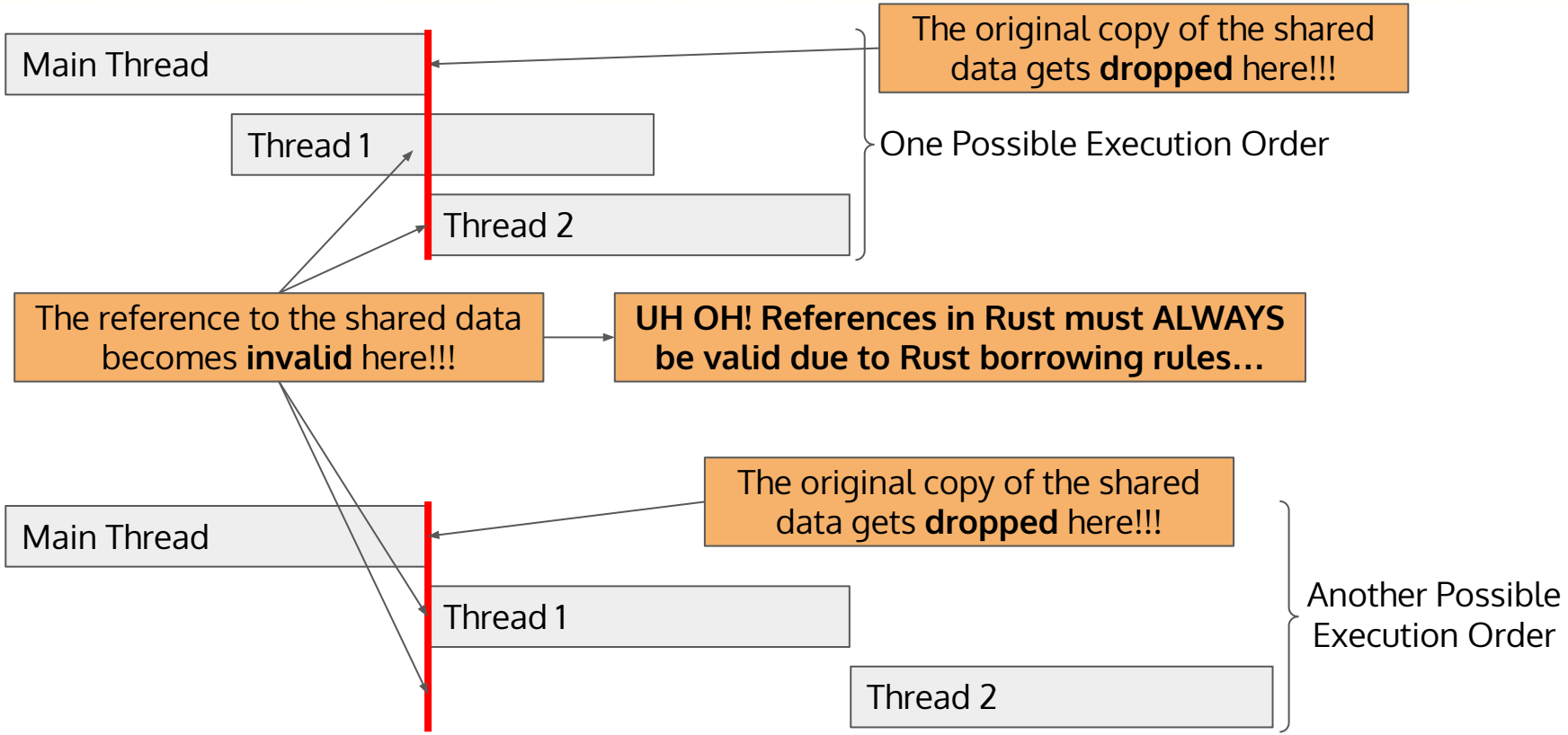
# Thread Execution Order (Spot the Safety Issue?)




REMEMBER!

- There is no guarantee of thread execution order!

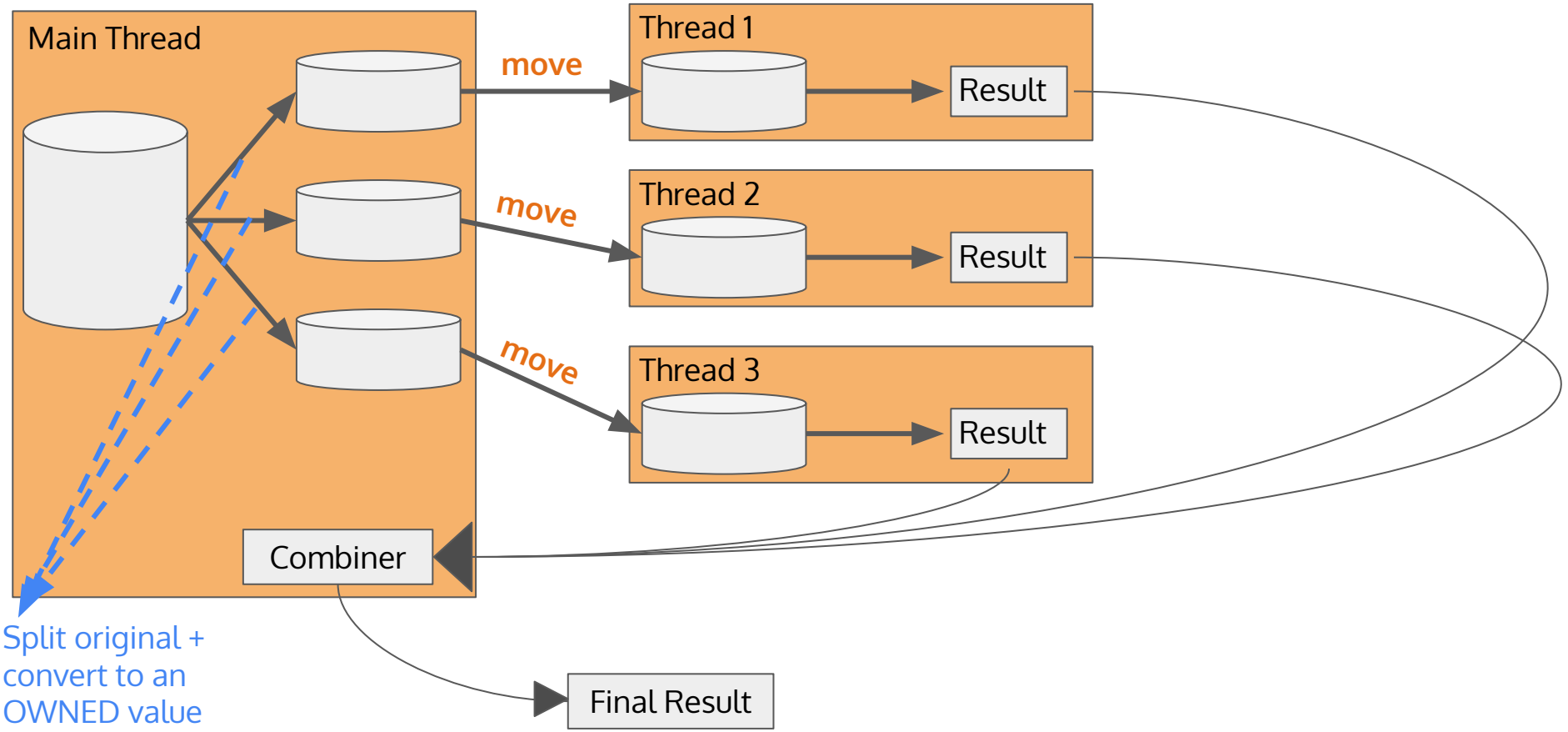
# Thread Execution Order (Spot the Safety Issue?)



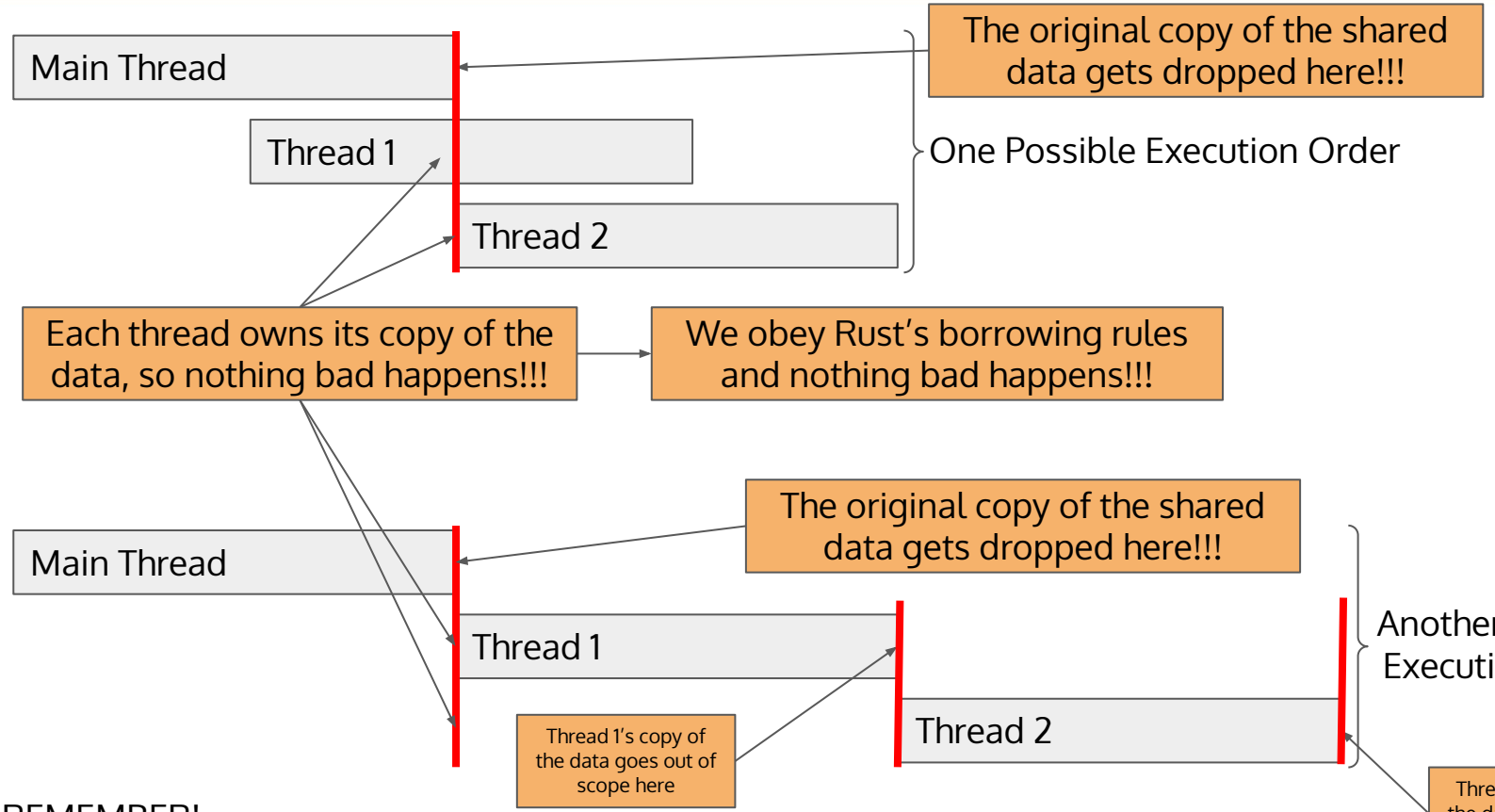

REMEMBER!

- There is no guarantee of thread execution order!

# Takeaway: Each Thread Gets Its Own Data



# Thread Execution Order




REMEMBER!

- There is no guarantee of thread execution order!

# Giving Threads Ownership (MP3 Hints)


- Split your data up into owned chunks
  - **Vec<T>** becomes **Vec<Vec<T>>**
  - 1000 elements becomes a list of lists, each with 100 elements
  - MP3: Take a vector of **String**s, clone each **String**, place the clone into a one of your vectors (**Vec<T>**) inside the chunks vector **Vec<Vec<T>>**
- Use an iterator that takes ownership and consumes your data when sending individual data chunks to your new threads
  - **let chunks: Vec<Vec<String>> = split_into_chunks(&original_data);**
  - **for chunk in chunks.into_iter() { thread::spawn(move || { // do computation on `chunk` here…**
  - Use **into_iter()** to get the OWNED elements of your vector


# That’s All Folks!