Intro to Multithreading

Ryan Eberhardt and Julio Ballista
May 10, 2021

Logistics

- Weekly survey for last week: https://forms.gle/5WTTQFXAtQuDwdpFA
- Project 1 due on Thursday
  - Post questions in #questions-project-1
  - Please let us know if we can help!
- Week 7 exercises going out today, due next Tuesday

Perils of concurrency

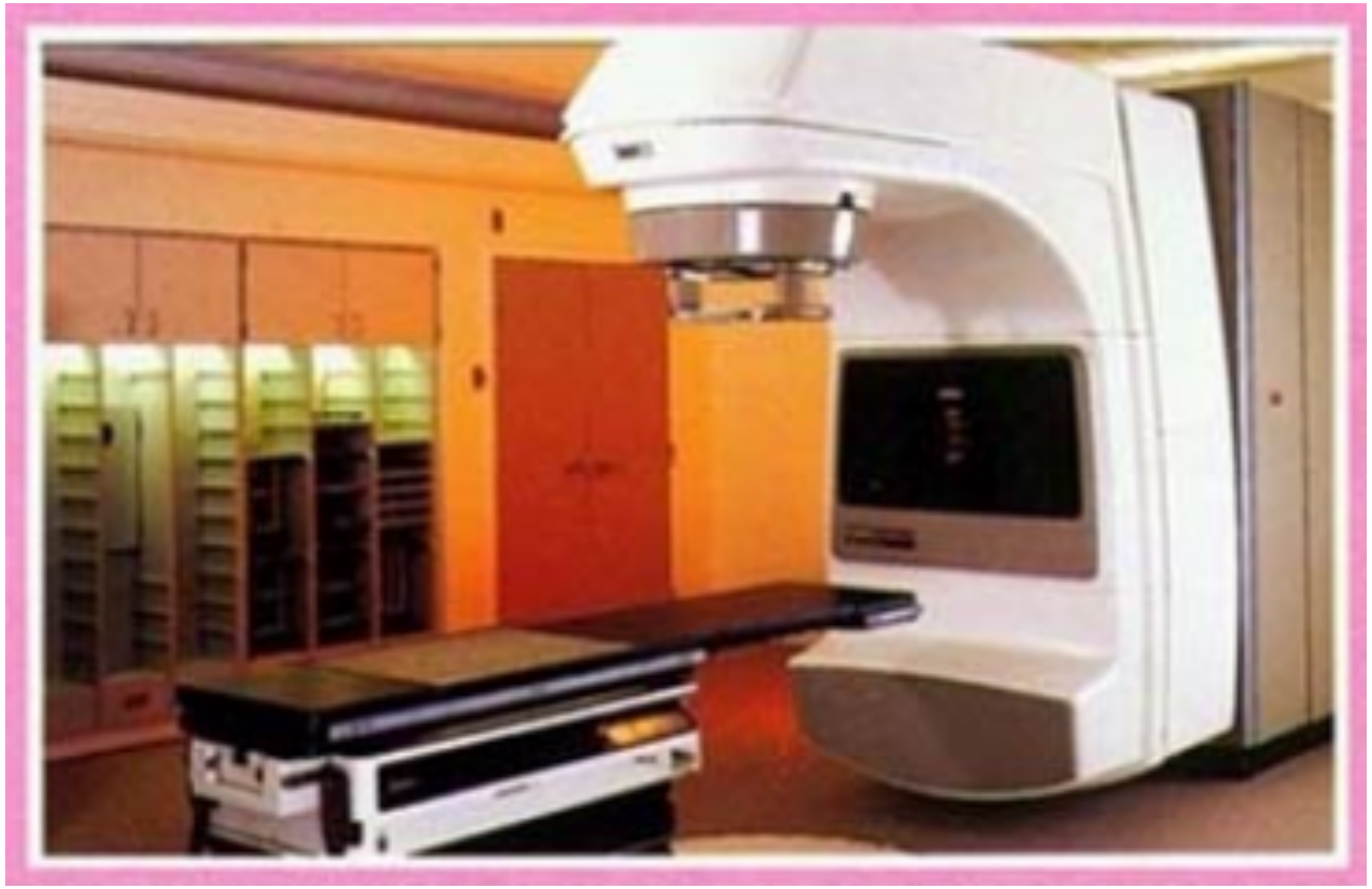
- Why is multithreading nice?
- Why is multithreading dangerous?
    - Race conditions
    - Deadlock (more on Thursday and next week)

Perils of concurrency

- Race conditions are bad because:
  - They cause the program to not work
  - But only sometimes! They're easy to miss in testing, and extremely hard to debug

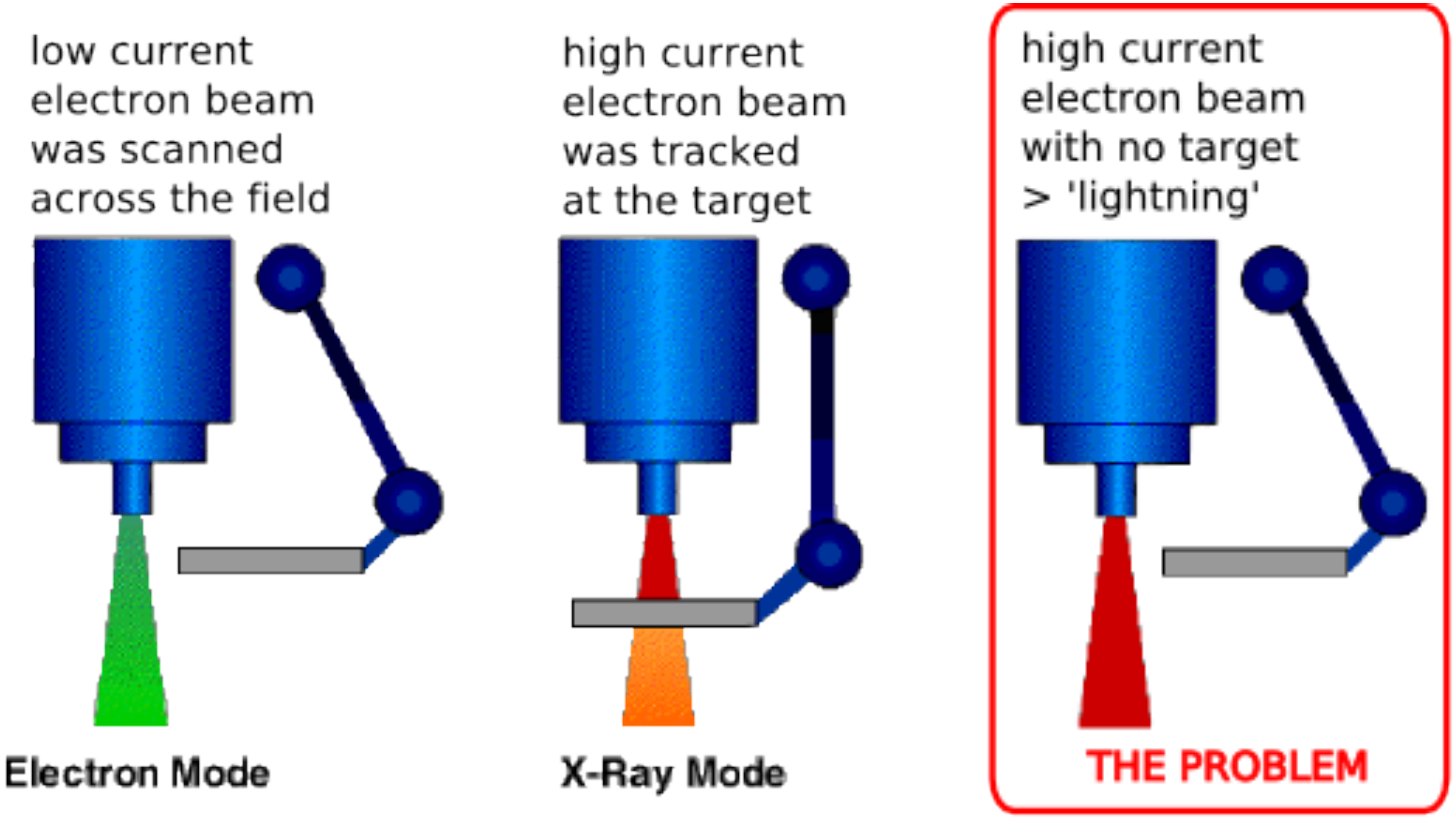
Okay, but why should I care?

Race conditions have killed people



https://hci.cs.siue.edu/NSF/Files/Semester/Week13-2/PPT-Text/Slide13.html
https://hackaday.com/2015/10/26/killed-by-a-machine-the-therac-25/

Race conditions have killed people



http://radonc.wikidot.com/radiation-accident-therac25

Race conditions have killed people

After each overdose the creators of Therac-25 were contacted. After the first incident the AECL responses was simple: "After careful consideration, we are of the opinion that this damage could not have been produced by any malfunction of the Therac-25 or by any operator error (Leveson,1993)."

After the 2nd incident the AECL sent a service technician to the Therac-25 machine, he was unable to recreate the malfunction and therefore conclude nothing was wrong with the software. Some minor adjustments to the hardware were changed but the main problems still remained.

It was not until the fifth incident that any formal action was taken by the AECL. However it was a physicist at the hospital where the 4th and 5th incident took place in Tyler, Texas who actually was able to reproduce the mysterious "malfunction 54". The AECL finally took action and made a variety of changes in the software of the Therac-25 radiation treatment system.

http://radonc.wdfiles.com/local--files/radiation-accident-therac25/Therac_UGuelph_TGall.pdf

Race conditions have killed people

- Investigation results:
- ***The failure occurred only when a particular nonstandard sequence of keystrokes was entered*** *on the VT-100 terminal which controlled the PDP-11 computer: an "X" to (erroneously) select 25 MeV photon mode followed by "cursor up", "E" to (correctly) select 25 MeV Electron mode, then "Enter", all within eight seconds.*
- *The equipment control task did not properly synchronize with the operator interface task, so that race conditions occurred if the operator changed the setup too quickly.* ***This was missed during testing, since it took some practice before operators were able to work quickly enough to trigger this failure mode.***
- https://en.wikipedia.org/wiki/Therac-25 and http://sunnyday.mit.edu/papers/therac.pdf

Race conditions are everywhere!

- Starbucks: possible to get unlimited coffee
- GitHub: possible to get logged in as a different user
- Unlimited bitcoin, voting multiple times, using Instacart coupons multiple times (from Jack Cable)
- Kernel race condition in CPlayground, caused by yours truly

Small probabilities are deceiving

- “Given the scale that Twitter is at, a one-in-a-million chance happens 500 times a day.” (Del Harvey, 2014)

Compounding effects

- "I'm just working on my hot new social media app… Who cares if it breaks 0.01% of the time?"
- Let's say that downloading/displaying a post involves 20 steps
  - Selecting the post to display, serializing, transmitting over the network, receiving, rendering, etc…
- You weren't very careful, and 5 of those steps have race conditions that each manifest 0.01% of the time. Displaying a post will crash 0.05% of the time
- Let's say the average user quickly scrolls through 300 posts/day. A user now has a ~15% chance of crashing the app every day
- Next, you add a messaging feature. Sending/receiving a message also fails 0.05% of the time
- A typical user sends/receives 100 messages a day. Now your app has a ~20% chance of crashing for a user on any given day
- Who would want to use an app like this? (Not me!)

Compounding effects

- Production codebases have millions of lines of code
- When working with concurrency, you must be meticulous and disciplined
- Yet even the very best programmers make mistakes! We need more tools to help us prevent and identify problems

Preventing data races

What are race conditions?

- Race condition:
  A race condition or race hazard is the condition of an electronics, software, or other system where the system's substantive behavior is dependent on the sequence or timing of other uncontrollable events. (Wikipedia)
- Data race:
  Multiple threads access a value, where at least one of them is writing
  - This should sound familiar!

Rust's design pays off

- Rust's design goals:
  - How do you do safe systems programming?
  - How do you make concurrency painless?
  - How do you make it fast?
- *"Initially these [first two] problems seemed orthogonal, but to our amazement, the solution turned out to be identical: the same tools that make Rust safe also help you tackle concurrency head-on."* (Rust blog)
- Compiler enforces rules for safe concurrency. *"Thread safety isn't just documentation; it's law."*
- There's very little in the core language specific to threading! (Only two traits!) Almost all thread safety comes from the ownership model you already know

Hello world!

```
use std::{thread, time};
use rand::Rng;

const NUM_THREADS: u32 = 20;

fn main() {
    let mut threads = Vec::new();
    println!("Spawning {} threads...", NUM_THREADS);
    for _ in 0..NUM_THREADS {
        threads.push(thread::spawn(|| {
            let mut rng = rand::thread_rng();
            thread::sleep(time::Duration::from_millis(rng.gen_range(0, 5000)));
            println!("Thread finished running!");
        }));
    }
    // wait for all the threads to finish
    for handle in threads {
        handle.join().expect("Panic happened inside of a thread!");
    }
    println!("All threads finished!");
}
```

Parameters for closure function (none, in this case)

Closure/lambda function borrows any referenced variables

A panic in a thread will not crash the entire program
Need to check if the thread panicked

[Playground]

Extroverts demo (CS 110)

```
static const char *kExtroverts[] = {
    "Frank", "Jon", "Lauren", "Marco", "Julie", "Patty",
    "Tagalong Introvert Jerry"
};
static const size_t kNumExtroverts = sizeof(kExtroverts)/sizeof(kExtroverts[0]) - 1;

int main() {
    vector<thread> threads;
    for (size_t i = 0; i < kNumExtroverts; i++) {
        threads.push_back(thread([&i]() {
                cout << "Hello from extrovert " << kExtroverts[i] << "!" << endl;
                }));
    }
    // wait for all the threads to finish
    for (thread& handle : threads) {
        handle.join();
    }
    return 0;
}
```

Passes a reference/pointer to i, but then the main thread changes i on the next iteration of the for loop. By the time the new thread runs, i is 7

[Cplayground]()

Can we do the same in Rust?

```
use std::thread;

const NAMES: [&str; 7] = ["Frank", "Jon", "Lauren", "Marco", "Julie", "Patty",
    "Tagalong Introvert Jerry"];

fn main() {
    let mut threads = Vec::new();
    for i in 0..6 {
        threads.push(thread::spawn(|| {
            println!("Hello from extrovert {}!", NAMES[i]);
        }));
    }
    // wait for all the threads to finish
    for handle in threads {
        handle.join().expect("Panic occurred in thread!");
    }
}
```

Closure/lambda function *borrows* referenced variables by default (whenever possible)

[Rust playground]()

Can we do the same in Rust?

```
error[E0373]: closure may outlive the current function, but it borrows `i`, which is owned by the
current function
  --> src/main.rs:9:36
   |
9  |         threads.push(thread::spawn(|| {
   |                                    ^^ may outlive borrowed value `i`
10 |             println!("Hello from extrovert {}!", NAMES[i]);
   |                                                        - `i` is borrowed here
   |
note: function requires argument type to outlive `'static`
  --> src/main.rs:9:22
   |
9  |           threads.push(thread::spawn(|| {
   |  ______________________^
10 | |             println!("Hello from extrovert {}!", NAMES[i]);
11 | |         }));
   | |_________^
help: to force the closure to take ownership of `i` (and any other referenced variables), use the
`move` keyword
   |
9  |         threads.push(thread::spawn(move || {
   |                                    ^^^^^^^
```

Can we do the same in Rust?

```
error[E0373]: closure may outlive the current function, but it borrows `i`, which is owned by the
current function
  --> src/main.rs:9:36
   |
9  |         threads.push(thread::spawn(|| {
   |                                    ^^ may outlive borrowed value `i`
10 |             println!("Hello from extrovert {}!", NAMES[i]);
   |                                                        - `i` is borrowed here
   |
note: function requires argument type to outlive `'static`
  --> src/main.rs:9:22
   |
9  |           threads.push(thread::spawn(|| {
   |  ______________________^
10 | |             println!("Hello from extrovert {}!", NAMES[i]);
11 | |         }));
   | |_________^
help: to force the closure to take ownership of `i` (and any other referenced variables), use the
`move` keyword
   |
9  |         threads.push(thread::spawn(move || {
   |                                    ^^^^^^^
```

Can we do the same in Rust?

```rust
use std::thread;

const NAMES: [&str; 7] = ["Frank", "Jon", "Lauren", "Marco", "Julie", "Patty",
    "Tagalong Introvert Jerry"];

fn main() {
    let mut threads = Vec::new();
    for i in 0..6 {
        threads.push(thread::spawn(move || {
            println!("Hello from extrovert {}!", NAMES[i]);
        }));
    }
    // wait for all the threads to finish
    for handle in threads {
        handle.join().expect("Panic occurred in thread!");
    }
}
```

i is moved into the closure function; closure now has ownership

[Rust playground]

Ticket agents demo (CS 110)

Multiple threads get mutable reference to remainingTickets

Value decremented simultaneously: ends up underflowing!

```
static void ticketAgent(size_t id, size_t& remainingTickets) {
    while (remainingTickets > 0) {
        handleCall(); // sleep for a small amount of time to emulate conversation time.
        remainingTickets--;
        cout << oslock << "Agent #" << id << " sold a ticket! (" << remainingTickets
             << " more to be sold)." << endl << osunlock;
        if (shouldTakeBreak()) // flip a biased coin
            takeBreak();       // if comes up heads, sleep for a random time to take a break
    }
    cout << oslock << "Agent #" << id << " notices all tickets are sold, and goes home!"
         << endl << osunlock;
}

int main(int argc, const char *argv[]) {
    thread agents[10];
    size_t remainingTickets = 250;
    for (size_t i = 0; i < 10; i++)
        agents[i] = thread(ticketAgent, 101 + i, ref(remainingTickets));
    for (thread& agent: agents) agent.join();
    cout << "End of Business Day!" << endl;
    return 0;
}
```

Cplayground

Attempt 1

```
fn main() {
    let mut remaining_tickets = 250;

    let mut threads = Vec::new();
    for i in 0..10 {
        threads.push(thread::spawn(move || {
            ticket_agent(i, &mut remaining_tickets)
        }));
    }
    // wait for all the threads to finish
    for handle in threads {
        handle.join().expect("Panic occurred in thread!");
    }
    println!("End of business day!");
}
```

This code only compiles because i32 is Copy. Every thread is getting its own copy of the number! Not at all what we want!

If remaining_tickets were a non-Copy type, we would get an error when trying to give ownership to multiple threads

[Rust playground]()

Attempt 2: Shared ownership

- We want to have *one* `remaining_tickets` counter that is shared between all threads
- Rust allows shared ownership using *reference counting*
  - Take the thing you want to share and allocate it on the heap, along with a reference count
  - Whenever you share the object with another owner, increment the reference count

Attempt 2: Shared ownership

- We want to have *one* `remaining_tickets` counter that is shared between all threads
- Rust allows shared ownership using *reference counting*
  - Take the thing you want to share and allocate it on the heap, along with a reference count
  - Whenever you share the object with another owner, increment the reference count
  - Whenever an owner drops the object, decrement the reference count

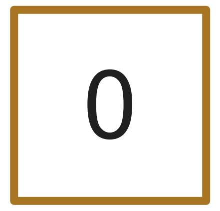
Attempt 2: Shared ownership

- We want to have *one* `remaining_tickets` counter that is shared between all threads
- Rust allows shared ownership using *reference counting*
  - Take the thing you want to share and allocate it on the heap, along with a reference count
  - Whenever you share the object with another owner, increment the reference count
  - Whenever an owner drops the object, decrement the reference count
  - When the reference count hits 0, free the memory

Note that this is different from references! References cannot outlive their owners, but with shared ownership, owners don't need to worry about each others' lifetimes

Attempt 2: Shared ownership

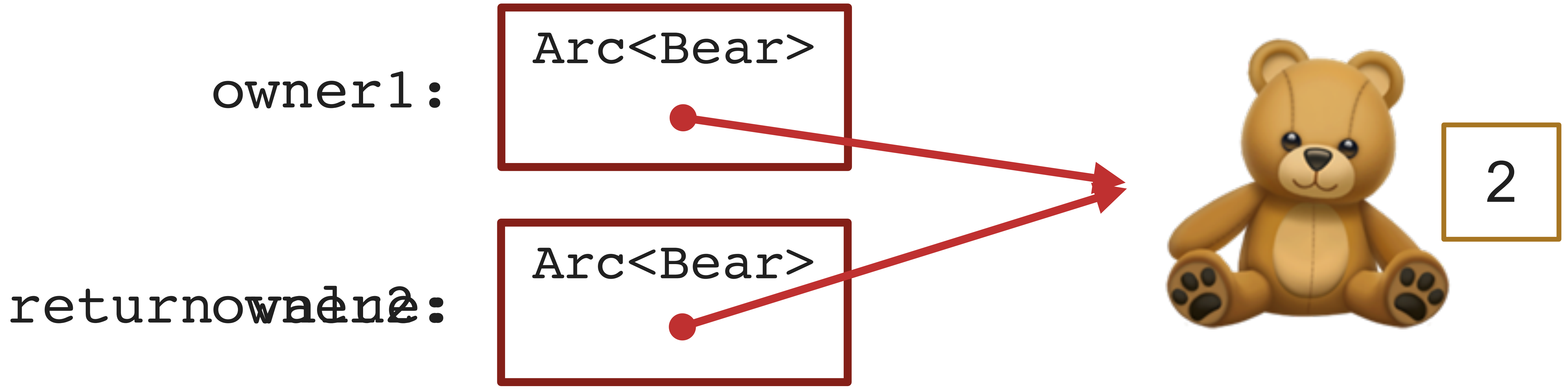
- We want to have *one* `remaining_tickets` counter that is shared between all threads
- Rust allows shared ownership using *reference counting*
  - Take the thing you want to share and allocate it on the heap, along with a reference count
  - Whenever you share the object with another owner, increment the reference count
  - Whenever an owner drops the object, decrement the reference count
  - When the reference count hits 0, free the memory



Attempt 2: Shared ownership

- We want to have *one* `remaining_tickets` counter that is shared between all threads
- Rust allows shared ownership using *reference counting*
  - Take the thing you want to share and allocate it on the heap, along with a reference count
  - Whenever you share the object with another owner, increment the reference count
  - Whenever an owner drops the object, decrement the reference count
  - When the reference count hits 0, free the memory
- [Arc type]: Atomically Reference Counted
  - Atomic: safe for multithreaded use
  - You may see the [Rc] type used in non-multithreaded settings

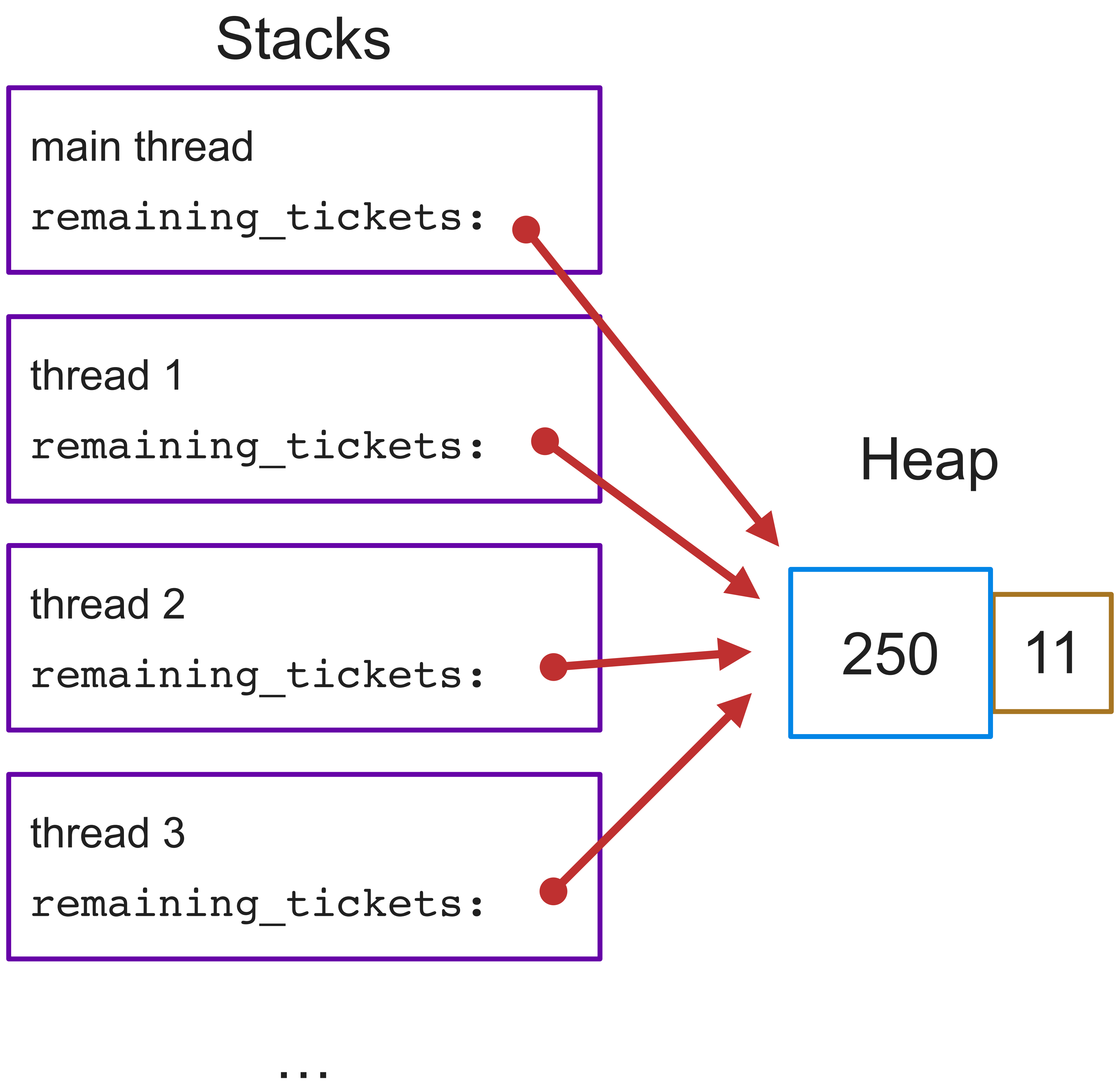
```
fn make_bear() -> Arc<Bear> {
    let owner1 = Arc::new(Bear {});
    let owner2 = owner1.clone();
    return owner2;
}
```



Attempt 2: Shared ownership

```rust
fn main() {
    let remaining_tickets = Arc::new(250);

    let mut threads = Vec::new();
    for i in 0..10 {
        let remaining_tickets_handle = remaining_tickets.clone();
        threads.push(thread::spawn(move || {
            ticket_agent(i, remaining_tickets_handle)
        }));
    }
    // wait for all the threads to finish
    for handle in threads {
        handle.join().expect("Panic occurred in thread!");
    }
    println!("End of business day!");
}
```



[Rust playground]

Problem: We can’t modify data in an Arc!

```
error[E0594]: cannot assign to data in an `Arc`
  --> src/main.rs:24:9
   |
24 |         *remaining_tickets -= 1;
   |         ^^^^^^^^^^^^^^^^^^^^^^^ cannot assign
   |
   = help: trait `DerefMut` is required to modify through a dereference, but it is not implemented for `Arc<usize>`
```

- Arc allows us to have multiple owners, but multiple ownership is only safe if the data is immutable
  - Otherwise, we could have someone altering the bear while someone else is painting it
- We need a way to safely coordinate access so that if someone wants to modify the bear, we ensure no one else is currently using it

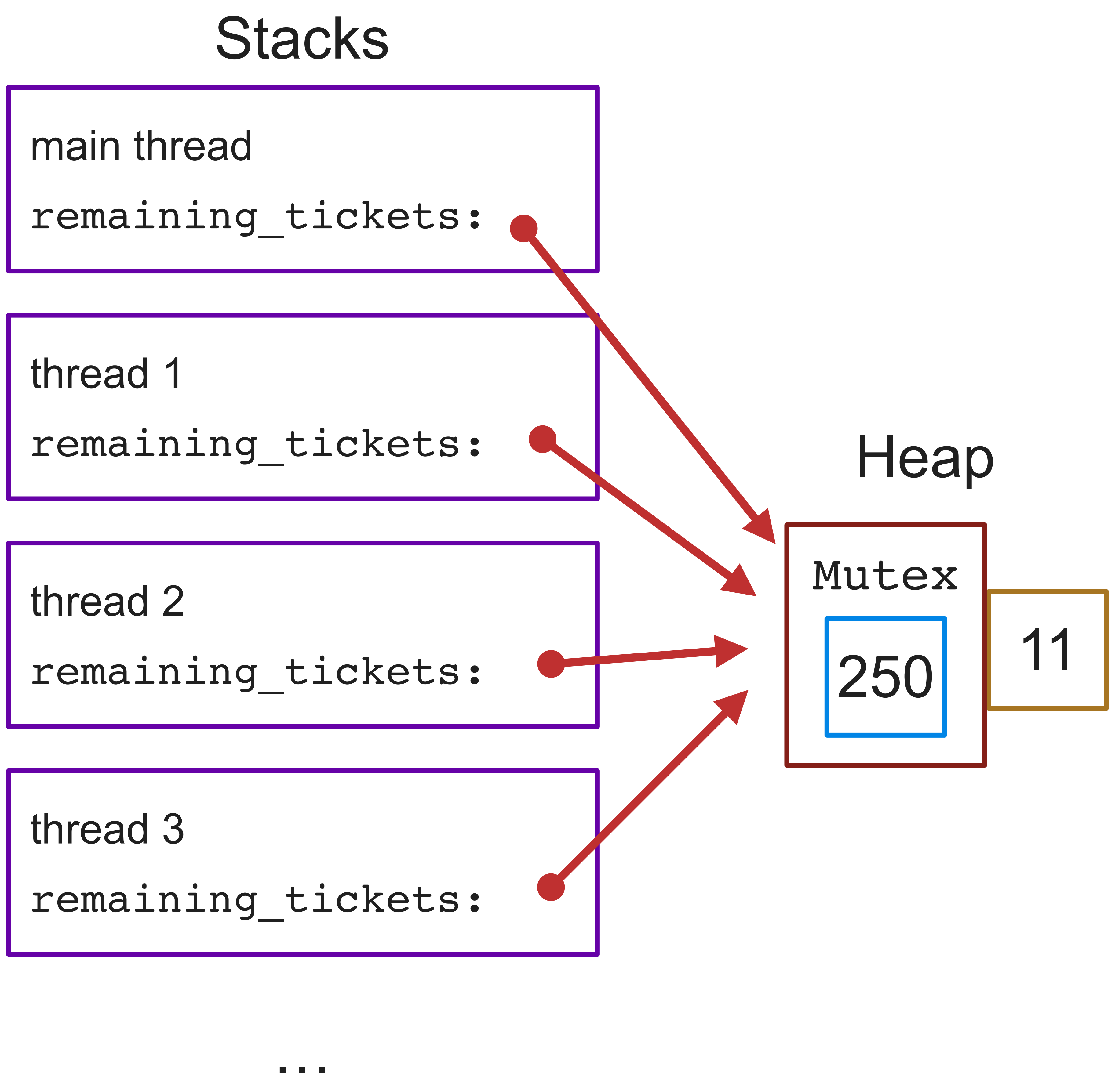
Attempt 3: Coordinated access with mutexes

- In Rust, the data goes *inside* the mutex
- The Mutex acts like a bathroom lock, where only one owner can pass at a time
- Unlike in C/C++, it is *impossible* to forget to lock a mutex! You can't access the data without going inside and locking the lock

Attempt 3: Coordinated access with mutexes

```
fn main() {
    let remaining_tickets: Arc<Mutex<usize>>
        = Arc::new(Mutex::new(250));

    let mut threads = Vec::new();
    for i in 0..10 {
        let remaining_tickets_handle = remaining_tickets.clone();
        threads.push(thread::spawn(move || {
            ticket_agent(i, remaining_tickets_handle);
        }));
    }
    // wait for all the threads to finish
    for handle in threads {
        handle.join().expect("Panic occurred in thread!");
    }
    println!("End of business day!");
}
```

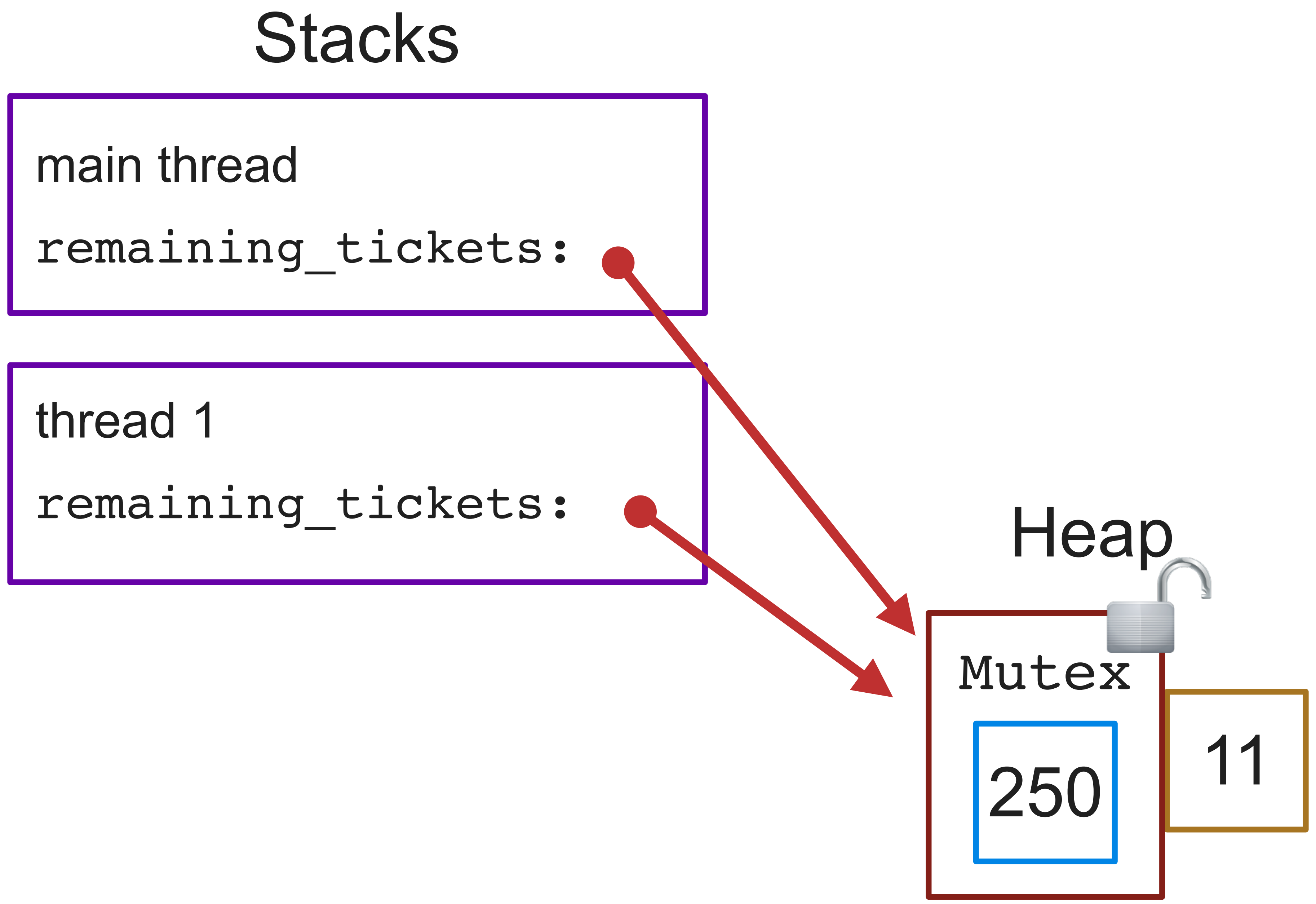


[Rust playground]

Attempt 3: Coordinated access with mutexes

```rust
fn ticket_agent(id: usize, remaining_tickets: Arc<Mutex<usize>>) {
    loop {
        let mut remaining_tickets_ref =
            remaining_tickets.lock().unwrap();
        if *remaining_tickets_ref == 0 {
            break;
        }
        handle_call();
        *remaining_tickets_ref -= 1;
        println!("Agent #{} sold a ticket! ({} more to be sold)",
            id, *remaining_tickets_ref);
        if should_take_break() {
            take_break();
        }
    }

    println!("Agent #{} notices all tickets are sold, and goes home!", id);
}
```



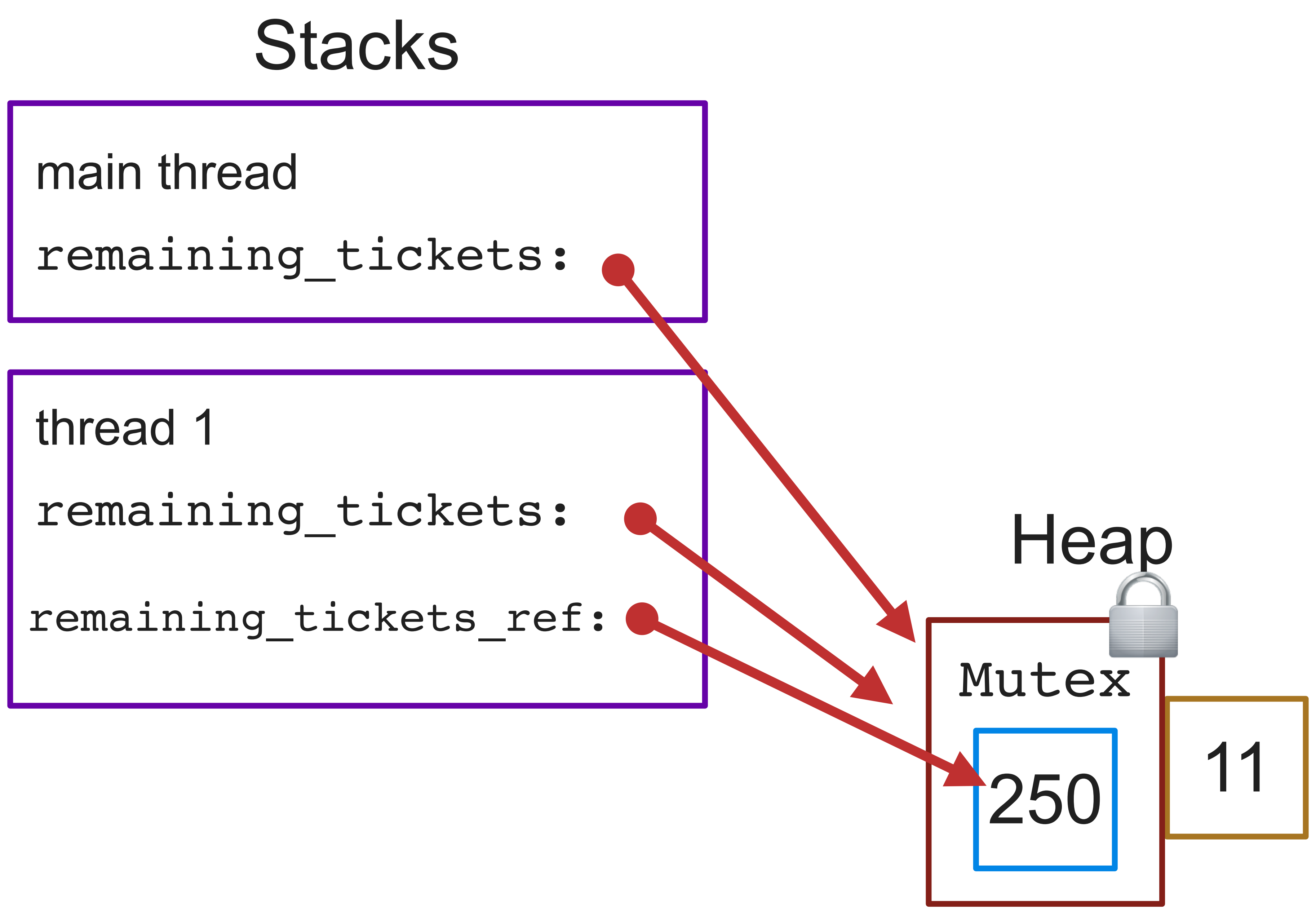
Rust playground

Attempt 3: Coordinated access with mutexes

```
fn ticket_agent(id: usize, remaining_tickets: Arc<Mutex<usize>>) {
    loop {
        let mut remaining_tickets_ref =
            remaining_tickets.lock().unwrap();
        if *remaining_tickets_ref == 0 {
            break;
        }
        handle_call();
        *remaining_tickets_ref -= 1;
        println!("Agent #{} sold a ticket! ({} more to be sold)",
            id, *remaining_tickets_ref);
        if should_take_break() {
            take_break();
        }
    }

    println!("Agent #{} notices all tickets are sold, and goes home!", id);
}
```

remaining_tickets_ref dropped at end of scope, lock is unlocked



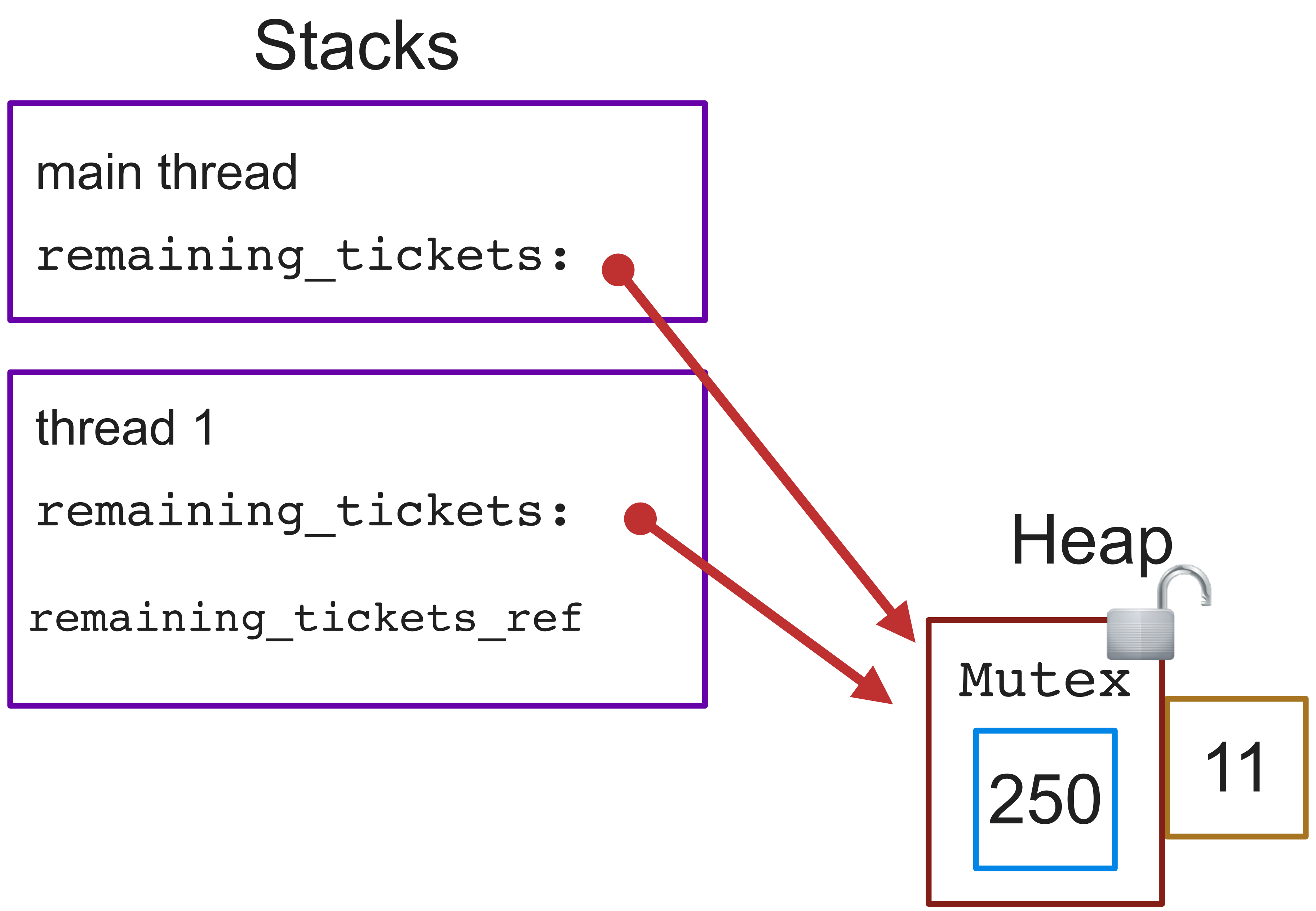
Rust playground

Attempt 3: Coordinated access with mutexes

```
fn ticket_agent(id: usize, remaining_tickets: Arc<Mutex<usize>>) {
    loop {
        let mut remaining_tickets_ref =
            remaining_tickets.lock().unwrap();
        if *remaining_tickets_ref == 0 {
            break;
        }
        handle_call();
        *remaining_tickets_ref -= 1;
        println!("Agent #{} sold a ticket! ({} more to be sold)",
            id, *remaining_tickets_ref);
        if should_take_break() {
            take_break();
        }
    }

    println!("Agent #{} notices all tickets are sold, and goes home!", id);
}
```

remaining_tickets_ref dropped at end of scope, lock is unlocked



Can't forget to unlock the lock 👍
But this code is completely serialized!!

[Rust playground]

Attempt 4: Releasing lock early

```
fn ticket_agent(id: usize, remaining_tickets: Arc<Mutex<usize>>) {
    loop {
        let mut remaining_tickets_ref =
            remaining_tickets.lock().unwrap();
        if *remaining_tickets_ref == 0 {
            break;
        }
        handle_call();
        *remaining_tickets_ref -= 1;
        println!("Agent #{} sold a ticket! ({} more to be sold)",
            id, *remaining_tickets_ref);
        drop(remaining_tickets_ref);
        if should_take_break() {
            take_break();
        }
    }

    println!("Agent #{} notices all tickets are sold, and goes home!", id);
}
```

Rust playground

Summary

- Rust does not prevent all race conditions, but it does prevent *data races*
  - Most common type of race condition in systems programming — big win!
  - This is also a huge advantage over other memory-safe languages. Garbage collection provides memory safety but not thread safety
- You still must be careful to avoid inadvertently serializing your code
- Deadlock can still be a problem