Multithreading in Rust: Shared Data

Ryan Eberhardt and Armin Namavari
May 7, 2020

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;

static void *recharge(void *args) {
  const char *name = kExtroverts[*(size_t *)args];
  printf("Hey, I'm %s.  Empowered to meet you.\n", name);
  return NULL;
}

int main() {
  printf("Let's hear from %zu extroverts.\n", kNumExtroverts);
  pthread_t extroverts[kNumExtroverts];
  for (size_t i = 0; i < kNumExtroverts; i++)
    pthread_create(&extroverts[i], NULL, recharge, &i);
  for (size_t j = 0; j < kNumExtroverts; j++)
    pthread_join(extroverts[j], NULL);
  printf("Everyone's recharged!\n");
  return 0;
}
```

Passes a pointer to i, but then the main thread changes i on the next iteration of the for loop

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 printer {}!", NAMES[i]);
        }));
    }
    // wait for all the threads to finish
    for handle in threads {
        handle.join().expect("Panic occurred in thread!");
    }
}
```

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 printer {}!", 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 printer {}!", 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 printer {}!", 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 printer {}!", 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?

```
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 printer {}!", NAMES[i]);
        }));
    }
    // wait for all the threads to finish
    for handle in threads {
        handle.join().expect("Panic occurred in thread!");
    }
}
```

Closure function takes ownership of i (under the hood, value of i is copied into thread's stack)

Rust playground

Ticket agents demo (CS 110)

```
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;
}
```

Multiple threads get mutable reference to remainingTickets

Value decremented simultaneously: ends up underflowing!

Cplayground

Attempt 1: Just Pass it in :^)

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

    let mut threads = Vec::new();
    for i in 0..10 {
        threads.push(thread::spawn(|| {
            ticketAgent(i, &mut remainingTickets)
        }));
    }
    // 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 2: RefCell and Rc

- Oh right, we need to move the value in
- Let’s just use RefCell and Rc
- Let's see how the Rust compiler feels about it

Attempt 3: Mutex and Arc

- We need to have memory that we can safely share between threads
- You can think of “Arc” as a thread safe version of the Rc safe pointer
- You can think of “Mutex” as a thread safe version of RefCell that allows exclusive access to the piece of data it wraps.
- Association between the lock and the data it protects!
- Deadlock danger: although the lock is released once the value returned by “.lock()” is dropped, you can still create situations with deadlock.
- Finished Example

Send and Sync

- Marker traits — you don’t implement functions for them, they serve a symbolic purpose
- Send: Transfer ownership (move) between threads
  - Rc can’t be Send: what if you clone() an Rc (so there are two handles to the underlying object + reference count), give one of those handles to a different thread, and the two threads update the reference count at the same time?
  - Arc implements the Send trait since the refcount update happens atomically. So does Mutex
- Sync: Allow this thing to be referenced from multiple threads
  - Mutex and Arc both implement Sync.
- Read more here

Link Explorer

- You and your friends are bored so you decided to play a game where you go to a random Wikipedia page and try to find a link to another wikipedia page that is the longest (by length of the html)
  - Trust me, it’s fun!
- You decide to enlist Rust (along with the reqwest and select crates) to help you.

Sequential Link Explorer

- The most straightforward approach
- No threads => no race conditions :^)
- Let’s see how fast it is…
- (code)

Multithreaded Link Explorer

- The web requests are network bound, so we can easily overlap the wait times for these requests by running them in separate threads.
- You can see this runs considerably faster!
- Problems
  - We have this funky batching thing going on — it's not super flexible and generalizable (what if we want to dynamically handle requests?)
  - We can easily reuse threads (really, we should be using a **threadpool** which you will implement in assignment 6 of CS110)

```
real    2m55.927s
user    0m7.964s
sys     0m1.722s
```

Sequential

```
real    0m9.932s
user    0m6.843s
sys     0m1.926s
```

Multithreaded

Next time

- Other synchronization primitives
- Beyond shared memory