LECTURE 22

Theory and Design of PL (CS 538)

April 13, 2020

SHARED-STATE CONCURRENCY

IDEA: IT'S GOOD TO SHARE

- All threads can modify shared data
- Benefits
  - Threads can work on data in place
  - Big savings if the shared data is big
  - One, consistent view of shared data
- Drawbacks
  - Not always safe to interrupt threads
  - Need to prevent data races

CRITICAL SECTIONS

- All parts of code accessing shared piece of data
- Only one thread allowed in section at a time
  - Otherwise: possible race condition
- Other threads must wait (“block”) until safe to enter

A BANK ACCOUNT

- Bank account tracks current account balance
- Operations to deposit/withdrawn money

```
struct Account { balance: i32 };

impl Account {
    fn deposit(&mut self, amt: i32) { /* ... */ }

    fn withdraw(&mut self, amt: i32) {
        if self.balance < amt {
            println!("Insufficient funds.");
        } else {
            self.balance -= amt;
        }
    }
}
```

WHAT COULD GO WRONG?

- Suppose `acct: Account` has balance 100
- Suppose two threads call `acct.withdraw(75)`
- Interleaving may cause **negative** balance
  1. T1 checks if balance is enough: OK
  2. Before deduct balance, T2 runs
  3. T2 checks if balance is enough: OK
  4. T1 deducts 75: balance is 25
  5. T2 deducts 75: balance is -25

LOCKS (MUTEXES)

- Main tool to enforce critical sections: *locks*
  - Threads can *acquire* or *release* locks
- Only one thread can hold a lock at a time
  - If T1 tries to acquire lock held by T2, it has to wait until the lock is released by T2
- Prevent data races by carefully using locks

A BANK ACCOUNT, BUT SAFER

- Following code gives the idea
  - Note: **not** real Rust code (stay tuned)

```
struct Account { balance: i32, mutex: Mutex };

impl Account {
    fn withdraw(&mut self, amt: i32) {
        self.mutex.get_lock();
        if self.balance < amt {
            println!("Insufficient funds.");
        } else {
            self.balance -= amt;
        }
        self.mutex.unlock();
    }
}
```

WHY DOES THIS WORK?

- Only one thread can get the lock
- If other thread tries to get lock, it must wait (block)
- Result: thread runs `withdraw` with no interruptions
  - Programmer doesn't pick which thread goes first

LOCKING DISCIPLINE

- More locks = more problems
- Programmer must coordinate how threads use locks
  - Which locks to acquire, when and where
  - What order to acquire locks
  - When and where to release locks
- Specified by programmer, not checked by compiler

EXAMPLES

- “When reading/writing, must hold global lock”
  - Strongly limits concurrency
- “When reading/writing x, must hold lock for x”
  - What if you need operate on two variables?
- Real schemes are usually much more complicated

MANY BUGS ARE POSSIBLE

- Too little locking or forget to take lock?
  - Data races and unpredictable behavior
- Too much locking?
  - Lots of threads waiting, little concurrency
- Forget to release lock?
  - Waiting threads will block forever

MESSAGE-PASSING CONCURRENCY

IDEA: SHARING IS A BAD IDEA

- Don’t share data between threads
  - Each thread operates on private data only
  - Threads send/receive messages
- Benefits
  - All interaction confined to thread input queues
  - No sharing = no data races = no manual locks
- Drawbacks
  - Inefficient if we need to send lots of data
  - Harder to synchronize, no common view of data

A BANK ACCOUNT, MESSAGE PASSING

- Idea: Clients send withdrawl messages
- Fancier: Clients can have message queues too

```
struct Client { the_acct: &mut Account };

impl Client {
  fn withdraw(&self, amt: i32) {
    the_acct.send_withdraw(amt);
  }
}
```

A BANK ACCOUNT, MESSAGE PASSING

- Idea: give Account a message queue
  - Again: not real Rust code (stay tuned)

```
enum AcctMsg { Withdraw(i32), // ... other messages ...  }
struct Account { balance: i32, msg: VecDeque<AcctMsg> };

impl Account {
  fn send_withdraw(&mut self, amt: i32) {
    self.msg.push_back(Withdraw(amt));
  }
  fn run_acct(&mut self) {
    loop {
      if let Some(Withdraw(amt)) = self.msg.pop_front() {
        if self.balance < amt {
          println!("Insufficient funds.");
        } else { self.balance -= amt; }
      } } } }
```

WHAT'S THE POINT?

- Account and Clients run in **separate** threads
- Account processes messages one at a time
  - Single thread: no overlapping withdraws!
- Synchronization needed only at message queue
  - If two clients send msgs, update queue correctly
- Reduce and restrict shared state
  - As much as possible: run logic in a single thread

CHANNELS

- Central abstraction for message passing
- Goes from thread(s) to other thread(s)
- Messages might arrive in any order
- Receiver sees a single stream of messages

CHANNEL OPERATIONS

- Each thread can send or receive message via channel
- Synchronous channels
  - Receiving waits until something arrives (blocking)
- Asynchronous channels
  - Try-receive: does not block if nothing incoming
  - Select: wait for msg. from any of these channels

DEADLOCK

CIRCULAR WAIT

- All threads blocked waiting for other threads
- Shared-state example
  - T1: take lock x, take lock y
  - T2: take lock y, take lock x
  - T1 takes x and T2 takes y: Deadlock!
- Obvious here, but harder with more locks/threads...
- Can happen under message-passing too
  - Two threads both waiting for message

DINING PHILOSOPHERS

- N philosophers (threads), sitting in a circle
- N forks (locks), one between every 2 philosophers
- Philosophers think, then eat, then think, …
- To eat: philosopher takes left fork, then right fork
- If all philosophers take left fork: stuck!

HOW TO FIX?

- One special philosopher: take **right** fork, then left
- Other philosophers: take left fork, then right
- Break the symmetry between threads
- In general: fixing deadlocks is very challenging

RUST THREADS

SPAWNING THREADS

- Rust function: `thread::spawn`
  - Caller passes in a closure for new thread to run
- Terminology: caller is *parent*, new thread is *child*

```
fn main() {
  thread::spawn( || {  // begin closure (child runs this)
      for i in 1..10 {
        println!("hi number {} from the spawned thread!", i);
        // do some stuff, sleep for 10 seconds, ...
      }
    } );  // end closure
}
```

RETURNS IMMEDIATELY

- Returns a handle, parent continues running
- Child thread may not finish before parent

```
fn main() {
  let child = thread::spawn( || {  // start child
      for i in 1..10 {
        println!("hi number {} from the spawned thread!", i);
        // do some stuff, sleep for 10 seconds, ...
      } });

  // parent thread continues
  for i in 1..5 {
    println!("hi number {} from the main thread!", i);
    // do some stuff, sleep for 10 seconds, ...
  }
}
```

JOINING THREADS

- `join`: wait for threads to finish
- Call with handle from spawn to wait for that thread

```
fn main() {
  let child = thread::spawn( || {
      for i in 1..10 {
        println!("hi number {} from the spawned thread!", i);
        // do some stuff, sleep for 10 seconds, ...
      } });

  for i in 1..5 {
    println!("hi number {} from the main thread!", i);
    // do some stuff, sleep for 10 seconds, ...
  }

  child.join(); // wait for child to finish
}
```

THREAD ENVIRONMENT

- Common case: parent wants to put data into thread
- Mechanism: closure mentions external variables
  - Must move or clone environment into each thread

```
let env_var: String = String::from("foo");
let child_one = thread::spawn( || {
  // Not OK: env_var doesn't live long enough
  let my_var = env_var;
  println!("Child 1 printing: {}", my_var);
});

let child_two = thread::spawn( move || {
  // OK: thread takes ownership of env_var
  let my_var = env_var;
  println!("Child 2 printing: {}", my_var);
});
```

EACH DATA HAS ONE OWNER

- Compiler: variable moved into at most one thread

```
let mut mut_var = String::new("foo");
let child_one = thread::spawn( move || {
  // OK: thread takes ownership of mut_var
  mut_var.push_str(" and bar");
});

let child_two = thread::spawn( move || {
  // Not OK: can't move mut_var again!
  mut_var.push_str(" and baz");
});
```

Rust compiler prevents data races!

OTHER RACES POSSIBLE

- Many resources not covered by aliasing rules
  - Printing lines to screen
  - Reading/writing from file system
  - Sending/receiving network packets
- Races in other *effects* not controlled by Rust

WHAT ABOUT SHARING?

- Can't share mutable access to same data
- But what about sharing *immutable* access?
- Does this work?

```
let env_var = String::new("foo");
let child_one = thread::spawn( || {
  // Thread borrows env_var immutably
  println!("Child 1 says: {}", env_var);
});

let child_two = thread::spawn( || {
  // Thread borrows env_var immutably
  println!("Child 2 says: {}", env_var);
});
```

DOESN'T LIVE LONG ENOUGH

- Parent may finish early, deallocate `env_var`
- Child threads may hold dangling reference...
- Try: use `Rc` to share ownership

```
let env_var_old = Rc::new(String::new("foo"));
let env_var_one = Rc::clone(&env_var_old);
let env_var_two = Rc::clone(&env_var_old);
let child_one = thread::spawn(move || {
  println!("Child 1 says: {}", env_var_one);
});

let child_two = thread::spawn(move || {
  println!("Child 2 says: {}", env_var_two);
});

// String will live as long as one Rc is still alive
```

STILL NOT HAPPY

- “Rc doesn’t implement Sync, try Arc”
- Now it works. But what was the problem?

```
let env_var_old = Arc::new(String::new("foo"));
let env_var_one = Arc::clone(&env_var_old);
let env_var_two = Arc::clone(&env_var_old);
let child_one = thread::spawn(move || {
  println!("Child 1 says: {}", env_var_one);
});

let child_two = thread::spawn(move || {
  println!("Child 2 says: {}", env_var_two);
});
```

TRAITS FOR THREAD-SAFETY

THREAD-SAFETY

- *Thread safe* data: safe to share between threads
- Interface and internals must be carefully designed
  - Multiple threads may operate on same data
  - Threads may call different operations
  - Resumed/paused/interrupted at any time

CHECKED BY RUST COMPILER

- By default, custom types are not thread safe
  - If you share between threads, bad stuff happens
- Much of standard library is thread safe
  - Vec, HashMap, String, …

Rust compiler complains if you don't use thread safe libraries with threads!

THREAD-SAFETY TRAITS

- Tracked at the type level through traits
  - `Send` trait: can be sent to another thread
  - `Sync` trait: can be shared by multiple threads
- Marker traits: no required implementations
  - Can't implement in safe Rust, essentially a promise
- Examples
  - `Rc` doesn't implement `Send` or `Sync`: not safe!
  - `Arc` implements `Send` and `Sync`: thread safe!