LECTURE 25

Theory and Design of PL (CS 538)

April 22, 2020

SCHEDULING NOTES

TAKEHOME FINAL: MAY 4 (MONDAY)

- Covers up to next Monday (inclusive)
- Open notes/computer/internet
- Think: mini versions of HW/WR assignments
- You will have 24 hours to complete it
  - Don’t spend 24 hours on it

Stay tuned for more detailed instructions

THREE MORE LECTURES

1. Asynchronous concurrency
2. More async in Rust
3. Unsafe Rust and wrapup

ASYNCHRONOUS RUST

- This stuff is very new: stabilized end of 2019
- Initial result of 4-5 years of discussions
- Ecosystem/libraries/patterns are still evolving
    - More extensions are planned

VERY IMPRESSIVE FEATURE

- Unlike C#/Python/JS, implemented as a Rust library
    - No runtime support built into language
- Similar to C or C++
    - Suitable for low=level settings: OS, embedded, etc.
- With all the benefits of Rust
    - No GC, no memory leaks/errors, no data races, …
- A bit of compiler support to make it easy to use

WE WILL GO (TOO) FAST

1. How do I use this thing?
    - Syntax, meaning, common pitfalls
2. Why are things designed this way?
    - Motivation, constraints, requirements
3. What’s really going on under the hood?
    - Gory details, optimizations, implementation

Not all material is equally important!

THE CONCURRENCY STORY SO FAR

CENTRAL TOOL: THREADS

1. Write a bunch of closures
2. Spawn a bunch of threads
3. Wait for threads to finish

PRE-EMPTIVE CONCURRENCY

- In Rust: threads provided by the OS
- OS scheduler decides which threads to run
- **Pre-emptive**: threads can be switched at any time
  - Don’t want one OS process to hog CPU

WHAT'S WRONG WITH THREADS?

- OS threads are heavy: require a lot of resources
    - Each thread has an execution stack
    - Tracks state; may pause at any point in time
- All OSes have thread limits
    - Few hundred threads OK, few thousand not OK

SOMETIMES, WE DON'T HAVE THREADS

- Implementing an OS kernel
- Code for embedded systems
    - E.g., programming a Raspberry Pi
- Code running client-side in browser
    - E.g., Javascript code or WebAssembly (wasm)

Q: *What if we need more concurrency?*

EXAMPLE: “C10K PROBLEM”

- Server: how to handle 10k concurrent connections?
    - Each one: tiny bit of compute, lots of waiting
- Need concurrency, but can’t spawn 10k threads
    - Spawn 1000 threads: lots of waiting
- Today: 1 machine can handle 1M-10M connections

COOPERATIVE MULTITASKING

THE MAIN IDEA

1. Run many tasks on the same thread
2. Tasks yield control when they need to wait
    - Yield = let other tasks run
3. **Cooperative**: tasks work together
    - Task keeps running until it yields

> *Use a small number of threads to handle a lot of concurrent tasks*

COROUTINES

- Proposed by Melvin Conway in 1958
- Generalizes subroutine/function call
- Subroutine: call, compute, return, done.
- Coroutine: call, compute, yield, call, compute, yield, ...
  - Can call/return more than once
  - Remembers state between calls

COROUTINE EXAMPLES: PYTHON

```
# Generator producing 0, 1, ..., n-1 one at a time
def firstn(n):
    num = 0
    while num < n:
        yield num  # return num to caller, suspend execution
        num += 1   # resume here next time generator called

gen = firstn(100);  # initialize generator

res0 = next(gen);   # 0
res1 = next(gen);   # 1
res2 = next(gen);   # 2
res3 = next(gen);   # 3
```

WHY COROUTINES?

1. PROGRAMMING PATTERN

- Natural for producers/consumers, pipelines
- Producer: coroutine that computes and yields
- Consumer: coroutine that accepts and computes
- Can be awkward to write with regular subroutines
    - Who is the caller? Who is called?
    - Don’t want to mash producer/consumer together

EXAMPLE: PRODUCER-CONSUMER

- Yield to another coroutine, not just to caller

```
var q := new queue

coroutine produce
    loop
        while q is not full
            create some new items
            add the items to q
        yield to consume

coroutine consume
    loop
        while q is not empty
            remove some items from q
            use the items
        yield to produce
```

2. BETTER PERFORMANCE

- Scheduler can be lighter
  - Don’t need to interrupt processes
- Tasks can be lighter
  - Yield when ready, not at random points in time
- More efficient context switches
  - Tasks can prepare for yield, save less state

NETWORKING AND DISK I/O

- One of the original motivating applications
- Network transmission, disk I/O are slow (vs CPU)
- Ideal properties for cooperative multitasking

1. Operations involve very little computation
2. Operations involve a lot of waiting

> *Potential for a lot more concurrency than one task per thread!*

COOPERATIVE VERSUS PRE-EMPTIVE

- Cooperative is not “better than” pre-emptive
  - Often more complex, error prone, messy
- Drawbacks can be avoided with runtime support
  - See Erlang, Go
- Sometimes: cooperative gives better performance

Generally: only use it if you need it!

IMPLEMENTING COROUTINES

LARGE DESIGN SPACE

- Many languages have coroutines
  - E.g., C#, Erlang, Go, JS, Kotlin, Python, Scala
- Many different tradeoffs. Two main choices:
  1. Stack or no stack?
  2. Who decides when to yield?

CHOICE 1: STACK OR NO STACK?

- Stackful coroutines (as seen in Go, …)
  - Each coroutine has an execution stack
  - AKA “green threads”, “fibers”, “goroutines”
- Stackless coroutines (as seen in Kotlin, Rust, …)
  - Coroutines do not have execution stacks

CHOICE 2: WHO DECIDES WHEN TO YIELD?

- Who decides when coroutines are ready to yield?
- Runtime (as seen in Go, …)
  - Runtime automatically swaps task when it blocks
  - Programmer doesn’t need to write yield
- Programmer (as seen in Kotlin, Rust, …)
  - Runtime doesn’t automatically swap tasks
  - Programmer write yield, makes sure not to block

BUILDING COROUTINES: STATE MACHINES

SIMPLE EXAMPLE: RESTAURANT WAITER

- Restaurant process
  1. Take food order
  2. Take drink order
  3. Make food: burger or pizza
  4. Make drink: milkshake or iced tea
  5. Wash dishes
- Each step may be very slow
  - Don’t block: yield control after each step

IN PSEUDOCODE

```
food = order_food();    // yield until order ready
drink = order_drink();  // yield until drink order ready
if food == burger
  make_burger();        // yield until burger ready
else
  make_pizza();         // yield until pizza ready
if drink == milkshake
  make_milkshake();     // yield until milkshake ready
else
  make_iced_tea();      // yield until tea ready
wash_dishes();          // yield until dishes ready
```

MODEL AS A STATE MACHINE

- States: places where we may need to wait
- Process starts in Start state
- At each step:
  - If process is ready, change state
  - If process not ready, yield control
- At end: process reaches Done state

STATE MACHINES TYPES

- First things first: let’s set up the types

```
enum Food { Burger, Pizza }
enum Drink { Milkshake, Tea }

enum WaiterState {
  Start,
  WaitingForFood,
  WaitingForDrink(Food),    // remember food order
  WaitingForBurger(Drink),  // remember drink order
  WaitingForPizza(Drink),   // remember drink order
  WaitingForMilkshake,
  WaitingForTea,
  WaitingForDishes,
  Done,
}
```

STATE MACHINE CODE

```rust
struct Waiter { state: WaiterState }
impl Waiter {
  fn step (&mut self) {
    match self.state {
      Start => { start_order_food(); self.state = WaitingForFood
      WaitingForFood => {
        if let Ready(food) = get_food_order() {
          start_order_drink();
          self.state = WaitingForDrink(food)
        }
      }
      // ...
    }
  }
}
```

STATE MACHINE, CONT'D

```
match self.state {
  // ...
  WaitingForDrink(food) => {
    if let Ready(drink) = get_drink_order() {
      if food == Burger {
        start_burger();
        self.state = WaitingForBurger(drink)
      } else {
        start_pizza();
        self.state = WaitingForPizza(drink)
      }
    }
  }
  WaitingForBurger(drink) => { /* ... */ }
  // ...
```

STATE MACHINE DRIVER

```
let mut waiter = Waiter { state: Start };

while waiter.state != Done {
  waiter.step();
}
```

WHAT’S WRONG WITH STATE MACHINES?

- Enums can be very complex
  - Complicated to track what data to save in states
- Easy to make mistakes
  - Need to make sure transitions are correct
- Just a pain in the ass to write!

COMPLEX EXAMPLE: A FASTER WAITER

- Two food items: burger or pizza
- Two drink items: milkshake or iced tea
- Restaurant process
    1. Take two food orders, then make food
    2. Take two drink orders, then make drinks
    3. After everything, wash dishes
- Fast waiter: 1. and 2. can happen simultaneously

WHAT DOES THE STATE LOOK LIKE?

- It’s looking pretty pretty ugly here...

```
enum WaiterState {
  Start,
  WaitFood1_WaitFood2,
  WaitFood1_WaitDrink2(Food),    // food for 2
  WaitFood1_WaitBurger2(Drink),  // drink for 2
  WaitFood1_WaitPizza2(Drink),   // drink for 2
  WaitFood1_WaitMilkshake2,
  WaitFood1_WaitTea2,
  WaitFood1_WaitDishes2,

  WaitDrink1_WaitFood2(Food),          // food for 1
  WaitDrink1_WaitDrink2(Food, Food),   // food for 1 and 2
  WaitDrink1_WaitBurger2(Food, Drink), // food for 1, drink for 2
  WaitDrink1_WaitPizza2(Food, Drink),  // food for 1, drink for 2
  WaitDrink1 WaitMilkshake2(Food),     // food for 1
```

BUILDING COROUTINES: (SIMPLE) FUTURES

COMBINE STATE MACHINES TOGETHER

- Two ingredients
    1. Building block state machines
    2. Ways to combine state machines
- We’ve seen this pattern before (e.g., parser)
- A state machine type has Future trait (“is a Future”)

SIMPLE FUTURES

- A simple version of the Rust Future trait

```
enum Poll<T> {
  NotReady,   // value not ready yet
  Ready(T)    // a value of type T is ready
}

trait Future {
  type Output;   // the thing that is produced

  // try to make a step in state machine
  // if state machine done, return `Ready`
  fn poll (&mut self) -> Poll<Self::Output>
}
```

HIDE STATES BEHIND ABSTRACTION

- Caller only cares about: are we there yet?
  - If done: get me the final result
  - If not done: try to make progress
- Each call to `poll` might advance state machine
  - Returns `Ready`: state machine done
  - Returns `NotReady`: did some work, not done yet
- Caller doesn't need to think about state!

COMBINING FUTURES: SEQUENCING

- State machines, just hidden

```
enum ThenState<Fut1, Fut2, F, T> {
  Start(Fut1, F),
  WaitingSecond(Fut2),
  Done(T),
}

fn then<Fut1, Fut2, F, T>(fst: Fut1, f: F)
    -> ThenState<Fut1, Fut2, F, T>
where
    Fut1: Future<Output = S>,
    F: FnOnce(S) -> Fut2,
    Fut2: Future<Output = T>,
{ Start(fst, f) }
```

COMBINING FUTURES: SEQUENCING

- How to run this state machine?

```
impl Future for ThenState<Fut1, Fut2, F, T> {
  type Output = T;
  fn poll(&mut self) -> Poll<T> {
    match self {
      Start(fut1, f) => {
        if let Ready(res1) = fut1.poll() {
          *self = WaitingSecond(f(res1)); return NotReady
        }
      }
      WaitingSecond(fut2) => {
        if let Ready(res2) = fut2.poll() {
          *self = Done(res2); return NotReady
        }
      }
      Done(res) => return Ready(res)
```

THIS PATTERN AGAIN...

- What the heck is this crazy type?

```
fn then<Fut1, Fut2, F, T>(first: Fut1, f: F)
   -> ThenState<Fut1, Fut2, F, T>
where
    Fut1: Future<Output = S>,
    F: FnOnce(S) -> Fut2,
    Fut2: Future<Output = T>,
```

- What would this look like in Haskell?

```
then :: Future S -> (S -> Future T) -> Future T

-- The same type as bind (>>=)... Future is a Monad!
```

COMBINING FUTURES: PARALLEL

- State machines, just hidden

```
enum JoinState<Fut1, Fut2, F, T> {
  Start(Fut1, Fut2),
  WaitingFirst(Fut1, T2),
  WaitingSecond(T1, Fut2),
  Done(T1, T2),
}

fn join<Fut1, Fut2, T1, T2>(fst: Fut1, snd: Fut2)
   -> JoinState<Fut1, Fut2, T1, T2>
where
    Fut1: Future<Output = T1>,
    Fut2: Future<Output = T2>,
{ Start(fst, snd)  }
```

COMBINING FUTURES: SEQUENCING

- How to run this state machine?

```
impl Future for JoinState<Fut1, Fut2, T1, T2> {
  type Output = (T1, T2);
  fn poll(&mut self) -> Poll<T> {
    match self {
      Start(fut1, fut2) => {
        match (fut1.poll(), fut2.poll()) {
          (Ready(res1), Ready(res2)) => *self = Done(res1, res2),
          (Ready(res1), NotReady) => *self = WaitingSecond(res1,
          (NotReady, Ready(res2)) => *self = WaitingFirst(fut1, r
          _ => (),
        }; return NotReady
      }
      WaitingFirst(fut1, res2) => {
        if let Ready(res2) = fut2.poll() {
          *self = Done(res1, res2); return NotReady
```

A DIFFERENT PATTERN...

- What the heck is this crazy type?

```
fn join<Fut1, Fut2, T1, T2>(fst: Fut1, snd: Fut2) -> JoinState<Fu
where
    Fut1: Future<Output = T1>,
    Fut2: Future<Output = T2>,
{ Start(fst, snd) }
```

- What would this look like in Haskell?

```
join :: Future S -> Future T -> Future (S, T)

-- For the curious: Future is a "strong monad"
```

RUNNING EXAMPLE

- Use Futures to model ops that take time to complete

```
// impl Future for FoodFuture { type Output = Food; ... }
let mut get_food_order: FoodFuture = ...;

// impl Future for DrinkFuture { type Output = (Drink, Food); ...
// Keep track of food order while getting drink
let mut get_drink_with_ord: Fn(Food) -> DrinkFuture = ...;

// impl Future for BurgerFuture { type Output = Drink; ... }
// Keep track of drink order while making burger
let mut make_burger_with_drink: Fn(Drink) -> BurgerFuture = ...;
let mut make_pizza_with_drink: Fn(Drink) -> PizzaFuture = ...;
let mut make_milkshake: MilkshakeFuture = ...;
let mut make_tea: TeaFuture = ...;
let mut wash_dishes: DishesFuture = ...;
```

RUNNING EXAMPLE

- Combine futures with combinators

```
let mut cust1 = get_food_order
  .then(|ord| get_drink_with_order(ord))
  .then(|(drink, ord)| {
      if ord == Burger { make_burger_with_drink(drink) }
      else { make_pizza_with_drink(drink) }
   })
  .then(|drink| {
      if drink == Milkshake { make_milkshake }
      else { make_tea }
   });
let mut cust2 = // ... same as cust1 ...
let mut waiter = future::join(cust1, cust2).then(|| wash_dishes);
```

EVEN FASTER WAITER

- Take food and drink orders in any order

```
let mut cust_food1 = get_food_order
  .then(|ord| {
      if ord == Burger { make_burger }
      else { make_pizza }
   })
let mut cust_drink1 = get_drink
  .then(|drink| {
      if drink == Milkshake { make_milkshake }
      else { make_tea }
   });
let mut cust_food2 = // ... same as cust_food1 ...
let mut cust_drink2 = // ... same as cust_drink1 ...
let mut waiter_future = future::join4(
  cust_food1, cust_food2, cust_drink1, cust_drink2)
  .then(|| wash_dishes);
```

FUTURE DRIVER

```
let mut waiter_future = ...;
let mut waiter_status = NotReady;

// repeatedly poll the future until it is ready
while waiter_status == NotReady {
  waiter_status = waiter_future.poll();
}

return waiter_status;  // at end: Ready(res)
```

WHAT'S GOOD ABOUT FUTURES?

- Much simpler than writing state machines by hand
- Combine in sequence or in parallel
- Uniform interface for futures: `poll`
- Libraries work generically with all futures
  - `FutureExt` for more combinators
  - `TryFutureExt` for working with Result futures

WHAT'S WRONG WITH FUTURES?

- Code can still be pretty ugly
  - Hard to understand, hard to debug
- Sometimes still need state machines by hand
  - What if we want to loop?
- Need to keep track of what state to save
  - E.g., drink order remembers food order
  - Especially tricky: remembering references