LECTURE 26

Theory and Design of PL (CS 538)

April 27, 2020

PLEASE COMPLETE COURSE EVALS!

RECAP: THE ASYNC STORY SO FAR

COOPERATIVE MULTITASKING

- Tasks decide when to yield, not forced to yield
- Scheduled by language runtime, not OS
- Useful when we want to run more tasks than OS limit

SUSPEND/RESUME TASKS

- Like threads, each task must be ready to suspend
- Suspends only happen at specific “yield” points
  - If a task doesn’t yield, it never suspends
- Tasks can be much lighter than OS threads

MOST BASIC: STATE MACHINES

- Model each task as a state machine
- Each state: task waits for X to happen/be ready
- To resume from state: check if X is ready
  - If X is ready, task goes to next state
  - If X not ready, yield control and try later
- Conceptually clean, but a huge pain to write

BETTER: THE FUTURES ABSTRACTION

- A Future: a value that will be ready later
  - Also known as a “promise”
- Wraps a state machine, client polls to check if ready
  - Ready: state machine is done, get final value
  - NotReady: made some progress, but not done yet
- Futures can be cleanly composed
  - Build complex state machines out of simple ones
- Still, writing code with futures is awkward

BUILDING COROUTINES: COMPILER SUPPORT

PYTHON GENERATORS, AGAIN

```
# 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
```

ISN'T THIS JUST AN ITERATOR?

- Indeed, we can do encode it as an Iterator

```
struct FirstNState { max: u32, num: u32 }
impl Iterator for FirstNState {
  type Item = u32;
  fn next(&mut self) -> Option<Self::Item> {
    if self.num < self.max {
      self.num += 1;
      Some(self.num - 1)
    } else {
      None
    }
  }
}
fn firstn(n: u32) -> FirstNState {
  FirstNState { max: n, num: 0 }
}
```

TRYING IT OUT

- Works just like we expected:

```
let mut gen = firstn(100);

res0 = gen.next();  // Some(0)
res1 = gen.next();  // Some(1)
res2 = gen.next();  // Some(2)
res3 = gen.next();  // Some(3)
```

BUT THIS IS A LOT OF TROUBLE

- Need to do a bunch of stuff:
  - Define iteration struct (FirstNState)
  - Implement Iterator correctly (next)
  - Define constructor (firstn)
- Python code with yield is much more natural
  - Easily expresses more complex generators

Can we just write "normal" code instead?

COMPILER BUILDS FUTURES

- Programmer can mark certain code as “future mode”
  - Code uses regular programming language (Rust)
- Programmer marks places where program may yield
- Compiler turns code into a future
  - Automatically generates the states (big enum)
  - Automatically figures out what state to remember
  - Automatically generates state transitions

ASYNC/AWAIT

- The idea and syntax is called async/await
- Adopted by many languages (C#, Python, JS, …)
- “async”: marks “future mode” code
- “await”: call other “future mode” code
  - Can only be done in “future mode”
  - Marks yield points: if called future not ready, yield

IN RUST: ASYNC BLOCK

- An async block looks something like this:

```
async { /* regular rust code */ }

async move { /* moves in env. variables */ }
```

- Last expr. is returned as the “result” of block
  - Should be a “regular” value, not a future
- Types: suppose “regular” return type is T
  - Then: async block has type “something implementing Future with Output = T”

EXAMPLE: ASYNC BLOCK

- Rust compiler turns an async block into a Future
- Can store this future in a variable, pass to fn, etc.

```
let my_async_block = async { 42 };  // you write this

// Compiler generates (something like) this:
enum AsyncState42 { Start, Done };
struct AsyncBlock42 { state: AsyncState42 };
impl Future for AsyncBlock42 {
  type Output = i32;
  fn poll(&mut self) -> Poll<i32> {
    if self.state == Start {
      *self.state = Done; Ready(42)
    } else {
      panic!("Already returned 42")
} } }
let my_async_block = AsyncBlock42 { state: Start };
```

IN RUST: ASYNC FN

- An async function ≈ async block with arguments
  - Inside function, write (mostly normal) Rust code
- Returns Future, but type doesn't mention Future

```
// you write this:
async fn my_async_fn(arg: Vec<i32>) -> String {
  /* body */
}

// compiler generates this:
fn my_async_fn(arg: Vec<i32>) -> FutStr {
  /* body converted into a Future */
}

// FutStr implements Future with Output = String
```

EVEN MORE GENERALLY

- FutStr name is compiler-generated, we don’t know it
- Can write this code:

```
// you write this:
async fn my_async_fn(arg: Vec<i32>) -> String {
  /* body */
}

// compiler generates this:
fn my_async_fn(arg: Vec<i32>) -> impl Future<Output = String> {
  /* body converted into a Future */
}

// Returns "something" impl. Future with Output = String
```

CALLING ASYNC FN

- Async fn are called just like regular fn
- Beware: they return a Future, not a “regular” value
  - They return a “recipe”, not a “cake”
- Calling an async fn doesn’t really do anything!
  - Doesn’t do I/O, send network packets, etc.

BIG PITFALL: THIS DOESN'T DO ANYTHING

```
let my_fut = async {
  let my_str = my_async_fn(vec![1, 2, 3]);
  // ... type of my_str isn't String ...
}
```

- When `my_fut` is polled, it doesn't do anything:
  1. Gets a Future and just stores it
  2. Doesn't do the work to produce the String!

IN RUST: AWAIT

- In async blocks/fns, can write `.await` after a Future
  - Can only use `await` in **async** context!
- If `fut` is a Future, `fut.await` means:

1. Check if `fut` is Ready (use `poll()`)
2. If Ready(val), unwrap it to val and continue
3. If NotReady, yield (return NotReady)

AWAIT IS MORE THAN A BIT LIKE ?

- In fns. returning Result, can write `?` after a Result
- If `res` is a Result, `res?` means:
  1. Check if `res` is Ok(...)
  2. If Ok(val), unwrap it to val and continue
  3. If Err(e), return Err(e) from fn

THIS CALL IS BETTER

```
let my_fut = async {
  let my_str = my_async_fn(vec![1, 2, 3]).await;
  // ... do stuff with my_str ...
}
```

- When polled, runs future from `my_async_fn`
  1. If it is Ready(str), assign str to `my_str`
  2. If it is NotReady, return NotReady
- "Wait for this thing to finish, then continue"

RUNNING EXAMPLE

- Set up a bunch of async fns:

```
async fn get_food_order() -> Food { /* ... */ }
async fn get_drink_order() -> Drink { /* ... */ }
async fn make_food(the_food: Food) -> () {
  if the_food = Burger {
    make_burger.await;
  } else {
    make_pizza.await;
  }
}
async fn make_drink(the_drink: Drink) -> () { /* ... */ }
async fn wash_dishes() -> () { /* ... */ }
```

RUNNING EXAMPLE

- Now, we can write the waiter using async/await

```
let serve_cust1_fut = async {
  let food = get_food_order().await;
  let drink = get_drink_order().await;
  make_food(food).await;
  make_drink(drink).await;
}
let serve_cust2_fut = async { /* ... */ }

let waiter_fut = async move {
  join(serve_cust1_fut, serve_cust2_fut).await;
  wash_dishes().await;
}
```

WHAT'S GOOD ABOUT ASYNC/AWAIT?

- Code is very natural: looks almost like regular code
- Compiler figures out how to make all the futures
  - Figures out what to remember
  - Generates the state machine, transitions
- Clearly marks points where async fn. may yield

WHAT'S WRONG WITH ASYNC/AWAIT?

- Calling regular fn. from async fn.: easy
- Calling async fn. from async fn.: OK (await)
- Calling async fn. from regular fn.: impossible
- Splits the language: async fn, or regular fn.?
  - Might need duplication: two versions of fns.
- See pros and cons

BIG PITFALL: BLOCKING IN ASYNC CODE

- Many stdlib calls “block”: might take a long time
  - `std::sync::Mutex::lock` (all of std::sync)
  - `std::fs::read` (all of std::fs, std::net)
  - `std::thread::sleep` (all of std::thread)
  - … many, many more
- These calls **do not yield**: will block state machine!
  - No compiler error, but much slower performance

Never use blocking calls in async code!!!

HOW TO RUN THE FUTURE?

A FUTURE IS A RECIPE

- So far, we've focused only on building Futures
- Future is just a recipe: it doesn't run itself!
- After building a Future, we want to run it
    - This runner is called an "async runtime"

A SIMPLE RUNTIME

- Takes a Future, polls it until it is done

```
fn run_fut<F, T>(fut: &mut F) -> T
where
  F: Future<Output = T>
{
  loop {
    if let Ready(result) = fut.poll() {
      return result;
    }
    // else, loop and try again
  }
}
```

WHAT'S WRONG WITH THIS SOLUTION?

- Only runs one Future
  - What if we want to run more than one?
- Repeatedly looping is wasteful
- Single threaded

WE WANT A FEW MORE THINGS

- Ability to run a large number of Futures
  - Schedule futures efficiently, switch, etc.
- Poll less: only poll when a Future is ready
  - But how do we know it's ready before polling??
- Run many futures on a small number of threads
  - Also known as "M:N" threading

GENERAL DESIGN OF ASYNC RUNTIMES

THREE MAIN PARTS

1. Executor: the thing that calls poll
2. Reactor: signals when things are ready
    - Typically: hooks into OS or hardware devices
    - I/O operation is done, timer goes off, etc.
3. Waker: conveys signal to executor

EXECUTOR

- We'll call a started Future a "task"
- Maintains two queues of tasks
  1. Ready queue: tasks that may be ready
  2. Waiting queue: tasks that are waiting
- Repeatedly gets a ready task, calls poll
  - If returns Ready, task is finished
  - If returns NotReady, put back on waiting queue
- Often (but not always) multi-threaded
  - Executor decides where to run tasks

REACTOR

- A Future that is not ready is waiting on something
  - A is waiting on B is waiting on C is waiting on ...
- Ultimately: waiting for some hardware event(s)
  - File read/write to finish, network packet to arrive
- Reactor monitors hardware, signals new events
  - Uses OS syscalls: epoll, kqueue, IOCP (cf. mio)

WAKERS

- Reactor uses Waker to signal Executor
  - Essentially, a callback used when hardware ready
- Associated to a task and an operation:
  - “When this operation is done, try task again”
- Sequence of events:
  1. Task X waits on I/O op, registers Waker WX, yields
  2. Hardware says I/O operation is done
  3. Reactor gets the Waker WX, calls it
  4. WX goes to Executor, puts X on the ready queue

THE REAL FUTURES TRAIT

```rust
pub trait Future {
  type Output;
  fn poll(
    self: Pin<&mut Self>,   // ignore Pin for now
    cx: &mut Context
  ) -> Poll<Self::Output>;
}
```

- `Context` holds a Waker, argument to `poll`
- `poll` threads the Waker through
  - Polling other, “child” futures: pass `cx` along
  - Waiting for “leafs” (I/O): register `cx` with Reactor

POLLING A FUTURE, TOP TO BOTTOM

- Say we have three Futures: A, B
  - A waits on B, B waits on file read
- Sequence of events: polling

1. Executor polls A, passes in Waker for A
2. Polling A polls B, passes in Waker for A
3. Polling B tries file read, passes in Waker for A
4. File read not ready, save Waker for A for this op

REACTING TO AN EVENT, BOTTOM TO TOP

- Sequence of events: reacting
  1. Reactor gets signal: file read is done
  2. Looks up Waker for this op, calls it
  3. Waker tells Executor to move A to ready queue
  4. Executor polls A, which polls B, …

RUST ASYNC RUNTIMES

TODAY: TWO MAIN LIBRARIES

- tokio
  - First major async runtime for Rust
  - Heavier: more complex, more features
- async-std
  - More recent async runtime for Rust
  - Lighter: less complex, less features

> *We'll talk about tokio, though the Rust async ecosystem is evolving rapidly*

ENTRY POINT

- tokio::runtime
- Main method: `block_on`
  - Pass it a future, run the task until it is done

```
use tokio::runtime::Runtime;

let mut rt = Runtime::new()?;  // make the Runtime

rt.block_on(async {
  let food = get_food_order().await;
  let drink = get_drink_order().await;
  make_food(food).await;
  make_drink(drink).await;
  // ...
});
```

SPAWNING TASKS

FUTURES FOR I/O

- tokio::{net, fs, signal, process}
- Rust stdlib has networking and file system calls
  - E.g., read from a file, write to a file, etc.
- These are synchronous: they block while waiting
  - Not suitable for use in async code!
- tokio has async versions of these standard calls
  - tokio's "leaf futures"
  - When waiting for read, register a Waker and yield

OTHER GOODIES

- tokio::sync
  - Async channels: communicate between tasks
  - Async mutexes: yield instead of blocking
- tokio::time
  - Delays: Put a task to sleep for some time
  - Timeouts: Cancel a task if too much time passes

MUCH MORE ON ASYNC/AWAIT

STREAMS

- Futures yields one T when done, after waiting
- Streams yield multiple Ts, after waiting
- Async counterpart of Iterator
  - If next item not ready, yield instead of blocking
- Natural abstraction (e.g., stream of HTTP requests)

TRAIT LOOKS SOMETHING LIKE THIS

```
pub trait Stream {
    type Item;
    fn poll_next(
        self: Pin<&mut Self>,
        cx: &mut Context
    ) -> Poll<Option<Self::Item>>;
}
```

- This returns an `Poll<Option<Item>>`
  - `NotReady`: next item not ready
  - `Ready(Some(item))`: next item ready
  - `Ready(None)`: stream finished

MORE ON STREAMS/GENERATORS

- Stream traits: here
- Streams and concurrency: here
- parallel-stream: here and here
- Combinators: StreamExt and TryStreamExt
- Generator design: here and here

EXAMPLES AND RESOURCES

- Building an executor/reactor: here and here
- Cooperative multitasking in an OS kernel: here

DESIGN NOTES

- Removing green threads from Rust: RFC
- Futures: here, here, and here
- Pin trait: 1 2 3 4 5 6 7
- Wakers: here and here
- async and borrowing: here
- async and destructors: here
- async/await syntax: here and here
- Scheduler design: here and here

PLEASE COMPLETE COURSE EVALS!