Lecture 18

Zero-Cost Async IO (Part 2)

Example: Let's sing a song

```
async fn learn_song() -> Song { /* ... */ }
async fn sing_song(song: Song) { /* ... */ }
async fn dance() { /* ... */ }
```

- We have to learn the song before we can sing it, but it's possible to dance at the same time as learning and singing.
- We can create two separate async fn which can be run concurrently:

Example: Let's sing a song

```
async fn learn_and_sing() {
  let song = learn_song().await;
  sing_song(song).await;
}

async fn async_main() {
  let f1 = learn_and_sing();
  let f2 = dance();

  futures::join!(f1, f2);
}

fn main() {
  block_on(async_main());
}
```

- Wait until the song is learned before singing it.
- use .**await** instead of **block_on** to prevent blocking the thread, which makes it possible to \`dance\` at the same time.

- **join!** can wait for multiple futures concurrently.
- **If** temporarily blocked in the \`learn_and_sing\` future, the *dance* future will take over the current thread.
- If *dance* is blocked,
- *learn_and_sing* can take back over.
- If both futare blocked, then *async_main* is blocked and will yield to the executor.

The Future Trait

- *SimpleFuture – lets start with a dummy basic version.*

```
trait SimpleFuture {
   type Output;
   fn poll(&mut self, wake: fn()) -> Poll<Self::Output>;
}

enum Poll<T> {
   Ready(T),
   Pending,
}
```

- Futures can be advanced by calling the poll function.

Example: Reading from a socket

- If data exists → read it; return Poll::Ready(data).
- if no data is ready → future is blocked.
- We register wake to be called when data becomes ready which will tell the executor that our future is ready.

```
pub struct SocketRead<'a> {
 socket: &'a Socket,
}
impl SimpleFuture for SocketRead<'_> {
 type Output = Vec<u8>;
 fn poll(&mut self, wake: fn()) -> Poll<Self::Output> {
  if self.socket.has_data_to_read() {
      Poll::Ready(self.socket.read_buf())
  } else {
     self.socket.set_readable_callback(wake);
     Poll::Pending
}}}
```

Example: Running futures (simultaneously)

```
pub struct Join<FutureA, FutureB> {
 a: Option<FutureA>,
 b: Option<FutureB>,
}
impl<FutureA, FutureB> SimpleFuture for Join<FutureA, FutureB>
where
 FutureA: SimpleFuture<Output = ()>,
 FutureB: SimpleFuture<Output = ()>,
{
 type Output = ();
 fn poll(&mut self, wake: fn()) -> Poll<Self::Output> {
   // Attempt to complete future `a`.
   if let Some(a) = &mut self.a {
     if let Poll::Ready(()) = a.poll(wake) {
      self.a.take();    }
   }
   if let Some(b) = &mut self.b {
     if let Poll::Ready(()) = b.poll(wake) {
      self.b.take();
   }  }
   if self.a.is_none() && self.b.is_none() {
     Poll::Ready(())
     } else {
       Poll::Pending
   }  } }
```

the real Future trait

- The Future trait can be used to express asynchronous control flow without requiring multiple allocated objects and deeply nested callbacks.

```
trait Future {
  type Output;
  fn poll(
  // Note the change from `&mut self` to `Pin<&mut 
Self>`:
  self: Pin<&mut Self>,
  // and the change from `wake: fn()` to `cx: &mut 
Context<'_>`:
  cx: &mut Context<'_>,
  ) -> Poll<Self::Output>;
}
```

the real Future trait

```
trait SimpleFuture {
  type Output;
  fn poll(&mut self, wake: fn()) -> Poll<Self::Output>;
}
```

```
trait Future {
  type Output;
  fn poll(
  // change from `&mut self` to `Pin<&mut Self>`:
  self: Pin<&mut Self>,
  // change from `wake: fn()` to `cx: &mu Context<'_>`:
  cx: &mut Context<'_>,
  ) -> Poll<Self::Output>;
}
```

Pin<&mut Self>

- Allows us to create futures that are immovable.
- A pinned reference guarantees that the value it refers to will never be moved again.
- A type which knows it will be pinned can be self-referential because it knows it won't be moved.

Rust solves all its runtime problems with smart pointer or box types.

Task Wakeups with Waker

- Each time a future is polled, it is polled as part of a "task".
- Tasks are the top-level futures that have been submitted to an executor.
- Waker provides a wake() method that can be used to tell the executor that the associated task should be awoken.
- When wake() is called, the executor knows that the task associated with the Waker is ready to make progress, and its future should be polled again.

Example: Build a Timer

```
pub struct TimerFuture {
  shared_state: Arc<Mutex<SharedState>>,
}
```

use a shared Arc<Mutex<..>> value to communicate between the thread and the future.

```
struct SharedState {
  completed: bool,
  waker: Option<Waker>,
}
```

- The waker for the task that **TimerFuture** is running on.
- The thread can use this after setting **completed = true** to tell **TimerFuture**'s task to wake up, see that **completed = true**, and move forward.

Example: Build a Timer

- If *shared_state.completed = false*, we clone the Waker for the current task and pass it to shared_state.waker so that the thread can wake the task back up.
- Update the Waker every time the future is polled because the future may have moved to a different task.

```
impl Future for TimerFuture {
 type Output = ();
 fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) ->
       Poll<Self::Output> {
 let mut shared_state =
       self.shared_state.lock().unwrap();
  if shared_state.completed {
   Poll::Ready(())
  } else {
   shared_state.waker = Some(cx.waker().clone());
   Poll::Pending
  } } }
```

Example: Build a Timer

```
impl TimerFuture {
 pub fn new(duration: Duration) -> Self {
  let shared_state = Arc::new(Mutex::new(SharedState {
    completed: false,
    waker: None,
   }));
  let thread_shared_state = shared_state.clone();
  thread::spawn(move || {
   thread::sleep(duration);
   let mut shared_state =
       thread_shared_state.lock().unwrap();
       shared_state.completed = true;
       if let Some(waker) = shared_state.waker.take() {
        waker.wake()
       }
     });
  TimerFuture { shared_state }
 }
}
```

More on async/.await

- There are two main ways to use async:

  async fn & **async blocks.**
- Each returns a value that implements the Future trait:

```
async fn foo() -> u8 { 5 }
```

foo() returns a type that implements **Future<Output = u8>**.
foo().await returns a value of type **u8**.

```
fn bar() -> impl Future<Output = u8> {
  async {
    let x: u8 = foo().await;
    x + 5
  }
}
```

async Lifetimes

- async fns which take references or other non-static arguments return a Future which is bounded by the lifetime of the arguments.

```
async fn foo(x: &u8) -> u8 { *x }
```

```
fn foo_expanded<'a>(x: &'a u8) -> impl Future<Output =
u8> + 'a {
    async move { *x }
}
```

async Lifetimes

- The future returned from an async fn must be .awaited while its non-'static arguments are still valid.
- One common workaround for turning an async fn with references-as-arguments into a 'static future is to bundle the arguments with the call to the async fn inside an async block.

```
fn bad() -> impl Future<Output = u8> {
 let x = 5;
 borrow_x(&x) // ERROR: `x` does not live long enough
}
fn good() -> impl Future<Output = u8> {
 async {
  let x = 5;
  borrow_x(&x).await
 }
}
```

async move

- async blocks and closures allow the move keyword.

```
async fn blocks() {
  let my_string = "foo".to_string();
  let future_one = async {println!("{}", my_string);};
  let future_two = async {println!("{}", my_string);};
  let ((), ()) = futures::join!(future_one,
       future_two);
}


fn move_block() -> impl Future<Output = ()> {
  let my_string = "foo".to_string();
  async move {println!("{}", my_string); }
}
```

Pinning

- To poll futures, they must be pinned using a special type called Pin<T>.
- Pinning guarantees that an object won't ever be moved.
- Why this is necessary? remember how async/.await works:

```
let fut_one = /* ... */;
let fut_two = /* ... */;
async move {
    fut_one.await;
    fut_two.await;
}
```

Pinning

- Previous code creates an anonymous type that implements a Future, providing a poll method that looks something like this:

```
struct AsyncFuture {
    fut_one: FutOne,
    fut_two: FutTwo,
    state: State,
}
enum State {
    AwaitingFutOne,
    AwaitingFutTwo,
    Done,
}
```

Pinning

```
impl Future for AsyncFuture {
 type Output = ();
 fn poll(mut self: Pin<&mut Self>, cx: &mut Context<'_>) ->
        Poll<()> {
  loop {
   match self.state {
    State::AwaitingFutOne => match self.fut_one.poll(..) {
     Poll::Ready(()) => self.state = State::AwaitingFutTwo,
     Poll::Pending => return Poll::Pending,
   }
    State::AwaitingFutTwo => match self.fut_two.poll(..) {
     Poll::Ready(()) => self.state = State::Done,
     Poll::Pending => return Poll::Pending,
   }
    State::Done => return Poll::Ready(()),
   }
  }
 }
}
```

Pinning

- When poll is first called, it will poll fut_one.
- If fut_one can't complete, AsyncFuture::poll will return.
- Further calls to poll will pick up where the previous one left off.
- This process continues until the future is able to successfully complete.
- What happens if we have an async block that uses references?

```
async {
    let mut x = [0; 128];
    let read_into_buf_fut = read_into_buf(&mut x);
    read_into_buf_fut.await;
    println!("{:?}", x);
}
```

Pinning

- What struct does this compile down to?

```
struct ReadIntoBuf<'a> {
    buf: &'a mut [u8], // points to `x` below
}

struct AsyncFuture {
    x: [u8; 128],
    read_into_buf_fut: ReadIntoBuf<'what_lifetime?>,
}
```

- ReadIntoBuf holds a reference into the other field of x.
- However, if AsyncFuture is moved, the location of x will move, invalidating the pointer in read_into_buf_fut.buf.

Pinning prevents this problem, making it safe to create references to values inside an async block.

How to Use Pinning?

- Pin wraps pointer types, guaranteeing that the values behind the pointer won't be moved.
- Most types don't have a problem being moved.
- These types implement a trait called Unpin.
- Pointers to Unpin types can be freely placed into or taken out of Pin.

How to Use Pinning?

- Some functions require the futures they work with to be Unpin.
- To use a Future or Stream that isn't Unpin with a function that requires Unpin types:

```
use pin_utils::pin_mut;

fn execute_unpin_future(x: impl Future<Output = ()> + Unpin) { /* ...
*/ }
let fut = async { /* ... */ };
execute_unpin_future(fut); // Error: `fut` does not implement `Unpin`
trait
// Pinning with `Box`:
let fut = async { /* ... */ };
let fut = Box::pin(fut);
execute_unpin_future(fut); // OK

// Pinning with `pin_mut!`:
let fut = async { /* ... */ };
pin_mut!(fut);
execute_unpin_future(fut); // OK
```

The Stream Trait

- Stream is similar to Future but can yield multiple values before completing. Think Iterators!!!

```
trait Stream {

  type Item;

  fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>)
        -> Poll<Option<Self::Item>>;
}
```

The Stream Trait

- One common example of a Stream is the Receiver for the channel type from the futures crate.
- It will yield Some(val) every time a value is sent from the Sender end, and will yield None once the Sender has been dropped and all pending messages have been received:

```
async fn send_recv() {
    const BUFFER_SIZE: usize = 10;
    let (mut tx, mut rx) = mpsc::channel::<i32>(BUFFER_SIZE);

    tx.send(1).await.unwrap();
    tx.send(2).await.unwrap();
    drop(tx);

    // `StreamExt::next` is similar to `Iterator::next`, but returns a
    // type that implements `Future<Output = Option<T>>`.
    assert_eq!(Some(1), rx.next().await);
    assert_eq!(Some(2), rx.next().await);
    assert_eq!(None, rx.next().await);
}
```

The Stream Trait

- There are many different ways to iterate over and process the values in a Stream.

map, filter, fold, try_map, try_filter, and try_fold

```
async fn sum_with_next(mut stream: Pin<&mut dyn Stream<Item = i32>>) ->
i32 {
  use futures::stream::StreamExt; // for `next`
  let mut sum = 0;
  while let Some(item) = stream.next().await { sum += item; }
  sum
}
async fn sum_with_try_next(
  mut stream: Pin<&mut dyn Stream<Item = Result<i32,
         io::Error>>>,) -> Result<i32, io::Error> {
  use futures::stream::TryStreamExt; // for `try_next`
  let mut sum = 0;
  while let Some(item) = stream.try_next().await? {
      sum += item;
  }
  Ok(sum)
}
```

The Stream Trait

- To process multiple items from a stream concurrently, use the *for_each_concurrent* and *try_for_each_concurrent* methods:

```
async fn jump_around(mut stream: Pin<&mut dyn Stream<Item =
Result<u8, io::Error>>>,) -> Result<(), io::Error> {
  use futures::stream::TryStreamExt;
  const MAX_CONCURRENT_JUMPERS: usize = 100;

  stream.try_for_each_concurrent(MAX_CONCURRENT_JUMPERS, |num|
       async move {
    jump_n_times(num).await?;
    report_n_jumps(num).await?;
    Ok(())
  }).await?;
  Ok(())
}
```

Async I/O – Final Thoughts, JavaScript???

Async I/O –
Some Low-level Details

Futures

- **Async IO:** Immediately return a future which eventually evaluates to the response.
- **Timeout:** A future that finishes when that much time has passed.
- **CPU-intensive work:** Perform work on a separate thread pool, future resolves when the work is finished

State Machines

- Each future is represented as a state machine allocated in a single heap allocation.
- The state machine has one variant per IO event, which stores the state needed to restore the future at that point.
- This has perfect memory overhead: one allocation per task with exactly the right size, no larger.

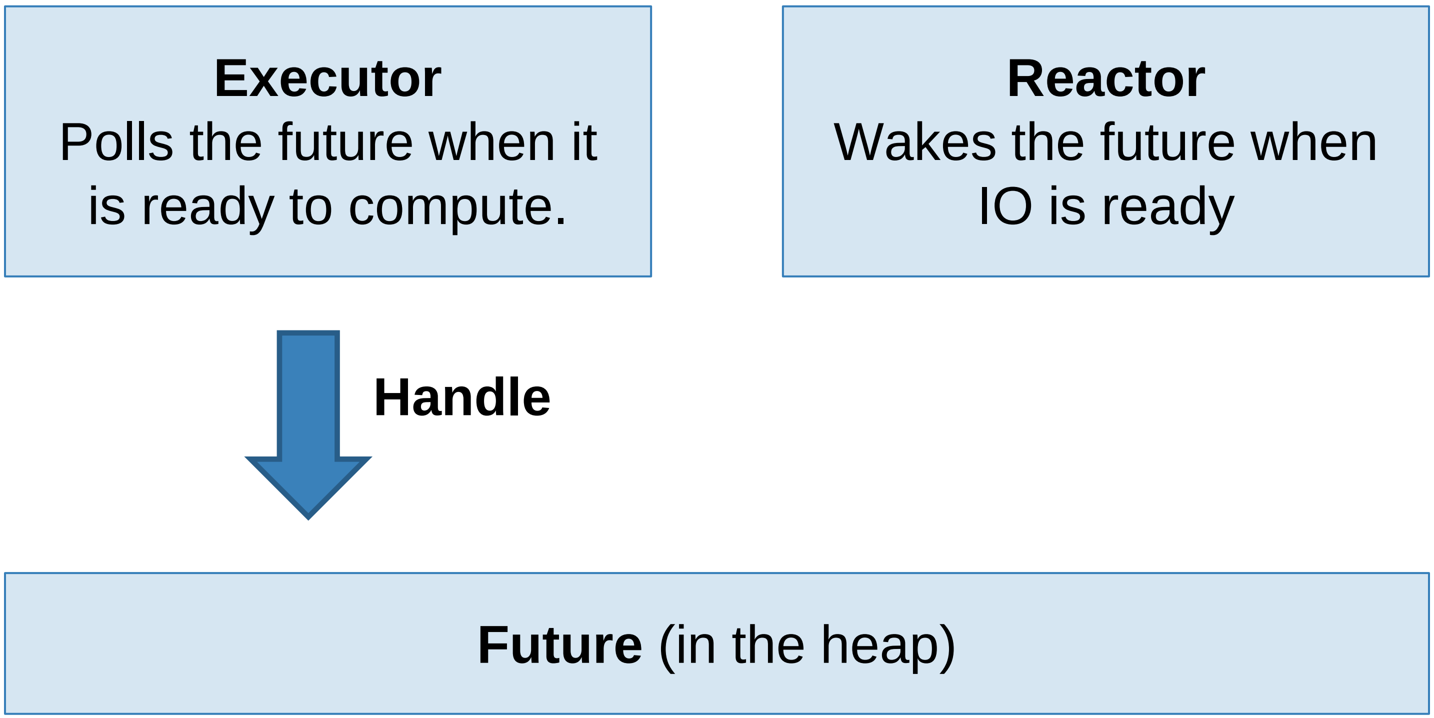
State Machines

Discriminant	State
Begin	Initial State
FirstIO	State needed to resume after 1st IO event
SecondIO	State needed after 2nd IO event
Finish	

Size of the entire future
(in a single heap
allocation)

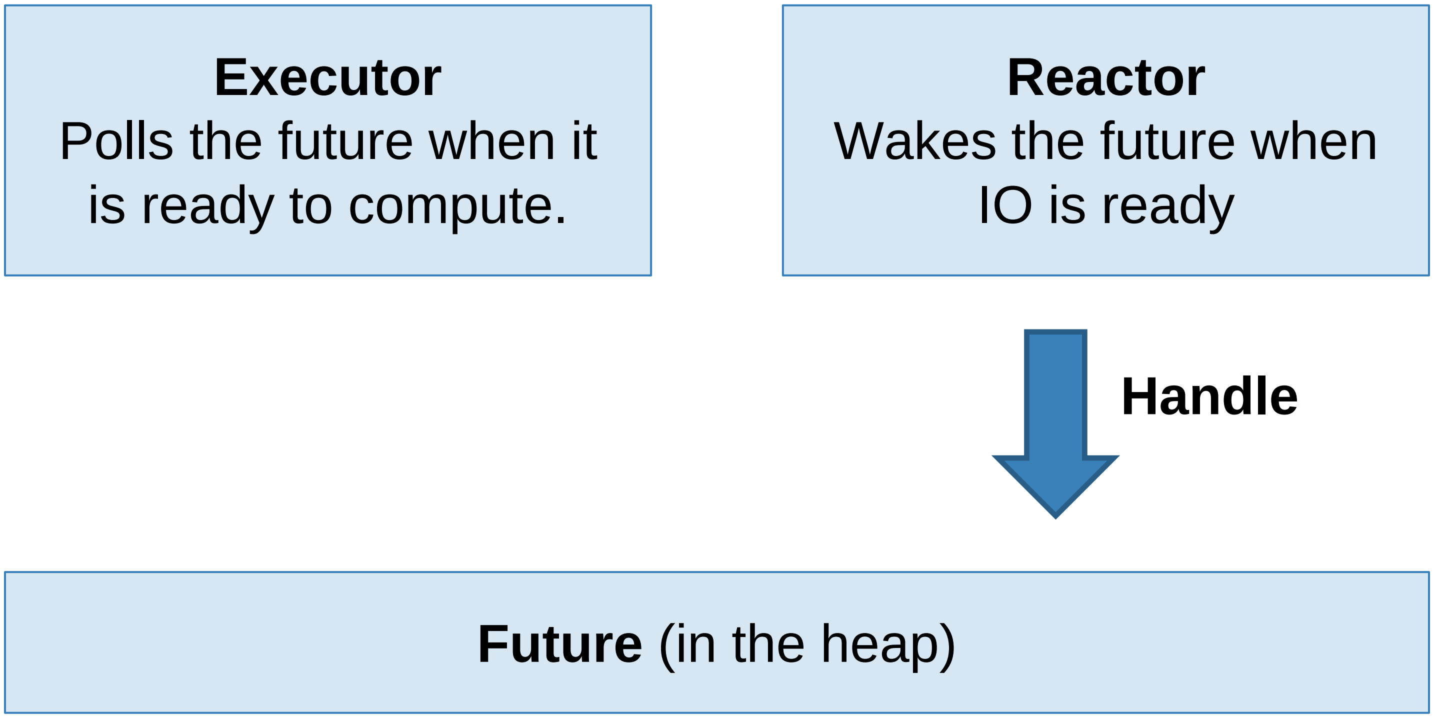
Poll/Wake Cycle

- Executor polls the future until it performs IO.



Poll/Wake Cycle

- Reactor wakes the future when the IO is complete.



Poll/Wake Cycle

- Future evaluates to its final result.

Executor
Polls the future when it is ready to compute.

Reactor
Wakes the future when IO is ready

Handle

Future (in the heap)

Poll/Wake Cycle

Poll
Executor polls the future until it returns Pending

Wait
Reactor registers that this future is listening for an event

Wake
The event occurs & the reactor wakes this future up to be polled again

Async/await Tomorrow

- **Streams:** asynchronous iterators. Very important for things like streaming HTTP, WebSockets, etc.
- **Async fn in traits:** currently not possible but a high priority addition.
- **Generators:** add the yield keyword to make it easy to write iterators and streams.

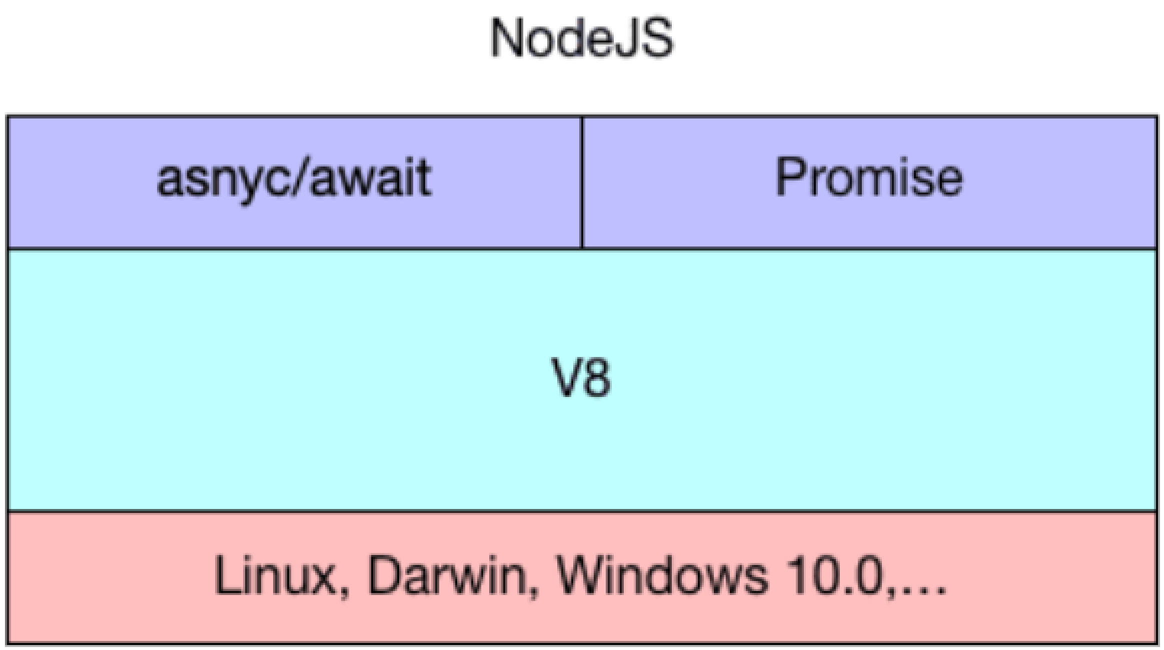
Async Rust vs. Node

NodeJS vs. Rust

- Now, Lets look at how Node and Rust are implementing the concepts we talked about, namely:
  - Syntax,
  - Type, and
  - Runtime.

NodeJS

- In NodeJS you have the async/await syntax and Promises.
- You can await a Promise (an action) which might need more time to process.



NodeJS

- In NodeJS you have the async/await syntax and Promises.
- You can await a Promise (an action) which might need more time to process.

```
const async_method = async () => {
    const dbResults = await dbQuery();
    const results = await serviceCall(dbResults);
    console.log(results);
}
```

Rust

- The Rust Async ecosystem is still in progress…
- Also use async/await.
- Instead of Promises, you have Futures.
- Does not to include any runtime.
- Rust wants to be as small as possible, and to be able to swap parts in and out as needed.
- Therefore you need to rely on crates to provide the appropriate runtime for you.

Rust

- The most popular one is tokio, which uses mio internally as its event queue.
- Even other runtimes are using mio since it's providing abstraction over kernel methods like epoll , kqueue and IOCP.

Rust Async in Detail