Asynchronous Programming in Rust

CS3211 Parallel and Concurrent Programming

Outline

- Non-blocking I/O
- Asynchronous programming paradigm
- From Futures to `async/.await`

Last week

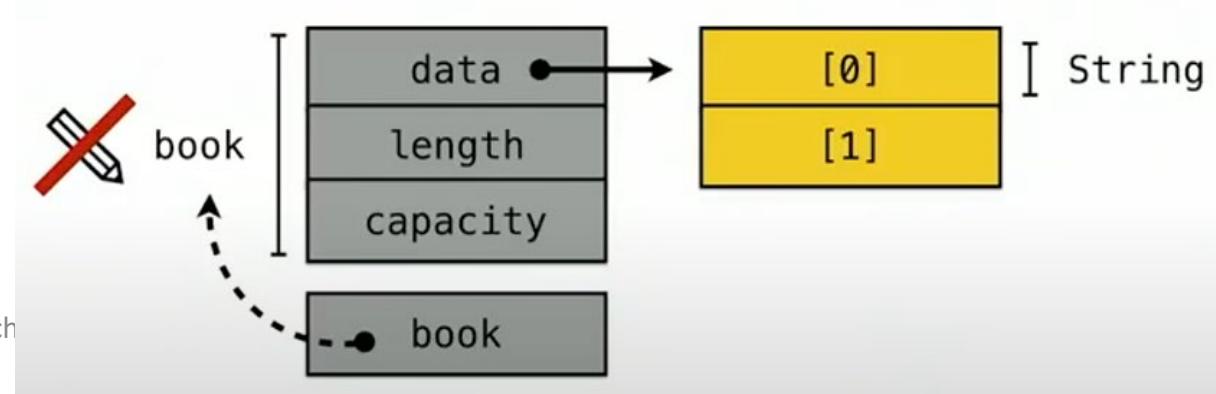
- Safety – due to ownership and borrowing
  - Memory safety
  - Thread safety
  - Rust code does not **compile** if it is not safe

Recap: ownership and borrowing

- Ownership
- Shared (immutable) borrow
- Mutable borrow

```
 1 fn main() {
 2     let mut book = Vec::new();
 3     book.push(...);
 4     book.push(...);
 5     publish(book);
 6     publish(book);
 7 }
 8
 9 fn publish(book: Vec<String>) {
10     ...
11 }
```

```
 1 fn main() {
 2     let mut book = Vec::new();
 3     book.push(...);
 4     book.push(...);
 5     publish(&book);
 6     publish(&book);
 7 }
 8
 9 fn publish(book: &Vec<String>) {
10     ...
11 }
```



Closures

- Can be called like a function
  - But they are not just simple functions
- Can be passed as parameters and returned to and from functions

Overheads of threads

- Context switching cost: threads give up the rest of the CPU time slice when **blocking** functions are called, and a switch happens
  - Registers get restored, virtual address space gets switched, cache gets stepped on, etc.
  - Big cost for high-performance situations (servers) with multiple threads, each handling a connection
- Memory overhead
  - Each thread has its own stack space that needs to get managed by the OS
  - Trying to have 5000 concurrent connections?
    - 5000 threads = 5000 stack segments = 40GB at 8MB/stack!

Time slice and threads

CPU:

Finally, the CPU is mine at long last!

```rust
fn main() {
    let listener: TcpListener = TcpListener::bind(addr: "127.0.0.1:25565").unwrap();
    for stream_res: Result<TcpStream, Error> in listener.incoming() {
        let mut stream: TcpStream = stream_res.unwrap();
        thread::spawn(move|| {
            let mut str: String = String::new();
            stream.read_to_string(buf: &mut str).unwrap();
        });
    }
}
```

Time slice and threads

CPU:

Finally, the CPU is mine at long last!

```rust
fn main() {
    let listener: TcpListener = TcpListener::bind(addr: "127.0.0.1:25565").unwrap();
    for stream_res: Result<TcpStream, Error> in listener.incoming() {
        let mut stream: TcpStream = stream_res.unwrap();
        thread::spawn(move|| {
            let mut str: String = String::new();
            stream.read_to_string(buf: &mut str).unwrap();
        });
    }
}
```

The read() system call can block! (Network connection slow, malicious client, etc).

Time slice and threads

CPU:

Nvm... Time to nap instead...

```rust
fn main() {
    let listener: TcpListener = TcpListener::bind(addr: "127.0.0.1:25565").unwrap();
    for stream_res: Result<TcpStream, Error> in listener.incoming() {
        let mut stream: TcpStream = stream_res.unwrap();
        thread::spawn(move|| {
            let mut str: String = String::new();
            stream.read_to_string(buf: &mut str).unwrap();
        });
    }
}
```

The read() system call can block! (Network connection slow, malicious client, etc).

What if this thread could've just served another request while also waiting for this one?

Mitigate the disadvantages of threads

- Context switches are expensive
  - Use lightweight threads (green threads)
    - Usually they need a runtime (like in Go)

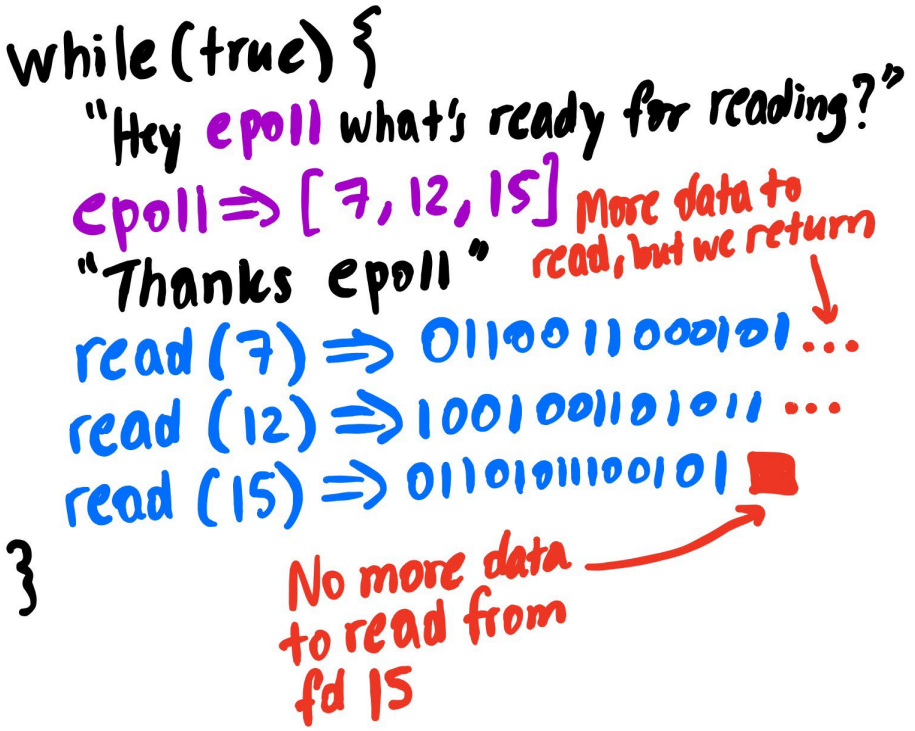
- Is there a way we can have concurrency with less penalties?
  - Non-blocking I/O

Introducing non-blocking I/O

- For example, the `read()` sys call would block if there is more data to be read but not available
  - Thread gets pulled off the CPU and it cannot do anything else in the meantime
- Instead, we could have `read()` return a special error value instead of blocking
  - If the client hasn't sent anything yet, the thread can do other useful work, e.g. reading from other descriptors
- **Non-blocking I/O enables concurrency with one thread!**

Non-blocking I/O

- `epoll` is a kernel-provided mechanism that notifies us of what file descriptors are ready for I/O
- Scenario: we are a server having conversations with multiple clients at the same time (multiple fds)
- `Epoll` notifies the thread when a client said something
- `read()` from each of those file descriptors, continue those conversations
- Rinse and repeat → event loop



State management

- Key problem: manage the state associated with each conversation
- Imagine trying to cook 10 dishes at the same time. Need to remember…
  - How long each thing has been on the stove
  - How long things have been in the oven
  - How long things have been marinating for
  - What the next step is for each dish

"Executor Thread"

State management

- Actual applications:
  - Was I waiting for the client to send me something, or was I in the middle of sending something to the client?
  - What was the client asking for before I got distracted?
  - Charlie the Client asked me for her emails, but I needed to get them from Dan the Database
  - Now Dan the Database responded with some info, but I can't remember what I was supposed to do with it
- Managing one connection in each thread is easy because each conversation is an independent train of thought

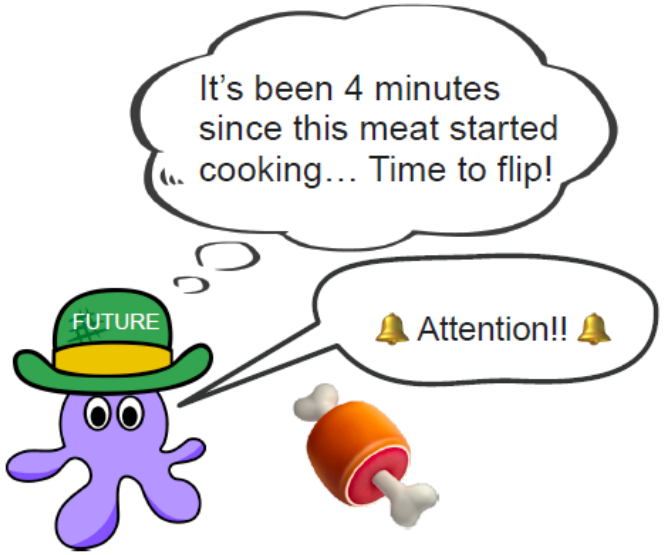
Non-blocking I/O is nice in theory, but managing state seems hard

"Executor Thread"

State management

- Rust (and a handful of other languages) take state management to the next level
- **Futures** allow us to keep track of in-progress operations along with associated state, in one package
  - Think of a future as a helper friend that oversees each operation, remembering any associated state

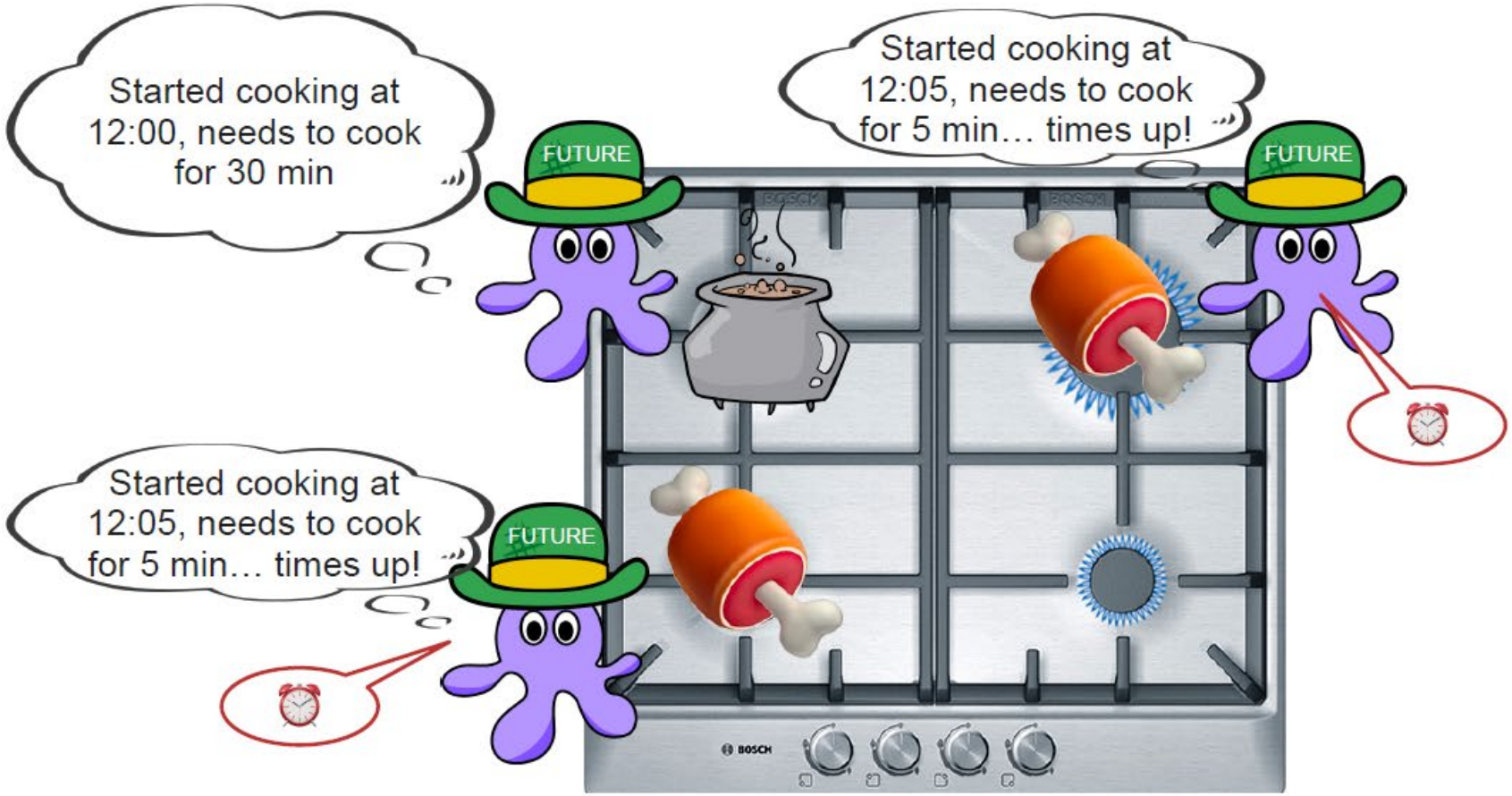
"Executor Thread"



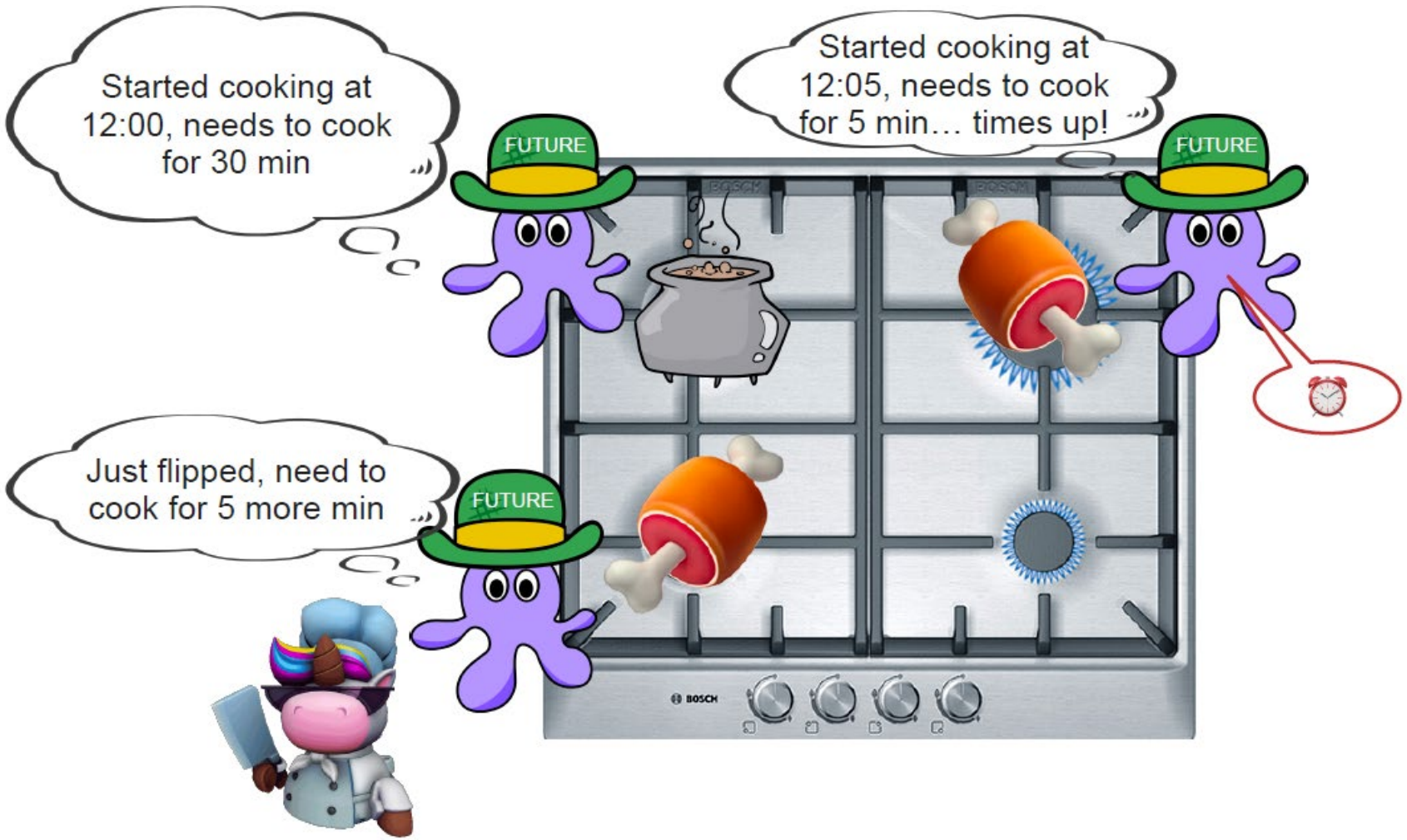
cookMeat future

Futures visualized

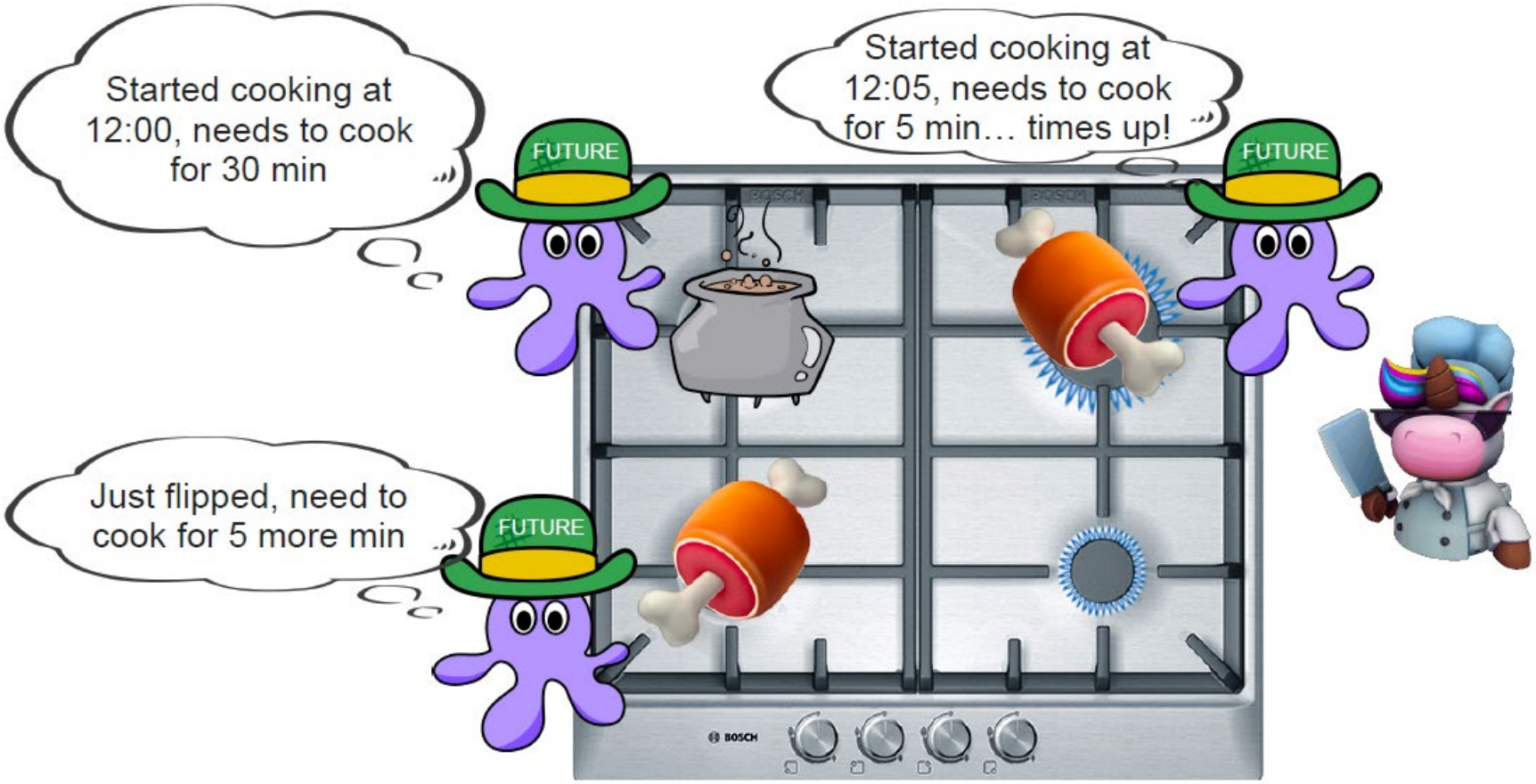
"Executor Thread"



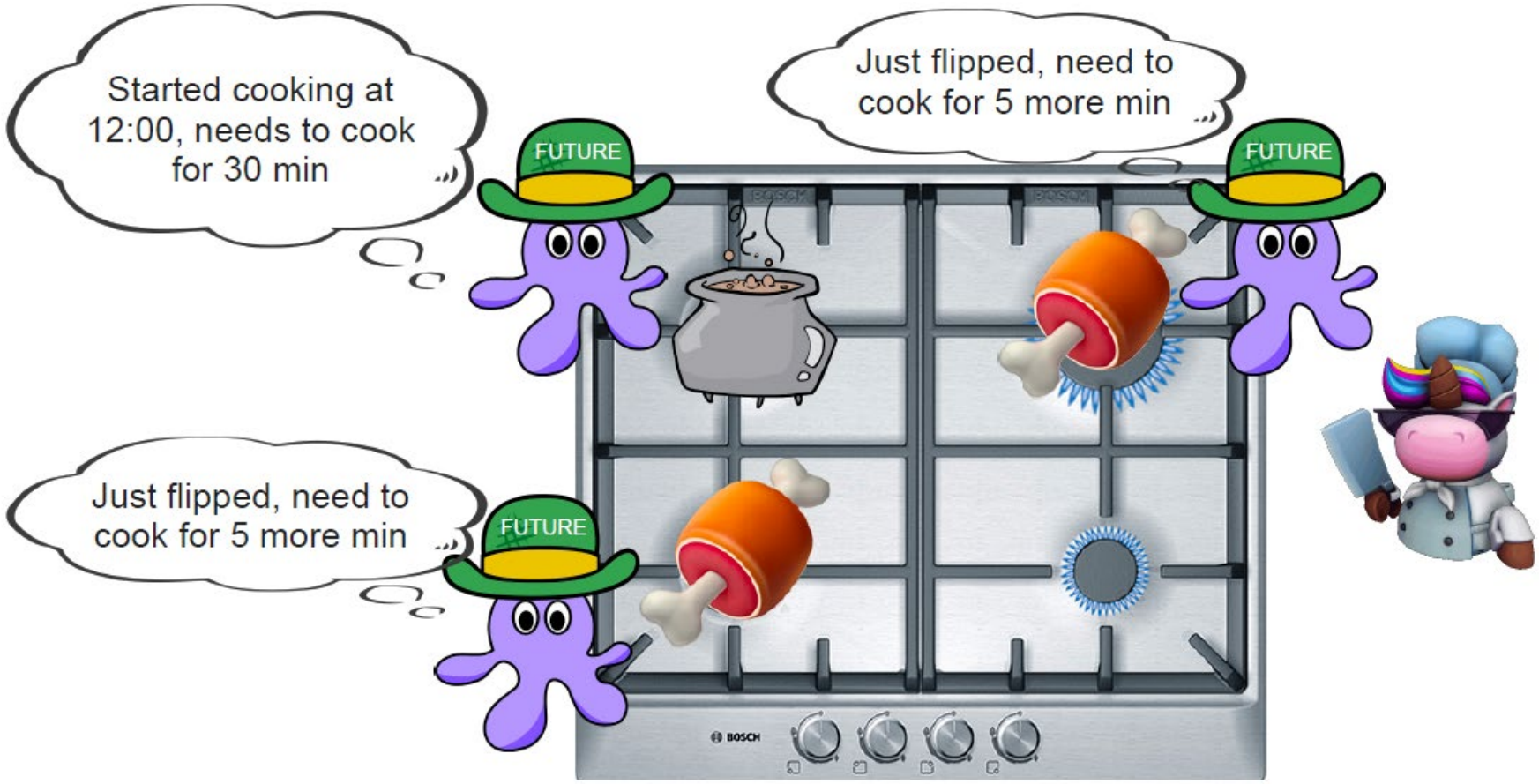
Futures visualized



Futures visualized

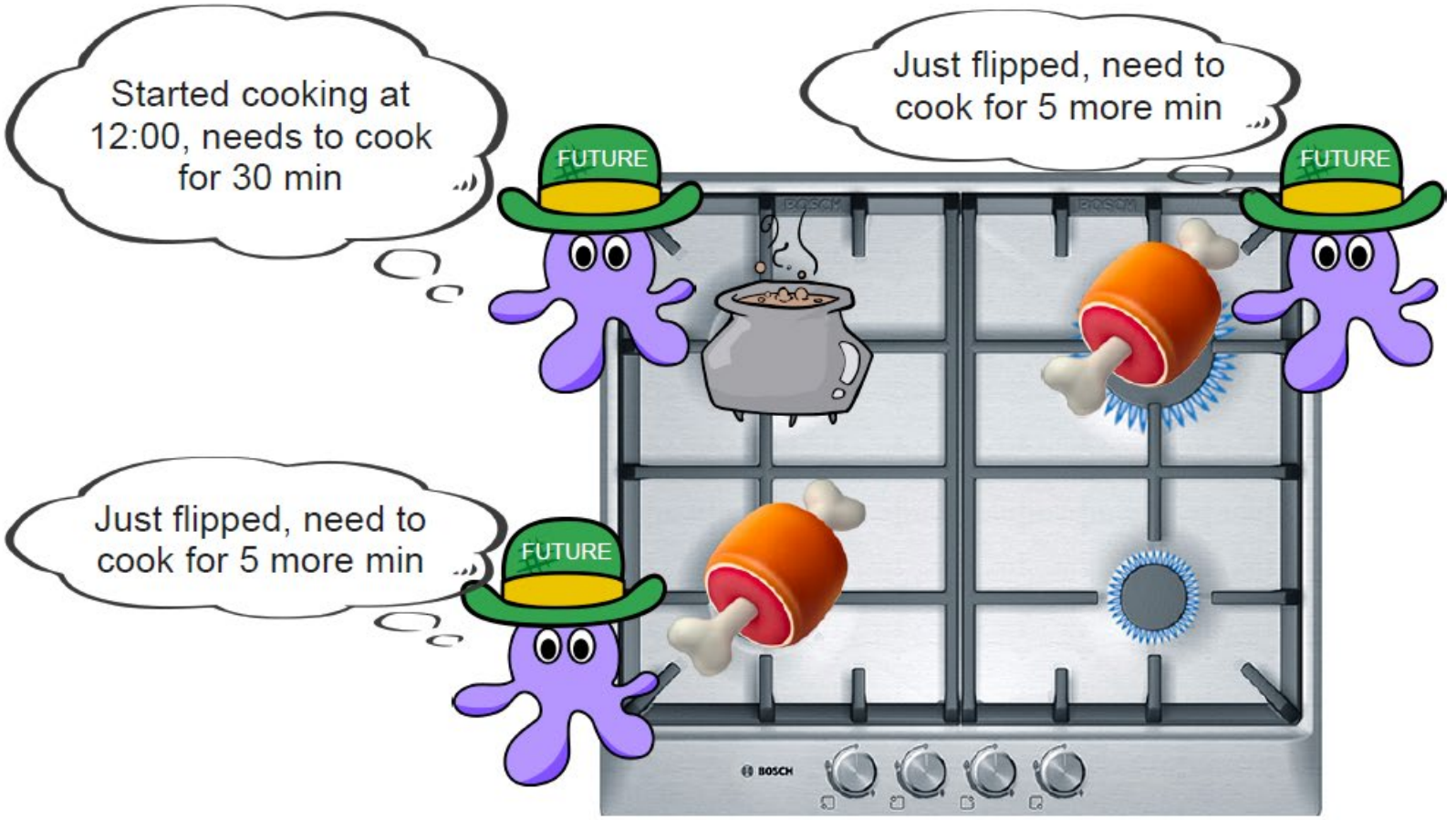


Futures visualized



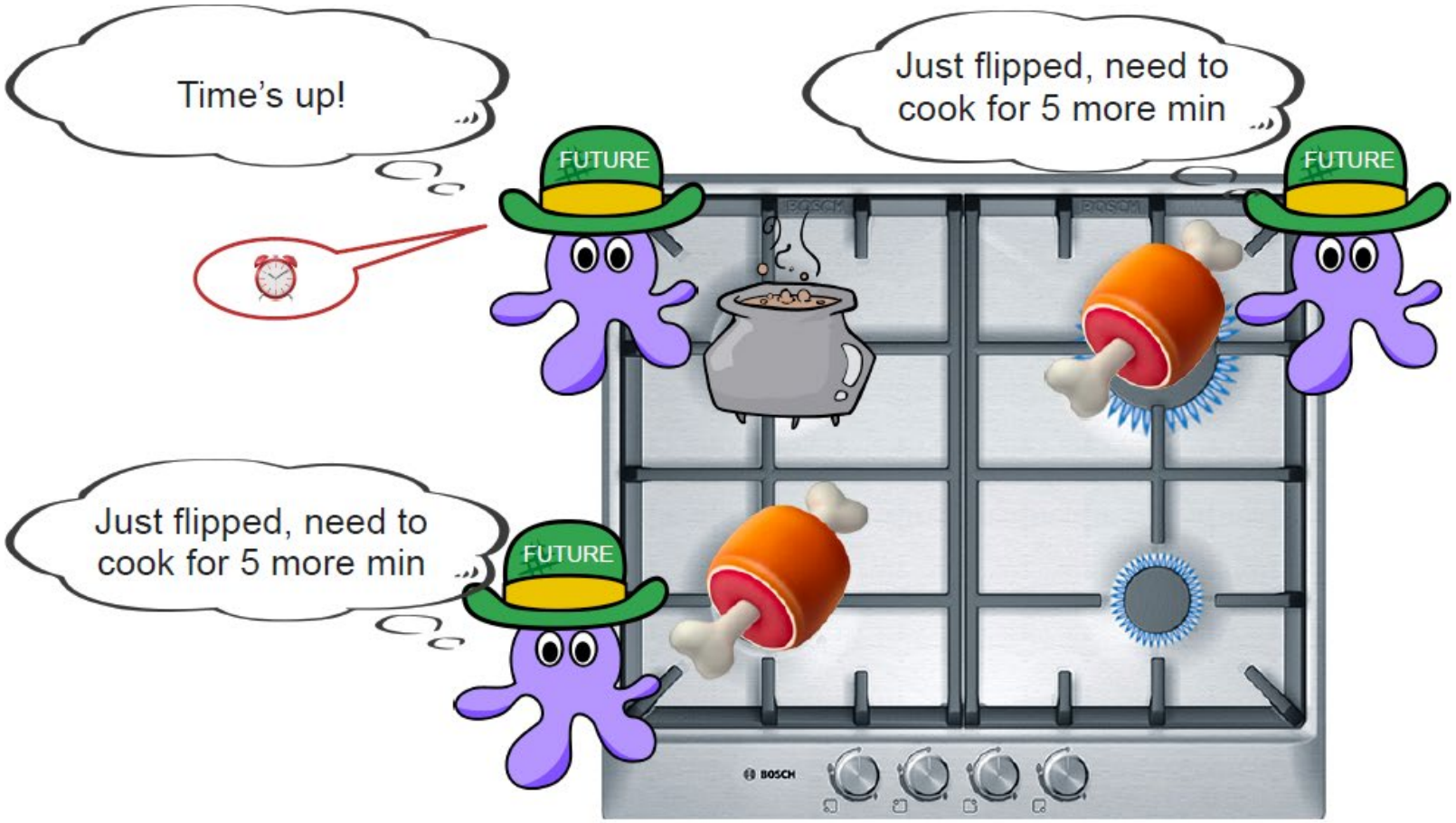
Futures visualized

Executor thread
(sleeping)

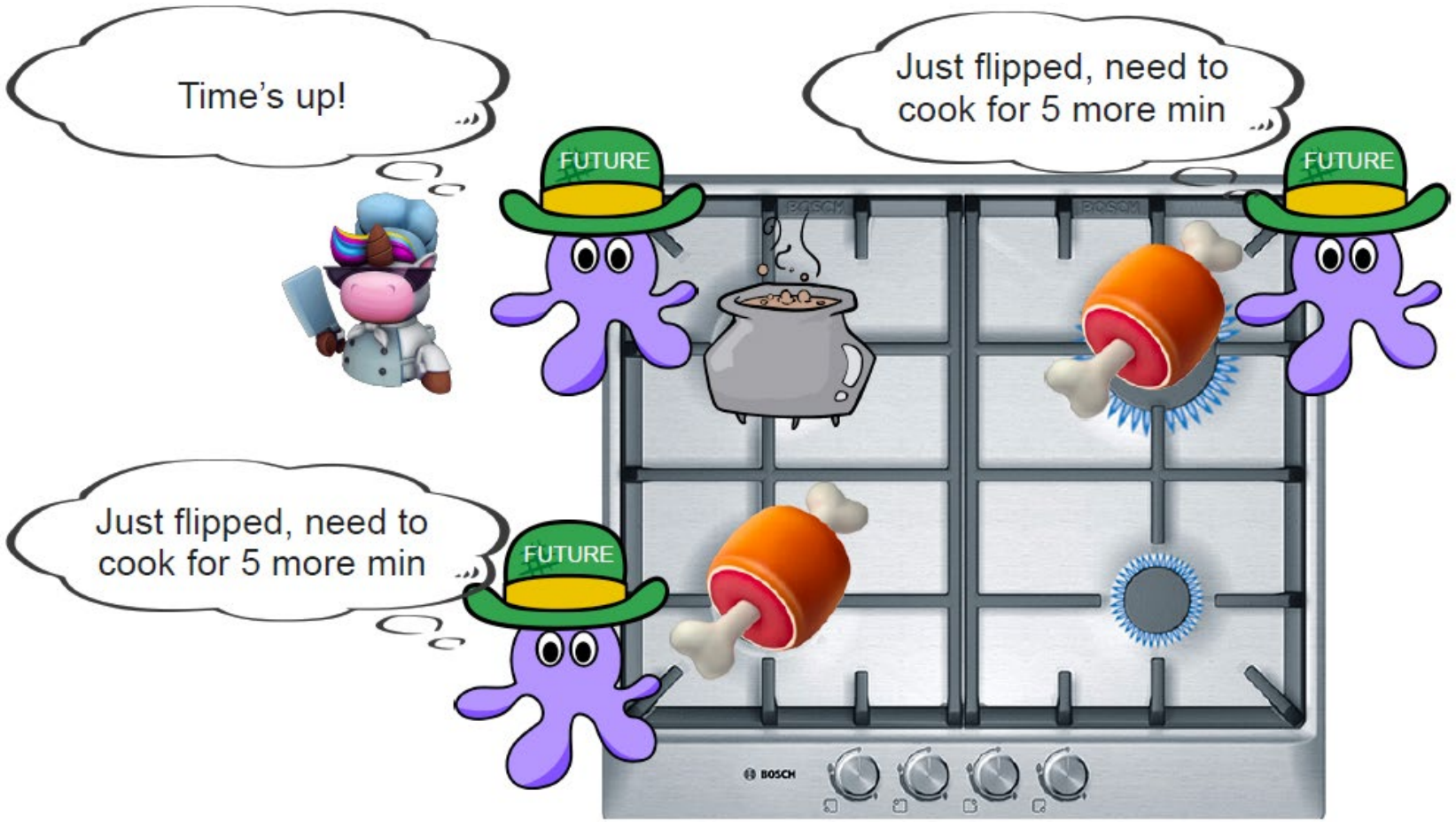


Futures visualized

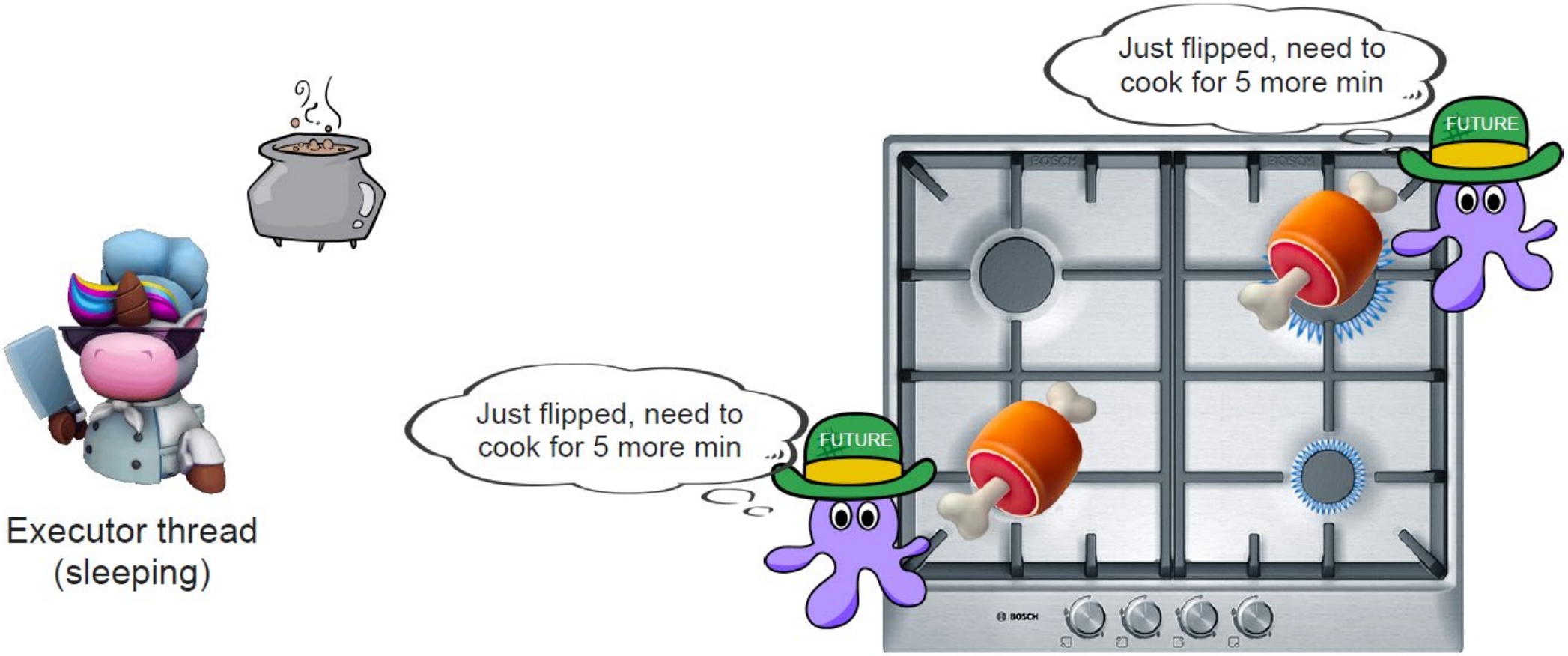
Executor thread
(sleeping)



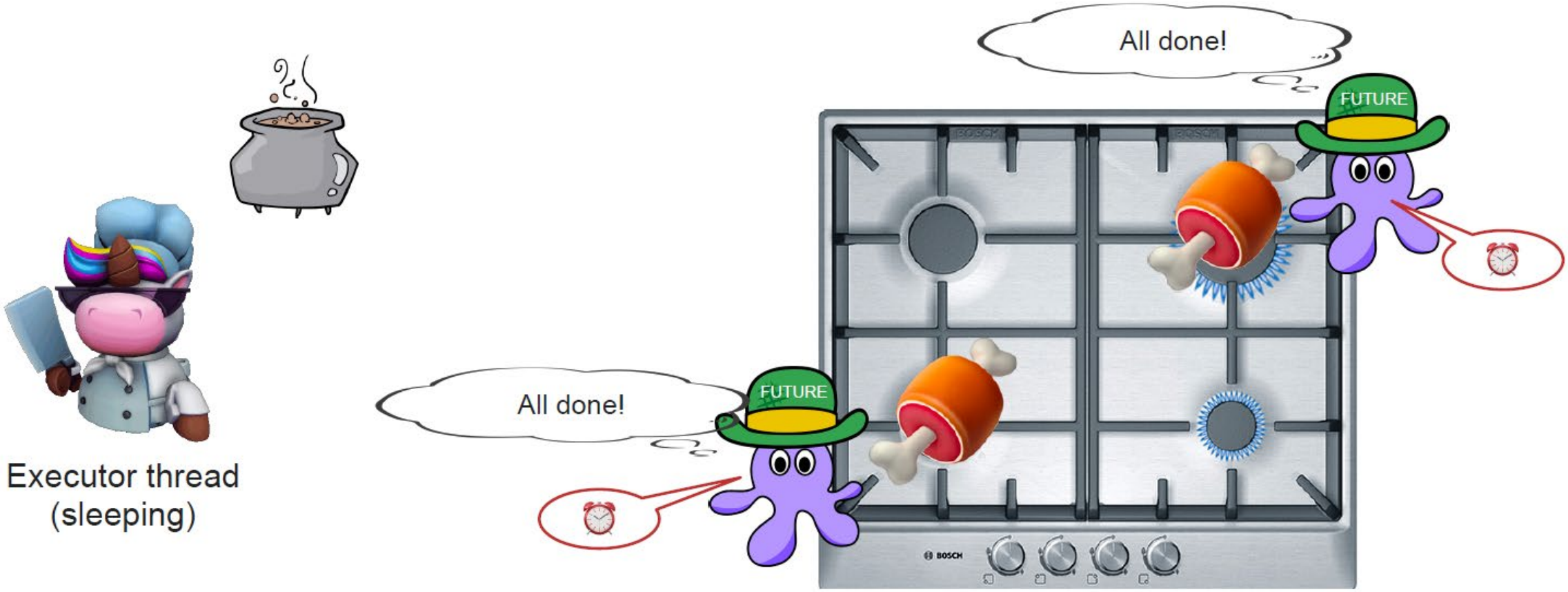
Futures visualized



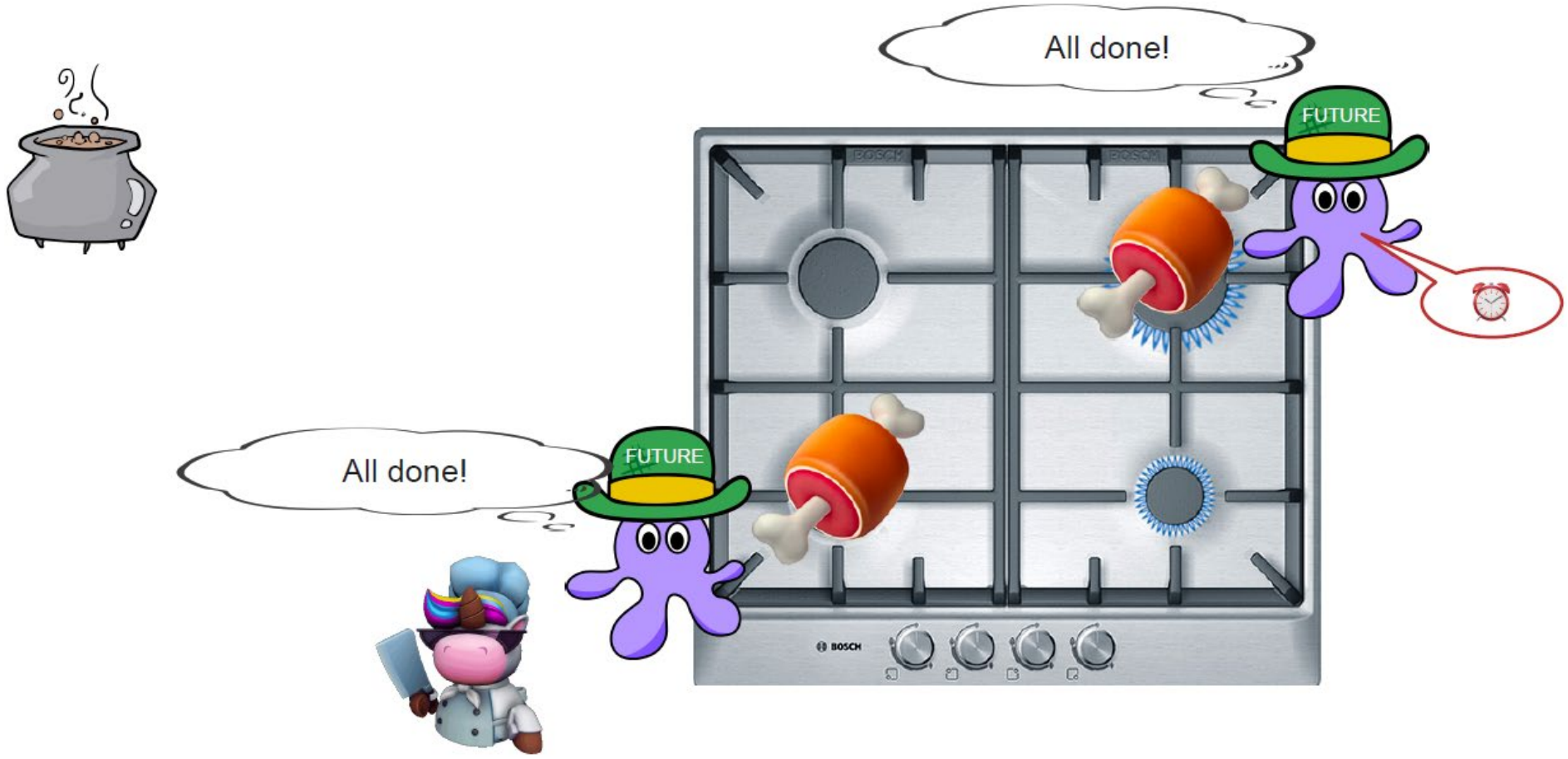
Futures visualized



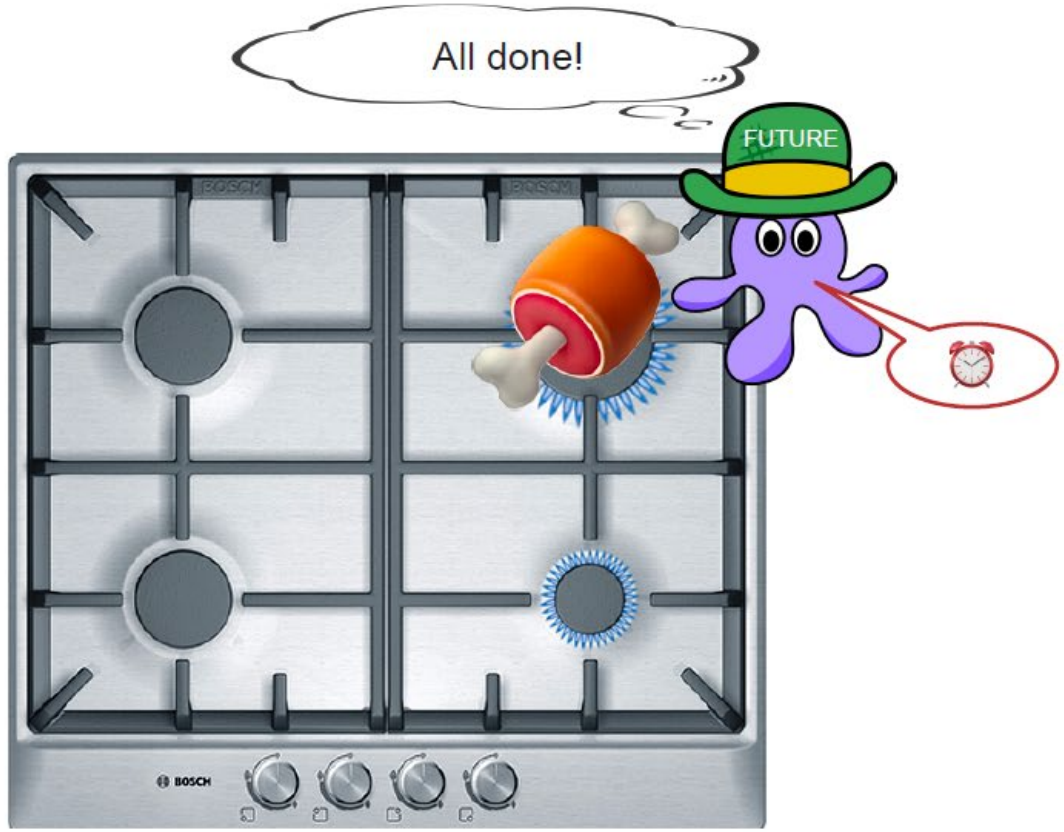
Futures visualized



Futures visualized

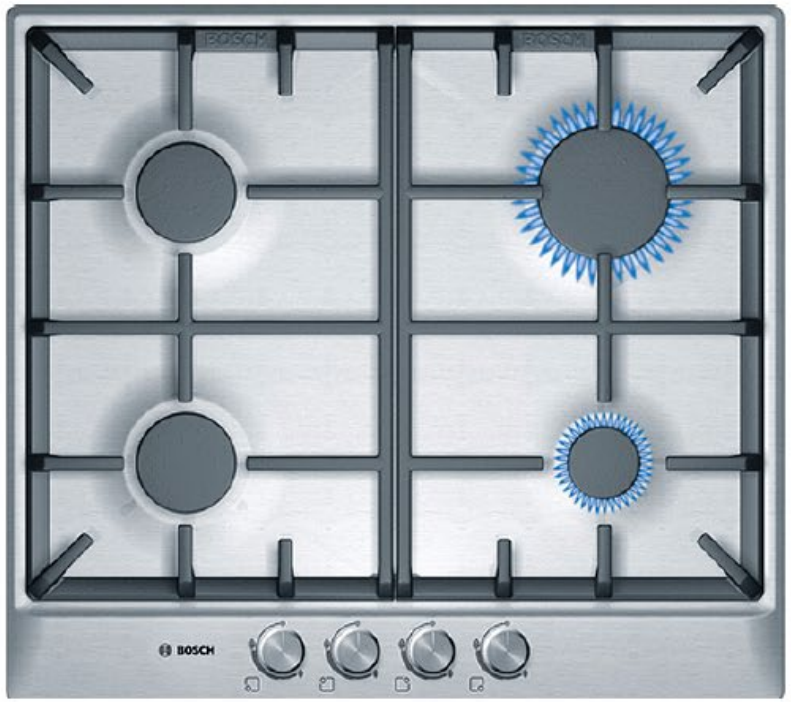


Futures visualized



Futures visualized

Futures visualized



Futures

- Represents a value that will exist sometime in the future
- Calculation that hasn’t happened yet
  - Is probably going to happen at some point
  - Just keep asking
- Event loop = runtime for Futures
  - Keeps polling Future until it is ready
  - Runs your code whenever it can be run
  - User-space scheduler for futures

Rust: zero-cost abstraction

- Code that you can’t write better by hand
  - Abstraction layers disappear at compile-time

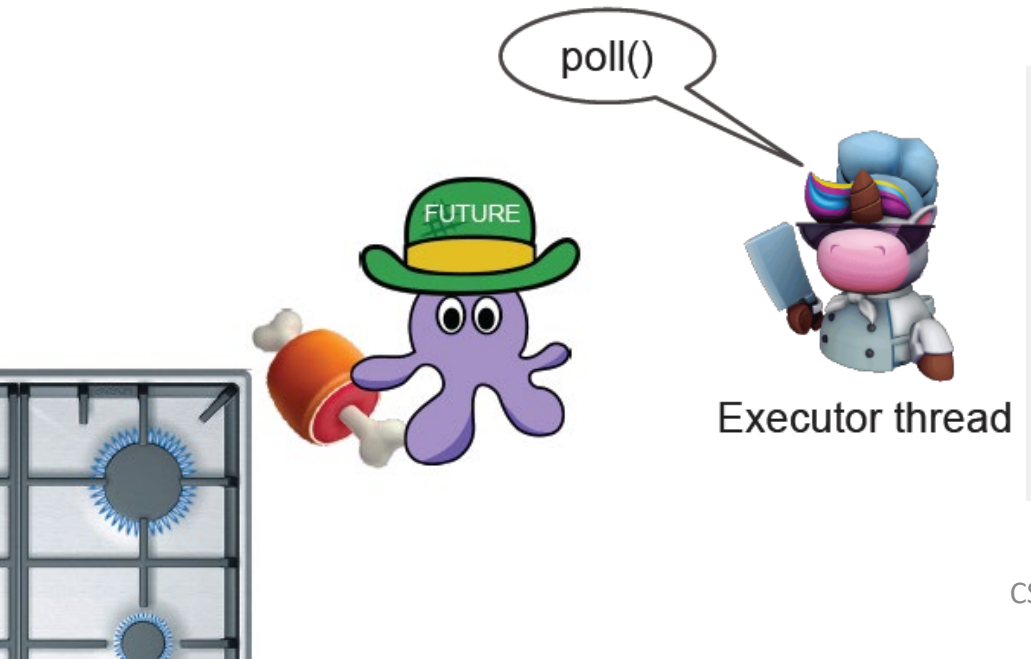
- Example: iterators API is a zero-cost abstraction
  - in release mode, the machine code
    - One can’t tell if it was done using iterators or implemented by hand
    - Compiler optimize the cache locality that would be difficult to implement by hand

Futures.rs – zero-cost abstraction for futures

- The code in the binary has no allocation and no runtime overhead in comparison to writing it by hand (non-assembly code ☺)
- Building async state machines
- The Future trait
- Event reactor

The `Future` trait

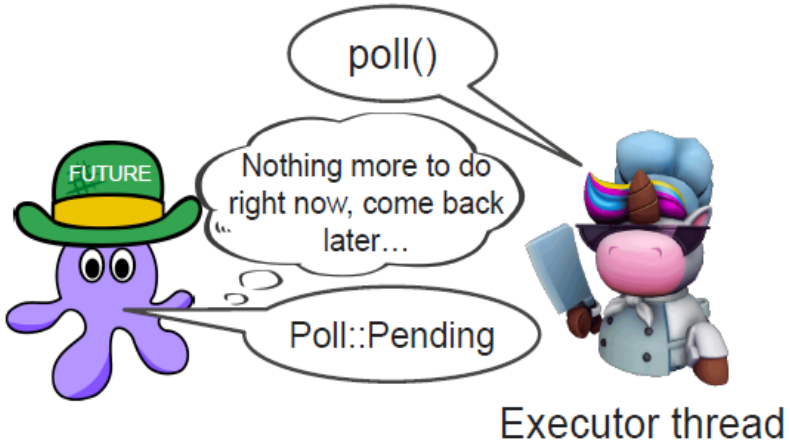
- The Future trait
  - The executor thread should call poll() on the future to start it off
  - It will run code until it can no longer progress




```
1  trait Future {
2  // This is a simplified version of the Future definition
3      type Output;
4      fn poll(&mut self, cx: &mut Context) -> Poll<Self::Output>;
5  }
6  enum Poll<T> {
7      Ready(T),
8      Pending,
9  }
```

The `Future` trait

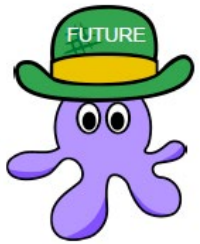
- The executor thread should call `poll()` on the future to start it off
- It will run code until it can no longer progress
  - If the future is complete, returns `Poll::Ready(T)`
  - If future needs to wait for some event, returns `Poll::Pending`, and allows the single thread to work on another future




```
1  trait Future {
2  // This is a simplified version of the Future definition
3      type Output;
4      fn poll(&mut self, cx: &mut Context) -> Poll<Self::Output>;
5  }
6  enum Poll<T> {
7      Ready(T),
8      Pending,
9  }
```

The `Future` trait

- The executor thread should call `poll()` on the future to start it off
- It will run code until it can no longer progress
  - If the future is complete, returns `Poll::Ready(T)`
  - If future needs to wait for some event, returns `Poll::Pending`, and allows the single thread to work on another future

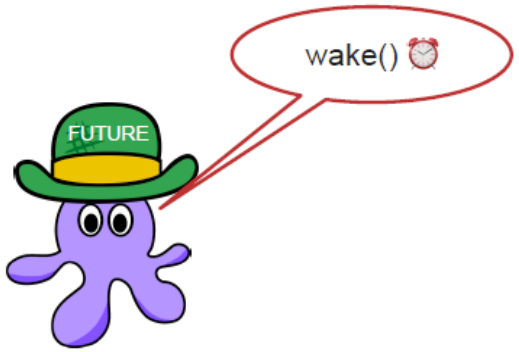


Executor thread
(doing other things)

```
1  trait Future {
2  // This is a simplified version of the Future definition
3      type Output;
4      fn poll(&mut self, cx: &mut Context) -> Poll<Self::Output>;
5  }
6  enum Poll<T> {
7      Ready(T),
8      Pending,
9  }
```

The `Future` trait

- When `poll()` is called, `Context` structure passed in
- `Context` includes a `wake()` function
  - Called when future can make progress again
  - Implemented internally using system calls)

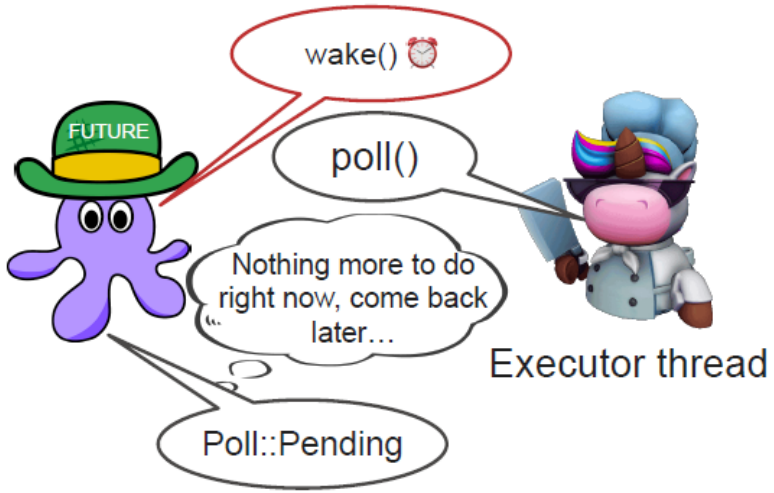


Executor thread
(sleeping)

```
1  trait Future {
2  // This is a simplified version of the Future definition
3      type Output;
4      fn poll(&mut self, cx: &mut Context) -> Poll<Self::Output>;
5  }
6  enum Poll<T> {
7      Ready(T),
8      Pending,
9  }
```

The `Future` trait

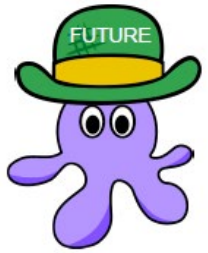
- When `poll()` is called, `Context` structure passed in
- `Context` includes a `wake()` function
  - Called when future can make progress again
  - Implemented internally using system calls)
- After `wake()` is called, executor can use `Context` to see which `Future` can be polled to make new progress




```
1  trait Future {
2  // This is a simplified version of the Future definition
3      type Output;
4      fn poll(&mut self, cx: &mut Context) -> Poll<Self::Output>;
5  }
6  enum Poll<T> {
7      Ready(T),
8      Pending,
9  }
```

The `Future` trait

- When `poll()` is called, `Context` structure passed in
- `Context` includes a `wake()` function
  - Called when future can make progress again
  - Implemented internally using system calls)
- After `wake()` is called, executor can use `Context` to see which `Future` can be polled to make new progress

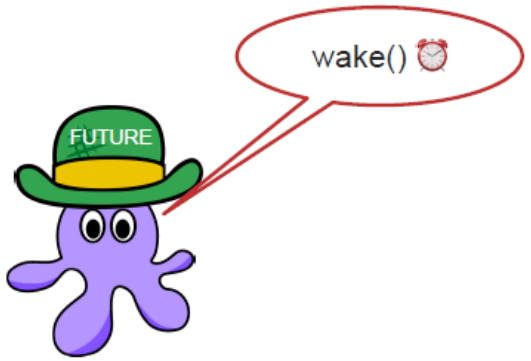


Executor thread
(sleeping)

```
1  trait Future {
2  // This is a simplified version of the Future definition
3      type Output;
4      fn poll(&mut self, cx: &mut Context) -> Poll<Self::Output>;
5  }
6  enum Poll<T> {
7      Ready(T),
8      Pending,
9  }
```

The `Future` trait

- When `poll()` is called, `Context` structure passed in
- `Context` includes a `wake()` function
  - Called when future can make progress again
  - Implemented internally using system calls)
- After `wake()` is called, executor can use `Context` to see which `Future` can be polled to make new progress

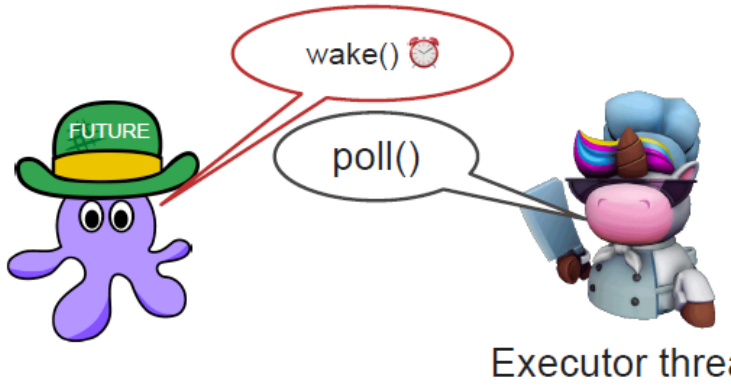


Executor thread (sleeping)

```
1  trait Future {
2  // This is a simplified version of the Future definition
3      type Output;
4      fn poll(&mut self, cx: &mut Context) -> Poll<Self::Output>;
5  }
6  enum Poll<T> {
7      Ready(T),
8      Pending,
9  }
```

The `Future` trait

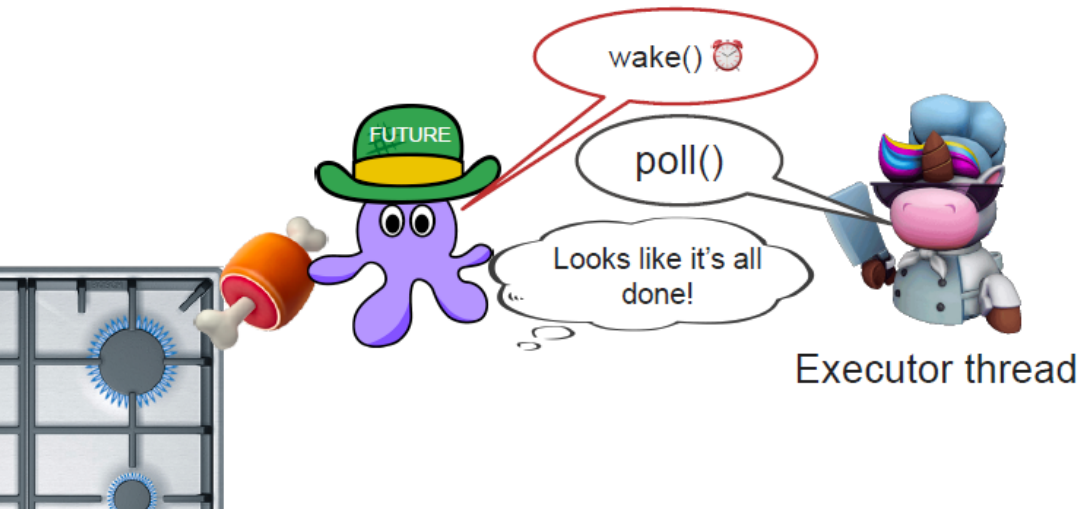
- When `poll()` is called, `Context` structure passed in
- `Context` includes a `wake()` function
  - Called when future can make progress again
  - Implemented internally using system calls)
- After `wake()` is called, executor can use `Context` to see which `Future` can be polled to make new progress




```
1  trait Future {
2  // This is a simplified version of the Future definition
3      type Output;
4      fn poll(&mut self, cx: &mut Context) -> Poll<Self::Output>;
5  }
6  enum Poll<T> {
7      Ready(T),
8      Pending,
9  }
```

The `Future` trait

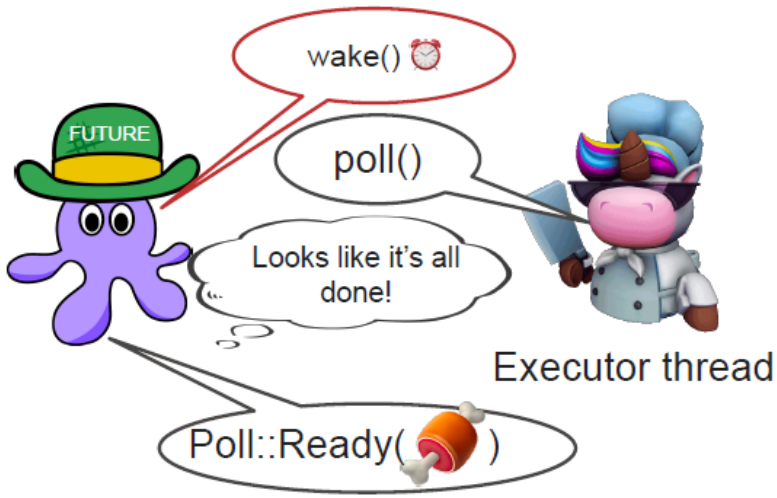
- When `poll()` is called, `Context` structure passed in
- `Context` includes a `wake()` function
  - Called when future can make progress again
  - Implemented internally using system calls)
- After `wake()` is called, executor can use `Context` to see which `Future` can be polled to make new progress



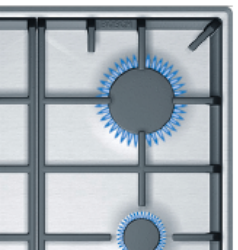

```
1  trait Future {
2  // This is a simplified version of the Future definition
3      type Output;
4      fn poll(&mut self, cx: &mut Context) -> Poll<Self::Output>;
5  }
6  enum Poll<T> {
7      Ready(T),
8      Pending,
9  }
```

The `Future` trait

- When `poll()` is called, `Context` structure passed in
- `Context` includes a `wake()` function
  - Called when future can make progress again
  - Implemented internally using system calls)
- After `wake()` is called, executor can use `Context` to see which `Future` can be polled to make new progress



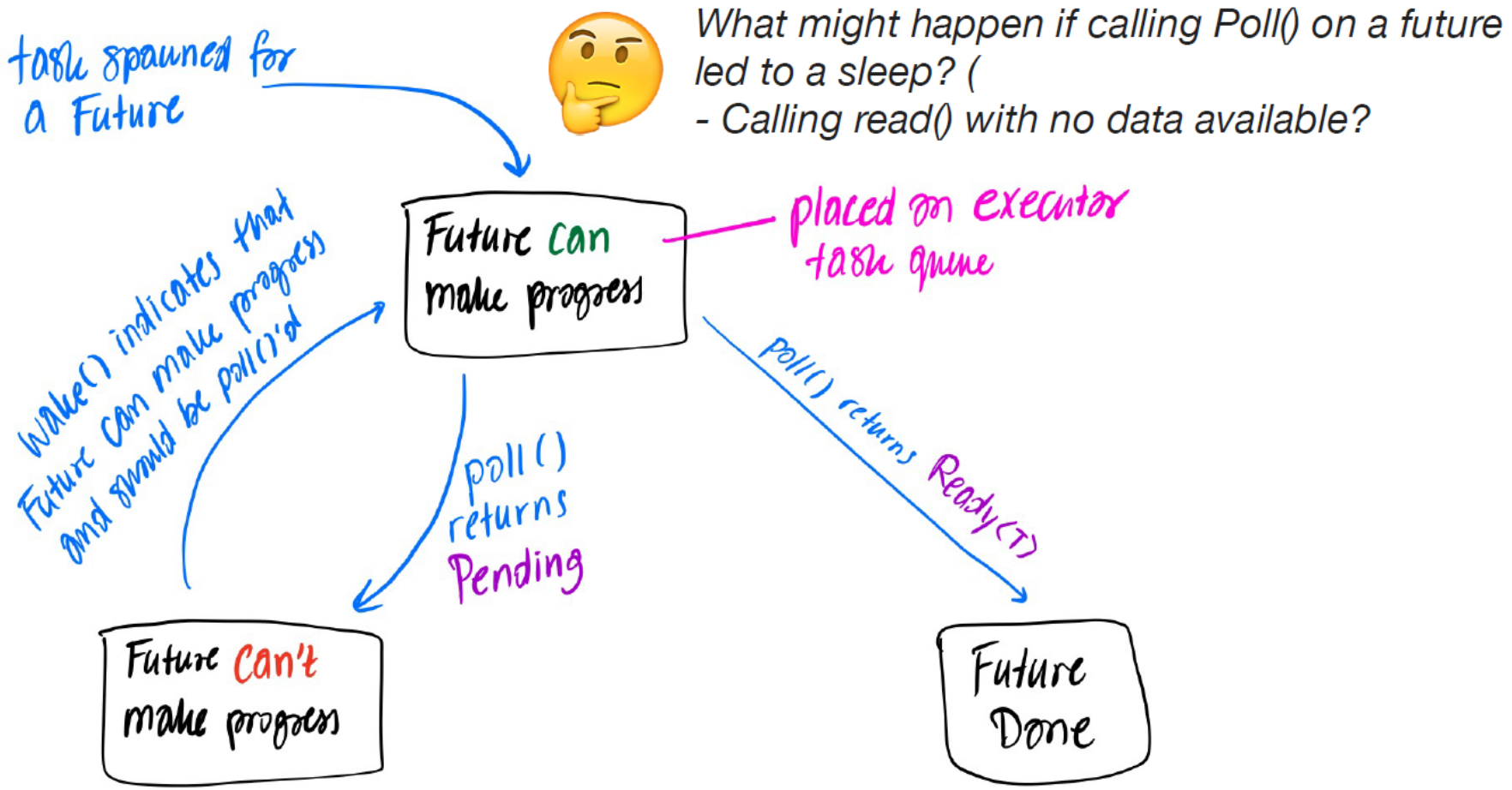

```
1  trait Future {
2  // This is a simplified version of the Future definition
3      type Output;
4      fn poll(&mut self, cx: &mut Context) -> Poll<Self::Output>;
5  }
6  enum Poll<T> {
7      Ready(T),
8      Pending,
9  }
```



Executors

- An executor loops over futures that can currently make progress
  - Calls `poll()` on them to give them attention until they need to wait again
  - When no futures can make progress, the executor goes to sleep until one or more futures calls `wake()`
- A popular executor in the Rust ecosystem is `Tokio`
  - Wraps around `mio.rs` and `futures.rs`
- Executors can be single threaded or multi-threaded
- Running on a multiple cores machine
  - Futures can truly run in parallel
  - Need to protect shared data using synchronization primitives (although the ownership model kind of already forces you to do this anyways)

Workflow of an executor



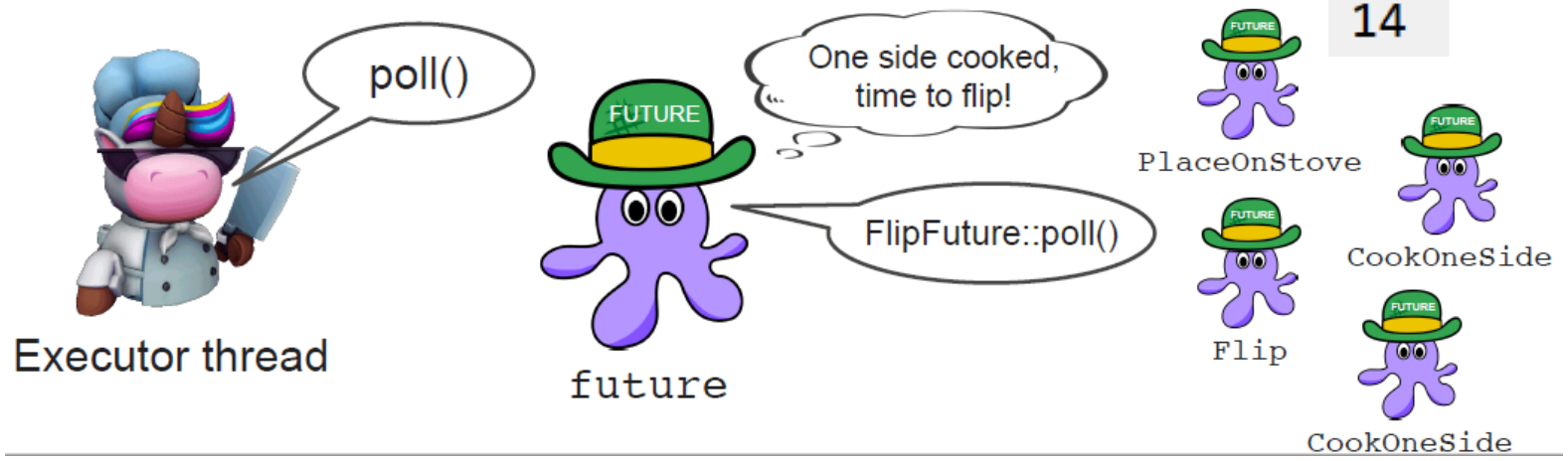
Futures should not block!

- If code within a future causes the thread to sleep, the executor running that code is going to sleep
- **Asynchronous** code needs to use non-blocking versions of everything, including mutexes, system calls that would normally block, or anything.
- Executor runtimes like `Tokio` provide these non-blocking implementations for your favorite synchronization primitives

Composition with futures

- Pretty much no one implements futures manually (unless you're a low-level library implementor)
- Instead, futures are combined with various **combinators**
  - Sequential
  - Concurrently

```
11  let future = placeOnStove(meat)
12      .then(|meat| cookOneSide(meat))
13      .then(|meat| flip(meat))
14      .then(|meat| cookOneSide(meat));
```



Not great ergonomics

```
11  let future = placeOnStove(meat)
12      .then(|meat| cookOneSide(meat))
13      .then(|meat| flip(meat))
14      .then(|meat| cookOneSide(meat));
```

- This code works
  - It's better than manually dealing with callbacks and state machines as you would in C/C++ with interfaces like `epoll`
- But can we do better?
  - The syntax is a little clunky… too much typing for Rust
  - Code becomes messier as complexity increases
  - Sharing mutable data (e.g. in local variables) can be painful: if there can only be one mutable reference at a time, only one closure can touch that data

Example of poor ergonomics

- Lines 21, 24, 25, 29: **asynchronous** functions returning Futures
- Lines 27: synchronous (normal) function
- Line 21: complicated syntax

```
21  fn addToInbox(email_id: u64, recipient_id: u64)
22      -> impl Future<Output=Result<(), Error>>
23  {
24      loadMessage(email_id)
25          .and_then(|message|
26              get_recipient(message, recipient_id))
27          .map(|(message, recipient)|
28              recipient.verifyHasSpace(&message))
29          .and_then(|(message, recipient)|
30              recipient.addToInbox(message))
31  }
```

Example of poor ergonomics

- Lines 26: why does `get_recipient` need to take a `message`?
  - To make this chain of futures work, since the next futures need both the `message` and `recipient` as input
  - Like a pipeline

```
21 fn addToInbox(email_id: u64, recipient_id: u64)
22     -> impl Future<Output=Result<(), Error>>
23 {
24     loadMessage(email_id)
25         .and_then(|message|
26             get_recipient(message, recipient_id))
27         .map(|(message, recipient)|
28             recipient.verifyHasSpace(&message))
29         .and_then(|(message, recipient)|
30             recipient.addToInbox(message))
31 }
```

Syntactic sugar

- Line 41: async function returns Future
- Line 44, 45, 47: .await waits for a future and gets its value
- Line 45: normal function usage

```
21 fn addToInbox(email_id: u64, recipient_id: u64)
22     -> impl Future<Output=Result<(), Error>>
23 {
24     loadMessage(email_id)
25         .and_then(|message|
26             get_recipient(message, recipient_id))
27         .map(|(message, recipient)|
28             recipient.verifyHasSpace(&message))
29         .and_then(|(message, recipient)|
30             recipient.addToInbox(message))
31 }
```

```
41 async fn addToInbox(email_id: u64, recipient_id: u64)
42     -> Result<(), Error>
43 {
44     let message = loadMessage(email_id).await?;
45     let recipient = get_recipient(recipient_id).await?;
46     recipient.verifyHasSpace(&message)?;
47     recipient.addToInbox(message).await
48 }
```

Async/.await

- An `async` function is a function that returns a Future
  - Any Futures used in the function are chained together by the compiler
- `.await` waits for a future and gets its value
  - `.await` can only be called in an async fn or block
- Structure of the code is similar to what we are used to!
- The Rust compiler transforms this code into a Future with a `poll()` method
  - Just as efficient as what you could implement by hand

Example: a simple server

- Lines 78, 81, 82: convert any blocking functions to asynchronous versions
  - They return Futures

```
51 use std::io::{Read, Write};
52 use std::net::TcpListener;
53 use std::thread;
54 fn main() {
55     let listener = TcpListener::bind("127.0.0.1:8080")
56         .unwrap();
57     loop {
58         let (mut socket, _) = listener.accept().unwrap();
59         thread::spawn(move || {
60             let mut buf = [0; 1024];
61             let n = socket.read(&mut buf).unwrap();
62             socket.write_all(&buf[0..n]).unwrap();
63         });
64     }
65 }
```

```
71 use tokio::io::{AsyncReadExt, AsyncWriteExt};
72 use tokio::net::TcpListener;
73 #[tokio::main]
74 async fn main() {
75     let listener = TcpListener::bind("127.0.0.1:8080").await
76         .unwrap();
77     loop {
78         let (mut socket, _) = listener.accept().await.unwrap();
79         tokio::spawn(async move {
80             let mut buf = [0; 1024];
81             let n = socket.read(&mut buf).await.unwrap();
82             socket.write_all(&buf[0..n]).await.unwrap();
83         });
84     }
85 }
```

Example: asynchronous programming

- Lines 78, 81, 82: `.await` the Futures
  - .await can only be used in an async function or block
  - The compiler will complain if you forget .await

```
51 use std::io::{Read, Write};
52 use std::net::TcpListener;
53 use std::thread;
54 fn main() {
55     let listener = TcpListener::bind("127.0.0.1:8080")
56         .unwrap();
57     loop {
58         let (mut socket, _) = listener.accept().unwrap();
59         thread::spawn(move || {
60             let mut buf = [0; 1024];
61             let n = socket.read(&mut buf).unwrap();
62             socket.write_all(&buf[0..n]).unwrap();
63         });
64     }
65 }
```

```
71 use tokio::io::{AsyncReadExt, AsyncWriteExt};
72 use tokio::net::TcpListener;
73 #[tokio::main]
74 async fn main() {
75     let listener = TcpListener::bind("127.0.0.1:8080").await
76         .unwrap();
77     loop {
78         let (mut socket, _) = listener.accept().await.unwrap();
79         tokio::spawn(async move {
80             let mut buf = [0; 1024];
81             let n = socket.read(&mut buf).await.unwrap();
82             socket.write_all(&buf[0..n]).await.unwrap();
83         });
84     }
85 }
```

Example: asynchronous programming

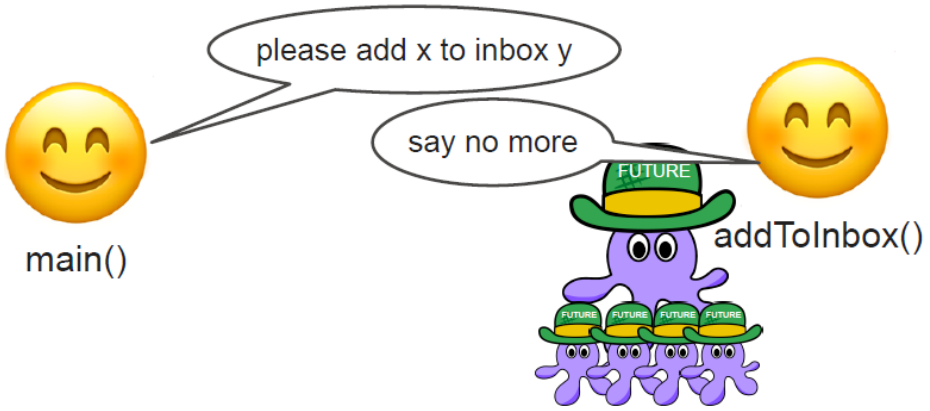
- `main()` now returns a Future
  - Futures don't actually do anything unless an executor executes them
  - Need to run `main()` and submit the returned `Future` to the executor
  - `Line 73: #[tokio::main]` macro submits the future to the executor

```
51 use std::io::{Read, Write};
52 use std::net::TcpListener;
53 use std::thread;
54 fn main() {
55     let listener = TcpListener::bind("127.0.0.1:8080")
56         .unwrap();
57     loop {
58         let (mut socket, _) = listener.accept().unwrap();
59         thread::spawn(move || {
60             let mut buf = [0; 1024];
61             let n = socket.read(&mut buf).unwrap();
62             socket.write_all(&buf[0..n]).unwrap();
63         });
64     }
65 }
```

```
71 use tokio::io::{AsyncReadExt, AsyncWriteExt};
72 use tokio::net::TcpListener;
73 #[tokio::main]
74 async fn main() {
75     let listener = TcpListener::bind("127.0.0.1:8080").await
76         .unwrap();
77     loop {
78         let (mut socket, _) = listener.accept().await.unwrap();
79         tokio::spawn(async move {
80             let mut buf = [0; 1024];
81             let n = socket.read(&mut buf).await.unwrap();
82             socket.write_all(&buf[0..n]).await.unwrap();
83         });
84     }
85 }
```

Async functions generate/return futures

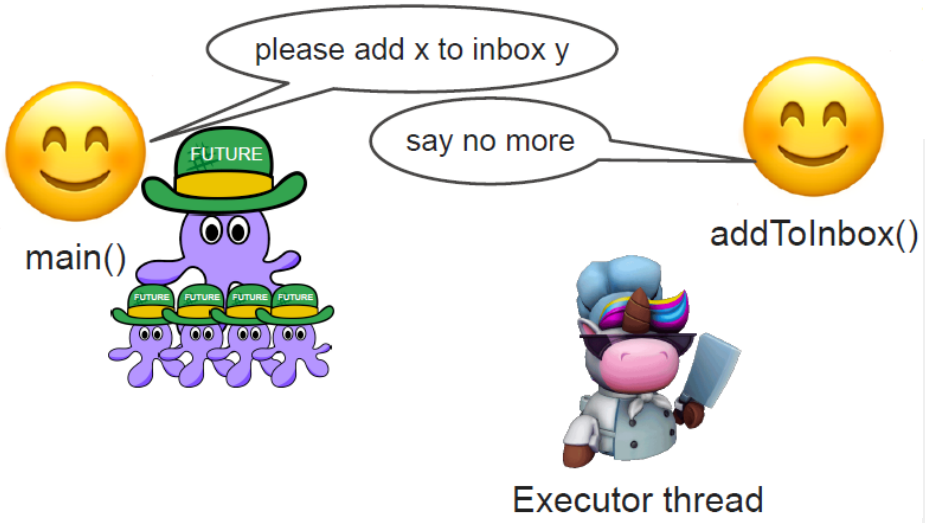
- If you run this function, it will not actually do any work with any messages
- This is still a function and you can still run it…
- But its purpose is now to produce a future that does the stuff that was written inside the function




```
91  async fn addToInbox(email_id: u64, recipient_id: u64)
92      -> Result<(), Error>
93  {
94      let message = loadMessage(email_id).await?;
95      let recipient = get_recipient(recipient_id).await?;
96      recipient.verifyHasSpace(&message)?;
97      recipient.addToInbox(message).await
98  }
```

Async functions generate/return futures

- If you run this function, it will not actually do any work with any messages
- This is still a function and you can still run it…
- But its purpose is now to produce a future that does the stuff that was written inside the function



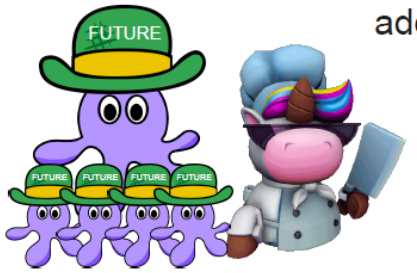

```
91  async fn addToInbox(email_id: u64, recipient_id: u64)
92      -> Result<(), Error>
93  {
94      let message = loadMessage(email_id).await?;
95      let recipient = get_recipient(recipient_id).await?;
96      recipient.verifyHasSpace(&message)?;
97      recipient.addToInbox(message).await
98  }
```

Async functions generate/return futures

- If you run this function, it will not actually do any work with any messages
- This is still a function and you can still run it…
- But its purpose is now to produce a future that does the stuff that was written inside the function

main()

addToInbox()



Executor thread

```
91  async fn addToInbox(email_id: u64, recipient_id: u64)
92      -> Result<(), Error>
93  {
94      let message = loadMessage(email_id).await?;
95      let recipient = get_recipient(recipient_id).await?;
96      recipient.verifyHasSpace(&message)?;
97      recipient.addToInbox(message).await
98  }
```

State management

- There are 5 places where we might be paused, not actively executing:
  - Before anything has happened yet (i.e. Future has been created but not yet `poll()`ed)
  - Await-ing for `loadMessage`
  - Await-ing for `get_recipient`
  - Await-ing for `addToInbox`
  - Future has completed
- Use an `enum` to store the state for these possibilities

```
enum AddToInboxState {
NotYetStarted { email_id: u64, recipient_id: u64 },
WaitingLoadMessage {
    recipient_id: u64, state: LoadMessageFuture },
WaitingGetRecipient {
    message: Message, state: GetRecipientFuture },
WaitingAddToInbox {
    state: AddToInboxFuture },
Completed { result: Result<(), Error> },
}
```

```
async fn addToInbox(email_id: u64, recipient_id: u64)
    -> Result<(), Error>
{
    let message = loadMessage(email_id).await?;
    let recipient = get_recipient(recipient_id).await?;
    recipient.verifyHasSpace(&message)?;
    recipient.addToInbox(message).await
}
```

State management

- Attempting to implement `poll()` for this Future
  - Look at the current state and execute the appropriate code from the async fn


```
enum AddToInboxState {
NotYetStarted { email_id: u64, recipient_id: u64 },
WaitingLoadMessage {
    recipient_id: u64, state: LoadMessageFuture },
WaitingGetRecipient {
    message: Message, state: GetRecipientFuture },
WaitingAddToInbox {
    state: AddToInboxFuture },
Completed { result: Result<(), Error> },
}
```

```
async fn addToInbox(email_id: u64, recipient_id: u64)
    -> Result<(), Error>
{
    let message = loadMessage(email_id).await?;
    let recipient = get_recipient(recipient_id).await?;
    recipient.verifyHasSpace(&message)?;
    recipient.addToInbox(message).await
}
```

Conceptually: state management

```
fn poll() {
    match self.state {
        NotYetStarted(email_id, recipient_id) => {
            let next_future = load_message(email_id);
            switch to WaitingLoadMessage state
        },
        WaitingLoadMessage(email_id, recipient_id, state) => {
            match state.poll() {
                Ready(message) => {
                    let next_future = get_recipient(recipient_id);
                    switch to WaitingGetRecipient state
                },
                Pending => return Pending,
            }
        },
        WaitingGetRecipient(message, recipient_id, state) => {
            match state.poll() {
                Ready(recipient) => {
                    recipient.verifyHasSpace(&message)?;
                    let next_future = recipient.addToInbox(message);
                    switch to WaitingAddToInbox state
                },
                Pending => return Pending,
            }
        },
        ...
```

```
enum AddToInboxState {
NotYetStarted { email_id: u64, recipient_id: u64 },
WaitingLoadMessage {
    recipient_id: u64, state: LoadMessageFuture },
WaitingGetRecipient {
    message: Message, state: GetRecipientFuture },
WaitingAddToInbox {
    state: AddToInboxFuture },
Completed { result: Result<(), Error> },
}
```


```
async fn addToInbox(email_id: u64, recipient_id: u64)
    -> Result<(), Error>
{
    let message = loadMessage(email_id).await?;
    let recipient = get_recipient(recipient_id).await?;
    recipient.verifyHasSpace(&message)?;
    recipient.addToInbox(message).await
}
```

Implications

- Async functions have no stack
  - Sometimes called *stackless coroutines*
  - The executor thread still has a stack (used to run normal/synchronous functions), but it isn't used to store state when switching between async tasks
  - All state is self contained in the generated Future
- No recursion
  - The Future returned by an async function needs to have a fixed size known at compile time
- Rust async functions are nearly optimal in terms of memory usage and allocations
  - Low overhead: the performance is as good as (or possibly better) what you could get tuning everything by hand

Usage of async code

- Taking a step back - problems to solve:
  - Memory usage from having so many stacks
  - Unnecessary context switching cost
- Async code makes sense when…
  - You need an extremely high degree of concurrency
    - Not as much reason to use async if you don't have that many threads
  - Work is primarily I/O bound
    - Context switching is expensive only if you're using a tiny fraction of the time slice
    - If you're doing a lot of work on the CPU for an extended period of time, you might prevent the executor from running other tasks

Similar tools in other languages

- Rust lets us write asynchronous code in the synchronous style that we're used to
- Javascript: very similar toolbox with Promises and async/await
  - Involves much more dynamic memory allocation, not as efficient
- Golang: goroutines are the asynchronous tasks, but unlike Rust they are not stackless
  - Resizable stacks - possible because Go is garbage collected
  - Runtime knows where all pointers are and can reallocate memory
- C++20 just got stackless coroutines
  - Still lots of sharp edges, may want to wait for more libraries to make this easier to use

Summary

- Never block in async code!
  - Asynchronous tasks are cooperative (not preemptive)
- You can only use `await` in `async` functions
- Rust won't let you write async functions in traits
  - You can use a crate called async-trait though
- References
  - Lectures 18 and 19 from Stanford's CS110L: https://reberhardt.com/cs110l/spring-2021/
  - Katharina Fey, 2018: https://www.youtube.com/watch?v=j0SIcN-Y-LA
  - https://www.youtube.com/watch?v=g2PmsO_C4eY