Event-Driven Programming (I)

Ryan Eberhardt and Julio Ballista
May 27, 2021

Today

- Today: Can we do better than threads?
    - Introducing an alternative paradigm called “event-driven programming”
    - Event-driven programming alone has nothing to do with safety…
    - But it’s a really important technique that is hard to pull off well. Often leads to complicated, error-prone designs
    - This week, will be talking about paradigms that make event-driven programming easier to reason about

Review: Threads

- A “lightweight process”
  - Control: the routine (i.e. function) running inside of the thread
  - State: a stack, CPU registers, status (ready/running/blocked), etc.
- **The OS manages threads**
  - The scheduler is responsible for assigning threads to run on cores, swapping them on and off as appropriate.

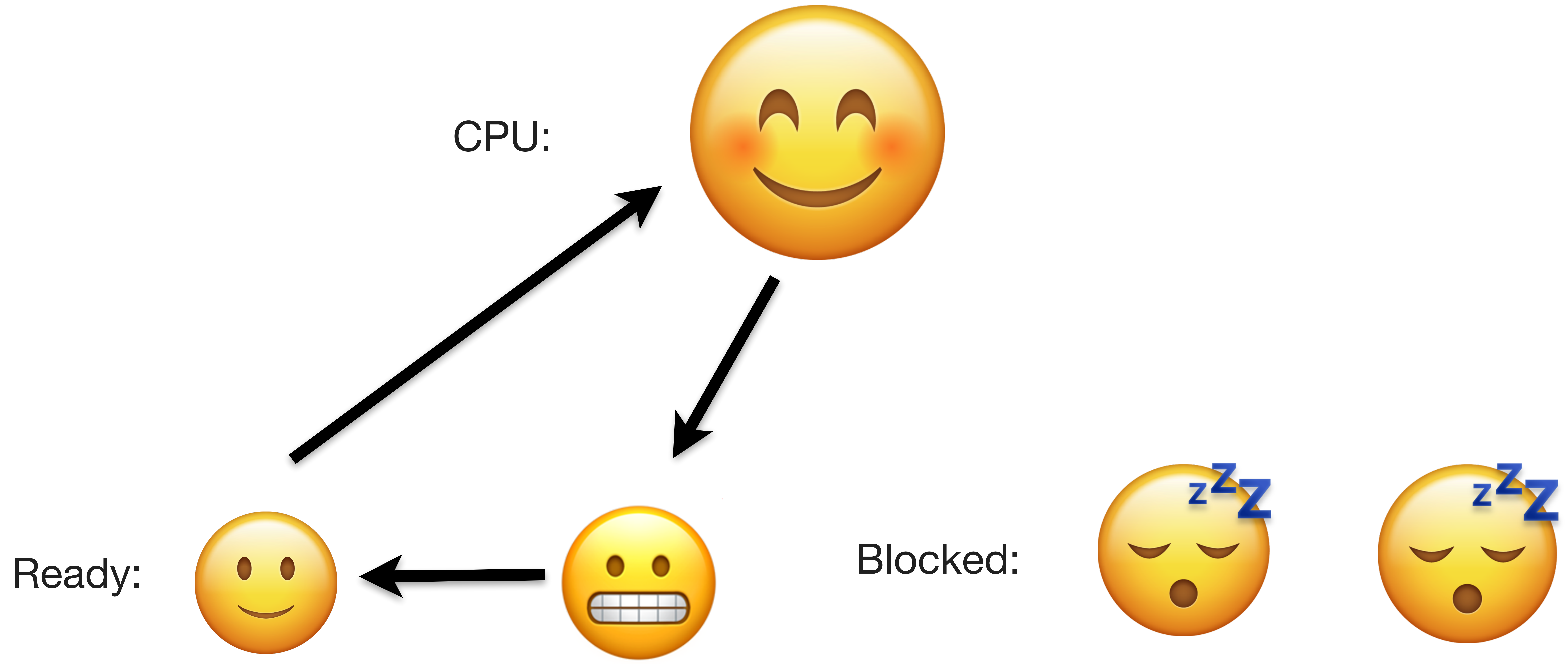
Review: Threads and the Scheduler

CPU: 😊

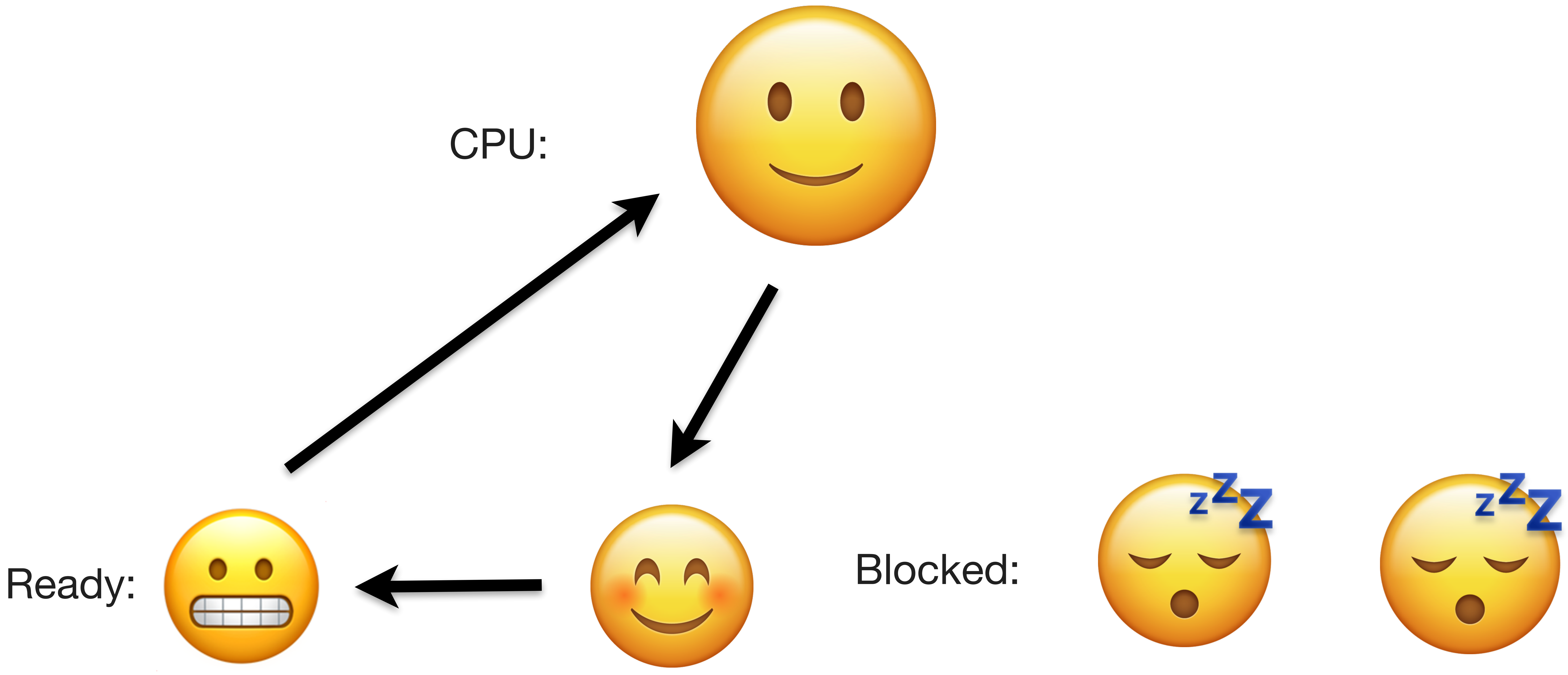
Ready: 🙂 😬

Blocked: 😴 😴

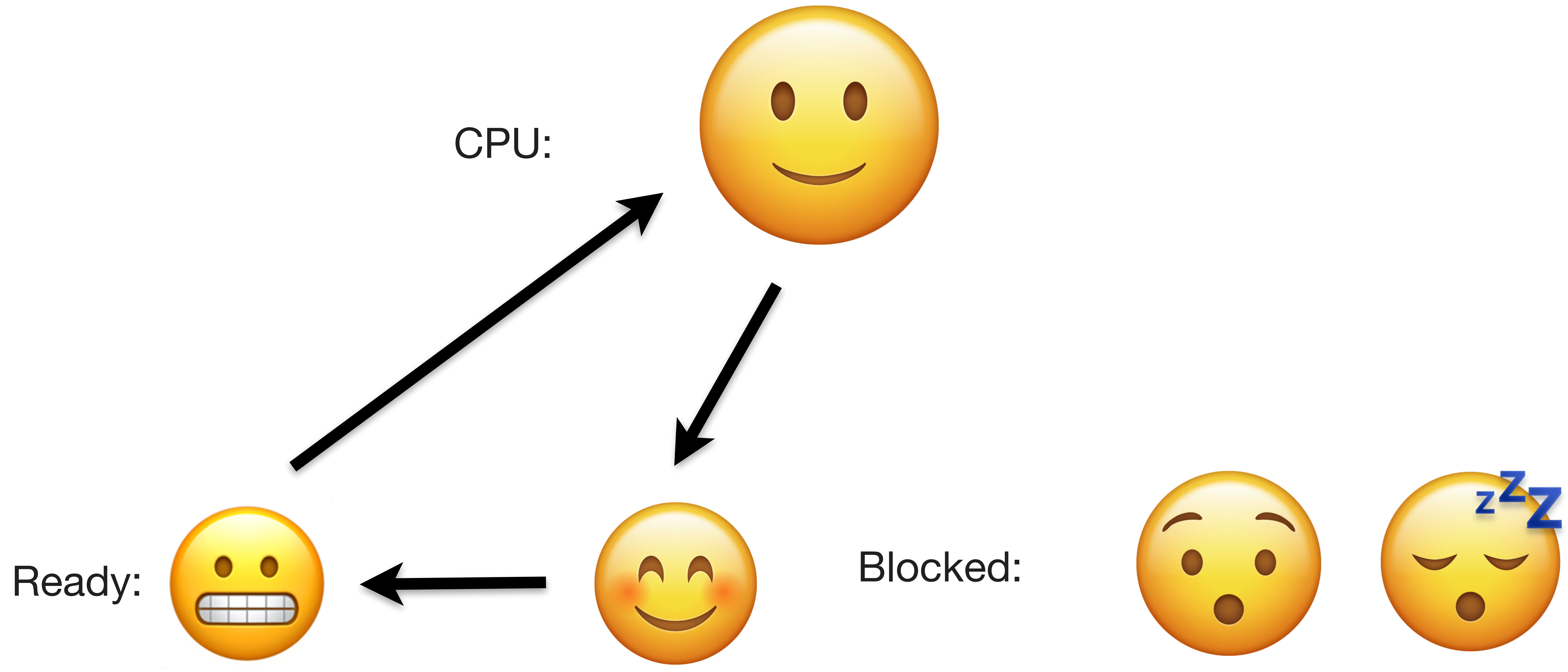
Review: Threads



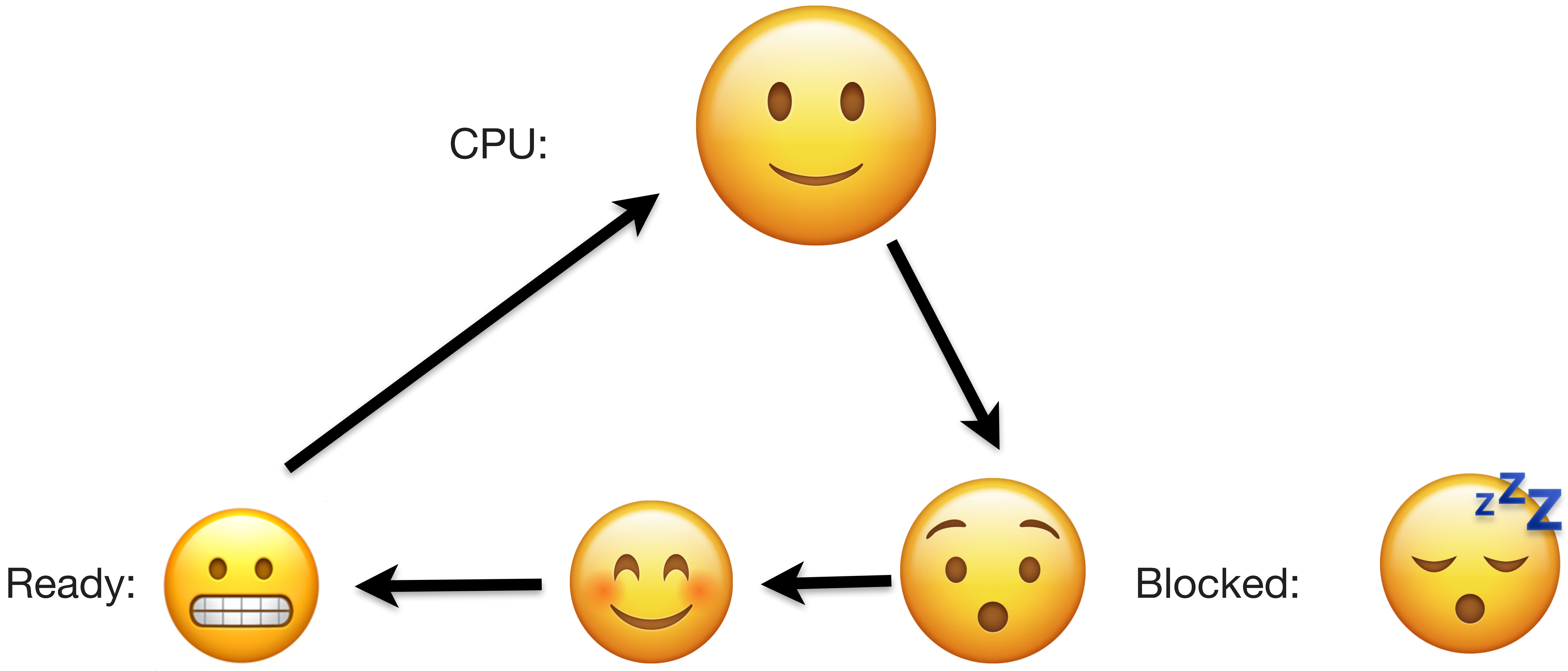
Review: Threads



Review: Threads

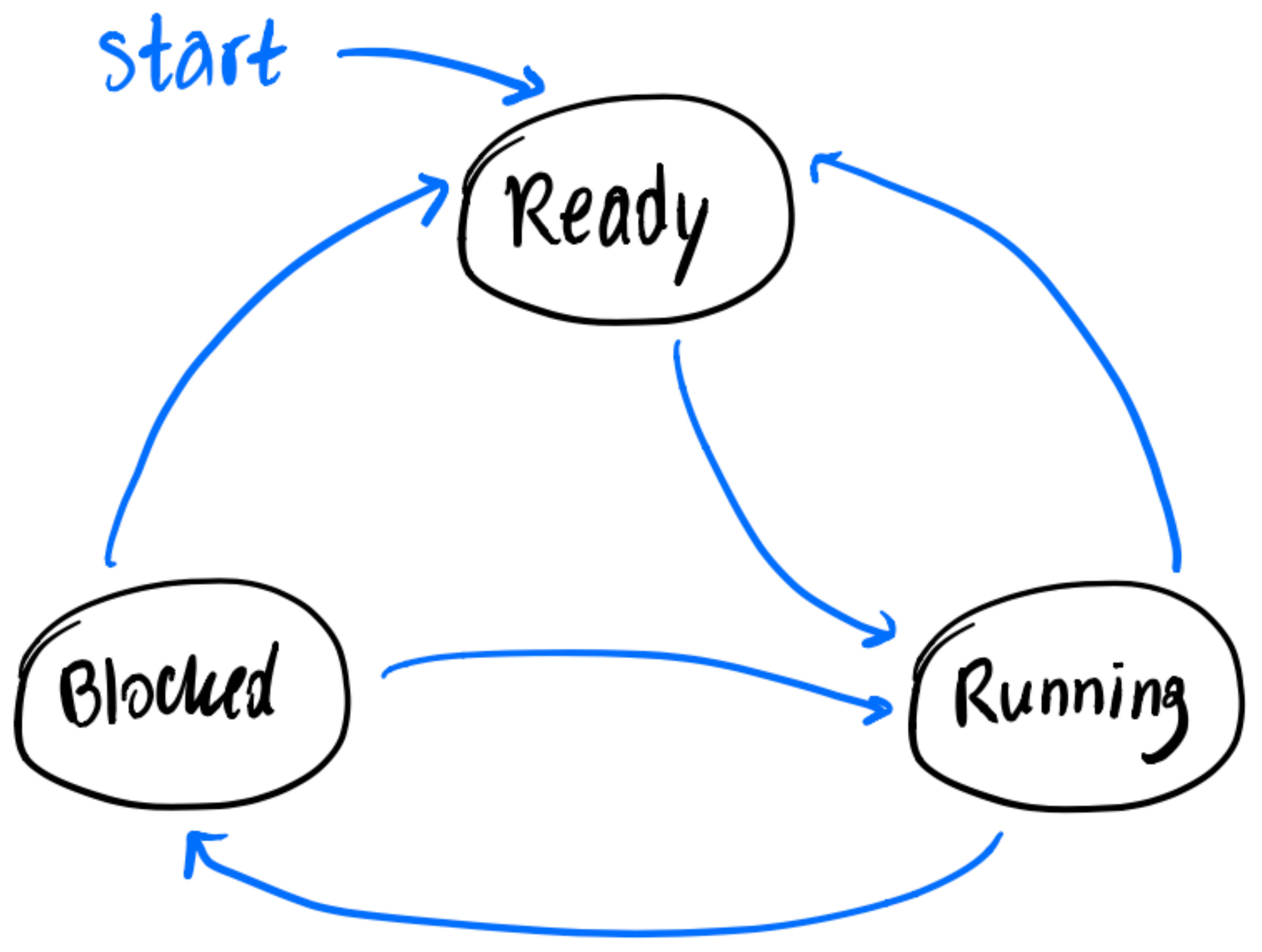


Review: Threads



Blocking syscalls

- Every thread runs for a “time slice” before the OS switches to a different thread
- A thread will give up its time slice early if it is “blocked,” needing to wait for something to happen
    - I/O: reading and writing
    - Waiting: waitpid, sigsuspend, join, cv.wait(…) etc.
    - lock()
    - sleep()
- This design is generally a good thing; getting off the CPU allows other threads to do useful work



Blocked:

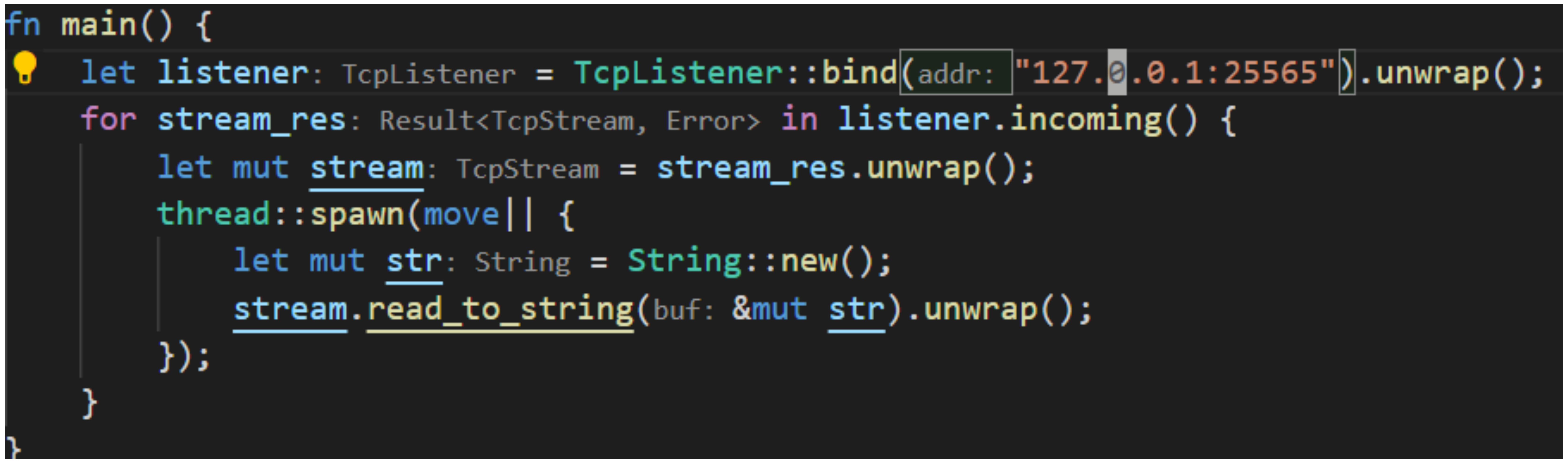
The Problem with Threads

- Context switching cost: When we use **blocking** functions within a thread, we discard the rest of the CPU time slice and incur a cost on switching the thread to be *blocked.*
    - Each switch is expensive! Virtual address space needs to get switched, registers need to get restored, cache gets stepped on, etc
    - This is a big cost for high-performance situations (servers). If we have to block on a client, maybe that thread could've done some other work instead.
- Memory overhead: If we have many threads, we consume a lot of memory
    - 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! (yike)

Review: Threads

CPU:

Finally, the CPU is mine at long last!

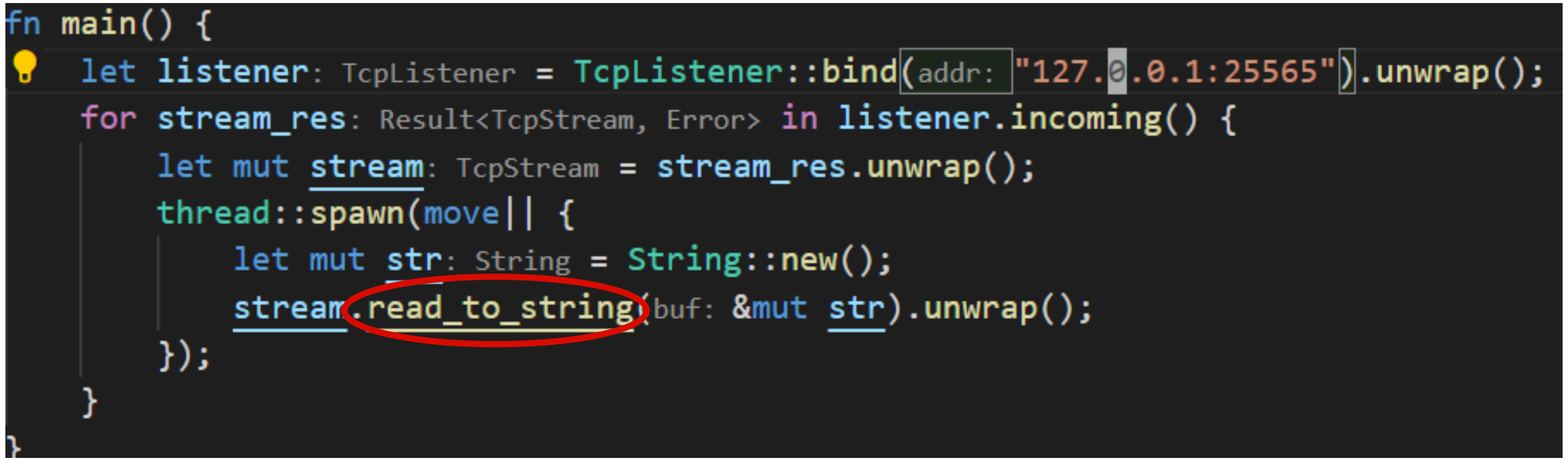


```
fn main() {
    let listener: TcpListener = TcpListener::bind(addr: "127.0.0.1:25565").unwrap();
    for stream_res: Result<TcpStream, Error> in listener.incoming() {
        let mut stream: TcpStream = stream_res.unwrap();
        thread::spawn(move|| {
            let mut str: String = String::new();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

Review: Threads

CPU:

Finally, the CPU is mine at long last!



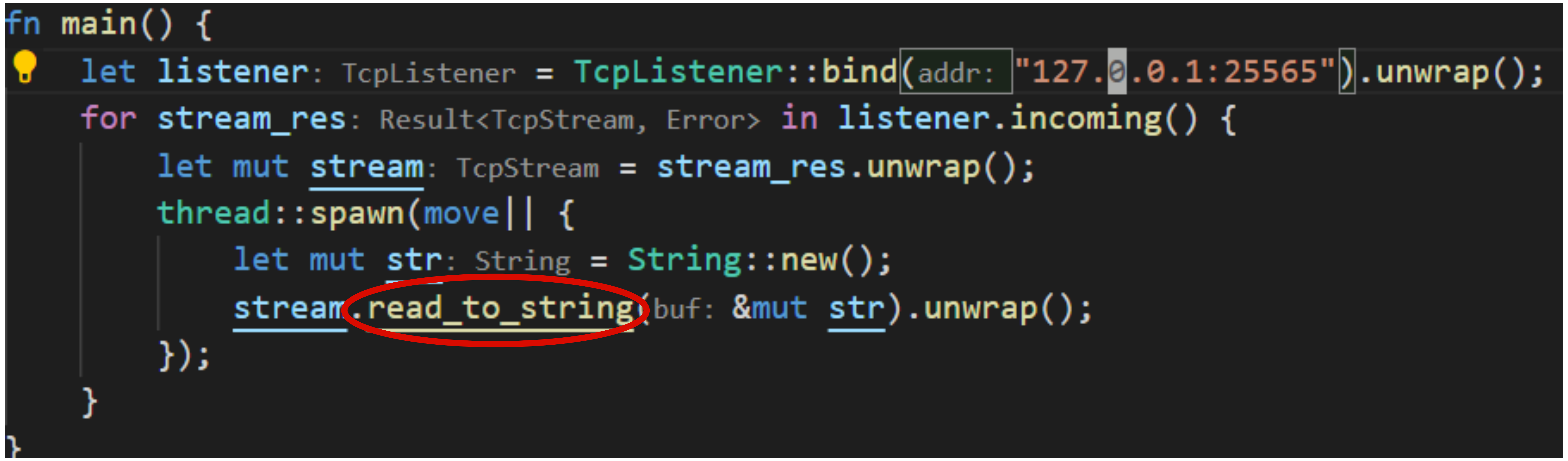
```
fn main() {
    let listener: TcpListener = TcpListener::bind(addr: "127.0.0.1:25565").unwrap();
    for stream_res: Result<TcpStream, Error> in listener.incoming() {
        let mut stream: TcpStream = stream_res.unwrap();
        thread::spawn(move|| {
            let mut str: String = String::new();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

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

Review: Threads

CPU:

Nvm... Time to nap instead...



```
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?

Roadmap

🥳 Threads are great!

😥 But we can’t have too many of them, and context switches are expensive

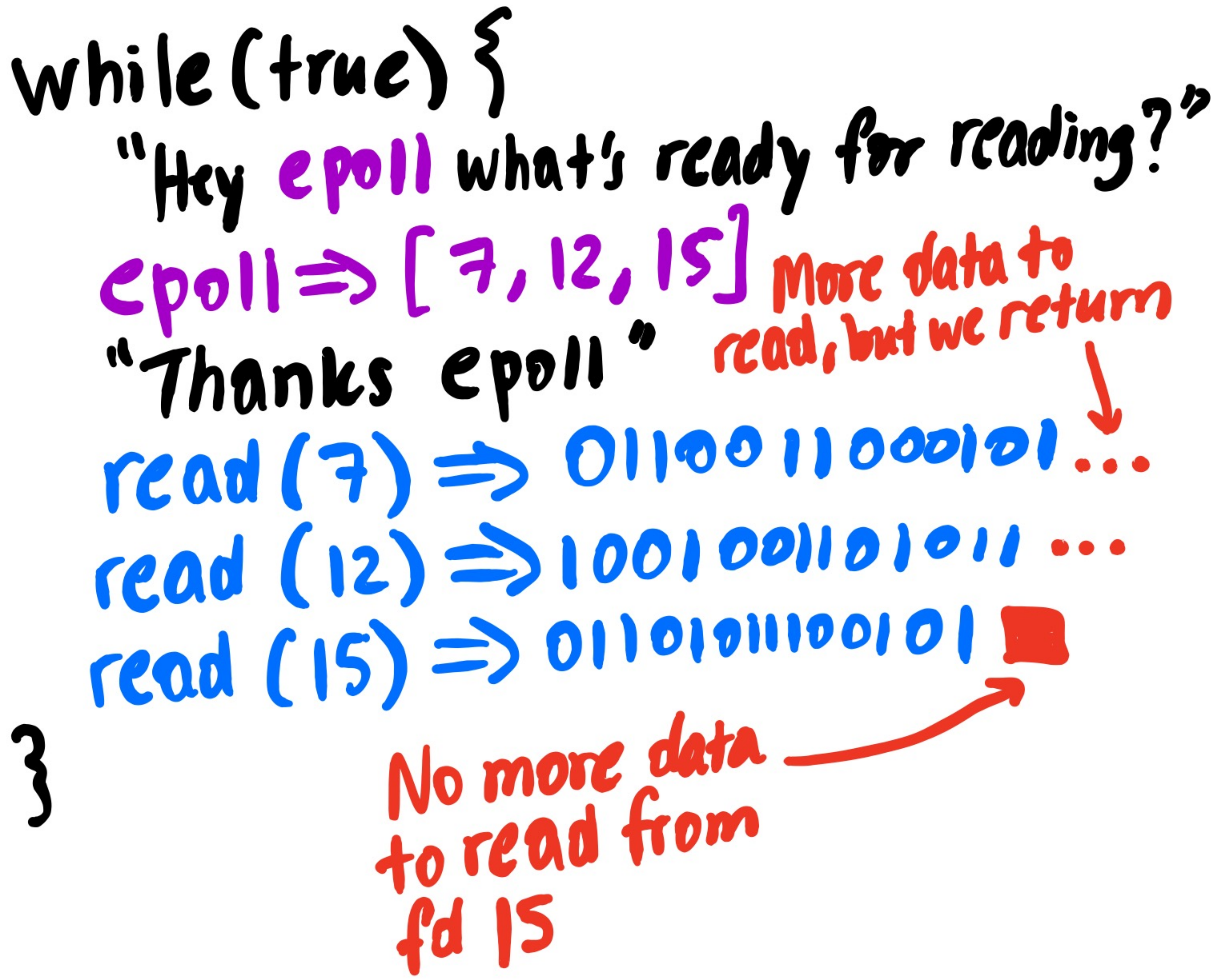
🤔 Is there a way we can have concurrency with less penalties?

Non-blocking I/O

- Traditionally, the read() sys call would block if there is more data to be read but not available.
  - This causes the thread to get pulled off the CPU. It can’t do anything else in the meantime.
- Instead, we could have read() return a special error value instead of blocking
  - If we see that a client hasn’t sent us anything yet, we can do other useful work on this thread e.g. reading from other descriptors we’re managing.
- **This allows us to have concurrent I/O with one thread!**

Non-blocking I/O visualized

- Epoll is a kernel-provided mechanism that notifies us of what fds are ready for I/O.
- Scenario: we are a server having conversations with multiple clients at the same time (a different fd is linked to each client)
- Start by asking epoll, “in which conversations has the client said something?”
- read() from each of those file descriptors, continue those conversations
- Rinse and repeat



```
while(true) {
    "Hey epoll what's ready for reading?"
    epoll => [7, 12, 15]
    "Thanks epoll"
    read(7) => 0110011000101...
    read(12) => 1001001101011 ...
    read(15) => 01101011100101 ■
}
```

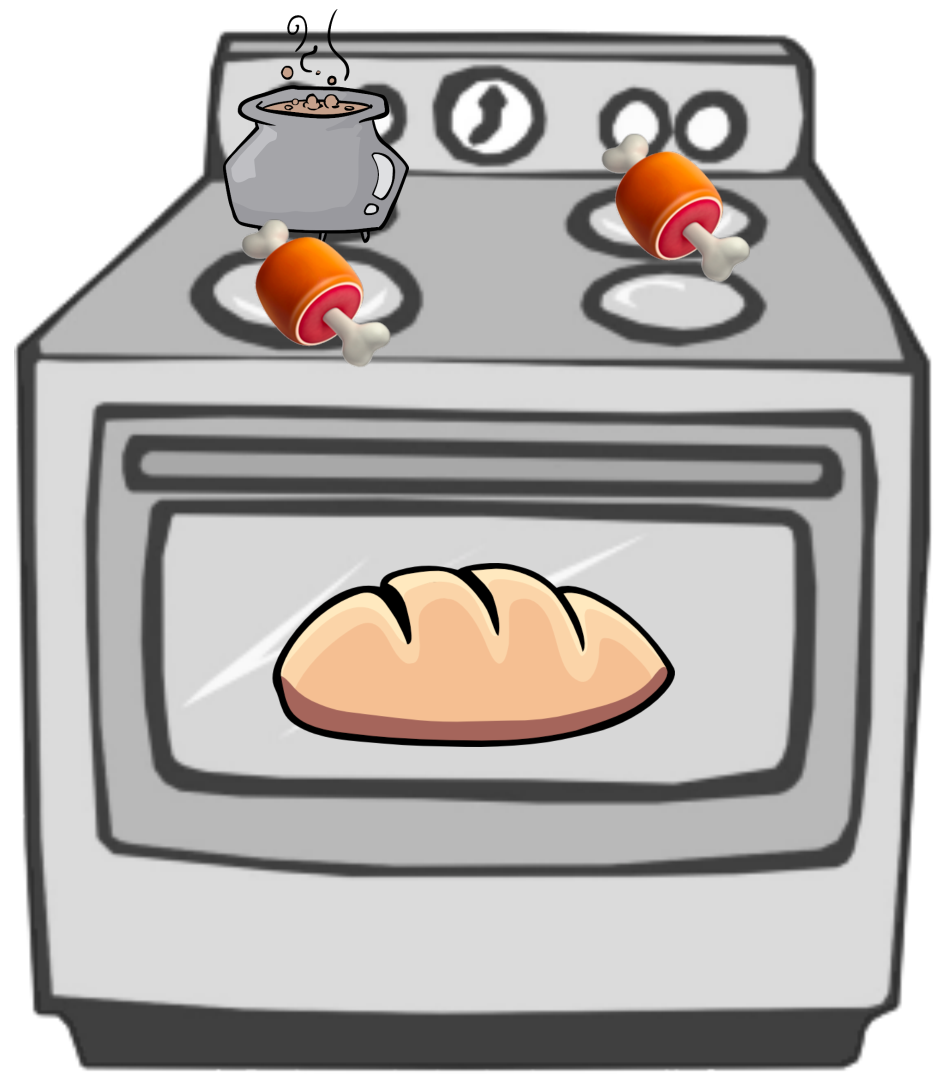
More data to read, but we return

No more data to read from fd 15

State management

- This sort of code looks okay in theory, but reality is much more complicated
- Key problem: need to figure out how to 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

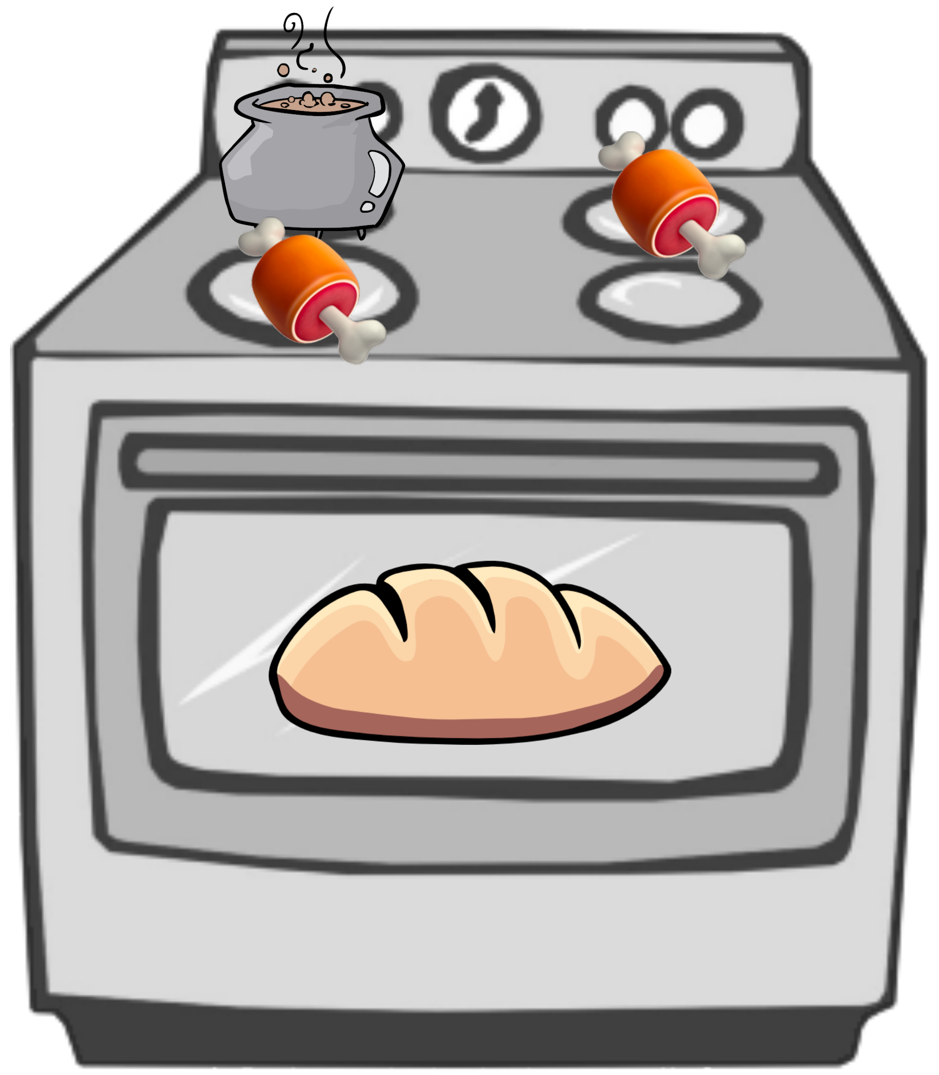
"Executor Thread"



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?
  - Alice the Client asked me for her emails, but I needed to get them from Bob the Database. Now Bob 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
- If we want to manage multiple connections in each thread, we now have a lot of jumbled trains of thoughts that we need to manage state for

“Executor Thread”



Roadmap

🥳 Threads are great!

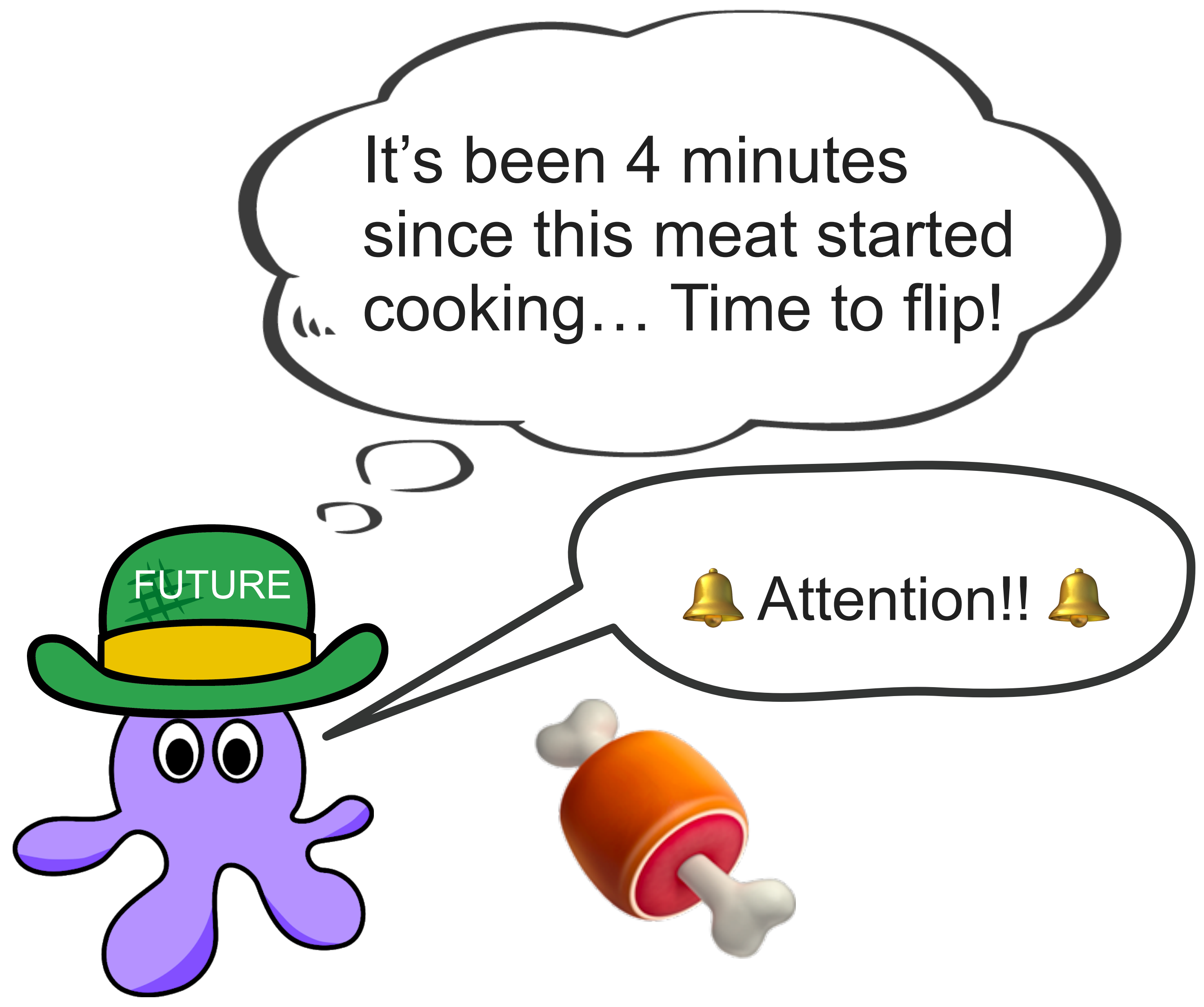
😥 But we can’t have too many of them, and context switches are expensive

⭐ 🤔 Event driven programming is nice in theory, but managing state seems hard

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

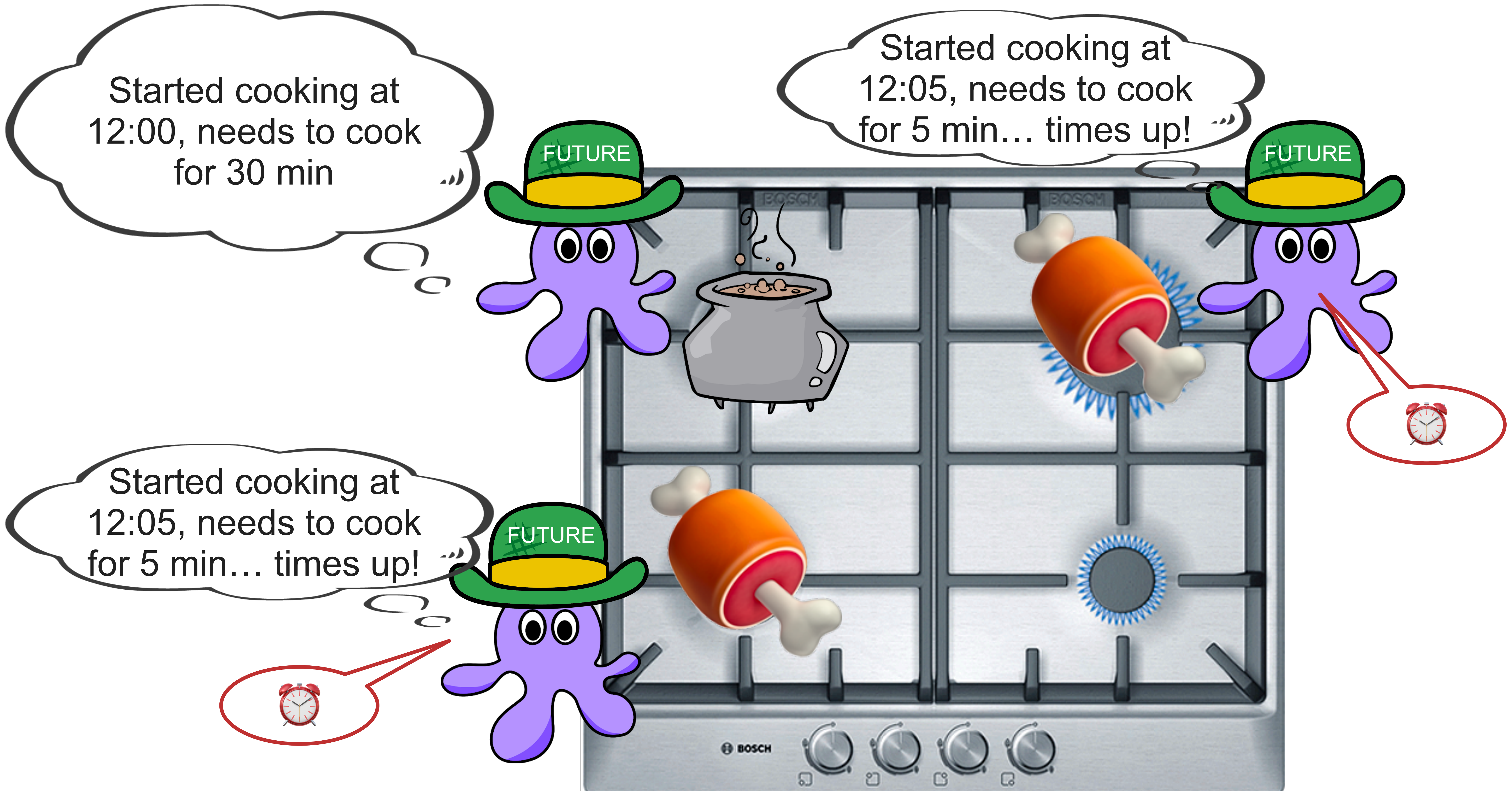
“Executor Thread”


It’s been 4 minutes since this meat started cooking… Time to flip!
FUTURE
🔔 Attention!! 🔔

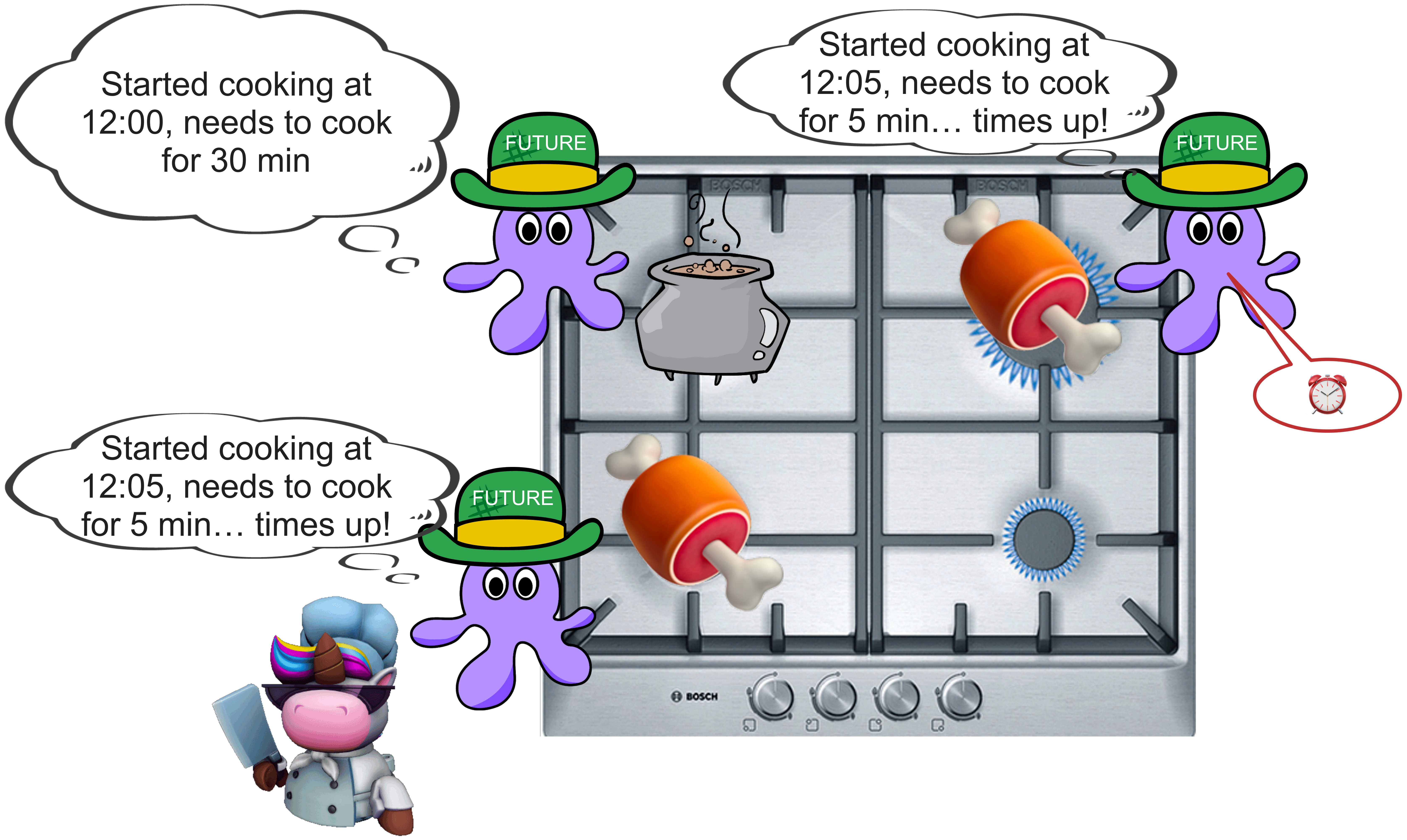
cookMeat future

Futures Visualized

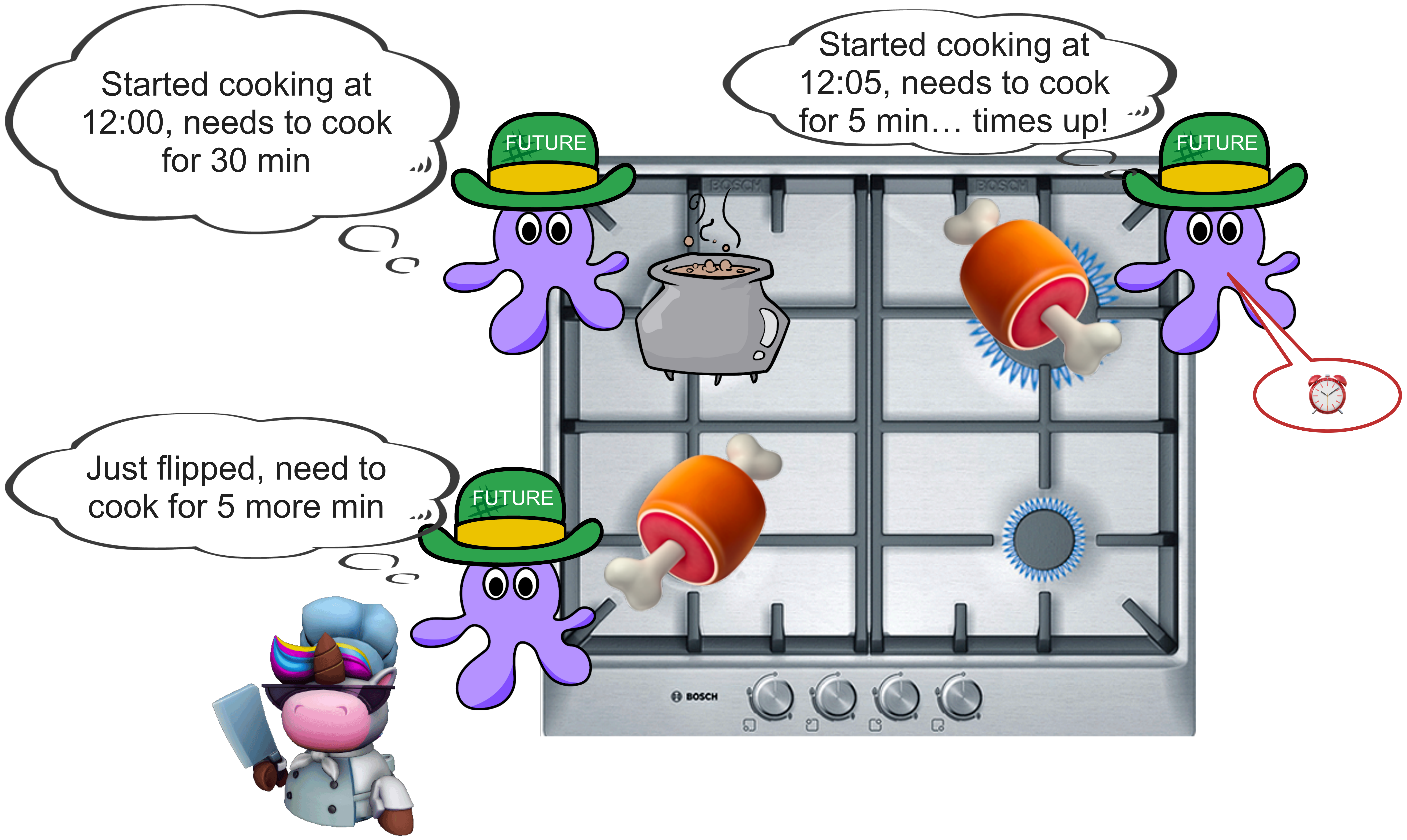
"Executor Thread"


Started cooking at 12:00, needs to cook for 30 min
FUTURE
Started cooking at 12:05, needs to cook for 5 min… times up!
FUTURE
Started cooking at 12:05, needs to cook for 5 min… times up!
FUTURE

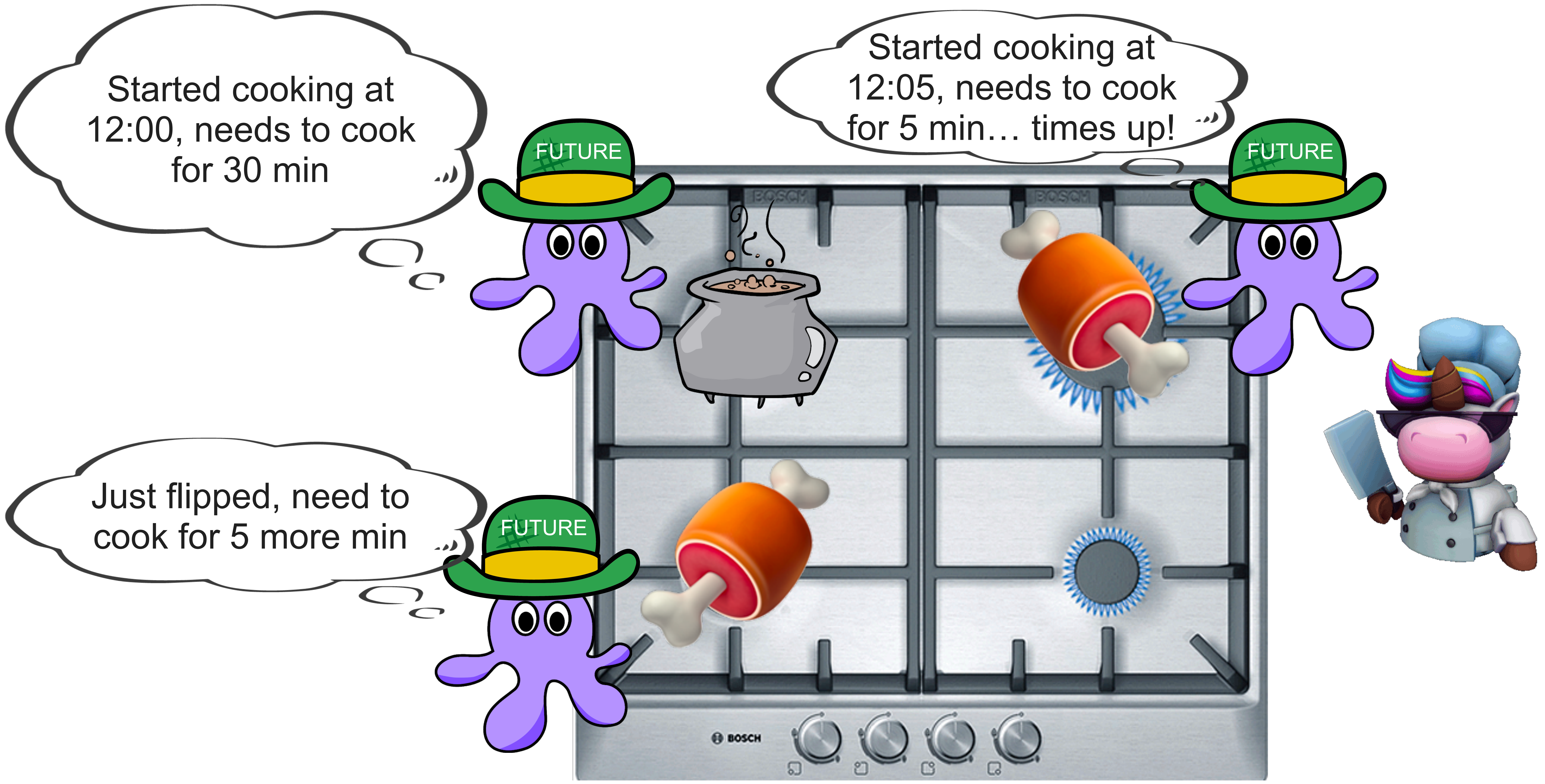
Futures Visualized


Started cooking at 12:00, needs to cook for 30 min
Started cooking at 12:05, needs to cook for 5 min… times up!
Started cooking at 12:05, needs to cook for 5 min… times up!
FUTURE
FUTURE
FUTURE
BOSCH

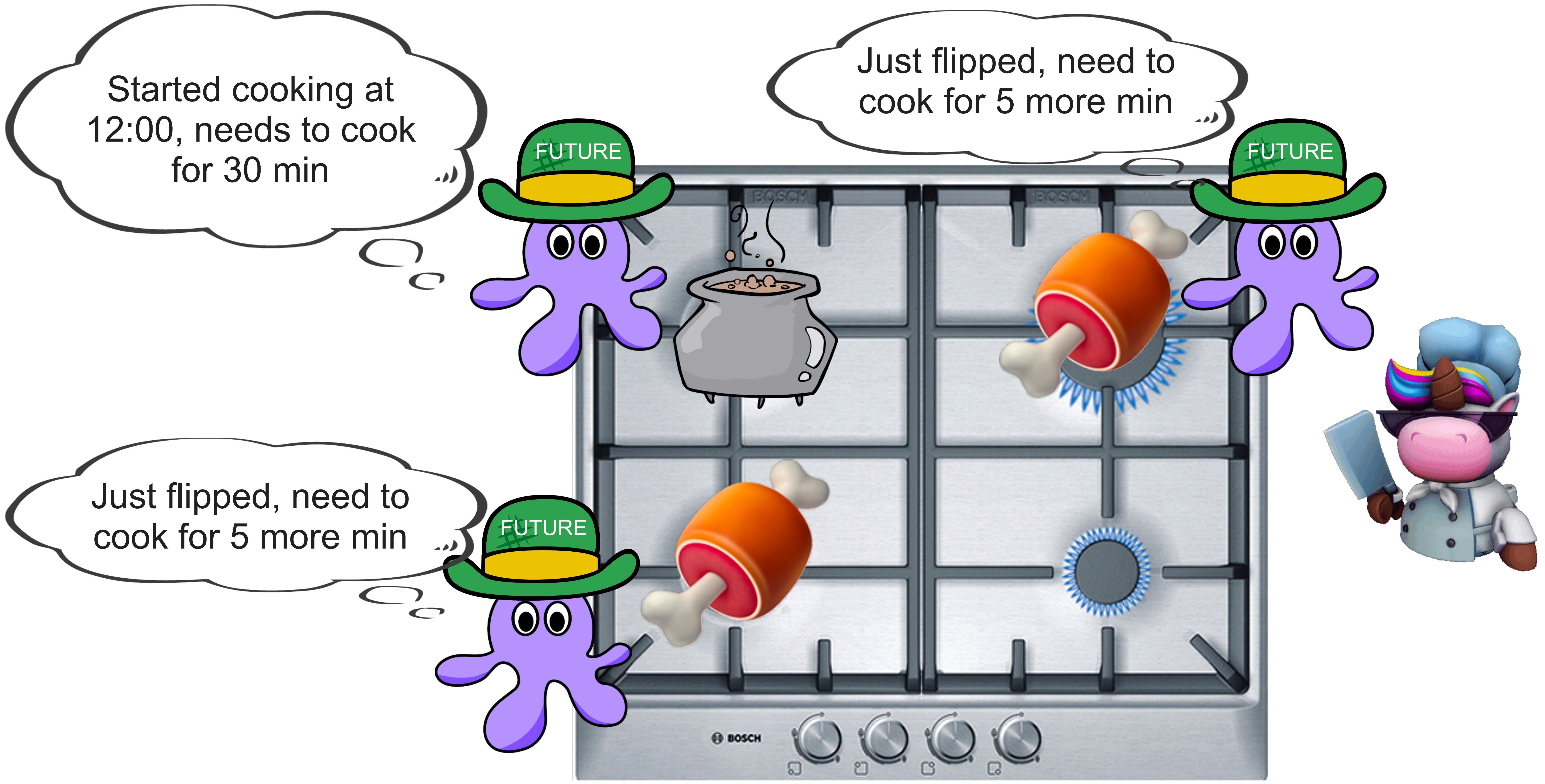
Futures Visualized


Started cooking at 12:00, needs to cook for 30 min
Started cooking at 12:05, needs to cook for 5 min… times up!
Just flipped, need to cook for 5 more min
FUTURE
FUTURE
FUTURE
BOSCH

Futures Visualized

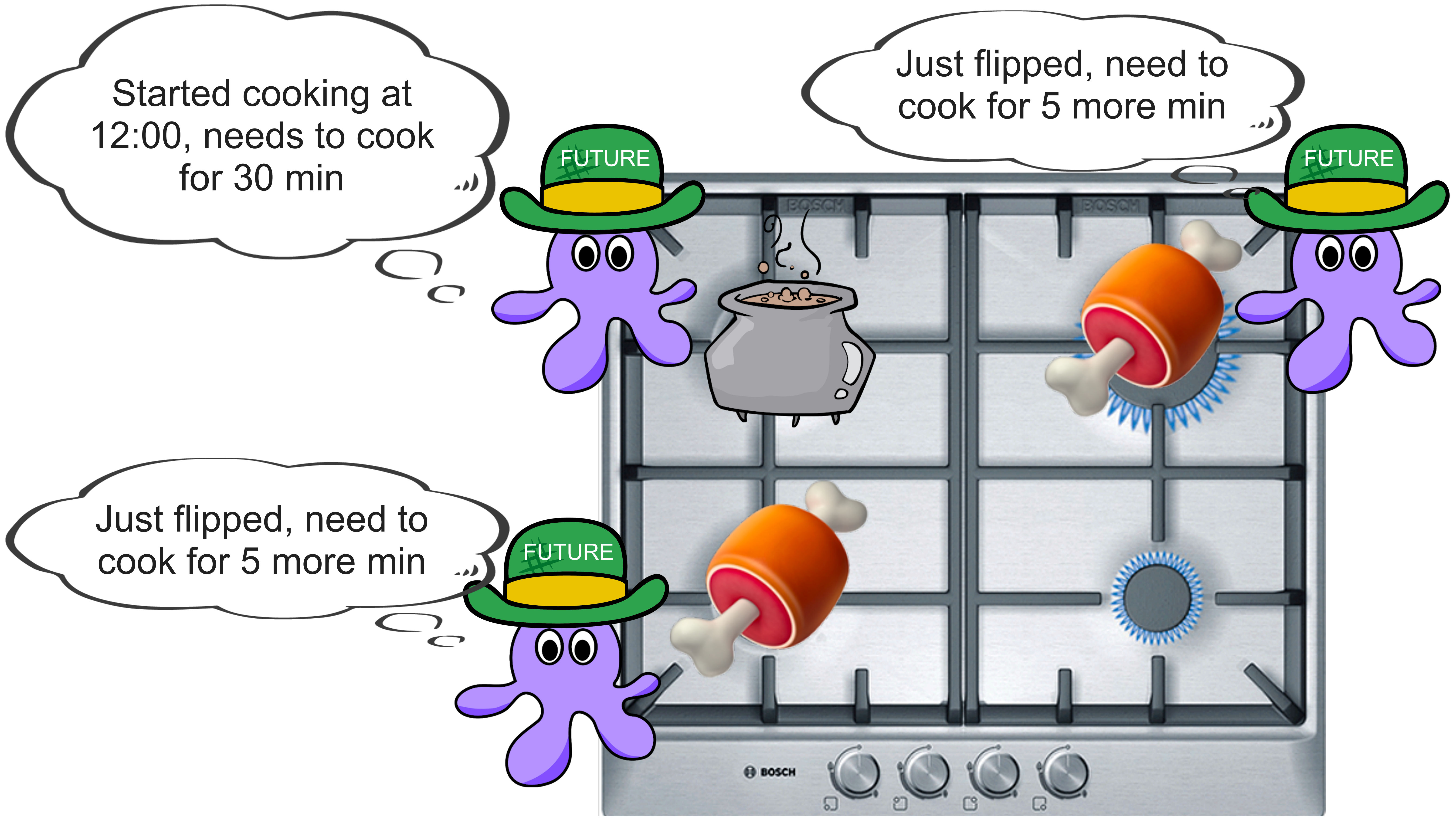

Started cooking at 12:00, needs to cook for 30 min
Started cooking at 12:05, needs to cook for 5 min… times up!
Just flipped, need to cook for 5 more min
FUTURE
FUTURE
FUTURE
BOSCH

Futures Visualized


Started cooking at 12:00, needs to cook for 30 min
Just flipped, need to cook for 5 more min
Just flipped, need to cook for 5 more min
FUTURE
FUTURE
FUTURE
BOSCH

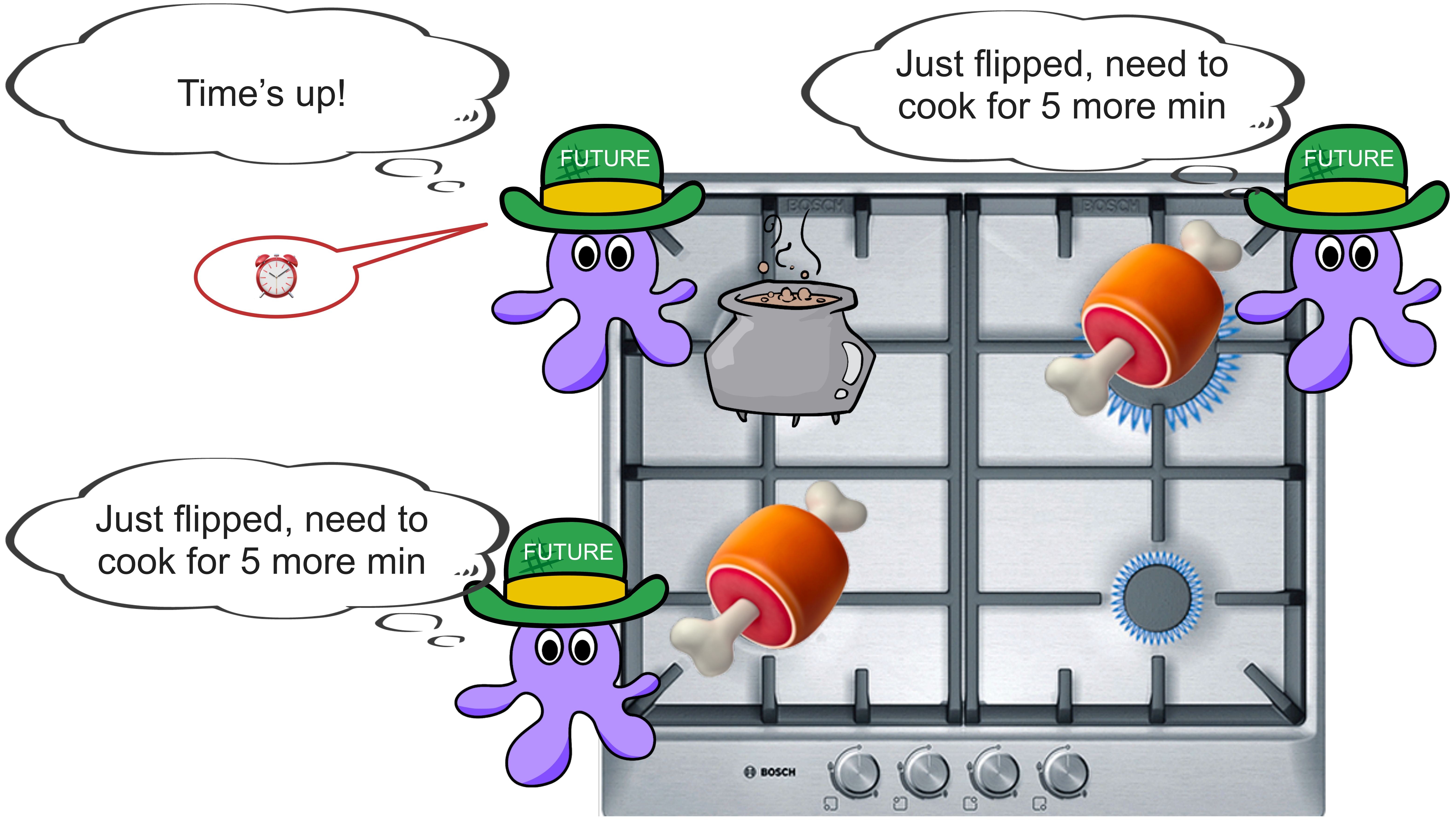
Futures Visualized

Executor thread
(sleeping)

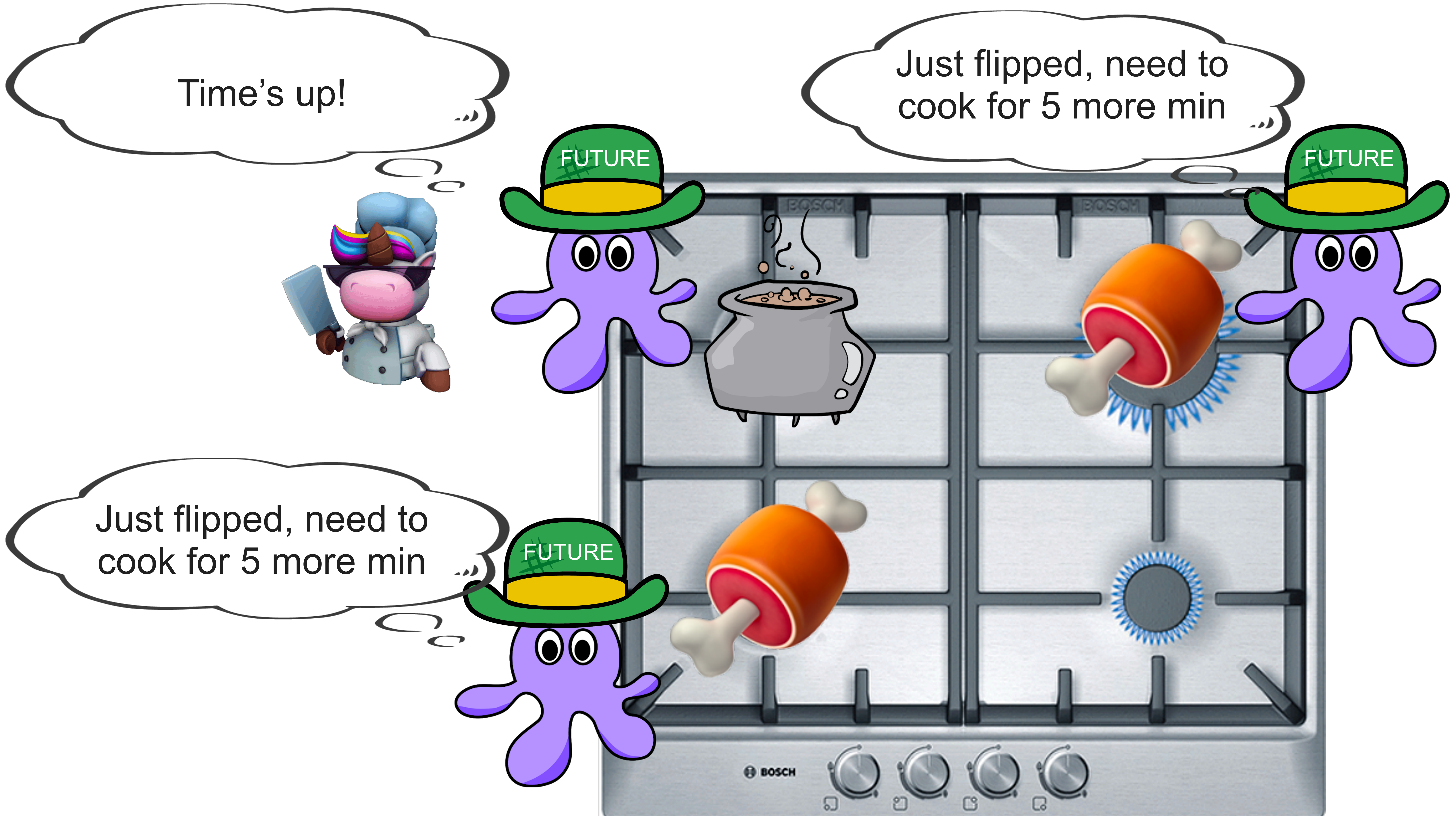

Started cooking at 12:00, needs to cook for 30 min
Just flipped, need to cook for 5 more min
Just flipped, need to cook for 5 more min
FUTURE
FUTURE
FUTURE

Futures Visualized

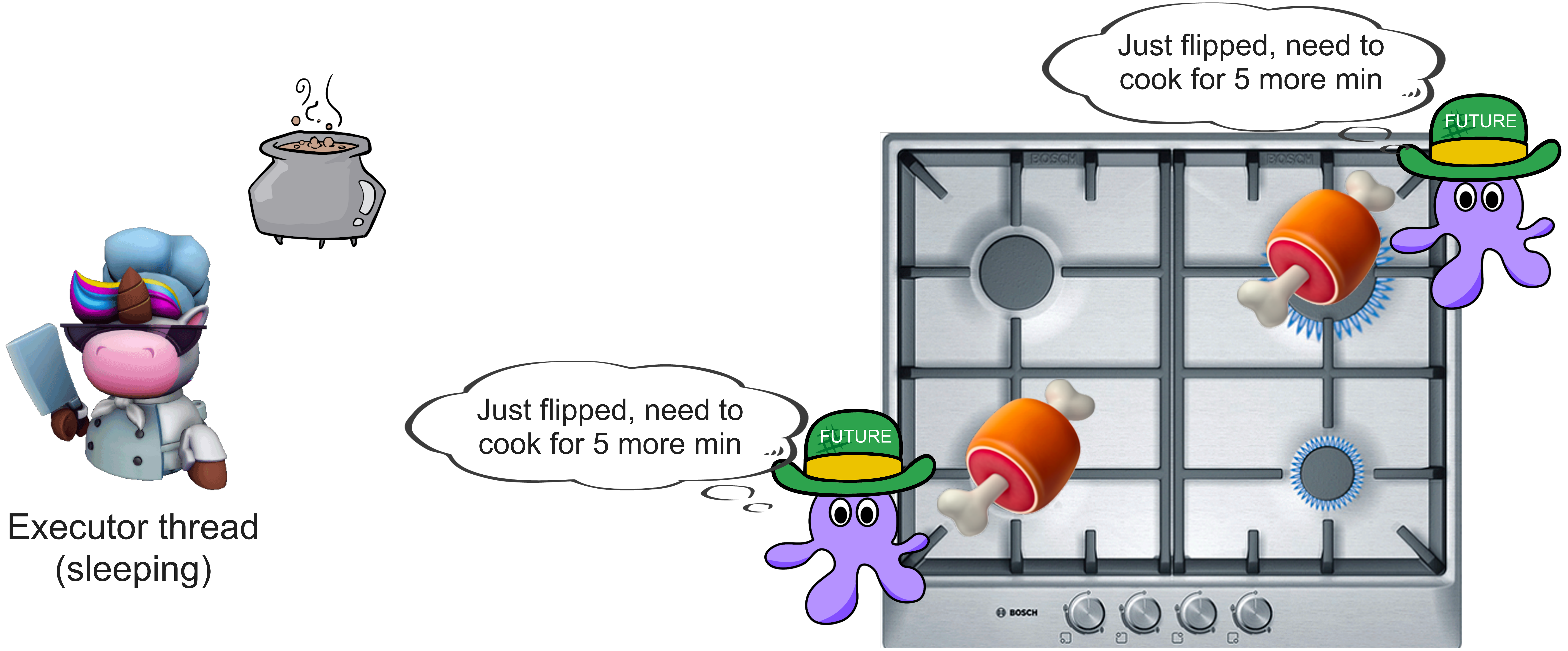
Executor thread
(sleeping)


Time’s up!
⏰
Just flipped, need to cook for 5 more min
Just flipped, need to cook for 5 more min
FUTURE
FUTURE
FUTURE

Futures Visualized

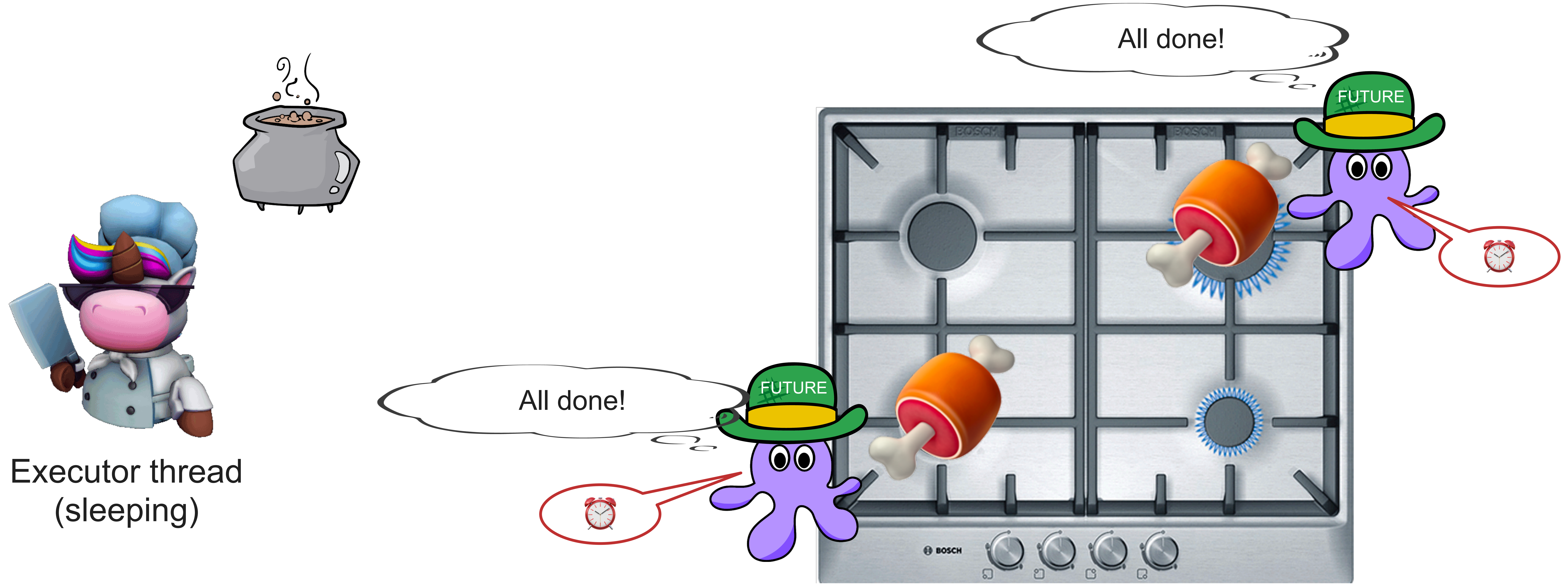

Time's up!
Just flipped, need to cook for 5 more min
FUTURE
FUTURE
Just flipped, need to cook for 5 more min
FUTURE

Futures Visualized


Just flipped, need to cook for 5 more min
FUTURE
Just flipped, need to cook for 5 more min
FUTURE
BOSCH

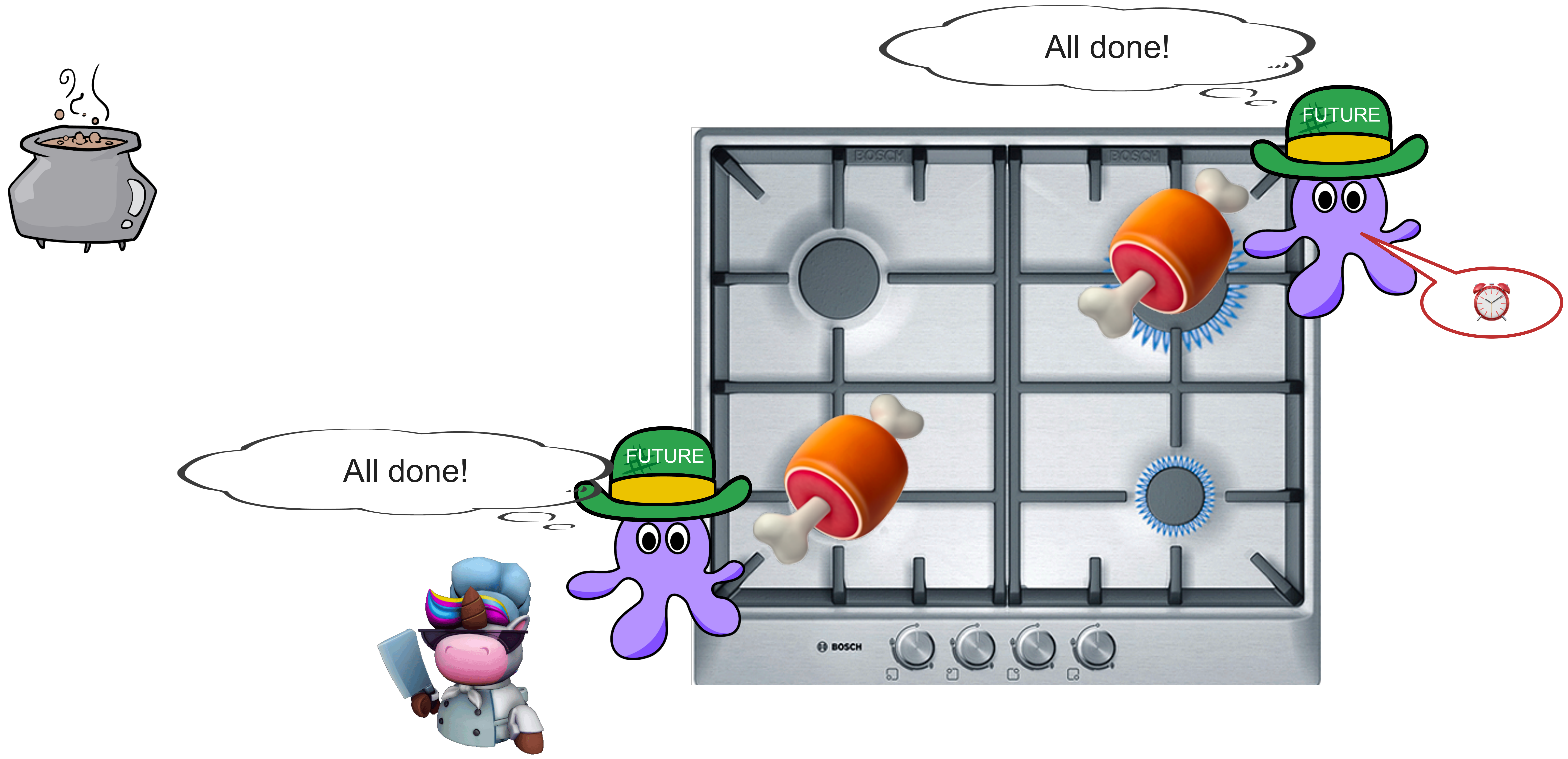
Executor thread
(sleeping)

Futures Visualized



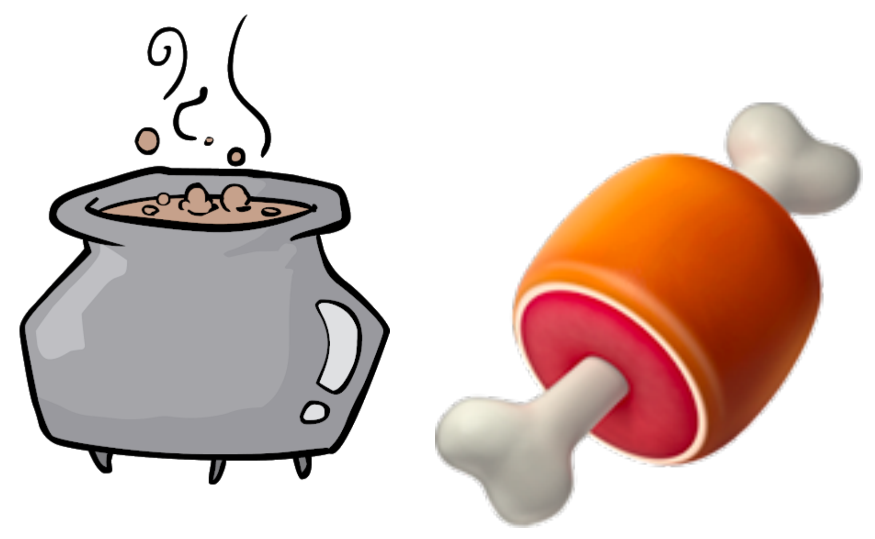
Executor thread
(sleeping)

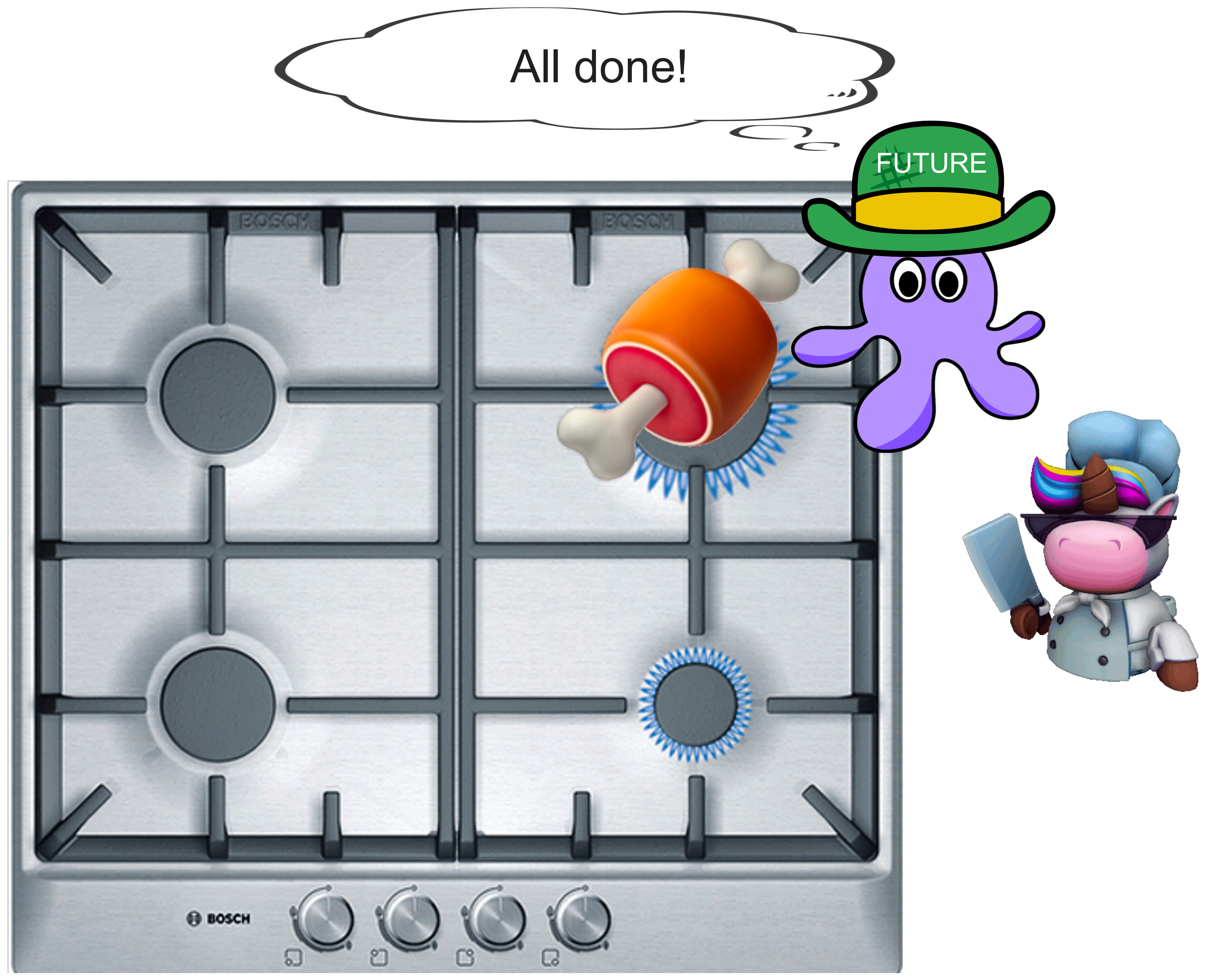
Futures Visualized



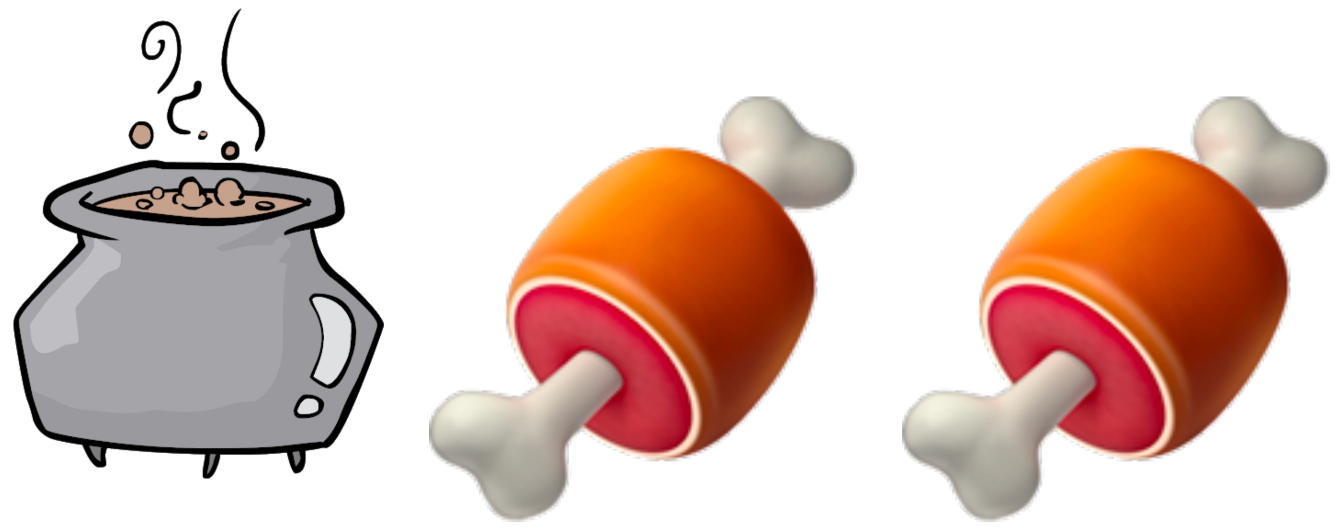
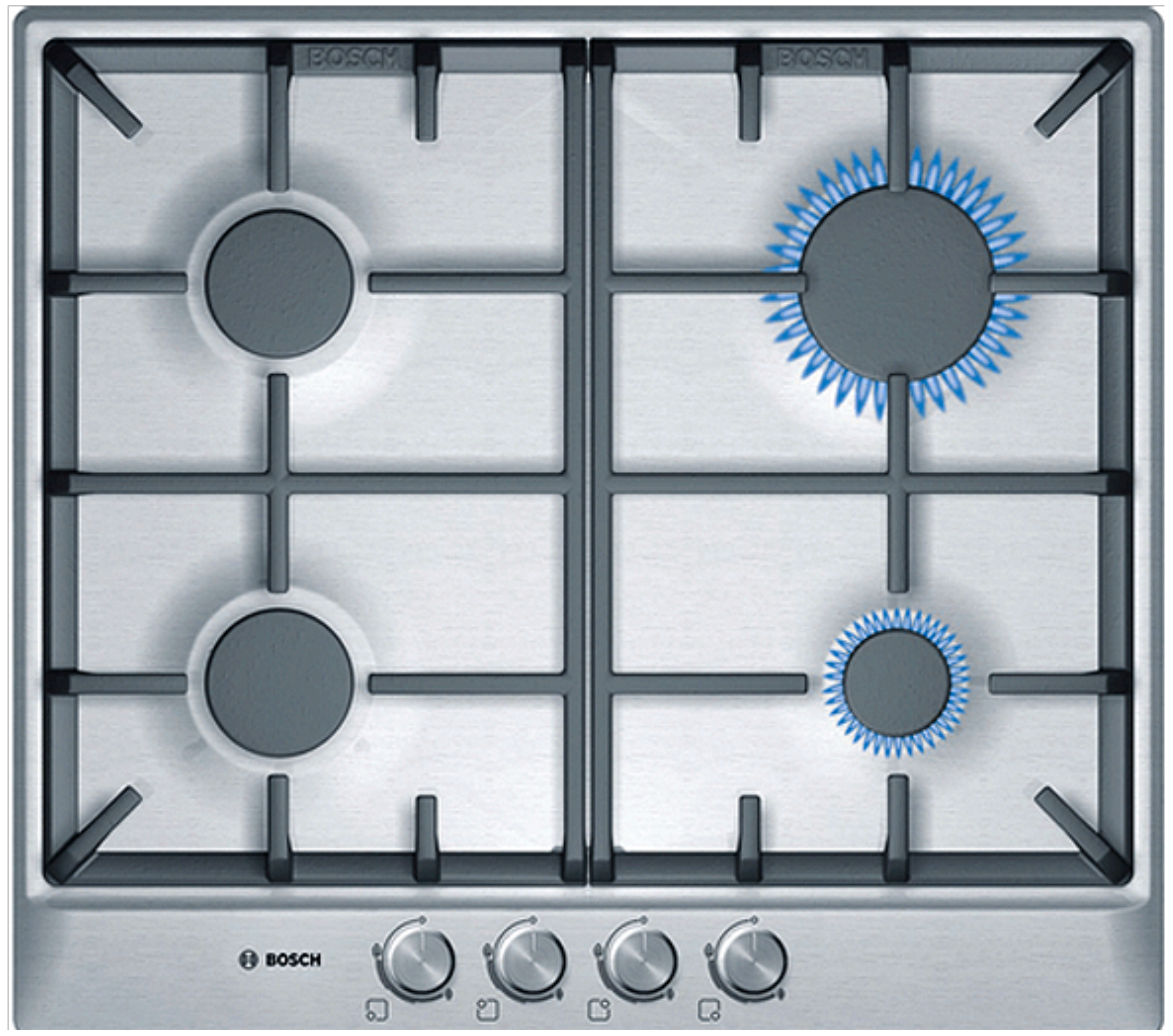
Futures Visualized

Futures Visualized





Futures Visualized

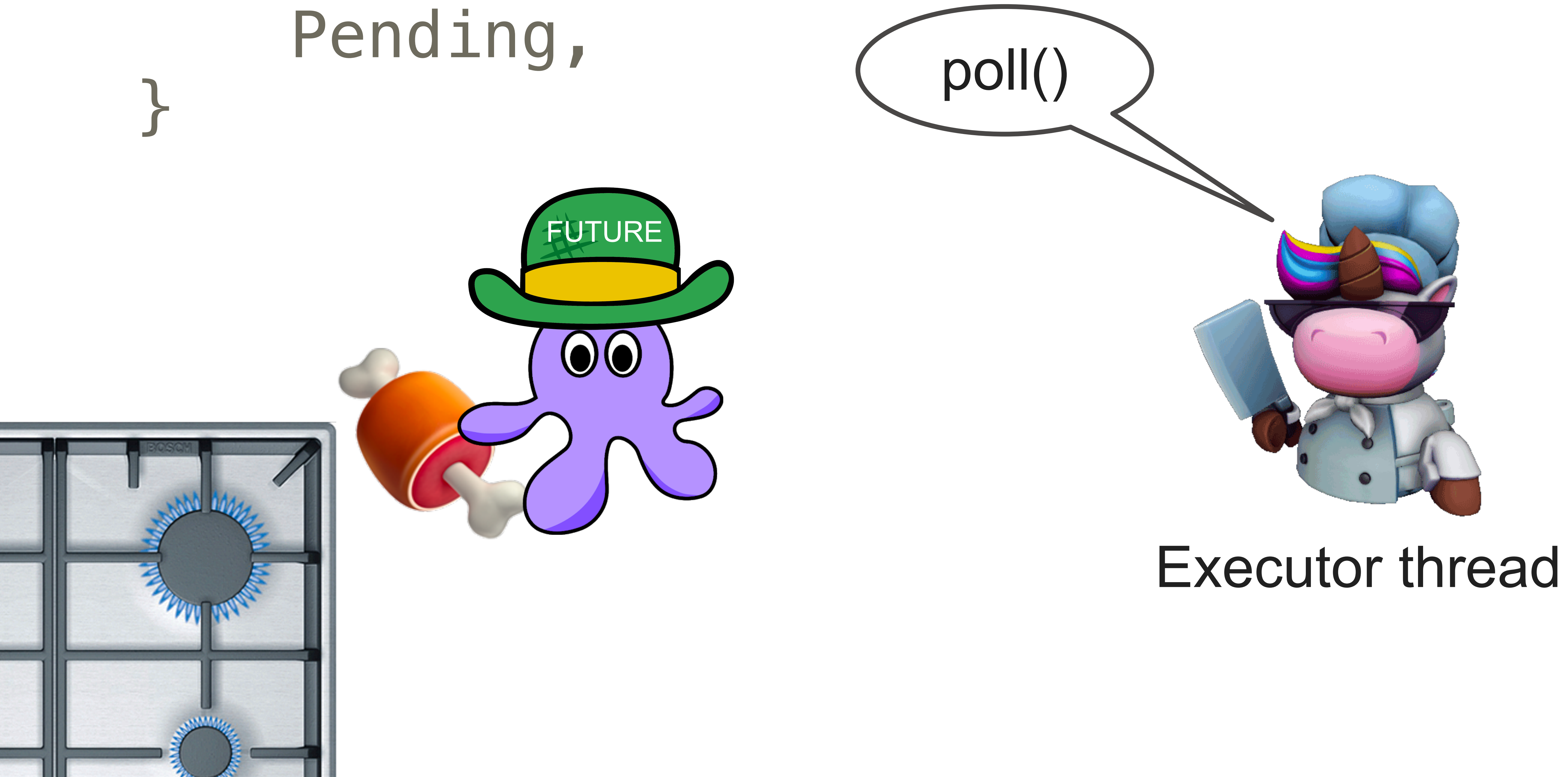



The Future Trait

```
trait Future { // This is a simplified version of the Future definition
    type Output;
    fn poll(&mut self, cx: &mut Context) -> Poll<Self::Output>;
}

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

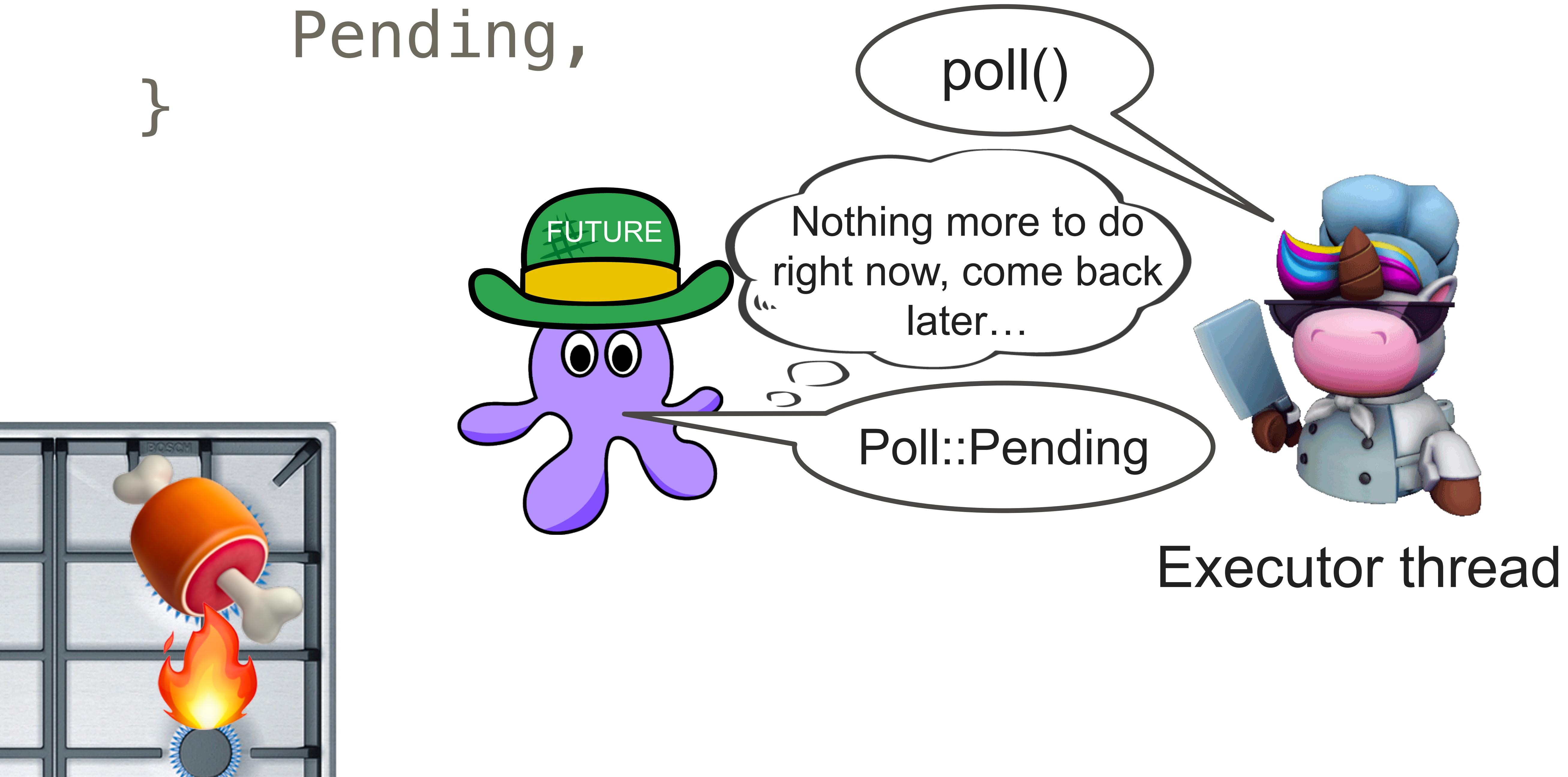
- The executor thread should call poll() on the future to start it off
- It will run code until it can no longer progress.



The Future Trait

```
trait Future { // This is a simplified version of the Future definition
    type Output;
    fn poll(&mut self, cx: &mut Context) -> Poll<Self::Output>;
}

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



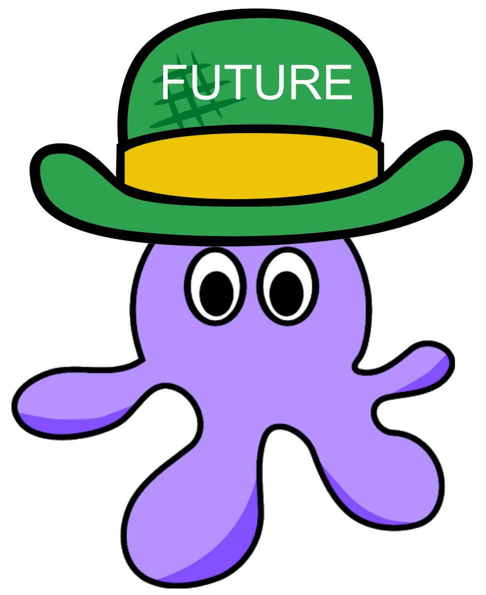
- The executor thread should call poll() on the future to start it off
- It will run code until it can no longer progress.
  - If the future is complete, returns Poll::Ready(T)
  - If future needs to wait for some event, returns Poll::Pending, and allows the single thread to work on another future

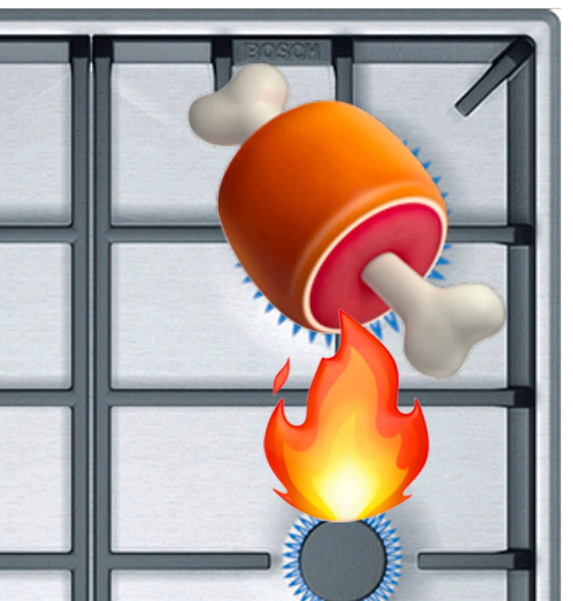
The Future Trait

```
trait Future { // This is a simplified version of the Future definition
    type Output;
    fn poll(&mut self, cx: &mut Context) -> Poll<Self::Output>;
}

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

- 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





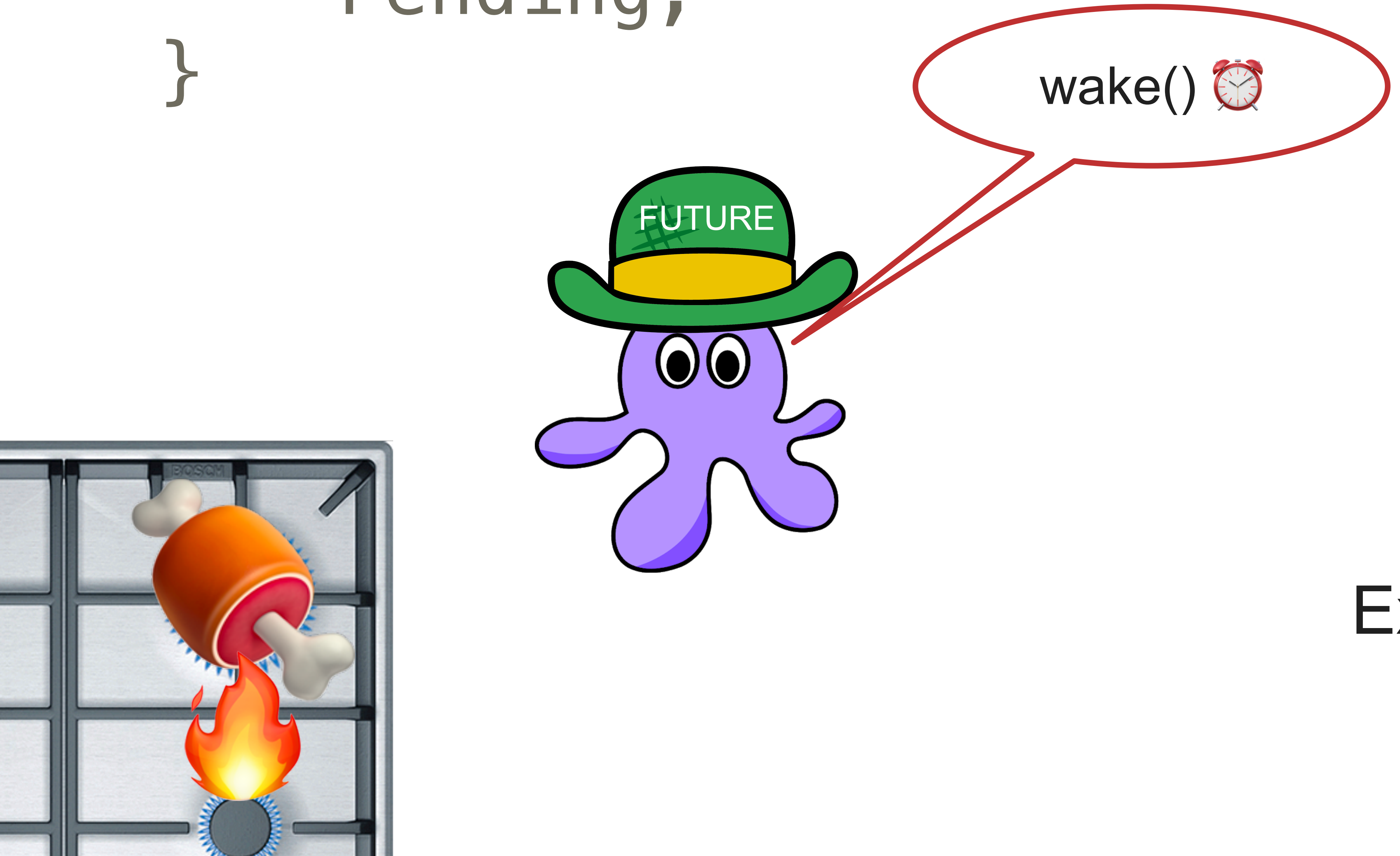
Executor thread (doing other things)

The Future Trait

```
trait Future { // This is a simplified version of the Future definition
    type Output;
    fn poll(&mut self, cx: &mut Context) -> Poll<Self::Output>;
}

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

- When poll() is called, Context structure passed in.
- Includes a “wake()” function that is set to be called when future can make progress again (This is implemented internally using system calls)
- After wake() called, executor can use Context to see which Future can be polled to make new progress



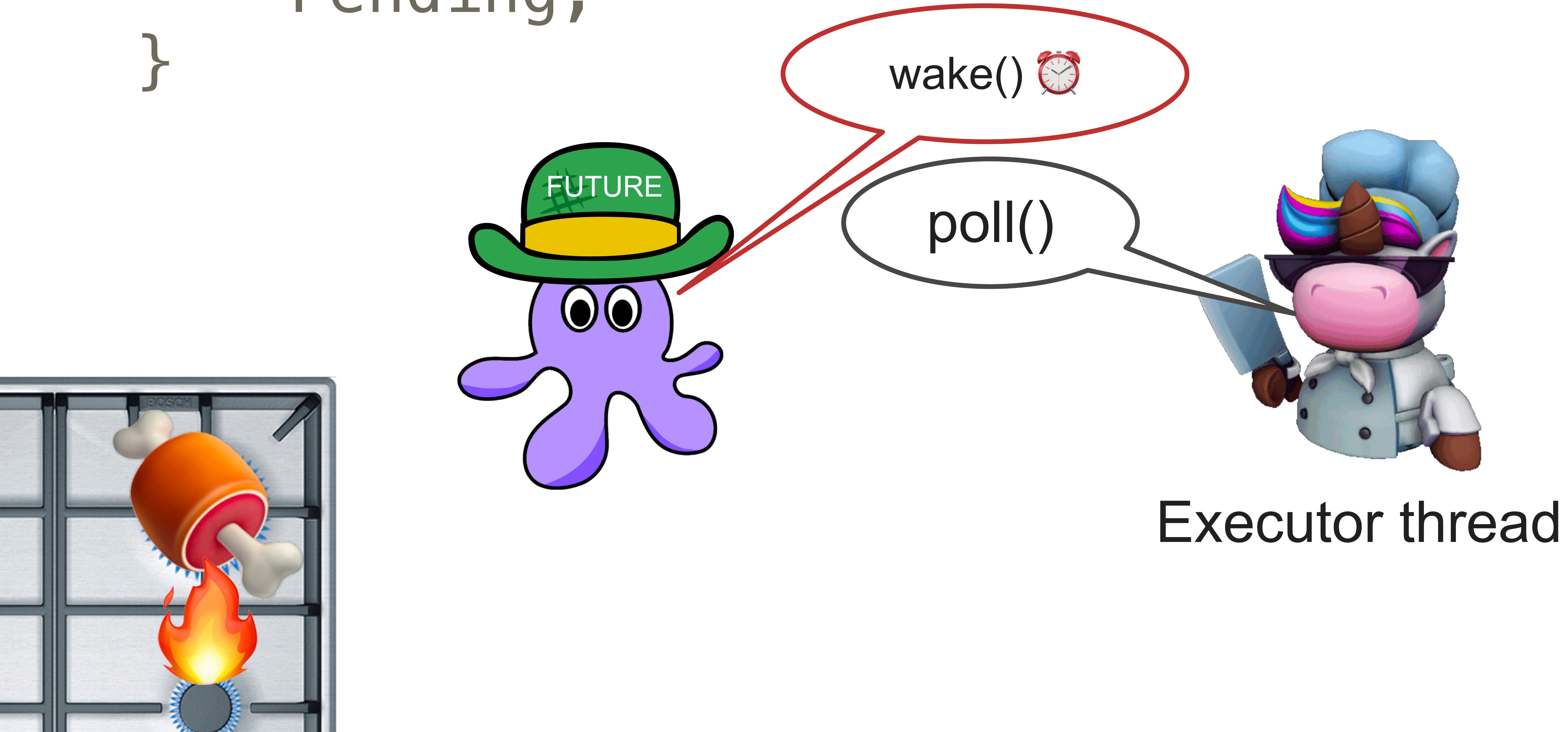
Executor thread (sleeping)

The Future Trait

```
trait Future { // This is a simplified version of the Future definition
    type Output;
    fn poll(&mut self, cx: &mut Context) -> Poll<Self::Output>;
}

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

- When poll() is called, Context structure passed in.
- Includes a “wake()” function that is set to be called when future can make progress again (This is implemented internally using system calls)
- After wake() called, executor can use Context to see which Future can be polled to make new progress

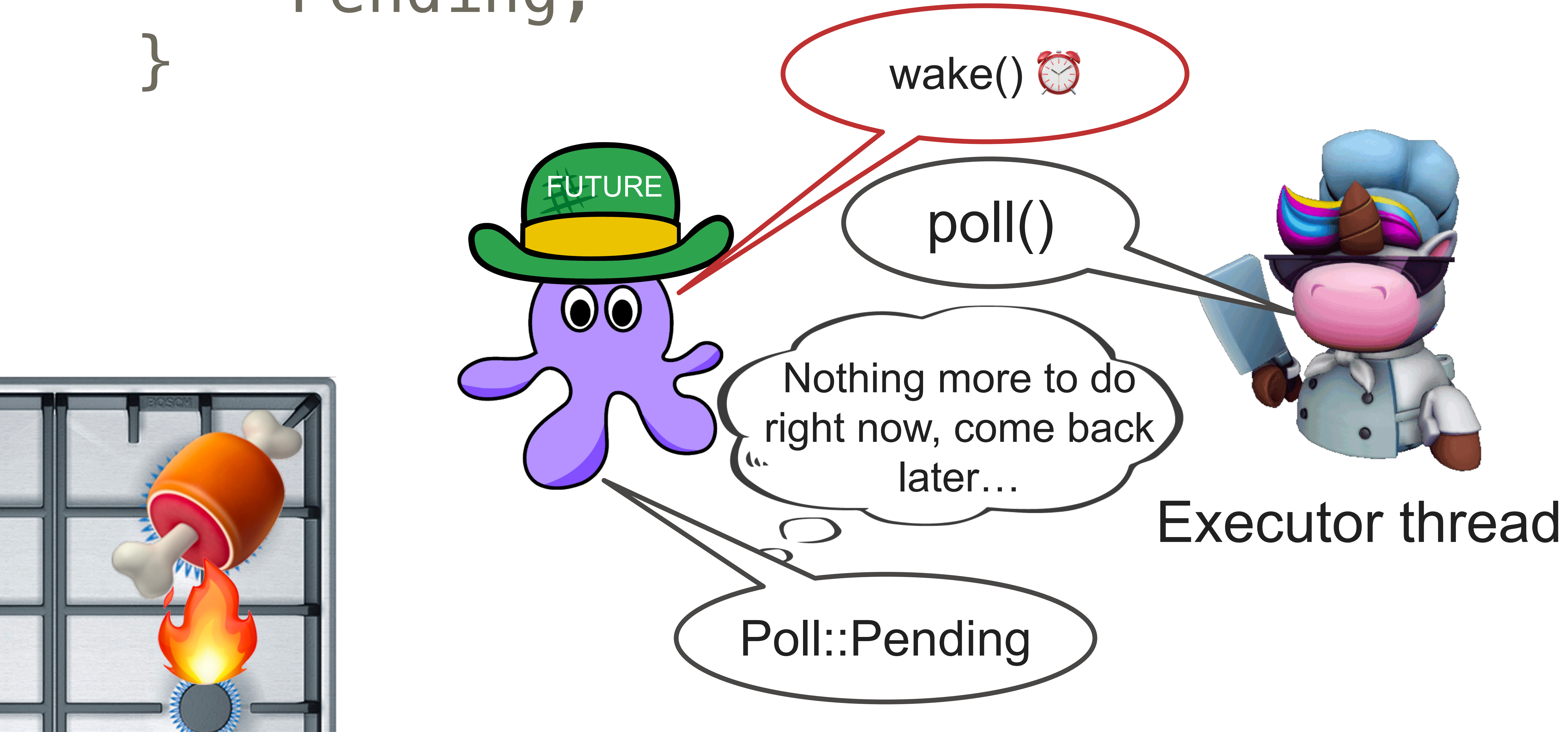


The Future Trait

```
trait Future { // This is a simplified version of the Future definition
    type Output;
    fn poll(&mut self, cx: &mut Context) -> Poll<Self::Output>;
}

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

- When poll() is called, Context structure passed in.
- Includes a "wake()" function that is set to be called when future can make progress again (This is implemented internally using system calls)
- After wake() called, executor can use Context to see which Future can be polled to make new progress

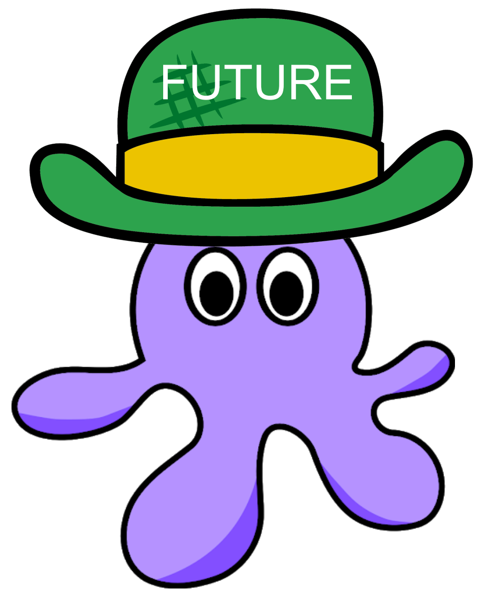


The Future Trait

```
trait Future { // This is a simplified version of the Future definition
    type Output;
    fn poll(&mut self, cx: &mut Context) -> Poll<Self::Output>;
}

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

- When poll() is called, Context structure passed in.
- Includes a “wake()” function that is set to be called when future can make progress again (This is implemented internally using system calls)
- After wake() called, executor can use Context to see which Future can be polled to make new progress

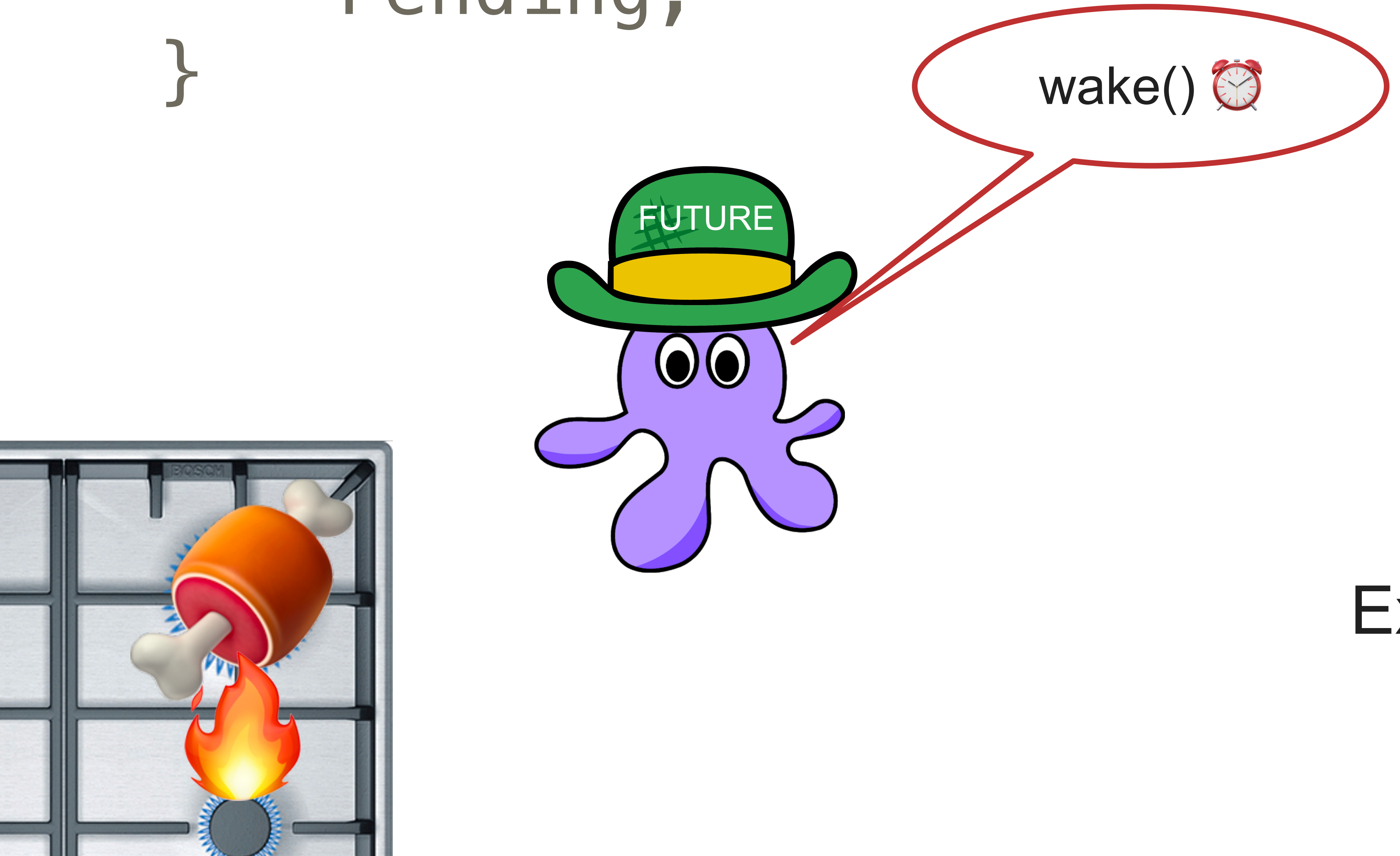


Executor thread (sleeping)

The Future Trait

```
trait Future { // This is a simplified version of the Future definition
    type Output;
    fn poll(&mut self, cx: &mut Context) -> Poll<Self::Output>;
}

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



Executor thread (sleeping)

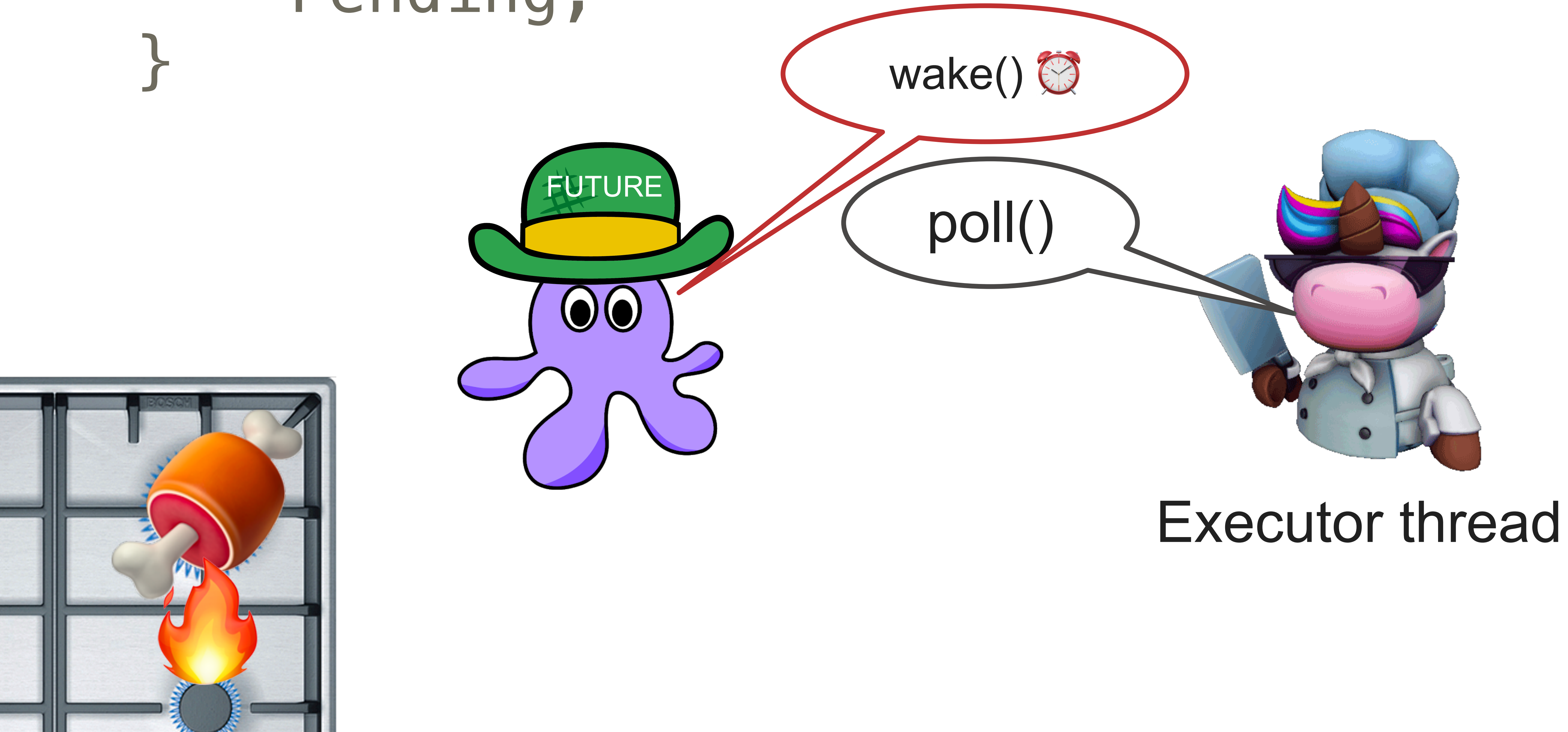
- When poll() is called, Context structure passed in.
- Includes a “wake()” function that is set to be called when future can make progress again (This is implemented internally using system calls)
- After wake() called, executor can use Context to see which Future can be polled to make new progress

The Future Trait

```
trait Future { // This is a simplified version of the Future definition
    type Output;
    fn poll(&mut self, cx: &mut Context) -> Poll<Self::Output>;
}

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

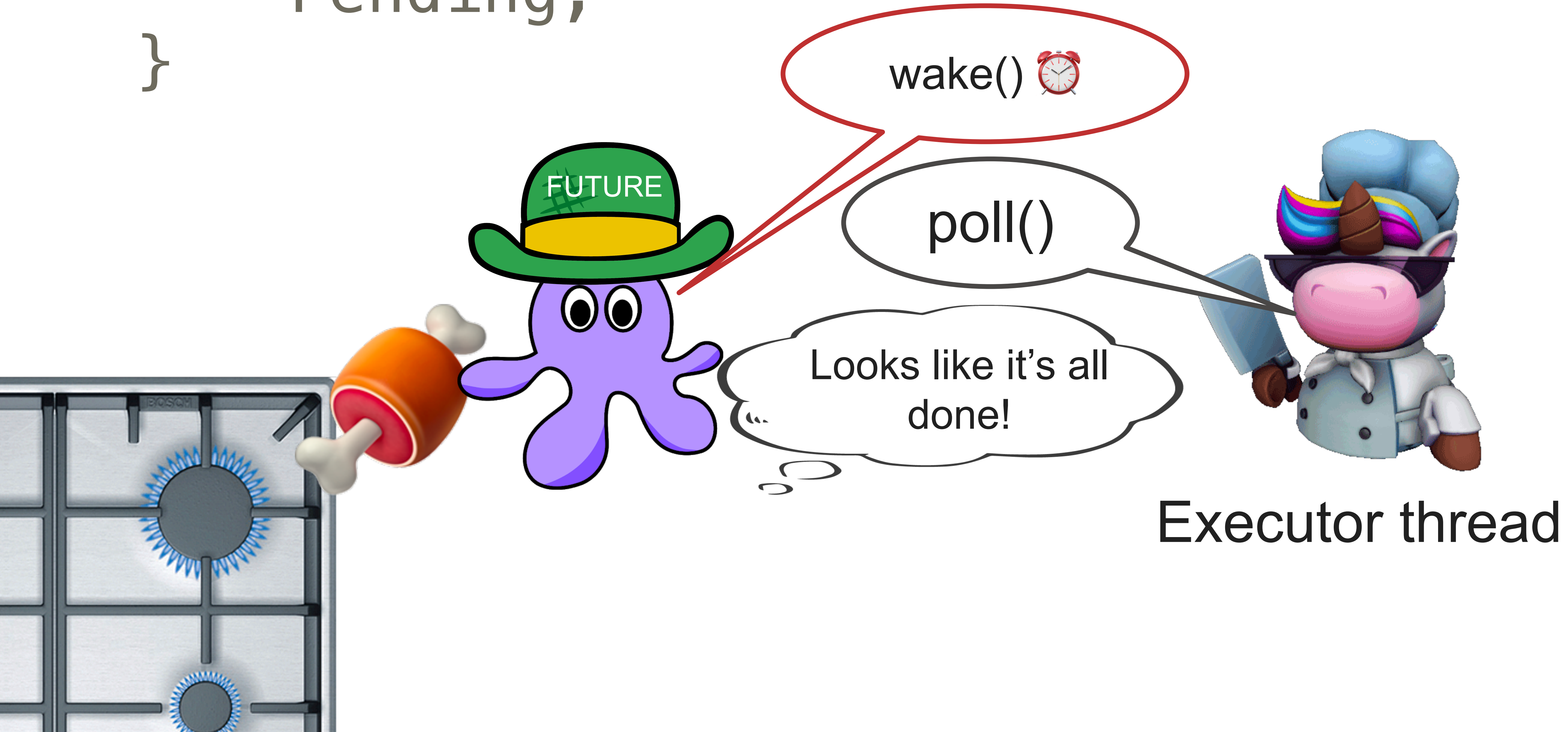
- When poll() is called, Context structure passed in.
- Includes a "wake()" function that is set to be called when future can make progress again (This is implemented internally using system calls)
- After wake() called, executor can use Context to see which Future can be polled to make new progress



The Future Trait

```
trait Future { // This is a simplified version of the Future definition
    type Output;
    fn poll(&mut self, cx: &mut Context) -> Poll<Self::Output>;
}

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



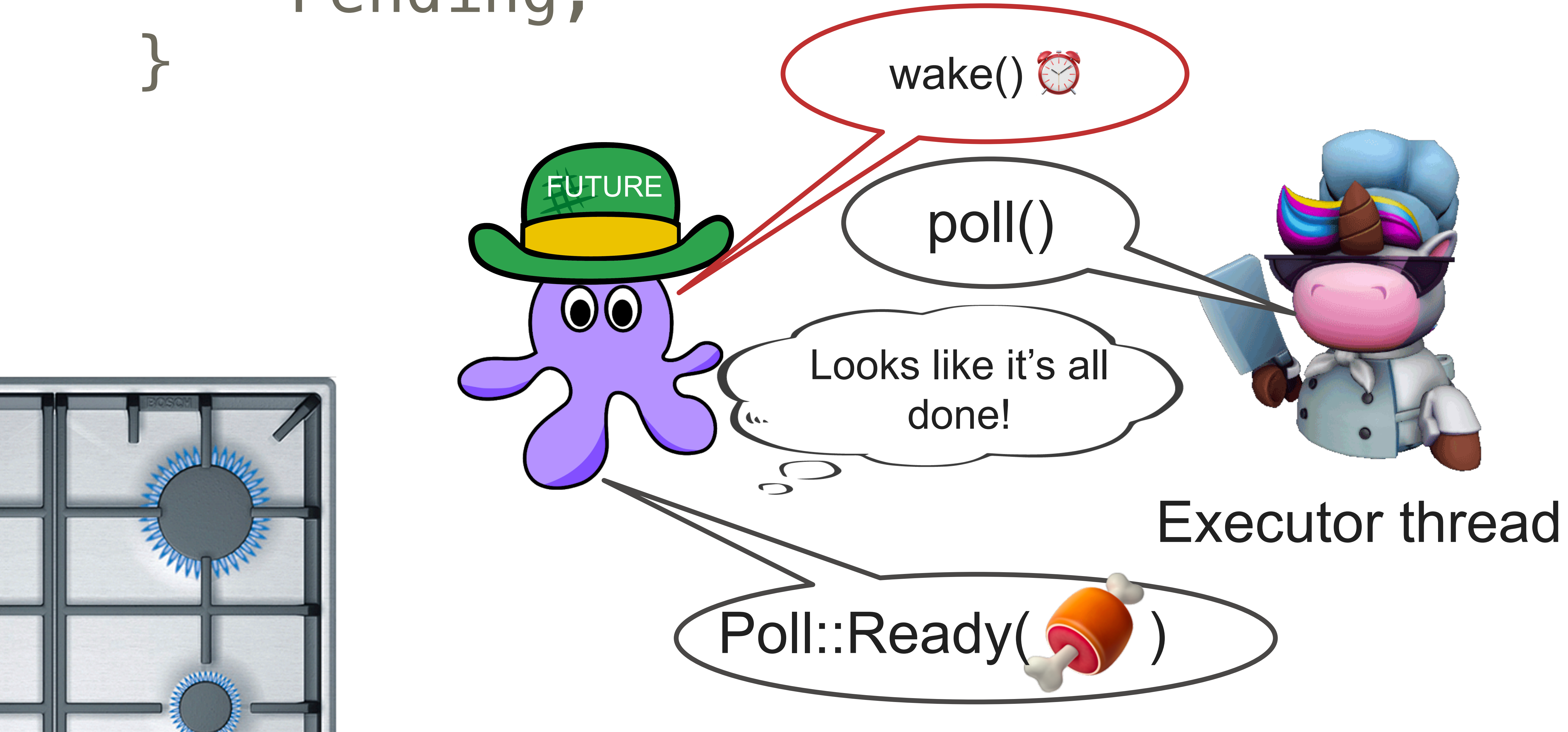
- When poll() is called, Context structure passed in.
- Includes a "wake()" function that is set to be called when future can make progress again (This is implemented internally using system calls)
- After wake() called, executor can use Context to see which Future can be polled to make new progress

The Future Trait

```
trait Future { // This is a simplified version of the Future definition
    type Output;
    fn poll(&mut self, cx: &mut Context) -> Poll<Self::Output>;
}

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

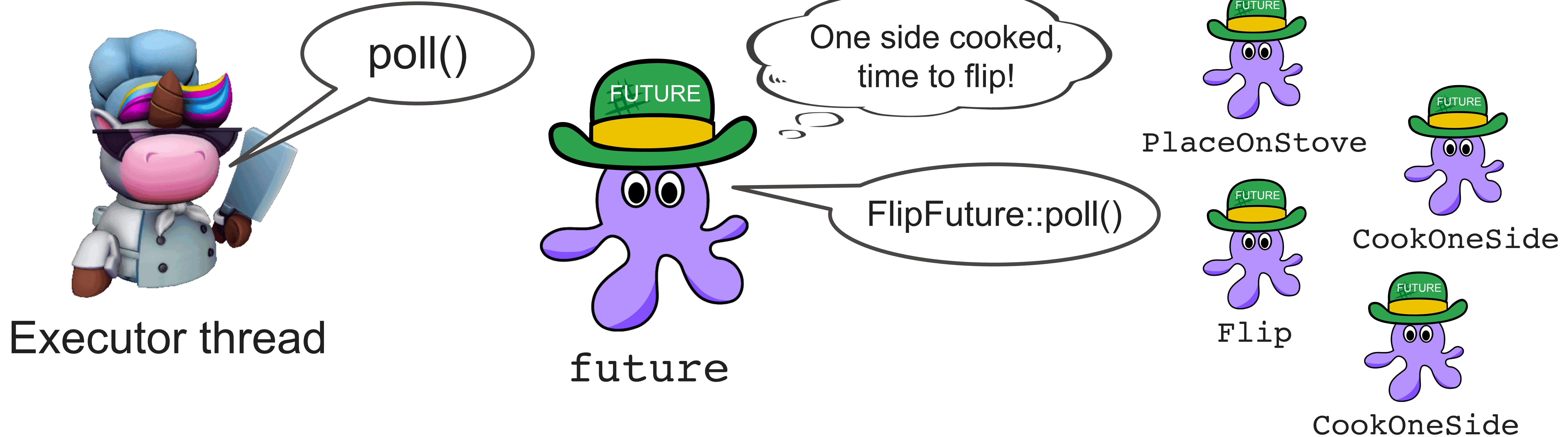
- When poll() is called, Context structure passed in.
- Includes a "wake()" function that is set to be called when future can make progress again (This is implemented internally using system calls)
- After wake() called, executor can use Context to see which Future can be polled to make new progress

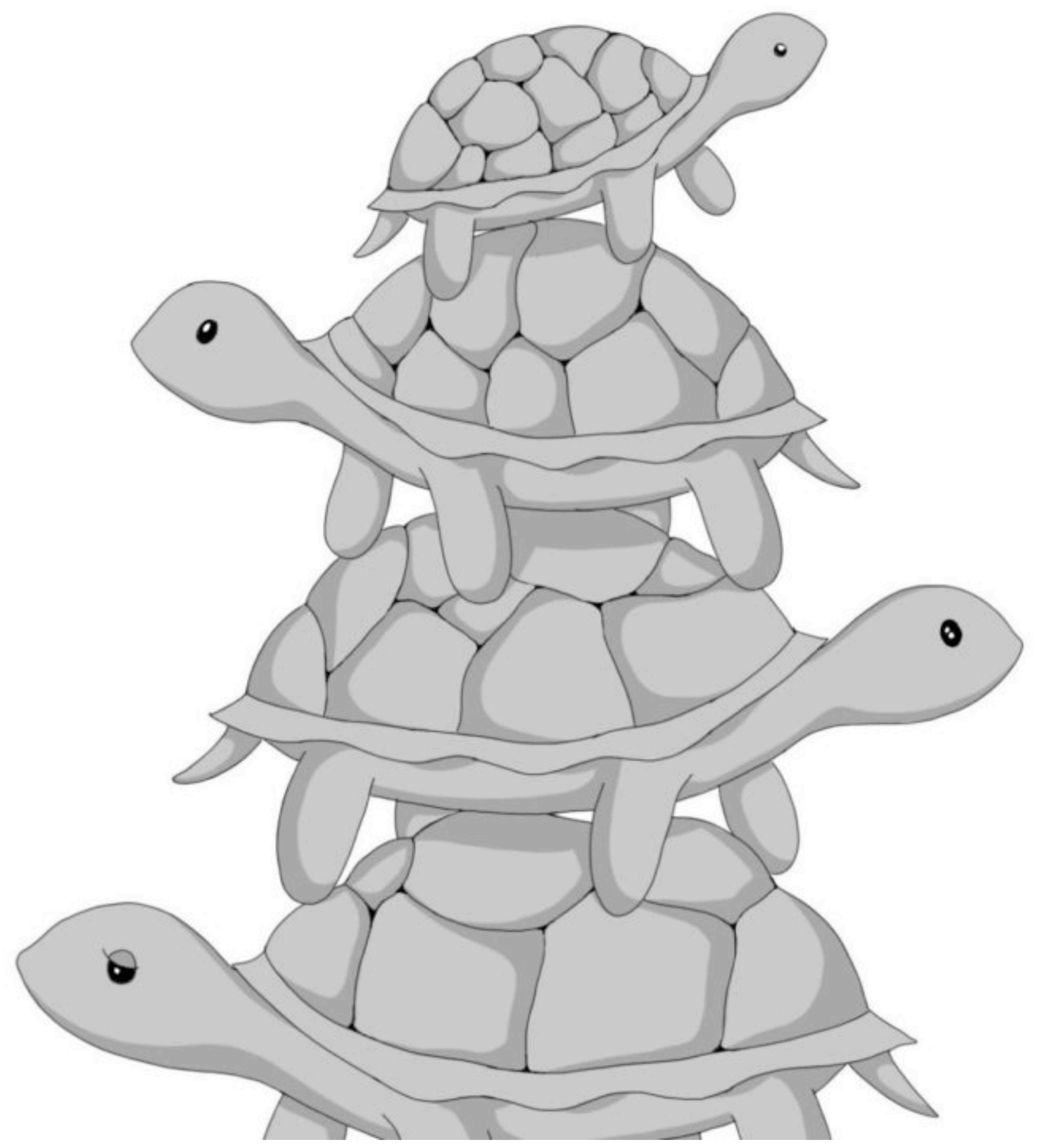


Implementing futures: Futures all the way down

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

```
let future = PlaceOnStoveFuture::new(meat)
    .then(|meat| CookOneSideFuture::new(meat))
    .then(|meat| FlipFuture::new(meat))
    .then(|meat| CookOneSideFuture::new(meat));
```

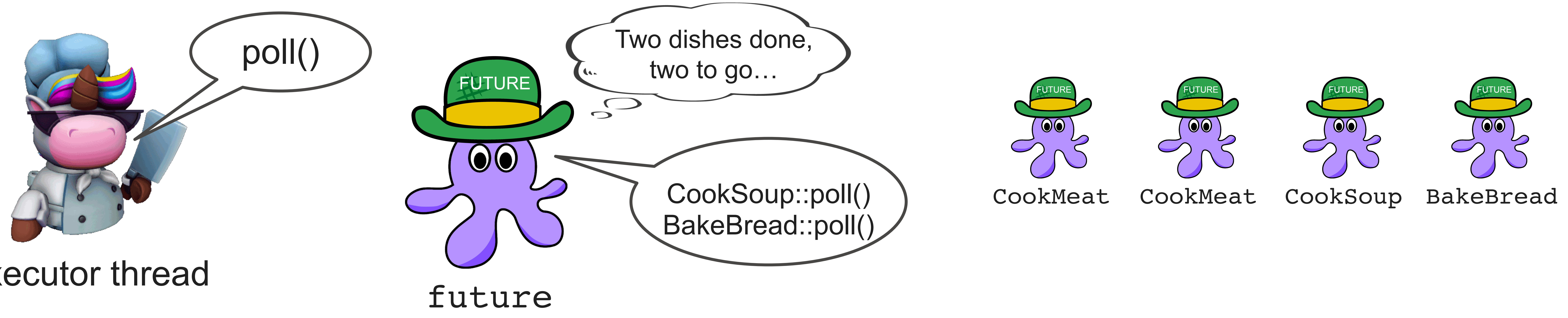




Parallelism with futures

- We can submit multiple futures to the executor to run concurrently
- You can even write futures that depend on multiple concurrently-executing futures!

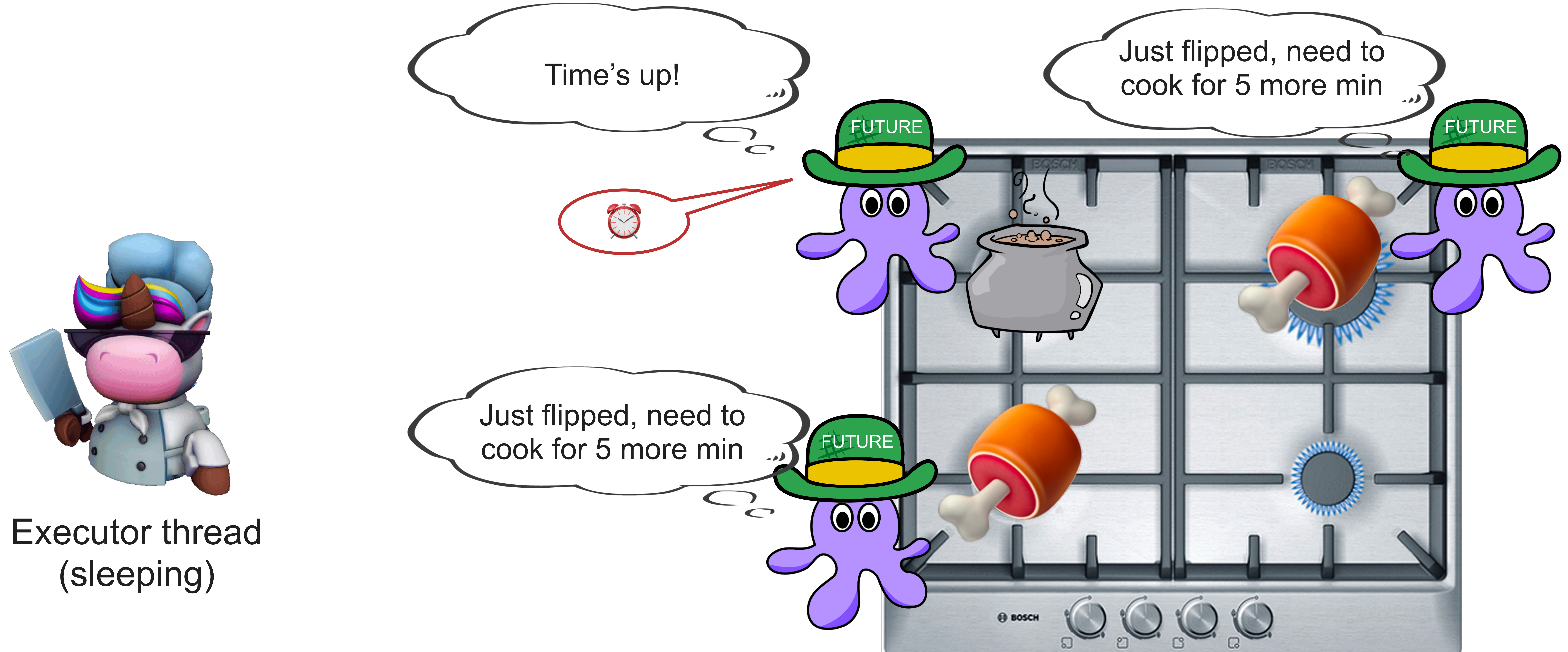
```
let cook_meal_future = futures::future::join_all(vec![
    CookMeatFuture::new(),
    CookMeatFuture::new(),
    CookSoupFuture::new(),
    BakeBreadFuture::new()
]);
```



- You can write code where the entire program is one huge future at the top level!

Parallelism with futures

- Because each future has self-contained state, we avoid the messy state management issues commonly associated with event-driven programming


Time's up!
Just flipped, need to cook for 5 more min
Just flipped, need to cook for 5 more min
FUTURE
FUTURE
FUTURE
Executor thread (sleeping)

Roadmap

- 🥳 Threads are great!
- 😥 But we can’t have too many of them, and context switches are expensive
- 🤔 Event driven programming is nice in theory, but managing state seems hard
- ⭐ 😯 Futures help us encapsulate state for each in-progress operation, making event-driven programming cleaner and more practical!

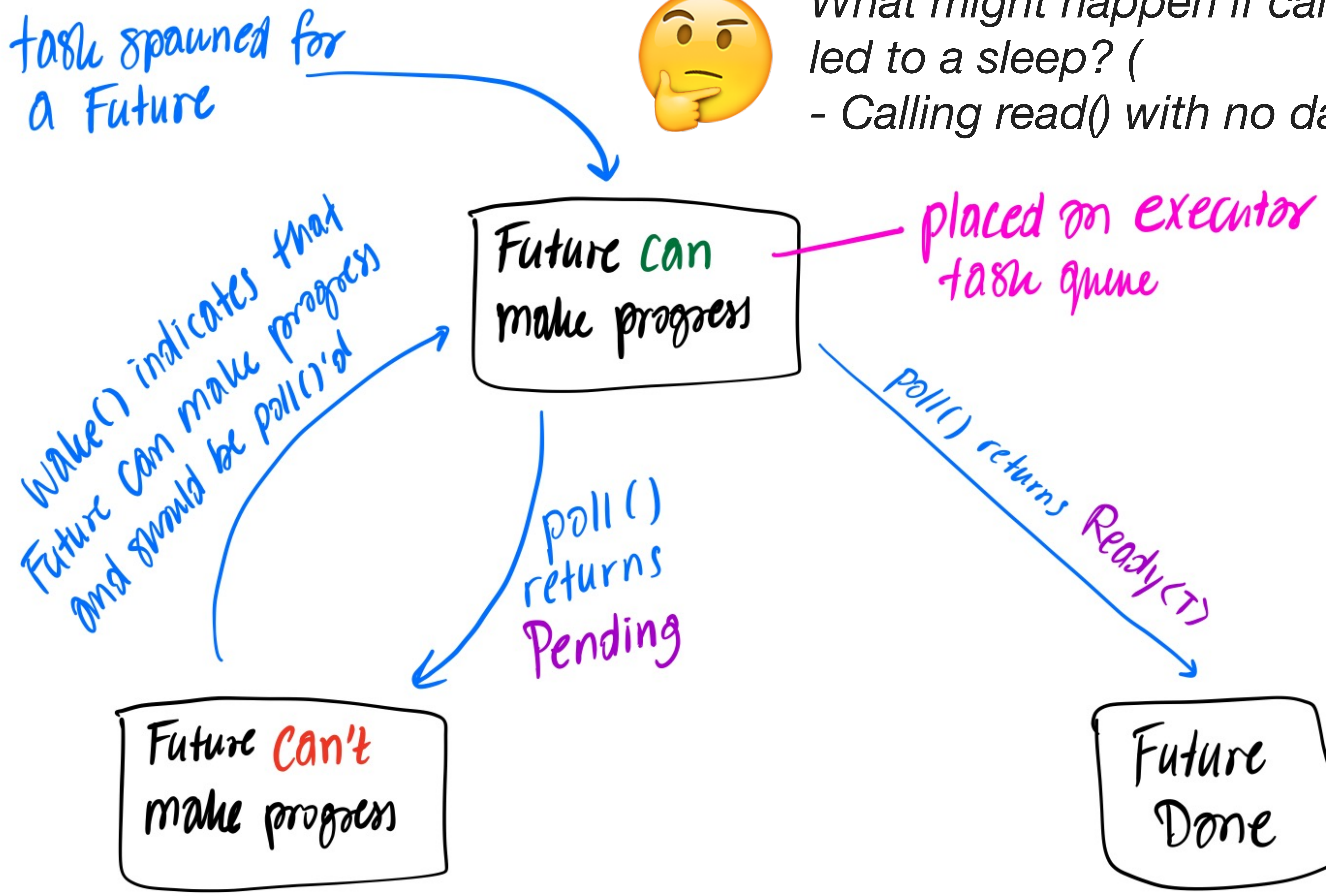
Questions?

How executors work

- An executor loops over futures that can currently make progress, calling 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()
    - Once awakened, the executor goes through those futures, poll()ing them
- A popular executor in the Rust ecosystem is Tokio and it's what you'll be using in Project 2!
- If you have multiple cores on your machine, you can actually execute futures truly in parallel!
    - This means that if you have multiple futures running concurrently, you need to protect shared data using synchronization primitives (although the ownership model kind of already forces you to do this anyways)

What is an executor really doing?

What might happen if calling Poll() on a future led to a sleep? (
- Calling read() with no data available?


task spawned for a Future
Future can make progress
placed on executor task queue
poll() returns Pending
Future can't make progress
wake() indicates that Future can make progress and should be poll()'d
poll() returns Ready(T)
Future Done

Futures cannot Block!

- If code within a future causes the thread to sleep, the executor running that code is going to sleep!
- Then it cannot continue to other futures! The joy of the system goes down the drain!
- 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.

Roadmap

🥳 Threads are great!

😥 But we can’t have too many of them, and context switches are expensive

🤔 Event driven programming is nice in theory, but managing state seems hard

😯 Futures help us encapsulate state for each in-progress operation, making event-driven programming cleaner and more practical!

⭐ 🥳 Thursday: new syntax for making programming with futures even easier

Additional Resources/References

- A great talk about how Rust arrived on the design for futures
- Another great talk about futures
- Phil Levis' CS110 Lecture on Events, Threads, and Async I/O
- The Rust Docs on Futures
- An article on futures
- John Ousterhout on why threads are a bad idea
- A great (and very accessible) Medium article explaining epoll (also has great illustrations!)
- A CS242 Assignment on Implementing Futures
- Note: the syntax for futures has changed over time so some of these articles may use outdated syntax — for the most up-to-date syntax, check out the docs.