FAKULTAS
ILMU
KOMPUTER

Module 10:
Asynchronous Programming
Advanced Programming

Compiled by: Ade Azurat

Editor: Teaching Team (Lecturer and Teaching Assistant)

Advanced Programming

Semester Genap 2024

Fasilkom UI

Author: *Ade Azurat*
Email: ade@cs.ui.ac.id

Table of Contents

Learning Objectives

1. Mahasiswa bisa menjelaskan bagaimana asynchronous programming bekerja dan apa bedanya dengan synchronous
2. Mahasiswa memahami teknologi terkait asynchronous seperti language support, websocket dll
3. Mahasiswa dapat membuat program asynchronous dan menerapkan pada project yang sesuai.

References

1. Asynchronous Programming in Rust, https://rust-lang.github.io/async-book/
2. Comprehensive Rust, Section: Broadcast Chat App, https://google.github.io/comprehensive-rust/concurrency/async-exercises/chat-app.html
3. Johnny Tordgeman, Websocket in Rust and Yew, https://github.com/jtordgeman/YewChat
4. Previous slide of Advanced Programming by Hafiyyan

What is Asynchronous Programming?

- **Asynchronous programming** is a means of parallel (concurrent) programming in which a unit of work runs separately from the main application thread and notifies the calling thread of its completion, failure or progress. (Eric Vogel : Visual Studio Magazine).
- **Asynchronous programming** is centered around the idea of an asynchronous operation:some operation that is started that will completes some time later. (S. Cleary)

What is Asynchronous Programming?

- **Asynchronous programming** is a *concurrent programming model* supported by an increasing number of programming languages.
  It lets you run a large number of concurrent tasks on a small number of OS threads, while preserving much of the look and feel of ordinary synchronous programming, through the async/await syntax. (https://rust-lang.github.io/async-book).
- **Asynchronous programming = Async**

The concept of Asynchronous Programming can be applied to any Programming Language that support it. It is a concept of programming, when a programmer also consider that the call of a function or execution of some statements can be done asynchronously.

Why Asynchronous Programming?

- In several cases, we need to start a task immediately in the middle of another task e.g.
  - Handling user event while loading(viewing) data in the web pages
- With existing resource (e.g. threads) synchronous programming causes too long wait for a task to be executed
- More and more requests come for services

There are several reasons why we need to understand about asynchronous programming. Some are as mentioned above. As a programmer, and an advanced programmer, we need to understand this concept. In term of programming in the small, such as implementing a function, most of the time, you may not need it. But once your system become more complicated, it is possible that you may need it. In web programming it even becoming more standard to have asynchronous call. However, in the web programming this asynchronous call has been encapsulated in the library of the framework. So, sometime programmer just needs to know how to use it. As Fasilkom's graduate, you should know more than just use a library. You need to understand the consequences of using a library. It is a responsible and professional programmer should be. Why it important? Because it may affect the use of CPU, Memory and network allocation. In small scale of course exercises, it probably would not matter. But in production scale, it definitely needs to be considered.

Benefits of Asynchronous Programming

- Improve application performance
- Improve application responsiveness
- Improve application throughput
- Improve resource utilization
- Improve code organization

Asynchronous Programming did not come uselessly. Yes, there are many benefits. If you understand Asynchronous Programming correctly and how it works, you will get the benefit as mentioned above. But we need to understand, when to use it compared to use standard synchronous programming as we have learned in the previous classes. Let's review synchronous programming.

Synchronous vs Asynchronous Programming?

- **Synchronous Programming** : A thread is assigned to one task and starts working on it. Once the task completes then it is available for the next task. In this model, it cannot leave the executing task in mid to take up another task
- **Asynchronous Programming** : Once start executing a task a thread can hold it in mid, save the current state and start executing another task

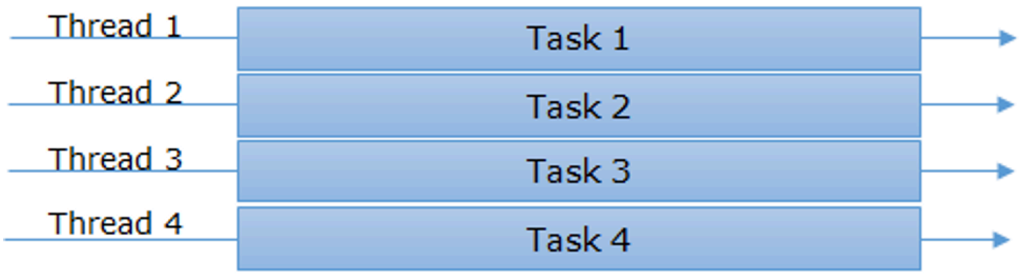
Synchronous Programming

Single-Threaded : If we have couple of tasks to be worked on and the current system provides just a single thread, then tasks are assigned to the thread one by one

It illustrates how several tasks a running in synchronous programming. It will run one by one. Consume input and then produce output, and then continues with the next one. While the computer is currently working on one task, it only we do that task only, and let the other tasks to wait until the previous one is finished. It is same things if we have multiple thread, each thread will handle on task at a time.

Synchronous Programming

Multi-Threaded : Each thread takes up one task and completes that. If we have more number of tasks than the number of available threads, then whichever thread gets free, it takes up another task



Async vs Other Concurrent Model

- **OS threads** don't require any changes to the programming model, which makes it very easy to express concurrency. However, synchronizing between threads can be difficult, and the performance overhead is large. Thread pools can mitigate some of these costs, but not enough to support massive IO-bound workloads.
- **Event-driven programming**, in conjunction with *callbacks*, can be very performant, but tends to result in a verbose, "non-linear" control flow. Data flow and error propagation is often hard to follow.

Async vs Other Concurrent Model (cont.)

- **Coroutines**, like threads, don't require changes to the programming model, which makes them easy to use. Like async, they can also support a large number of tasks. However, they abstract away low-level details that are important for systems programming and custom runtime implementors.
- **The actor model** divides all concurrent computation into units called actors, which communicate through fallible message passing, much like in distributed systems. The actor model can be efficiently implemented, but it leaves many practical issues unanswered, such as flow control and retry logic.

Concurrent programming is less mature and "standardized" than regular, sequential programming. As a result, we express concurrency differently depending on which concurrent programming model the language is supporting. A brief overview of the most popular concurrency models can help you understand how asynchronous programming fits within the broader field of concurrent programming.

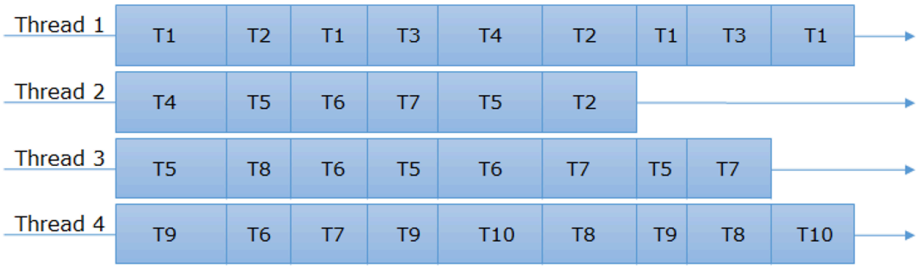
Asynchronous Programming

Single-Threaded : Single thread is responsible to complete all the tasks and tasks are interleaved to each other

In *Asynchronous Programming*, When a task is executing, and probably still waiting for internet access or other resources, the CPU is idle. While CPU is idle, asynchronous programming allows other tasks to be executed. For example, in the illustration, a task T1, is executing. It is not finished yet. Let's say it accessing a disk and still waiting for the disk, since disk processing is slower than CPU. While the CPU idle, it run T2, while actually the T1 is not yet finished. The some illustration can also occurs in a multi-threaded environment.

Asynchronous Programming

Multi-Threaded : Same task say T4, T5, T6.. are handled by multiple thread. T4 was started first in Thread 1 and completed by Thread 2. Similarly, T6 is completed by Thread 2, Thread 3 and Thread 4. It shows the maximum utilization of the threads.



What is the correlation with concurrency?

Concurrency means processing multiple requests (tasks) at a time. Based on the previous slides, we can implement concurrency by using multi-threaded synchronous programming and asynchronous programming (both single-threaded and multi-threaded)

Most of modern discussion on asynchronous are relate it with concurrency. While actually the concept of Asynchronous programming has been there long before the concurrency become more popular and standardized. The execution of asynchronous programming is done by applying concurrency in computing. But the concurrency model has their own discussion. So it is correct to relate it with concurrency especially as one approach to achieve concurrency.

In Summary

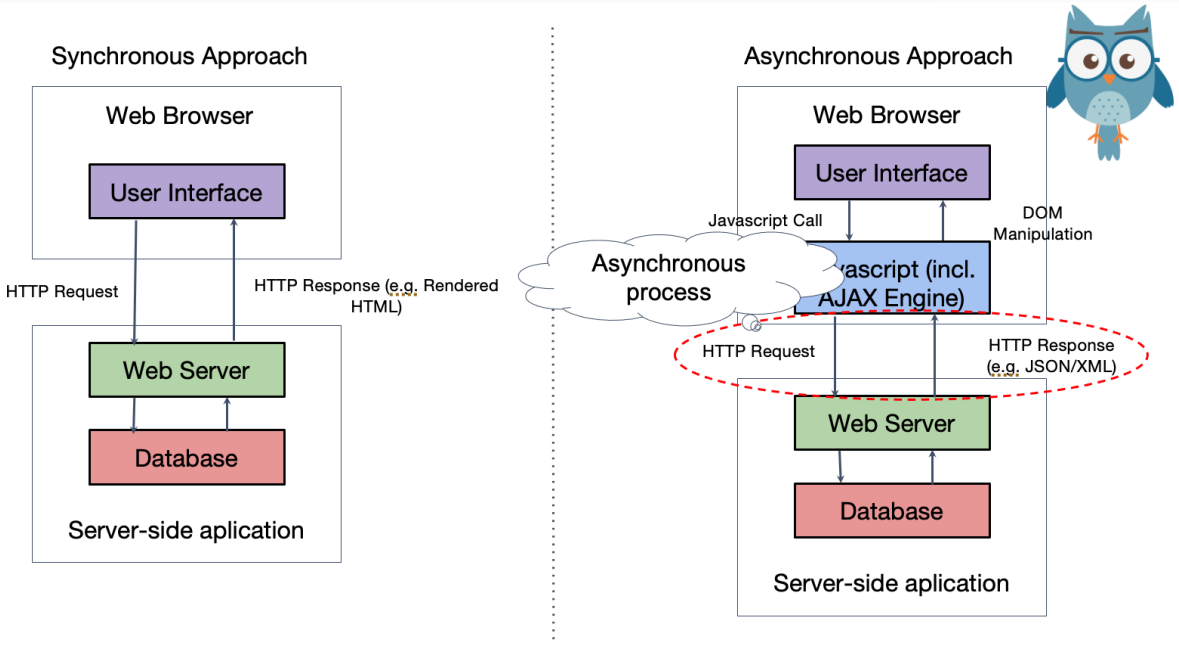
asynchronous programming allows highly performant implementations that are suitable for low-level languages like Rust, while providing most of the ergonomic benefits of threads and coroutines.

More About Asynchronous Programming

AJAX

- **Not a programming language.**
- While XML become the part of its name, nowadays AJAX mostly use JSON for exchanging data.

The popularity of Asynchronous Programming is supported by AJAX. AJAX is started since 1999. It is a concept of programming techniques. A programming technique that make web application more responsive by applying asynchronous call.



Promise

- A proxy object representing that will hold a value after the process completed
- Promise can only have these 3 states:
  - Pending: Initial state of Promise that does not categorized as Fulfilled or Rejected
  - Fulfilled: The process completed successfully.
  - Rejected: The process failed to be executed.
- We can assign asynchronous action, given a fulfilled or rejected state

25

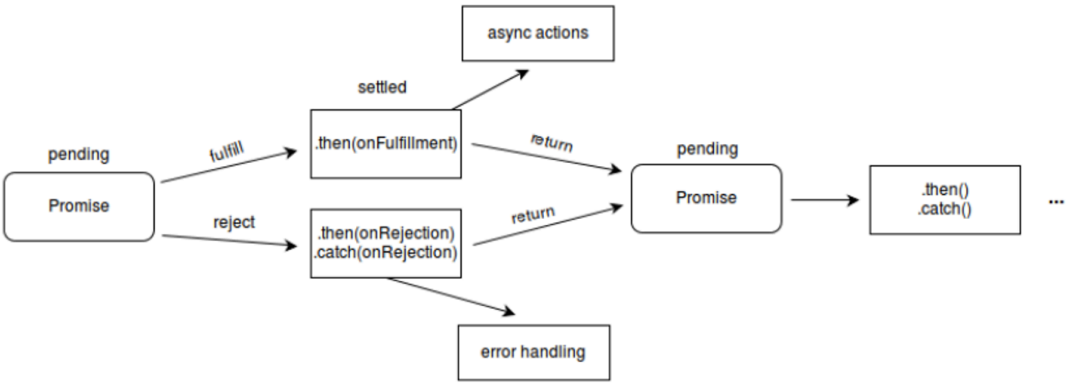
In web programming of javascript, we can achieve asynchronous programming using what we call “Promise” (which similar to Future in Java and Rust).

```
const promise = new Promise((resolve, reject) => {
  const request = new XMLHttpRequest();

  request.open("GET", "https://api.icndb.com/jokes/random");
  request.onload = () => {
    if (request.status === 200) {
      resolve(request.response);
    } else {
      reject(Error(request.statusText));
    }
  };

  request.onerror = () => {
    reject(Error("Error fetching data."));
  };

  request.send();
});

console.log("Asynchronous request made.");
```

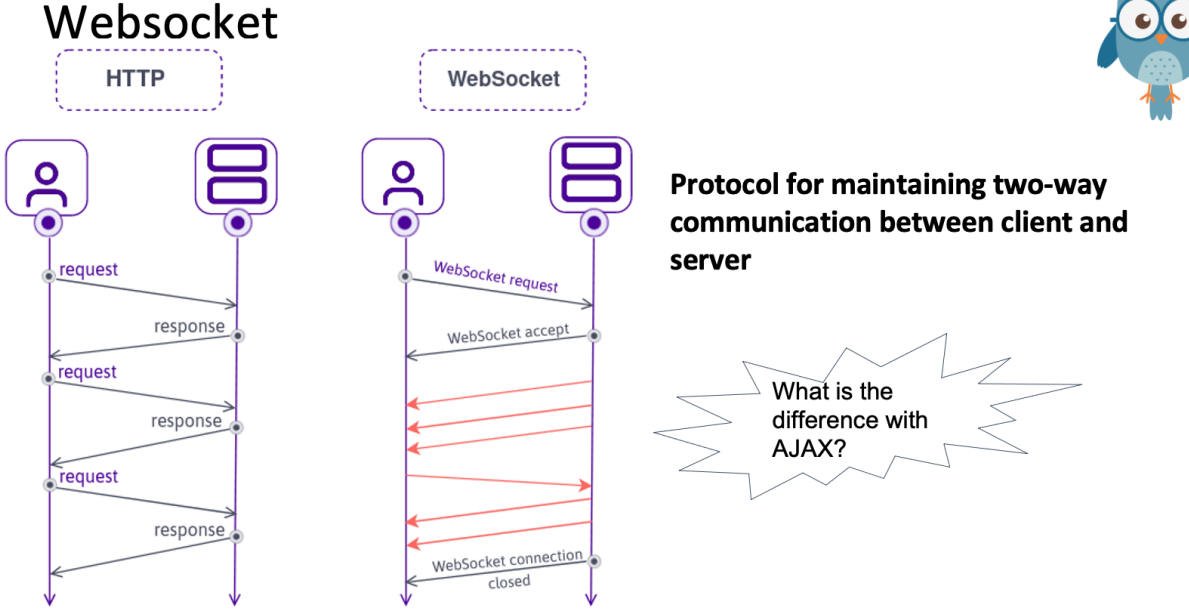
```
promise.then(
  data => {
    console.log("Got data! Promise fulfilled.");
    document.body.textContent = JSON.parse(data).value.joke;
  },
  error => {
    console.log("Promise rejected.");
    console.log(error.message);
  }
);
```

Example of Promise

27



Flow of Process in Promise



With the coming of websocket being standardized in 2011, the asnchronous programming become much more popular. It is because the websocket protocol allowing a more efficient network communication. Ajax that previously simulated, now can be achieved by the support of lower level communication protocol, which make it more efficient.

Asynchronous Programming in Rust

Source: https://rust-lang.github.io/async-book/

Async in Rust vs other languages

- **Futures are inert** in Rust and make progress only when polled. Dropping a future stops it from making further progress.
- **Async is zero-cost** in Rust, which means that you only pay for what you use. Specifically, you can use async without heap allocations and dynamic dispatch, which is great for performance! This also lets you use async in constrained environments, such as embedded systems.
- **No built-in runtime** is provided by Rust. Instead, runtimes are provided by community maintained crates.
- **Both single- and multithreaded** runtimes are available in Rust, which have different strengths and weaknesses.

Although asynchronous programming is supported in many languages, some details vary across implementations. Rust's implementation of async differs from most languages in a few ways.

Async vs threads in Rust

- The primary alternative to async in Rust is using OS threads.
- Migrating from threads to async or vice versa typically requires major refactoring work.

The primary alternative to async in Rust is using OS threads, either directly through std::thread or indirectly through a thread pool. Migrating from threads to async or vice versa typically requires major refactoring work, both in terms of implementation and (if you are building a library) any exposed public interfaces. As such, picking the model that suits your needs early can save a lot of development time.

OS Threads in Rust

- **OS threads** are suitable for a small number of tasks, since threads come with CPU and memory overhead.
- Spawning and switching between threads is quite expensive as even idle threads consume system resources.
- **Threads let you reuse existing synchronous code** without significant code changes—no particular programming model is required.

OS threads are suitable for a small number of tasks, since threads come with CPU and memory overhead. Spawning and switching between threads is quite expensive as even idle threads consume system resources. A thread pool library can help mitigate some of these costs, but not all. However, threads let you reuse existing synchronous code without significant code changes—no particular programming model is required. In some operating systems, you can also change the priority of a thread, which is useful for drivers and other latency sensitive applications.

Async in Rust

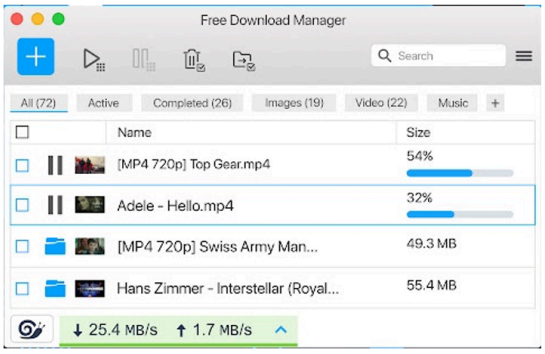
- **Async** provides significantly reduced CPU and memory overhead.
- an async runtime uses a small amount of (expensive) threads to handle a large amount of (cheap) tasks.
- async Rust results in larger binary blobs

Async provides significantly reduced CPU and memory overhead, especially for workloads with a large amount of IO-bound tasks, such as servers and databases. All else equal, you can have orders of magnitude more tasks than OS threads, because an async runtime uses a small amount of (expensive) threads to handle a large amount of (cheap) tasks. However, async Rust results in larger binary blobs due to the state machines generated from async functions and since each executable bundles an async runtime.

asynchronous programming is not *better* than threads,
but different.
If you don't need async for performance reasons,
threads can often be the simpler alternative.

Example: Concurrent Downloading

- Downloading may take quite sometimes
- In some cases, we may need to download from several URLs
- We need to apply concurrency approach to do multiple downloading, so that we do not need to wait for each download to finish before doing another download
- We can do it using thread or async



FACULTY OF COMPUTER SCIENCE

Concurrent Downloading: Threads

```
fn get_two_sites() {
    // Spawn two threads to do work.
    let thread_one = thread::spawn(|| download("https://www.foo.com"));
    let thread_two = thread::spawn(|| download("https://www.bar.com"));

    // Wait for both threads to complete.
    thread_one.join();
    thread_two.join();
}
```

FACULTY OF COMPUTER SCIENCE

Downloading a web page is a small task; creating a thread for such a small amount of work is quite wasteful.

Concurrent Downloading: Async

```
async fn get_two_sites_async() {
    // Create two different "futures" which, when run to completion,
    // will asynchronously download the webpages.
    let future_one = download_async("https://www.foo.com");
    let future_two = download_async("https://www.bar.com");

    // Run both futures to completion at the same time.
    join!(future_one, future_two);
}
```

FACULTY OF COMPUTER SCIENCE

For a larger application, it can easily become a bottleneck. In async Rust, we can run these tasks concurrently without extra threads. Here, no extra threads are created. Additionally, all function calls are statically dispatched, and there are no heap allocations! However, we need to write the code to be asynchronous in the first place, which this we will help you achieve.

Custom Concurrency models in Rust

Rust doesn't force you to choose between threads and async.

You can use both models within the same application, which can be useful when you have mixed threaded and async dependencies.

In fact, you can even use a different concurrency model altogether, such as event-driven programming, as long as you find a library that implements it.

FACULTY OF COMPUTER SCIENCE

The Future Traits

- The **Future** trait is at the center of asynchronous programming in Rust.
- A Future is an asynchronous computation that can produce a value (although that value may be empty, e.g. ()).
- But what is a trait?

FACULTY OF COMPUTER SCIENCE

Review of Trait

- By definition: trait is a distinguishing quality or characteristic, typically one belonging to a person. (Oxford Dictionary)
- Example: "His sense of humor is one of his better traits. Arrogance is a very unattractive personality/character trait."

- In Computer Science:
- Trait : Composable units of Behaviour (Schärli etal, ECOOP 2003)
- It is a better way to represents ***reuse***, compare to: *inheritance*, *mixin*.

FACULTY OF COMPUTER SCIENCE

Despite the undisputed prominence of inheritance as the fundamental reuse mechanism in object-oriented programming languages, the main variants — single inheritance, multiple inheritance, and mixin inheritance — all suffer from conceptual and practical problems. A trait is essentially a group of pure methods that serves as a building block for classes and is a primitive

unit of code reuse. In this model, classes are *composed* from a set of traits by specifying *glue code* that connects the traits together and accesses the necessary state. (Abstract: https://www.cs.cmu.edu/~aldrich/courses/819/Scha03aTraits.pdf)

The Future Traits

- A *simplified* version of the future trait might look something like this:

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

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

FACULTY OF COMPUTER SCIENCE

Futures can be advanced by calling the poll function, which will drive the future as far towards completion as possible. If the future completes, it returns Poll::Ready(result). If the future is not able to complete yet, it returns Poll::Pending and arranges for the wake() function to be called when the Future is ready to make more progress. When wake() is called, the executor driving the Future will call poll again so that the Future can make more progress.

Without wake(), the executor would have no way of knowing when a particular future could make progress, and would have to be constantly polling every future. With wake(), the executor knows exactly which futures are ready to be polled.

```
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() {
            // The socket has data -- read it into a buffer and return it.
            Poll::Ready(self.socket.read_buf())
        } else {
            // The socket does not yet have data.
            // Arrange for `wake` to be called once data is available.
            // When data becomes available, `wake` will be called, and the
            // user of this `Future` will know to call `poll` again and
            // receive data.
            self.socket.set_readable_callback(wake);
            Poll::Pending
        }
    }
}
```


A simple SocketRead future might look something like this. Consider the case where we want to read from a socket that may or may not have data available already. If there is data, we can read it in and return Poll::Ready(data), but if no data is ready, our future is blocked and can no longer make progress. When no data is available, we must register wake to be called when data becomes ready on the socket, which will tell the executor that our future is ready to make progress.

Task Wakeups with waker

- It's common that *futures* aren't able to complete the first time they are polled. When this happens, the future needs to ensure that it is polled again once it is ready to make more progress.
- This is done with the *Waker* type.
- 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.
- Waker also implements ***clone()*** so that it can be copied around and stored.

Example: Build a Timer

- we'll just spin up a new thread when the timer is created,
- sleep for the required time, and then
- signal the timer future when the time window has elapsed.

FACULTY OF COMPUTER SCIENCE

Build an executor

- Rust's Futures are lazy: they won't do anything unless actively driven to completion.
- One way to drive a future to completion is to .await it inside an async function, but that just pushes the problem one level up: who will run the futures returned from the top-level async functions?
- The answer is that we need a ***Future executor***.
- Future executors take a set of top-level Futures and run them to completion by calling poll when ever the Future can make progress.

FACULTY OF COMPUTER SCIENCE

Rust's Futures are lazy: they won't do anything unless actively driven to completion. One way to drive a future to completion is to .await it inside an async function, but that just pushes the problem one level up: who will run the futures returned from the top-level async functions? The answer is that we need a Future executor.

Future executors take a set of top-level Futures and run them to completion by calling pollwhenever the Future can make progress. Typically, an executor will poll a future once to start off. When Futures indicate that they are ready to make progress by calling wake(), they are placed back onto a queue and poll is called again, repeating until the Future has completed.

In this section, we'll write our own simple executor capable of running a large number of top-level futures to completion concurrently.

For this example, we depend on the futures crate for the ArcWake trait, which provides an easy way to construct a Waker.

```
/// Task executor that receives tasks off of a channel and runs them.
struct Executor {
    ready_queue: Receiver<Arc<Task>>,
}

/// `Spawner` spawns new futures onto the task channel.
#[derive(Clone)]
struct Spawner {
    task_sender: SyncSender<Arc<Task>>,
}

/// A future that can reschedule itself to be polled by an `Executor`.
struct Task {
    /// In-progress future that should be pushed to completion.
    ///
    /// The `Mutex` is not necessary for correctness, since we only have
    /// one thread executing tasks at once. However, Rust isn't smart
    /// enough to know that `future` is only mutated from one thread,
    /// so we need to use the `Mutex` to prove thread-safety. A production
    /// executor would not need this, and could use `UnsafeCell` instead.
    future: Mutex<Option<BoxFuture<'static, ()>>>,

    /// Handle to place the task itself back onto the task queue.
    task_sender: SyncSender<Arc<Task>>,
}
```

FACULTY OF COMPUTER SCIENCE

Our executor will work by sending tasks to run over a channel. The executor will pull events off of the channel and run them. When a task is ready to do more work (is awoken), it can schedule itself to be polled again by putting itself back onto the channel.

In this design, the executor itself just needs the receiving end of the task channel. The user will get a sending end so that they can spawn new futures. Tasks themselves are just futures that can reschedule themselves, so we'll store them as a future paired with a sender that the task can use to requeue itself.

```
fn new_executor_and_spawner() -> (Executor, Spawner) {
    // Maximum number of tasks to allow queueing in the channel at once.
    // This is just to make `sync_channel` happy, and wouldn't be present in
    // a real executor.
    const MAX_QUEUED_TASKS: usize = 10_000;
    let (task_sender, ready_queue) = sync_channel(MAX_QUEUED_TASKS);
    (Executor { ready_queue }, Spawner { task_sender })
}

impl Spawner {
    fn spawn(&self, future: impl Future<Output = ()> + 'static + Send) {
        let future = future.boxed();
        let task = Arc::new(Task {
            future: Mutex::new(Some(future)),
            task_sender: self.task_sender.clone(),
        });
        self.task_sender.send(task).expect("too many tasks queued");
    }
}
```

FACULTY OF COMPUTER SCIENCE

Let's also add a method to spawner to make it easy to spawn new futures. This method will take a future type, box it, and create a new Arc<Task> with it inside which can be enqueued onto the executor.

```
impl ArcWake for Task {
    fn wake_by_ref(arc_self: &Arc<Self>) {
        // Implement `wake` by sending this task back onto the task channel
        // so that it will be polled again by the executor.
        let cloned = arc_self.clone();
        arc_self
            .task_sender
            .send(cloned)
            .expect("too many tasks queued");
    }
}
```

FACULTY OF COMPUTER SCIENCE

To poll futures, we'll need to create a Waker. As discussed in the task wakeups section, Wakers are responsible for scheduling a task to be polled again once wake is called. Remember

that Wakers tell the executor exactly which task has become ready, allowing them to poll just the futures that are ready to make progress. The easiest way to create a new Waker is by implementing the ArcWaketrait and then using the waker_ref or .into_waker() functions to turn an Arc<impl ArcWake> into a Waker. Let's implement ArcWake for our tasks to allow them to be turned into Wakers and awoken:

```
impl Executor {
    fn run(&self) {
        while let Ok(task) = self.ready_queue.recv() {
            // Take the future, and if it has not yet completed (is still Some),
            // poll it in an attempt to complete it.
            let mut future_slot = task.future.lock().unwrap();
            if let Some(mut future) = future_slot.take() {
                // Create a `LocalWaker` from the task itself
                let waker = waker_ref(&task);
                let context = &mut Context::from_waker(&waker);
                // `BoxFuture<T>` is a type alias for
                // `Pin<Box<dyn Future<Output = T> + Send + 'static>>`.
                // We can get a `Pin<&mut dyn Future + Send + 'static>`
                // from it by calling the `Pin::as_mut` method.
                if future.as_mut().poll(context).is_pending() {
                    // We're not done processing the future, so put it
                    // back in its task to be run again in the future.
                    *future_slot = Some(future);
                }
            }
        }
    }
}
```


When a Waker is created from an Arc<Task>, calling wake() on it will cause a copy of the Arc to be sent onto the task channel. Our executor then needs to pick up the task and poll it. Let's implement main method for that:

```
fn main() {
    let (executor, spawner) = new_executor_and_spawner();

    // Spawn a task to print before and after waiting on a timer.
    spawner.spawn(async {
        println!("howdy!");
        // Wait for our timer future to complete after two seconds.
        TimerFuture::new(Duration::new(2, 0)).await;
        println!("done!");
    });

    // Drop the spawner so that our executor knows it is finished and won't
    // receive more incoming tasks to run.
    drop(spawner);

    // Run the executor until the task queue is empty.
    // This will print "howdy!", pause, and then print "done!".
    executor.run();
}
```

FACULTY OF COMPUTER SCIENCE

Async / .await

async/.await

- async/.await are special pieces of Rust syntax that make it possible to yield control of the current thread rather than blocking,
- allowing other code to make progress while waiting on an operation to complete.
- There are two main ways to use async:
  - async fn
  - async blocks. async { ... }
- Each returns a value that implements the Future trait.

FACULTY OF COMPUTER SCIENCE

```
// `foo()` returns a type that implements `Future<Output = u8>`.
// `foo().await` will result in a value of type `u8`.
async fn foo() -> u8 { 5 }

fn bar() -> impl Future<Output = u8> {
    // This `async` block results in a type that implements
    // `Future<Output = u8>`.
    async {
        let x: u8 = foo().await;
        x + 5
    }
}
```

FACULTY OF COMPUTER SCIENCE

As we saw in the first chapter, async bodies and other futures are lazy: they do nothing until they are run. The most common way to run a Future is to .await it. When .await is called on a Future, it will attempt to run it to completion. If the Future is blocked, it will yield control of the current thread. When more progress can be made, the Future will be picked up by the executor and will resume running, allowing the .await to resolve.

async lifetime

- Unlike traditional functions, async fn s which take references or other non-'static arguments return a Future which is bounded by the lifetime of the arguments

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

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

FACULTY OF COMPUTER SCIENCE

This means that the future returned from an async fn must be .awaited while its non-'staticarguments are still valid. In the common case of .awaiting the future immediately after calling the function (as in foo(&x).await) this is not an issue. However, if storing the future or sending it over to another task or thread, this may be an issue.

async lifetime

```
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
    }
}
```

UNIVERSITAS INDONESIA
FACULTY OF COMPUTER SCIENCE

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:

By moving the argument into the async block, we extend its lifetime to match that of the Future returned from the call to good.

When poll is first called, it will poll fut_one. If fut_one can't complete, AsyncFuture::poll will return. Future calls to poll will pick up where the previous one left off. This process continues until the future is able to successfully complete.

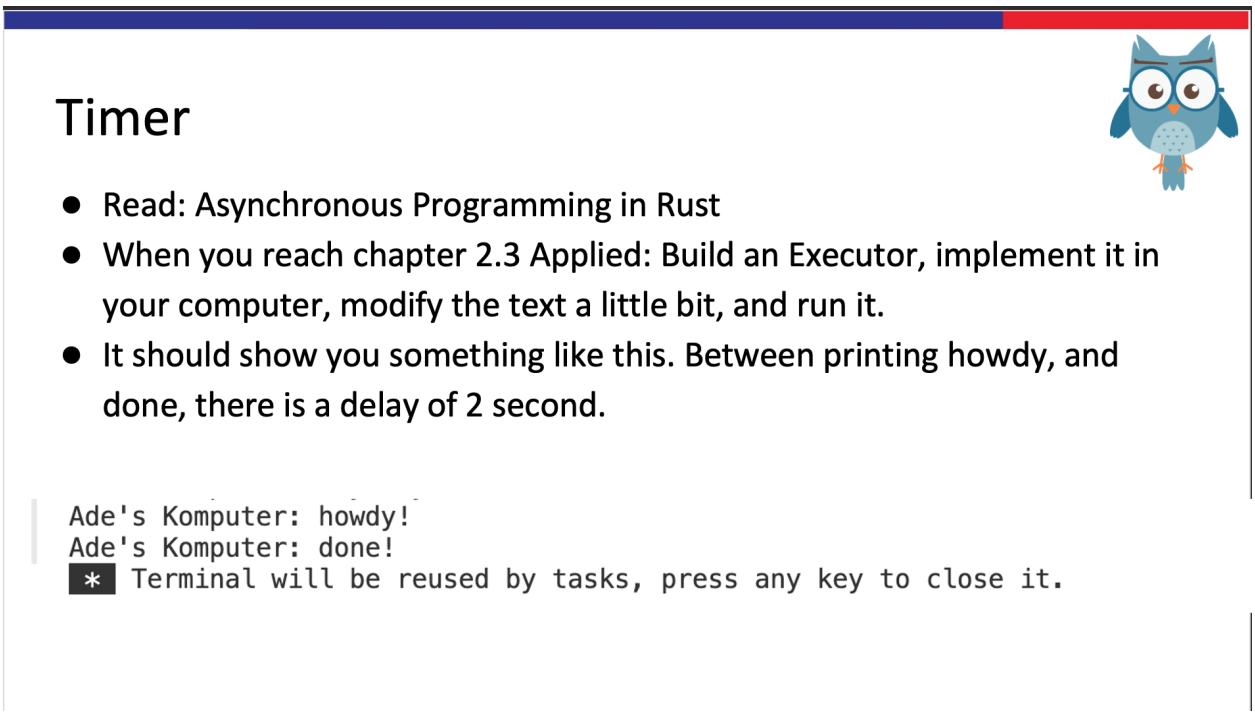
Tutorial

Tutorial 1: Timer

This tutorial is based on this part of the book: https://rust-lang.github.io/async-book/02_execution/04_executor.html

1.1. Initial Code

Prepare the code and run it so that it may have the result such as the following. You may need to change the text “Ade’s Komputer” with your own signature.



Commit and push your works. Put this text “Experiment 1.1: Original timer from the book” as the message commit.

Let's try to modify it:

- Try to add another print just after the spawn. As shown below:

```
97  fn main() {
98      let (executor: Executor, spawner: Spawner) = new_executor_and_spawner();
99
100     // Spawn a task to print before and after waiting on a timer.
101     spawner.spawn(future: async {
102         println!("Ade's Komputer: howdy!");
103         // Wait for our timer future to complete after two seconds.
104         TimerFuture::new(Duration::new(secs: 2, nanos: 0)).await;
105         println!("Ade's Komputer: done!");
106     });
107
108     println!("Ade's Komputer: hey hey");
109     // Drop the spawner so that our executor knows it is finished and won't
110     // receive more incoming tasks to run.
111     drop(spawner);
```

1.2. Understanding how it works.

Add such a sentence (you need to modify the text) right after the spawner.sparwn (...);

Take a look at what happened. Capture the result of your execution. Put it in your **Readme.md**, with some explanation why the result is as such.

Commit and push the modification and capture of your running program together with your explanation. Put a message "Experiment 1.2: Understanding how it works."

1.3. Multiple Spawn and removing drop

What is the effect of spawning?

What is the spawner for, what is the executor for, what is the drop for?

What is the correlation of all of that?

The book and the class has briefly inform you, but we need to see it ourselves and try to investigate it.

Let try to do the following:

Multiple Spawn

- Try to replicate the spawn, and run it:

```
100     // Spawn a task to print before and after waiting on a timer.
101     spawner.spawn(future: async {
102         println!("Ade's Komputer: howdy!");
103         // Wait for our timer future to complete after two seconds.
104         TimerFuture::new(Duration::new(secs: 2, nanos: 0)).await;
105         println!("Ade's Komputer: done!");
106     });
107     spawner.spawn(future: async {
108         println!("Ade's Komputer: howdy2!");
109         // Wait for our timer future to complete after two seconds.
110         TimerFuture::new(Duration::new(secs: 2, nanos: 0)).await;
111         println!("Ade's Komputer: done2!");
112     });
113     spawner.spawn(future: async {
114         println!("Ade's Komputer: howdy3!");
115         // Wait for our timer future to complete after two seconds.
116         TimerFuture::new(Duration::new(secs: 2, nanos: 0)).await;
117         println!("Ade's Komputer: done3!");
118     });
```

Removing statement: drop(spawner);

```
97  fn main() {
119     println!("Ade's Komputer: hey hey");
120     // Drop the spawner so that our executor knows it is finished and won't
121     // receive more incoming tasks to run.
122     // drop(spawner);
123
124     // Run the executor until the task queue is empty.
125     // This will print "howdy!", pause, and then print "done!".
126     executor.run();
127
128 } fn main
129
```

Try to remove and put it again. Pay attention to your console. Capture your screen as many as needed (but not more than three) and provide explanation why it is like that. Put in in **Readme.md**

Commit and push, with message “Experiment 1.3: Multiple Spawn and removing drop”.

Submit your repository and a llst of commit links to scele!

Tutorial 2: Broadcast Chat

Now we would like to understand how asynchronous is more properly being used in an application. Not anything is suitable to be made asynchronous. One case, where asynchronous programming is suitable with the support of websocket, is the chat application. Here is we would like to see how it works.

2.1. Original code of broadcast chat.

The original code can be found in:

https://google.github.io/comprehensive-rust/concurrency/async-exercises/chat-app.html

You can clone or fork or combine with your previous repository or create another new repository.

Try to run one server, and three clients. Try to type something in each client. Capture your screen. Put it in your **Readme.md**. Put some explanation. How to run it, and what happens when you type some text in the clients.

Commit and push to your repository, put a message “Experiment 2.1: Original code, and how it run”

2.2. Modifying the websocket port

The application is using websocket connection as can be seen:

Try to modify the **port** to be **8080.** Test it again. Make sure it still runs properly. Remember since it is a connection, it means there are at least two sides. The server side and the client side. Find where you should modify it. Take a look at the other files that need to be modified. Is it also using the same websocket protocol? Where is it defined?

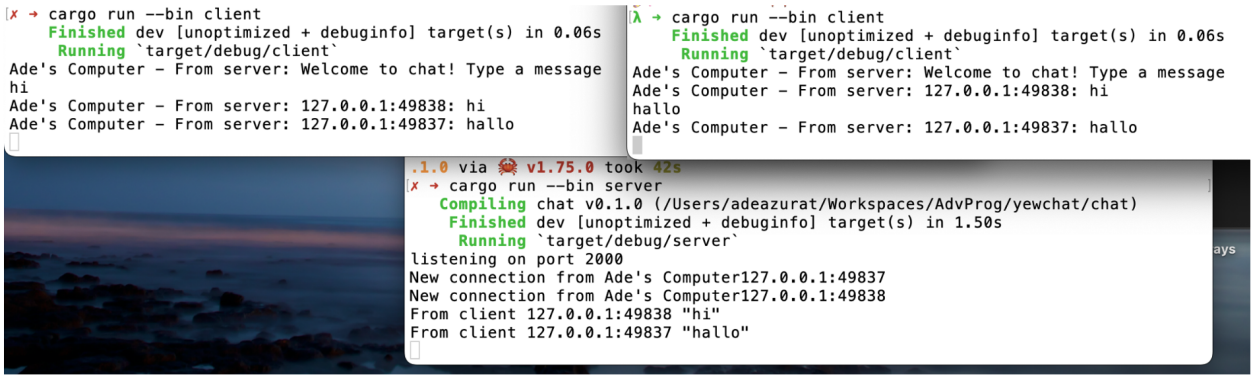
Put your explanation in the Readme.md.

Commit and push. Put a commit message “Experiment 2.2: Modifying port”.

2.3. Small changes. Add some information to client

Try to modify it a little. Let’s try to understand how the message is passed from each other.

Small changes



We try to add information about the sender to each client. Since we don’t have a name yet, we just sent information about the IP and Port of the sender as can be seen above.

Make your own modification. Capture the result, put it in the Readme.md and put some explanation why you change it there.

Commit and push. Put a commit message “Experiment 2.3: Small changes, add IP and Port”.

Tutorial 3: WebChat using yew

Having a chat system in the console is cool. But we live in the graphical-internet world now where people usually use browsers or other systems. Let's see how to build a webchat! There is a good example to start with. You can check the following blog:

https://blog.devgenius.io/lets-build-a-websockets-project-with-rust-and-yew-0-19-60720367399f

3.1. Original code

At this moment we are focusing on the client part. You can find another link to its websocket server. We will discuss the server later. Try to clone both projects. Run it, and play with it on your computer. Capture your screen, put it in your **Readme.md**.

Commit and push. Put a commit message "Experiment 3.1: Original code"

3.2. Add some creativities to the webclient

Let's be creative. Creativity is one of the most critical traits for success in your future career. As the WEF (World Economic Forum) has identified that creatify is the key to compete with AI in the future workforce. https://www.weforum.org/agenda/2020/11/ai-automation-creativity-workforce-skill-fute-of-work/

Try to think for a while, try to spend one or two hours. You can add an additional page, put some picture icons, add some creative sentences and some others. Your creativity is not limited to programming. Do whatever you please. Make it run, capture the screen. Put it in your Readme.md and provide some text to explain what you are doing.

Commit and push. Put a commit message "Experiment 3.2: Be Creative!"

Bonus: Change the websocket server!

If you pay attention to the Tutorial 3, the websocket server is actually built on javascript. We already have a server running on Rust on Tutorial 2. Are you not challenged to modify the server in tutorial 2 so that it can serve the webchat in tutorial 3? As a start, in the Tutorial 3, the communication is strong json format. While in tutorial 2, it is only simple text. But if you go deeper into how the message was sent in Tutorial 3, it is also sended as one text message. So the json format is serialized and deserialized to text messages.

Put your explanation of how you do it. Explain why it is a successful change. Also provide your opinion, which one you prefer, the javascript version or the Rust version.

Commit and push. Put a commit message "Bonus: Rust Websocket server for YewChat!"

Grading Scheme

Scale

All components will be scored on scale 0-4. Grade 4 is the highest grade, equivalent to A. Score 0 is for not submitting.

Components

- 60% - Commits
- 20% - Reflection/ Notes Explanation
- 20% - Correctness
- Bonus 10%:
    - 5% - Commit,
    - 3% - Run properly,
    - 2% - Enough explanation and opinion

Rubrics

	Score 4	Score 3	Score 2	Score 1
Correctness	Correct perfectly as requested.	Something is missing.	A lot of requirements are missing, but the program still runs.	Incorrect program
Reflection/ Notes Explanation	More than 5 sentences. The description is sound.	Less than or equal 5 sentences. The description is sound.	The description is not sound although still related.	The description is not sound. It is not related to the topics.
Commit	All requested commits are registered and correct.	At least 50% of the requested commits are registered and correct.	At least 25% of the requested commits are registered and correct.	Less than 25%.

Notes to Assistants: For 3.2, any creativity is accepted as full score (4), as long as it is provided with some explanation and some screenshot showing the creativity.