

# Asynchronous Execution Model

## Overview

As multicore and networked systems proliferate, performance of multi-threaded synchronous and asynchronous execution has become more critical. In addition to the standard low-level multithreaded model commonly used in operating systems, the **async/await** pattern, based on coroutines, promises, and futures, has blossomed. This document highlights the key elements of the async/await model and provides examples in Rust and JavaScript.

## Concepts

The async/await pattern uses **coroutines** (also called async functions / methods), which are function object generators; when executed, a coroutine returns a function object, but this function object does not usually immediately execute. This function object can be stored or scheduled. Once scheduled, the coroutine becomes a **task** (a function object that has been scheduled for execution).

```
[dependencies]
futures = "0.3"
```

```
<html><body><script
src="invade.js">
```

```
use futures::executor::block_on; // executor

async fn howdy() -> String // Return func.
object
{
    String::from("We come in peace!\n") //
    Fools!
}

fn main()
{
    let promise = howdy(); // Get function
    object
```

```
async function howdy() //
same
{
    return "We come in peace!\n";
}

function main()
{
    howdy()
    .then(hi => // promise
    {
        console.log(hi); //
    greet
```

Usually, we use a coroutine to generate a function object to guarantee completion of some computation; the computation is referred to as a **promise** (i.e., a promise to complete the computational task). This often (but not always!) results in some computational result, so a placeholder value is created – known as a **future** (i.e., a value that will be computed in the future). As they are closely related, we sometimes hear the terms “future” and “promise” used interchangeably. Strictly speaking, we generate a promise that provides us a future value.

## Await Pattern

Often within one asynchronous task, we may need to wait for another asynchronous task to complete – and ideally, we do not want to block the entire thread, as there may be other asynchronous tasks that could still complete. In these cases, we can use the **await** operator.

```
use futures::executor::block_on;

async fn prepLaser() -> u8
{
    let laserPower: u8 = 100;
    return laserPower;
}

async fn howdy() -> String
{
    String::from("We come in peace!");
}

async fn land()
{
    println!("Ship has landed.\n");
}

async fn contact()
{
    land().await;
    println!("{}", howdy().await);
}

async fn evilPlan()
{
    futures::join!(contact(),
    prepLaser());
}

fn main()
{
    block_on(evilPlan());
    println!("Done.\n");
}
```

```
async function prepLaser()
{
    laserPower = 100;
    return laserPower;
}

async function howdy()
{
    return "We come in peace!\n";
}

async function land()
{
    console.log("Ship has landed.\n");
}

async function contact()
{
    await land();
    hello = await howdy();
    console.log(hello);
}

async function evilPlan()
{
    p1 = contact();
    p2 = prepLaser();
    await Promise.allSettled([p1,
    p2]);
}

evilPlan()
.then(_ => console.log("Done.\n"));
```

While the syntax and use for “await” are slightly different in various languages, the concept is the same – from within an asynchronous routine, we can **await** the completion of another asynchronous routine before continuing, and do so in a way that does not block other routines in the same process / thread unnecessarily.