The Async/Await

The Rust syntax features async/await, which allow you to watch the running thread rather than interrupt it and make progress in another program while waiting for the process to finish. Async/await works on the principle of letting programmers write what appears to be standard synchronous code, but the compiler converts it to asynchronous code. The two keywords are awaited, and the async foundation is used. The synchronous function in the signature function can be changed into an asynchronous function that returns the future by using the asynchronous keyword. In synchronous functions, the await keyword might be used to retrieve the asynchronous value of the future because the async/await keywords alone would not be helpful.

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 and async : async fn and async blocks. Each return a value that implements the future trait :

// `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

}

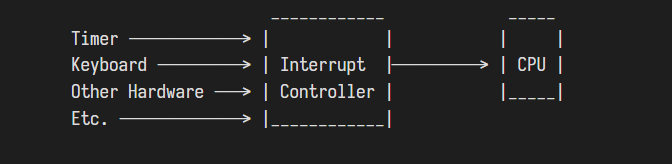
}

double fault

A double fault is a specific exception that happens when the CPU neglects to call an exception handler, to put it simply. It happens, for instance, when a page fault is raised but no page fault handler is registered in the interrupt descriptor table (IDT). The catch(...) or catch(Exception e) blocks in C++ or Java or C# are examples of catch-all blocks in programming languages with exceptions. A double fault functions just like any other exception. So can define a typical handler function for it in the IDT because it has the vector number. The provision of a double fault handler is crucial because, if left unattended, a double fault can result in a lethal triple fault.

Hardware interrupts

The CPU can be informed by associated hardware devices using interrupts. So the keyboard may inform the kernel of each keypress rather than allowing the kernel periodically inspect the keyboard for new characters (a process known as polling). Because the kernel only needs to react when something occurs, this is far more efficient. Additionally, because the kernel can respond right now rather than just at the subsequent poll, reaction times are sped up. It is impossible to connect all gear directly to the CPU. As an alternative, a different interrupt controller gathers all interrupts from all devices and then alerts the CPU:



The majority of interrupt controllers may be programmed, thus they support a range of interrupt priority levels. To ensure precise timekeeping, for instance, this enables giving timer interrupts a greater priority than keyboard interrupts. Hardware interrupts take place asynchronously, unlike exceptions. They can therefore happen at any time and are totally independent of the code that was actually executed. Consequently, all of the potential concurrency-related issues suddenly appear in our kernel in the shape of a type of concurrency. Because it bans changeable global state, Rust's rigid ownership model is helpful in this situation. However, as we will see later in this piece, deadlocks are still possible.