Asynchronous Development

Lecture 5

Asynchronous Development

- Concurrency
- Asynchronous Executor
- Future s
- Communication between tasks

Concurrency

Preemptive and Cooperative

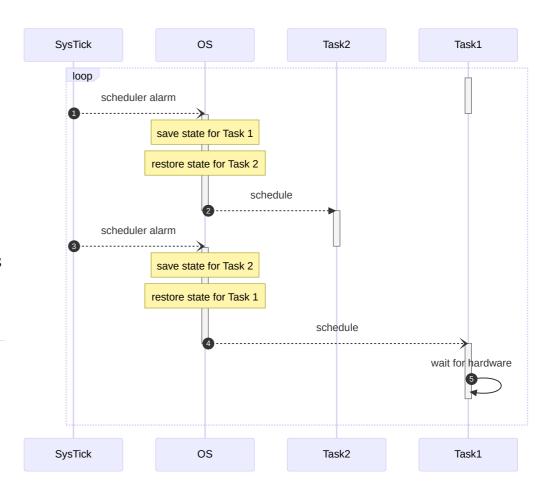
Bibliography

for this section

Brad Solomon, Async IO in Python: A Complete Walkthrough

Preemptive Concurrency

- MCUs are usually single core^[1]
- Tasks in parallel require an OS^[2]
- Tasks can be suspended at any time
- Switching the task is expensive
- Tasks that do a lot of I/O which makes the switching time longer than the actual processing time
- 1. RP2040 is a dual core MCU, we use only one core ←
- 2. Running in an ISR is not considered a normal task ↔



Asynchronous Executor

of Embassy

Bibliography

for this section

Embassy Documentation, *Embassy executor*

Tasks

- #[embassy_executor::main]
 - starts the Embassy scheduler
 - defines the main task
- #[embassy_executor::task] defines a new
 task
 - pool_size -is optional and defines how many identical tasks can be spawned
- the main task
 - initializes the the led
 - spawns the led_blink task (adds to the scheduler)
 - uses .await to give up the MCU while waiting form the button

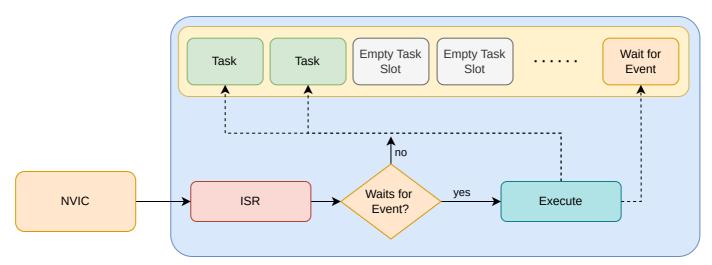
```
#[embassy executor::task(pool size = 2)]
async fn led blink(mut led:Output<'static, PIN X>) {
        Timer::after secs(1).await;
#[embassy executor::main]
async fn main(spawner: Spawner) {
```

Tasks can stop the executor

- unless awaited, async functions are not executed
- tasks have to use .await in loops, otherwise they block the scheduler

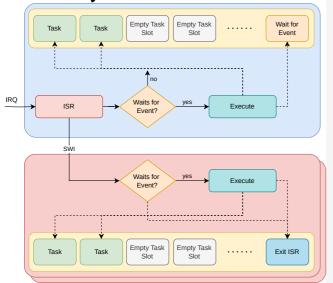
```
#[embassy executor::task]
    async fn led blink(mut led:Output<'static, PIN X>) {
        loop {
            led.toogle();
            // this does not execute anything
            Timer::after secs(1);
            // infinite loop without `.await`
            // that never gives up the MCU
9
    #[embassy executor::main]
    async fn main(spawner: Spawner) {
            button.wait for rising edge().await;
            info!("button pressed");
```

How it works



- sleep when all tasks wait for events
- after an ISR is executed.
 - if waiting for events, ask every task if it can execute (if the IRQ was what the task was .await ing for)
 - if a task is executing, continue the task until it .await s
- if a task never .await s, the executor does not run and never executes another task

Priority Tasks



```
#[interrupt]
unsafe fn SWI_IRQ_1() {
    EXECUTOR_HIGH.on_interrupt()
}
#[interrupt]
unsafe fn SWI_IRQ_0() {
    EXECUTOR_MED.on_interrupt()
}
```

```
static EXECUTOR LOW: StaticCell<Executor> = StaticCell::new();
     #[entry]
17
         // Low priority executor: runs in thread mode, using WFE/SEV
         let executor = EXECUTOR LOW.init(Executor::new());
18
19
         executor.run(|spawner| {
20
             unwrap!(spawner.spawn(run low()));
21
         });
```

priority executors run in ISRs, lower priority tasks are interrupted

The Future type

a.k.a Promise in other languages

Bibliography

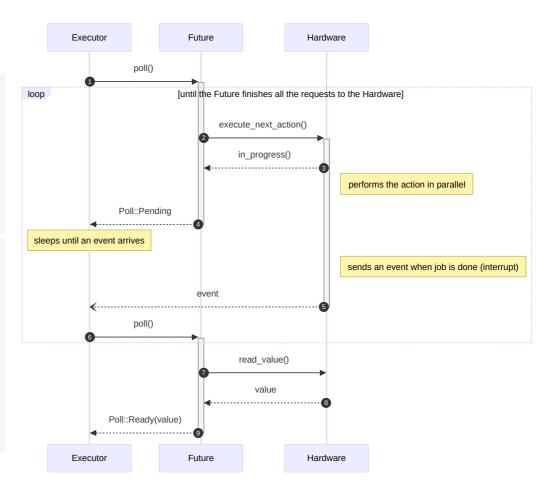
for this section

Bert Peters, How does async Rust work

Future

```
enum Poll<T> {
     Pending,
     Ready(T),
}
trait Future {
    type Output;
    fn poll(&mut self) -> Poll<Self::Output>;
}
```

```
fn execute<F>(mut f: F) -> F::Output
where
  F: Future
{
  loop {
    match f.poll() {
      Poll::Pending => wait_for_event(),
      Poll::Ready(value) => break value
    }
  }
}
```



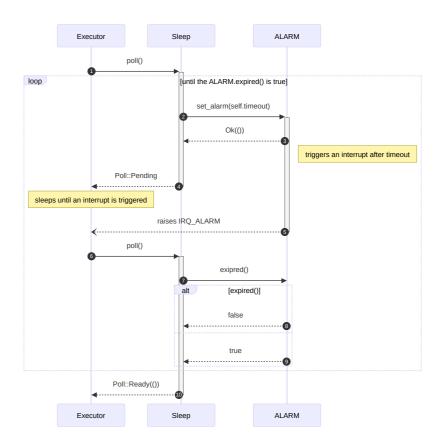
Implementing a Future

```
11
     impl Sleep {
12
         pub fn new(timeout: usize) -> Sleep {
13
             Sleep {
14
                 timeout,
15
                 status: SleepStatus::SetAlarm,
16
18 }
```

```
fn poll(&mut self) -> Poll<Self::Output> {
           ALARM.set alarm(self.timeout);
        SleepStatus::WaitForAlarm => {
           } else {
               Poll::Pending
```

Executing Sleep

```
fn poll(&mut self) -> Poll<Self::Output> {
           ALARM.set alarm(self.timeout);
        SleepStatus::WaitForAlarm => {
            if ALARM.expired() {
           } else {
                Poll::Pending
```



Async Rust

```
async fn blink(mut led: Output<'static, PIN_X>) {
    led.on();
    Timer::after_secs(1).await;
    led.off();
}
```

Rust rewrites

```
struct Blink {
    // status
    status: BlinkStatus,
    // local variables
    led: Output<'static, PIN_X>,
        timer: Option<impl Future>,
}
impl Blink {
    pub fn new(led: Output<'static, PIN_X>) -> Blink {
        Blink { status: BlinkStatus::Part1, led, timer: None }
    }
}
fn blink(led: Output<'static, PIN_X>) -> Blink {
        Blink::new(led)
}
```

```
fn poll(&mut self) -> Poll<Self::Output> {
   match self.status {
        self.timer1 = Some(Timer::after_secs(1));
        if self.timer.unwrap().poll() == Poll::Pending {
      BlinkStatus::Part3 => {
        self.led.off();
       return Poll::Ready(());
```

Async Rust

- the Rust compiler rewrites async function into Future
- it does not know how to execute them
- executors are implemented into third party libraries

```
use engine::execute;
     async fn blink(mut led: Output<'static, PIN_X>) {
     #[entry]
     fn main() -> ! {
14
         execute(blink()); // this works, as `execute` executes the Blink future
```

Executor

```
fn executor() {
                 if let Some(task) = task {
                     if Poll::Ready(_) = task.poll() {
                         *task = None
             // wait for interrupts
14
             cortex m::asm::wfi();
15
```

- this is a simplified version, Option<impl Future> does not work
- the executor is not able to use TASKS like this
- an efficient executor will not poll all the tasks, it uses a waker that tasks use to signal the executor

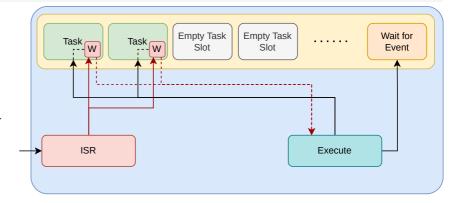
The Future trait

that Rust provides

```
trait Future {
    type Output;

fn poll(mut self: std::pin::Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}
```

- Pin to mut self, which means that self cannot be moved
- Context which provides the waker
 - tasks are polled only if they ask the executor (by using the wake function)
- embassy-rs provides the execution engine



Communication

between tasks

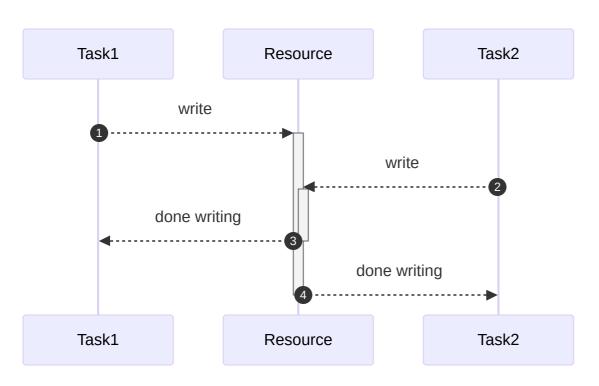
Bibliography

for this section

Omar Hiari, Sharing Data Among Tasks in Rust Embassy: Synchronization Primitives

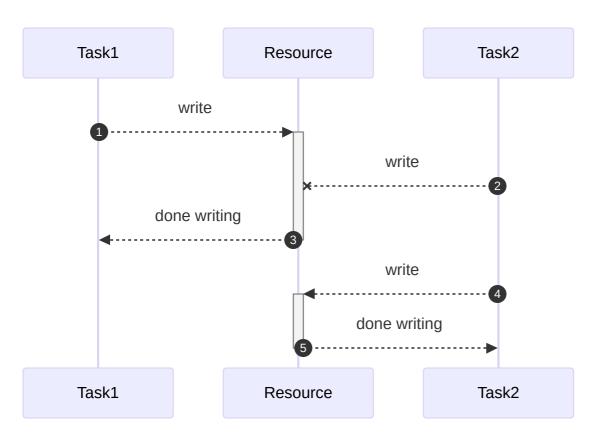
Simultaneous Access

Rust forbids simultaneous writes access



Exclusive Access

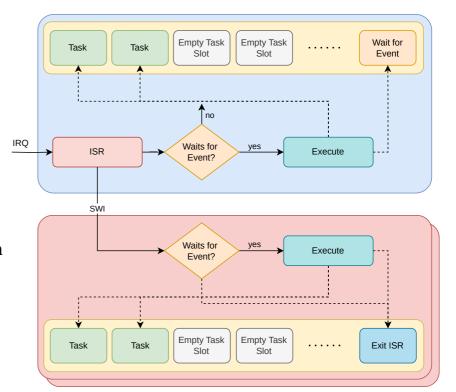
we want to sequentiality access the resource



Synchronization

safely share data between tasks

- NoopMutex used for data shared between tasks
 within the same executor
- Critical SectionMutex used for data shared
 between multiple executors, ISRs and cores
- ThreadModeMutex used for data shared between tasks within low priority executors (not running in ISRs mode) running on a single core



- ISRs are executed in parallel with tasks
- embassy allows registering priority executors, that run tasks in ISRs
- some MCUs have multiple cores

Blocking Mutex

no .await allowed while the mutex is held

```
#[embassy executor::task]
     async fn task1() {
10
         SHARED_DATA.lock(|f| {
11
             let data = f.borrow_mut();
12
             // edit data
13
             f.replace(data);
14
        });
```

Async Mutex

. await is allowed while the Mutex is held, it will release the Mutex while await ing

```
#[embassy executor::task]
     async fn task1() {
10
             let mut data = SHARED_DATA.lock().await;
11
12
             // edit *data
13
             Timer::after(Duration::from_millis(1000)).await;
14
```

Channels

send data from a task to another

Embassy provides four types of channels synchronized using Mutex s

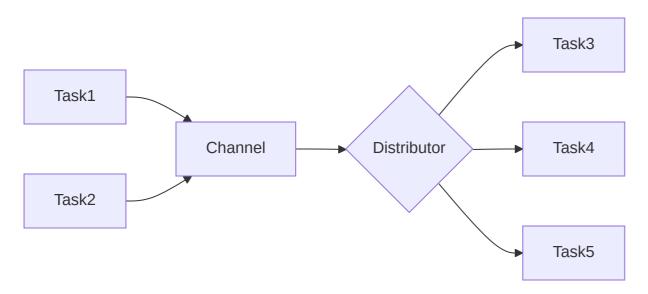
Type	Description		
Channel	A Multiple Producer Multiple Consumer (MPMC) channel. Each message is only received by a single consumer.		
PriorityChannel	A Multiple Producer Multiple Consumer (MPMC) channel. Each message is only received by a single consumer. Higher priority items are shifted to the front of the channel.		
Signal	Signalling latest value to a single consumer.		
PubSubChannel	A broadcast channel (publish-subscribe) channel. Each message is received by all consumers.		

Channel and Signal

sends data from one task to another

Channel - A Multiple Producer Multiple Consumer (MPMC) channel. Each message is only received by a single consumer.

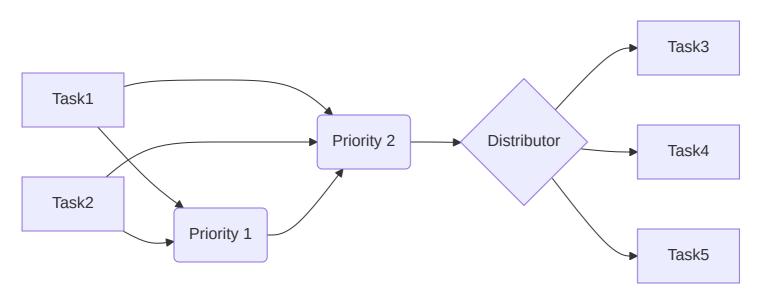
Signal - Signalling latest value to a single consumer.



PriorityChannel

sends data from one task to another with a priority

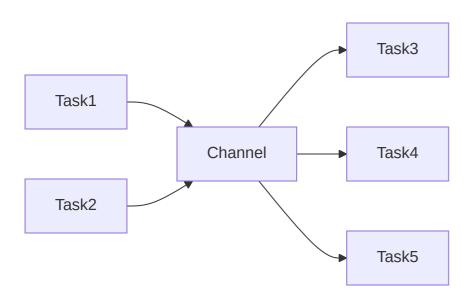
PriorityChannel - A Multiple Producer Multiple Consumer (MPMC) channel. Each message is only received by a single |consumer. Higher priority items are shifted to the front of the channel.



PubSubChannel

sends data from one task to all receiver tasks

PubSubChannel - A broadcast channel (publish-subscribe) channel. Each message is received by all consumers.



Channel Example

```
#[embassy executor::main]
     async fn main(spawner: Spawner) {
         loop {
             match CHANNEL.receive().await {
                 LedState::On => led.on(),
10
                 LedState::Off => led.off()
11
12
13
14
     #[embassy executor::task]
```

Conclusion

we talked about

- Preemptive & Cooperative Concurrency
- Asynchronous Executor
- Future s and how Rust rewrites async function
- Communication between tasks