Asynchronous Development

Lecture 4

Asynchronous Development

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

Concurrency

Preemptive and Cooperative

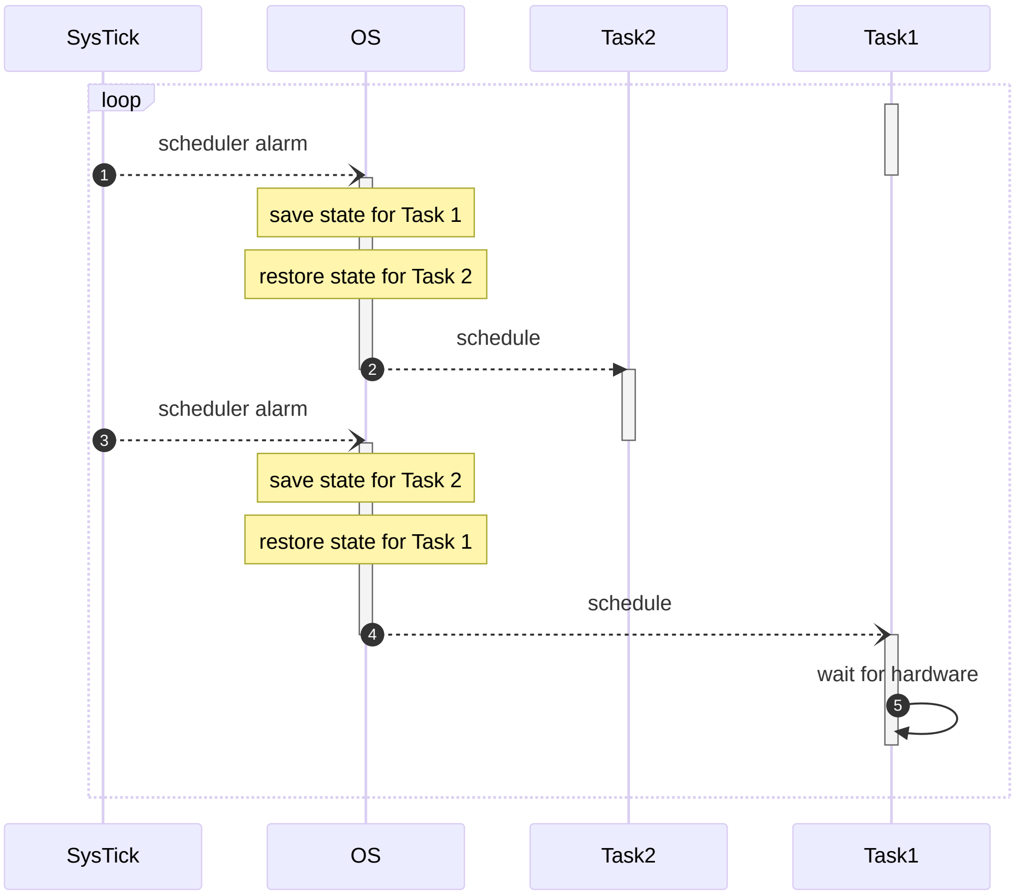
Bibliography

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**

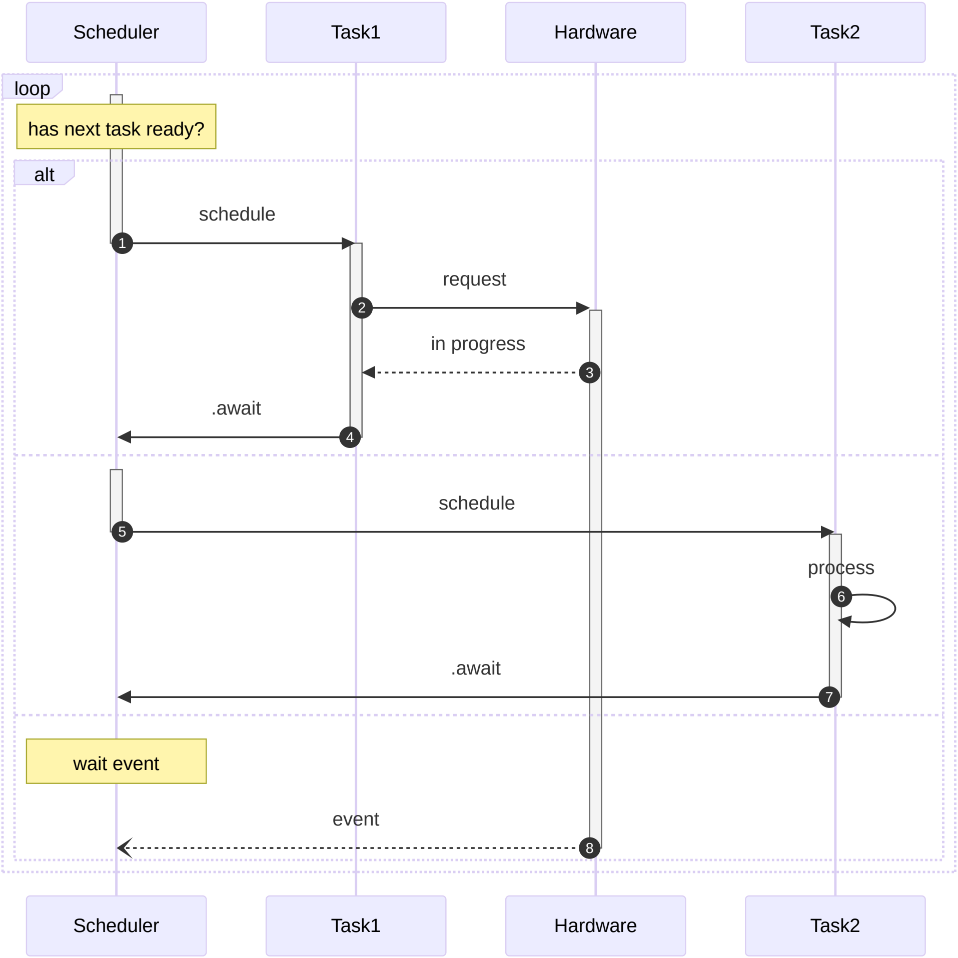


---

1. RP2350 is a dual core MCU, we use only one core ↩
2. Running in an ISR is not considered a normal task ↩

Cooperative Concurrency

- tasks cannot be interrupted[1]
- **hardware** works in an **asynchronous** way
- tasks cooperate
  - give up the MCU for other tasks to use it while they wait for hardware
- there is **no need for an OS**, everything is done in **one single flow**
- **no penalty** for saving and restoring the state



---

1. except for ISR ↩

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

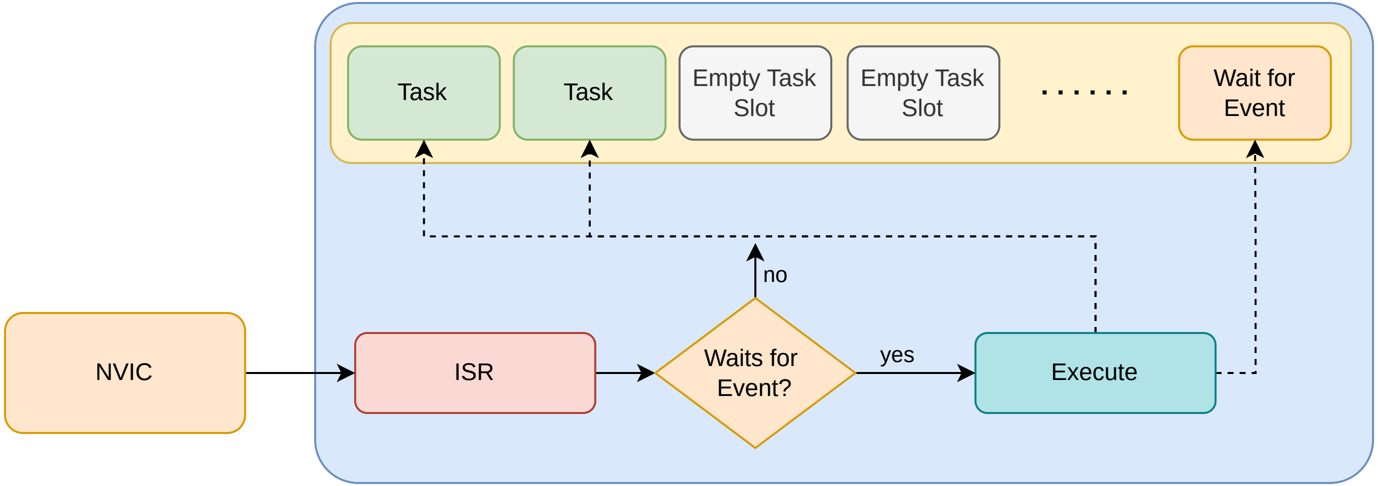
```rust
#[embassy_executor::task(pool_size = 2)]
async fn led_blink(mut led: AnyPin) {
    loop {
        led.toogle();
        Timer::after_secs(1).await;
    }
}

#[embassy_executor::main]
async fn main(spawner: Spawner) {
    // ...

    // init led
    spawner.spawn(led_blink(led)).unwrap();
    info!("task started");

    // init button
    loop {
        button.wait_for_rising_edge().await;
        info!("button pressed");
    }
}
```

Tasks can stop the executor

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

```rust
#[embassy_executor::task]
async fn led_blink(mut led: AnyPin) {
    loop {
        led.toogle();
        // this does not execute anything
        Timer::after_secs(1);
        // infinite loop without `.await`
        // that never gives up the MCU
    }
}

#[embassy_executor::main]
async fn main(spawner: Spawner) {
    // ..
    loop {
        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

- the tasks run in separate *executors*
- triggered from **interrupts**
- will **interrupt any task** running in the **main executor**

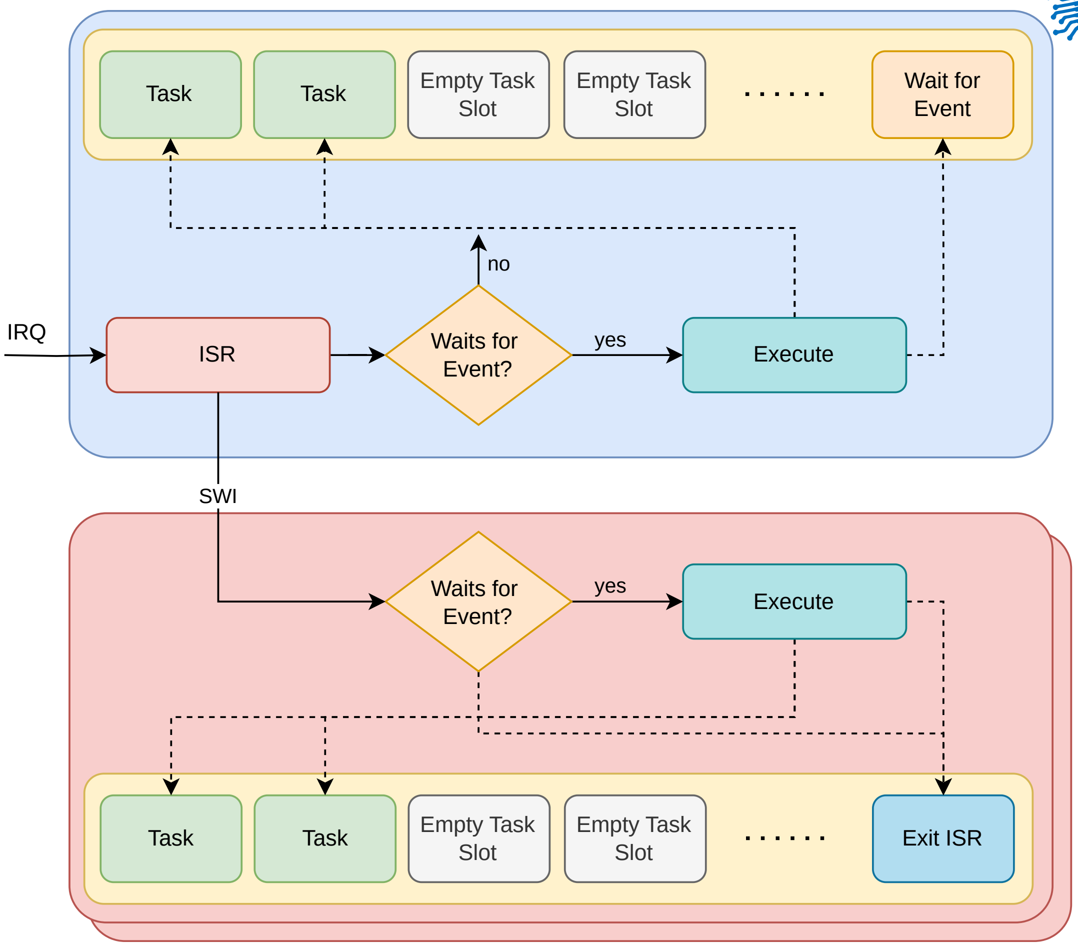
> ⚠️ Tasks that share data between executors require synchronization.

RP2

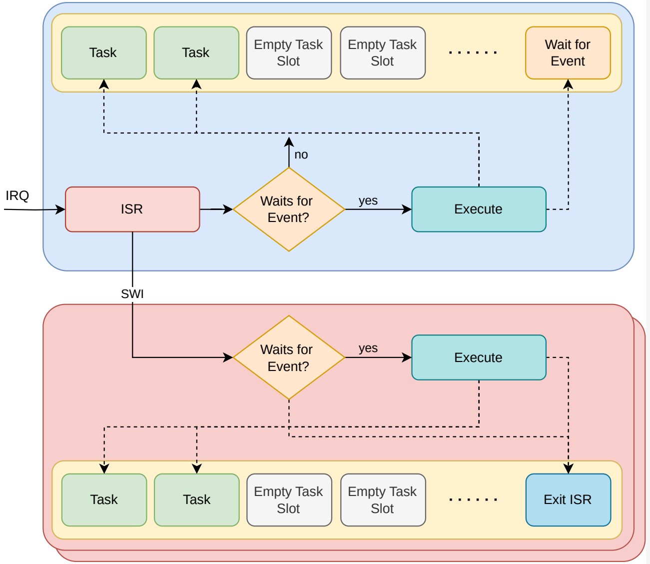
- use `SWI_IRQ_01` and `SWI_IRQ_01`

STM32U545RE

- use any interrupt (`UART`, `SPI`, ...) not used anywhere else



Priority Tasks




```rust
#[interrupt]
unsafe fn UART4() {
    EXECUTOR_HIGH.on_interrupt()
}
#[interrupt]
unsafe fn UART5() {
    EXECUTOR_MEDIUM.on_interrupt()
}
```

```rust
// STM32U545RE

static EXECUTOR_HIGH: InterruptExecutor = InterruptExecutor::new();
static EXECUTOR_MEDIUM: InterruptExecutor = InterruptExecutor::new();
static EXECUTOR_LOW: StaticCell<Executor> = StaticCell::new();

#[entry]
fn main() -> ! {
    // High-priority executor: UART4, priority level 2
    interrupt::SWI_IRQ_1.set_priority(Priority::P2);
    let spawner = EXECUTOR_HIGH.start(interrupt::UART4);
    spawner.spawn(run_high()).unwrap();

    // Medium-priority executor: UART5, priority level 3
    interrupt::SWI_IRQ_0.set_priority(Priority::P3);
    let spawner = EXECUTOR_MEDIUM.start(interrupt::UART5);
    spawner.spawn(run_med()).unwrap();

    // Low priority executor: runs in thread mode, using WFE/SEV
    let executor = EXECUTOR_LOW.init(Executor::new());
    executor.run(|spawner| {
        unwrap!(spawner.spawn(run_low()));
    });
}
```

💡 priority executors run in ISRs, lower priority tasks are interrupted

The `Future` type

a.k.a `Promise` in other languages

Bibliography

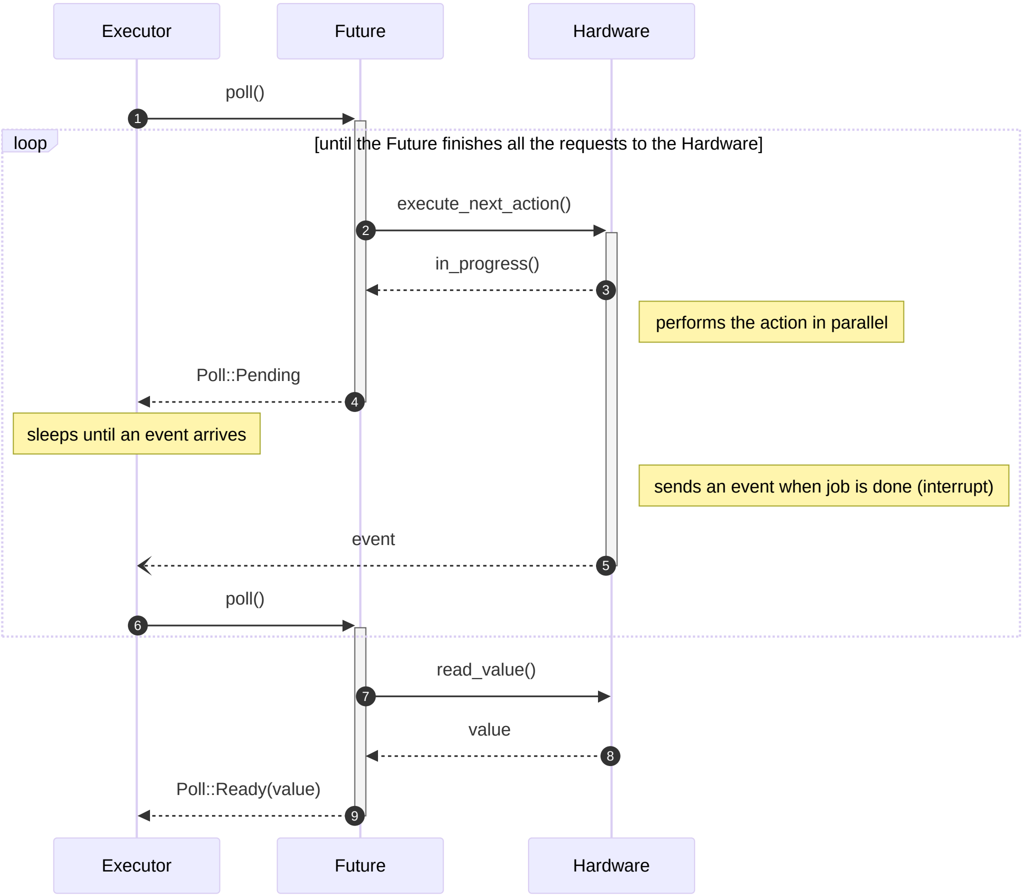
for this section

Bert Peters, *How does async Rust work*

Future

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

```rust
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
    }
  }
}
```



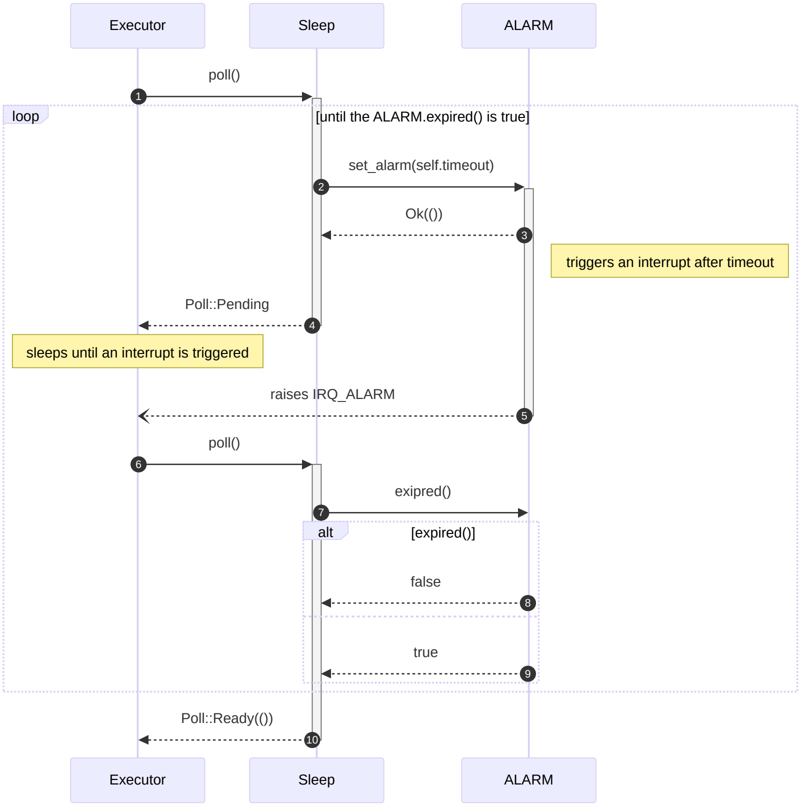
Implementing a Future

```rust
enum SleepStatus {
    SetAlarm,
    WaitForAlarm,
}

struct Sleep {
    timeout: usize,
    status: SleepStatus,
}

impl Sleep {
    pub fn new(timeout: usize) -> Sleep {
        Sleep {
            timeout,
            status: SleepStatus::SetAlarm,
        }
    }
}
```

```rust
impl Future for Sleep {
    type Output = ();

    fn poll(&mut self) -> Poll<Self::Output> {
        loop {
            match self.status {
                SleepStatus::SetAlarm => {
                    ALARM.set_alarm(self.timeout);
                    self.status = SleepStatus::WaitForAlarm;
                }
                SleepStatus::WaitForAlarm => {
                    if ALARM.expired() {
                        return Poll::Ready(());
                    } else {
                        return Poll::Pending
                    }
                }
            }
        }
    }
}
```

Executing Sleep

```rust
fn poll(&mut self) -> Poll<Self::Output> {
    loop {
        match self.status {
            SleepStatus::SetAlarm => {
                ALARM.set_alarm(self.timeout);
                self.status = SleepStatus::WaitForAlarm;
            }
            SleepStatus::WaitForAlarm => {
                if ALARM.expired() {
                    return Poll::Ready(());
                } else {
                    return Poll::Pending;
                }
            }
        }
    }
}
```



Async Rust

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

Rust rewrites

```rust
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)
}
```

```rust
impl Future for Blink {
  type Output = ();
  fn poll(&mut self) -> Poll<Self::Output> {
    loop {
      match self.status {
        BlinkStatus::Part1 => {
          self.led.on();
          self.timer1 = Some(Timer::after_secs(1));
          self.status = BlinkStatus::Part2;
        }
        BlinkStatus::Part2 => {
          if self.timer.unwrap().poll() == Poll::Pending {
            return Poll::Pending;
          } else {
            self.status = BlinkStatus::Part3;
          }
        }
        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

```rust
use engine::execute;

// Rust rewrites the function to a Future
async fn blink(mut led: Output<'static, PIN_X>) {
    led.on();
    Timer::after_secs(1).await;
    led.off();
}

#[entry]
fn main() -> ! {
    blink(); // this returns the Blink future, but does not execute it
    blink().await; // does not work, as `main` is not an `async` function
    execute(blink()); // this works, as `execute` executes the Blink future
}
```

Executor

```rust
static TASKS: [Option<impl Future>; N] = [None, N];

fn executor() {
    loop {
        // ask all tasks to continue if they have available data
        for task in TASKS.iter_mut() {
            if let Some(task) = task {
                if Poll::Ready(_) = task.poll() {
                    *task =  None
                }
            }
        }

        // wait for interrupts
        cortex_m::asm::wfi();
    }
}
```

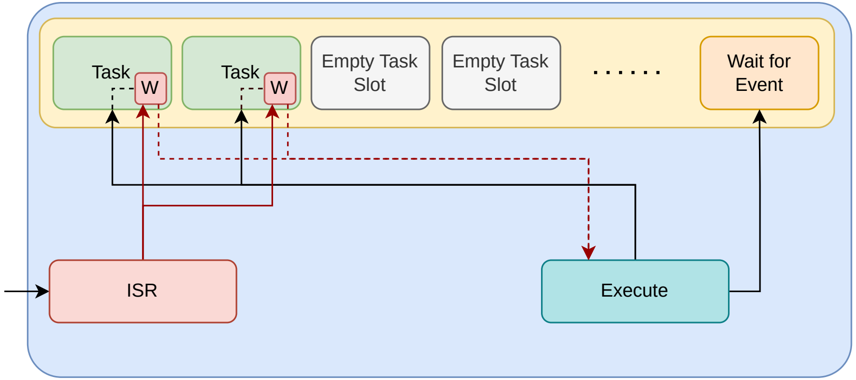
- this is a simplified version, `Option<impl Future>` does not work
- the executor is not able to use `TASKS` like this
- an efficient executor will not poll all the tasks, it uses a `waker` that tasks use to signal the executor

The `Future` trait

that Rust provides

```rust
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

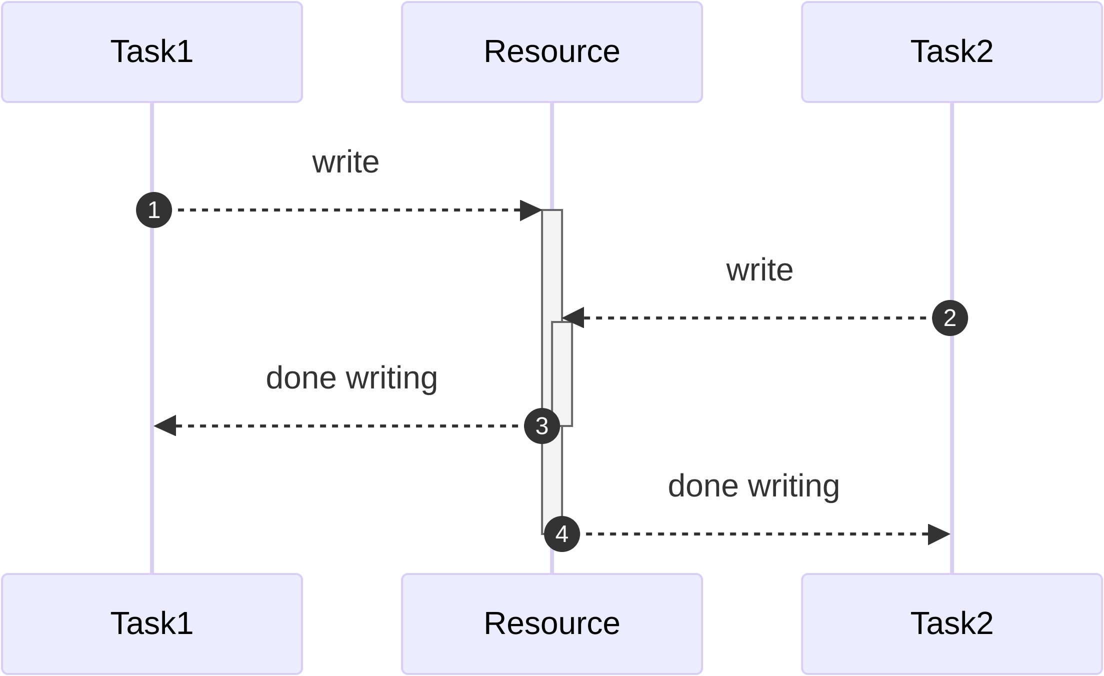
Bibliography

for this section

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

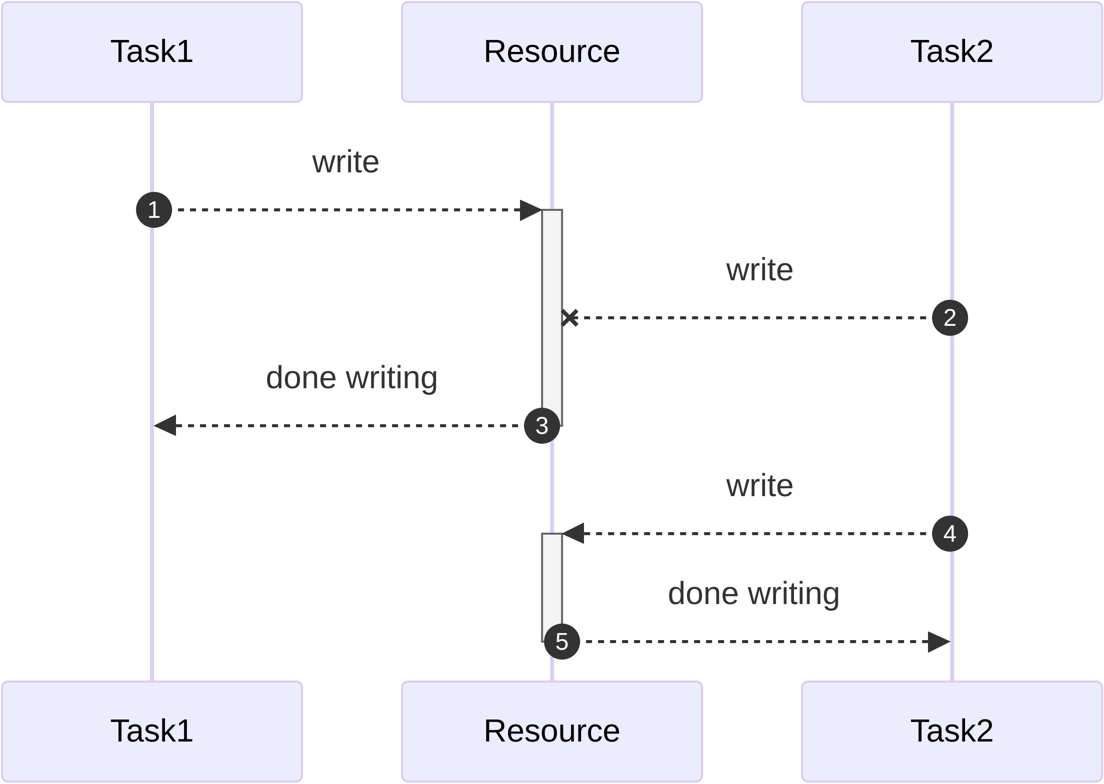
Simultaneous Access

Rust forbids simultaneous writes access



Exclusive Access

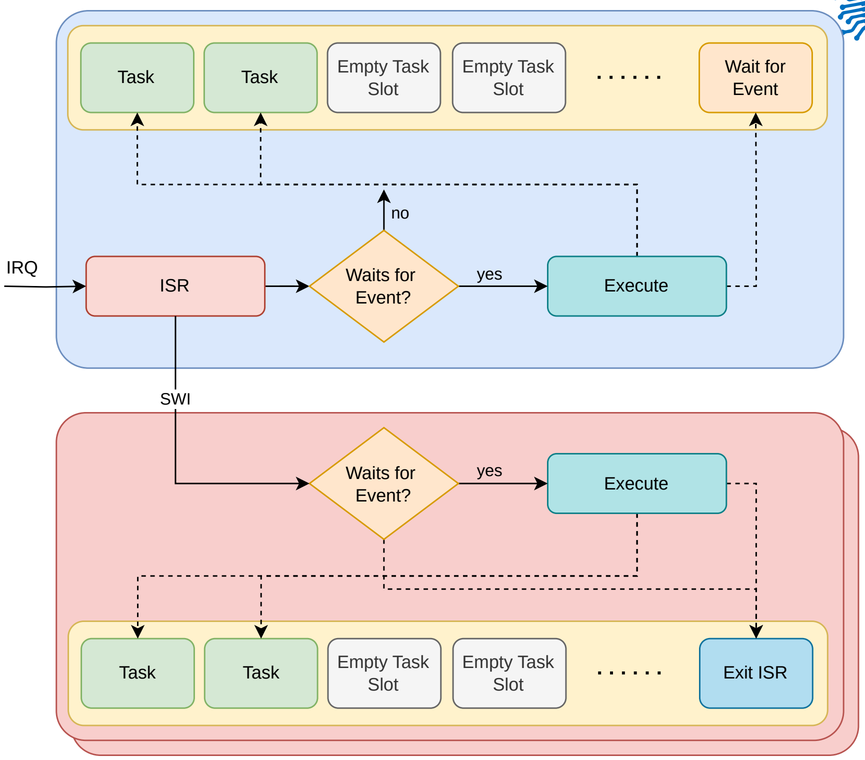
we want to sequentiality access the resource



Synchronization

safely share data between tasks

- `NoopMutex` - used for data shared between tasks within the **same executor**
- `CriticalSectionMutex` - 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

```rust
use embassy_sync::blocking_mutex::Mutex;

struct Data {/* ... */ }

static SHARED_DATA: Mutex<ThreadModeRawMutex, RefCell<Data>> = Mutex::new(RefCell::new(Data::new(/* ... */)));

#[embassy_executor::task]
async fn task1() {
    // Load value from global context, modify and store
    SHARED_DATA.lock(|f| {
        let data = f.borrow_mut();
        // edit data
        f.replace(data);
    });
}
```

Async Mutex

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

```rust
use embassy_sync::mutex::Mutex;

struct Data {/* ... */ }

static SHARED: Mutex<ThreadModeRawMutex, Data> = Mutex::new(Data::new(/* ... */));

#[embassy_executor::task]
async fn task1() {
    // Load value from global context, modify and store
    {
        let mut data = SHARED_DATA.lock().await;
        // edit *data
        Timer::after(Duration::from_millis(1000)).await;
    }
}
```

Channels

send data from a task to another

Embassy provides four types of channels synchronized using `Mutex`s

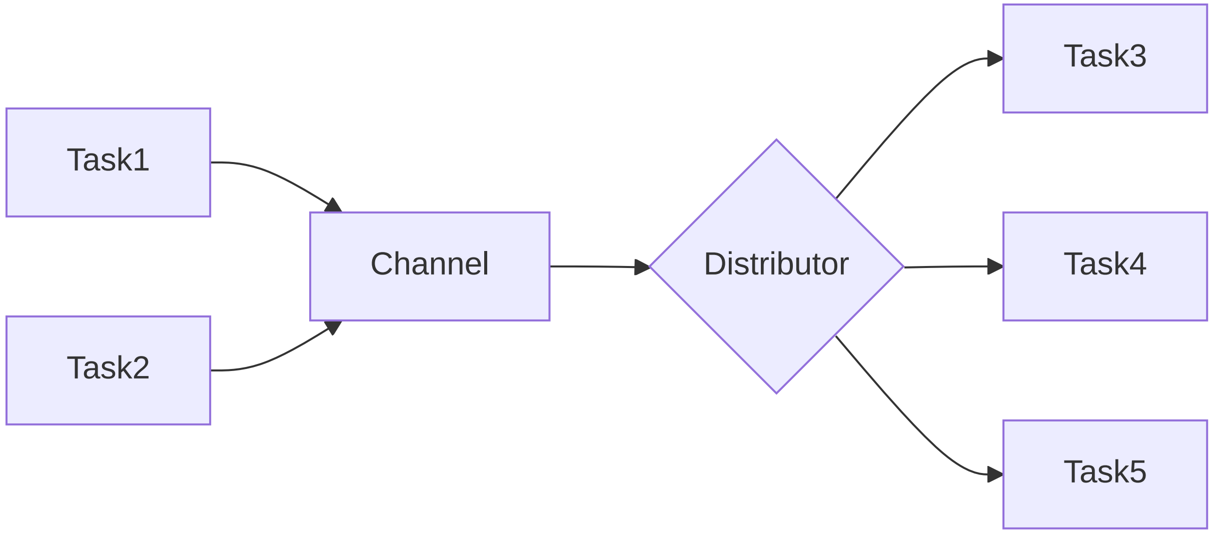
Type	Description
`Channel`	A Multiple Producer Multiple Consumer (MPMC) channel. Each message is only received by a single consumer.
`PriorityChannel`	A Multiple Producer Multiple Consumer (MPMC) channel. Each message is only received by a single consumer. Higher priority items are shifted to the front 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.

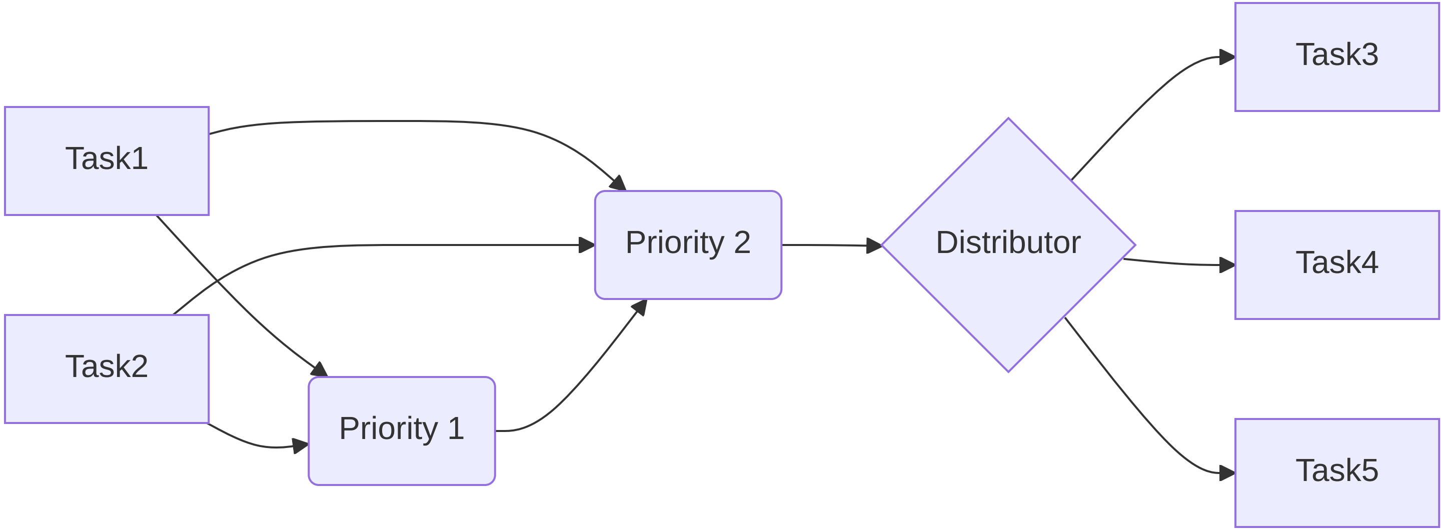
`Signal` - Signalling latest value to a single consumer.



PriorityChannel

sends data from one task to another with a priority

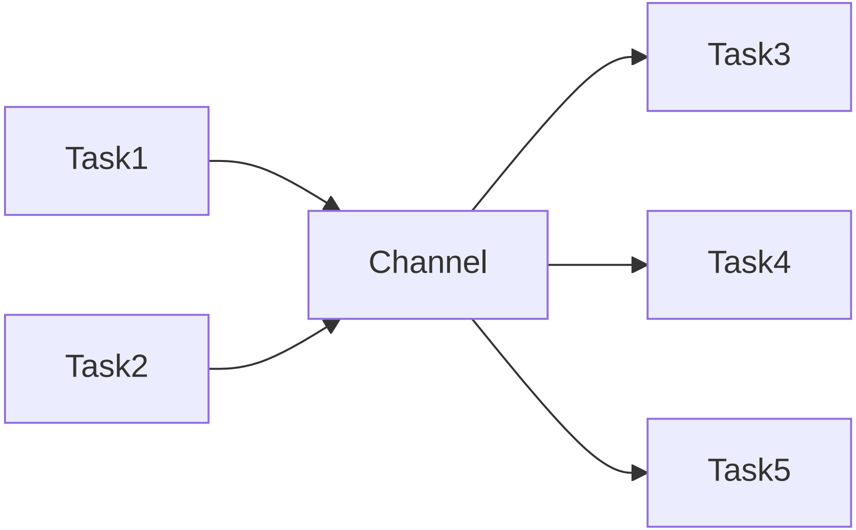
`PriorityChannel` - A Multiple Producer Multiple Consumer (MPMC) channel. Each message is only received by a single |consumer. Higher priority items are shifted to the front of the channel.



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

```rust
enum LedState { On, Off }
static CHANNEL: Channel<ThreadModeRawMutex, LedState, 64> = Channel::new();

#[embassy_executor::main]
async fn main(spawner: Spawner) {
    // init led
    spawner.spawn(execute_led(CHANNEL.sender(), Duration::from_millis(500)));
    loop {
        match CHANNEL.receive().await {
            LedState::On => led.on(),
            LedState::Off => led.off()
        }
    }
}

#[embassy_executor::task]
async fn execute_led(control: Sender<'static, ThreadModeRawMutex, LedState, 64>, delay: Duration) {
    let mut ticker = Ticker::every(delay);
    loop {
        control.send(LedState::On).await;
        ticker.next().await;
        control.send(LedState::Off).await;
        ticker.next().await;
```

Conclusion

we talked about

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