第二十一讲：异步编程 (Asynchronous Programming)

第 5 节：Waker and Reactor

向勇、陈渝

清华大学计算机系

xyong,yuchen@tsinghua.edu.cn

2020 年 5 月 5 日

Waker

- The *Waker* type allows for a loose coupling between the reactor-part and the executor-part of a runtime

Waker

- The *Waker* type allows for a loose coupling between the reactor-part and the executor-part of a runtime
- By having a wake up mechanism that is not tied to the thing that executes the future, runtime-implementors can come up with interesting new wake-up mechanisms

Waker

- The *Waker* type allows for a loose coupling between the reactor-part and the executor-part of a runtime
- By having a wake up mechanism that is not tied to the thing that executes the future, runtime-implementors can come up with interesting new wake-up mechanisms
- Creating a ‘Waker‘ involves creating a ‘vtable‘ which allows us to use dynamic dispatch to call methods on a type erased trait object we construct our selves

Fat pointers in Rust

Example ‘&[i32]‘

- The first 8 bytes is the actual pointer to the first element in the array (or part of an array the slice refers to)
- The second 8 bytes is the length of the slice.

Fat pointers in Rust

Example ‘&[i32]‘

- The first 8 bytes is the actual pointer to the first element in the array (or part of an array the slice refers to)
- The second 8 bytes is the length of the slice.

Example ‘&dyn SomeTrait‘

- The first 8 bytes points to the ‘data‘ for the trait object
- The second 8 bytes points to the ‘vtable‘ for the trait object

Fat pointers in Rust

Example ‘&[i32]‘

- The first 8 bytes is the actual pointer to the first element in the array (or part of an array the slice refers to)
- The second 8 bytes is the length of the slice.

Example ‘&dyn SomeTrait‘

- The first 8 bytes points to the ‘data‘ for the trait object
- The second 8 bytes points to the ‘vtable‘ for the trait object

Trait object

- ‘&dyn SomeTrait‘ is a reference to a trait, or what Rust calls a trait object.
- Implement ‘Waker:‘ we’ll actually set up a ‘vtable‘

Fat pointers in Rust

Example ‘&[i32]‘

- The first 8 bytes is the actual pointer to the first element in the array (or part of an array the slice refers to)
- The second 8 bytes is the length of the slice.

Example ‘&dyn SomeTrait‘

- The first 8 bytes points to the ‘data‘ for the trait object
- The second 8 bytes points to the ‘vtable‘ for the trait object

Trait object

- ‘&dyn SomeTrait‘ is a reference to a trait, or what Rust calls a trait object.
- Implement ‘Waker:‘ we’ll actually set up a ‘vtable‘

Example:

- Fat pointers in Rust

Reactor

- To actually abstract over this interaction with the outside world in an asynchronous way

Reactor

- To actually abstract over this interaction with the outside world in an asynchronous way
  - Receive events from the operating system or peripherals
  - Forward them to waiting tasks

Reactor

- To actually abstract over this interaction with the outside world in an asynchronous way
  - Receive events from the operating system or peripherals
  - Forward them to waiting tasks
- Mio: Library of reactors in Rust

Reactor

- To actually abstract over this interaction with the outside world in an asynchronous way
  - Receive events from the operating system or peripherals
  - Forward them to waiting tasks
- Mio: Library of reactors in Rust
  - Provide non blocking APIs and event notification for several platforms

Reactor example

- The example task is a timer that only spawns a thread and puts it to sleep for the number of seconds we specify.

Reactor example

- The example task is a timer that only spawns a thread and puts it to sleep for the number of seconds we specify.
- The reactor we create here will create a leaf-future representing each timer.

Reactor example

- The example task is a timer that only spawns a thread and puts it to sleep for the number of seconds we specify.
- The reactor we create here will create a leaf-future representing each timer.
- In return the Reactor receives a waker which it will call once the task is finished.

Reactor example

- The example task is a timer that only spawns a thread and puts it to sleep for the number of seconds we specify.
- The reactor we create here will create a leaf-future representing each timer.
- In return the Reactor receives a waker which it will call once the task is finished.
- Our Reactor
    - Be dependent on thread::spawn

Async implementation in kernel mode

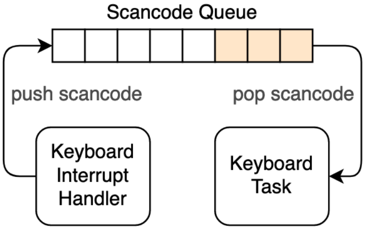
```
// in src/task/simple_executor.rs
pub struct SimpleExecutor {
    task_queue: VecDeque<Task>,
}
impl SimpleExecutor {
    pub fn new() -> SimpleExecutor {
        SimpleExecutor {
            task_queue: VecDeque::new(),
        }
    }
    pub fn spawn(&mut self, task: Task) {
        self.task_queue.push_back(task)
    }
}
```

Asynchronous task based on the keyboard interrupt

- The executor has proper support for ‘Waker‘ notifications
    - The simple executor does not utilize the ‘Waker‘ notifications
    - Simply loops over all tasks until they are done

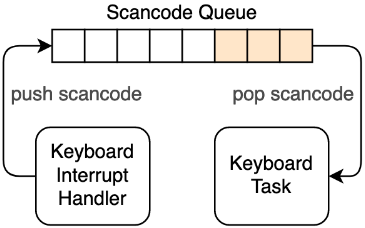
Asynchronous task based on the keyboard interrupt

- The executor has proper support for ‘Waker‘ notifications
  - The simple executor does not utilize the ‘Waker‘ notifications
  - Simply loops over all tasks until they are done
- Create an asynchronous task based on the keyboard interrupt



Asynchronous task based on the keyboard interrupt

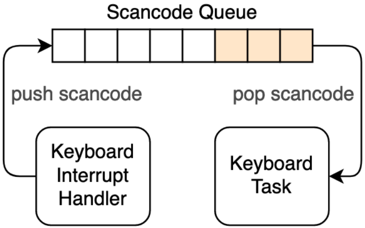
- The executor has proper support for ‘Waker‘ notifications
  - The simple executor does not utilize the ‘Waker‘ notifications
  - Simply loops over all tasks until they are done
- Create an asynchronous task based on the keyboard interrupt



- A simple implementation of that queue could be a mutex-protected ‘VecDeque‘
  - Using mutexes in interrupt handlers is not a good idea since it can easily lead to deadlocks.

Asynchronous task based on the keyboard interrupt

- The executor has proper support for ‘Waker‘ notifications
  - The simple executor does not utilize the ‘Waker‘ notifications
  - Simply loops over all tasks until they are done
- Create an asynchronous task based on the keyboard interrupt



- A simple implementation of that queue could be a mutex-protected ‘VecDeque‘
  - Using mutexes in interrupt handlers is not a good idea since it can easily lead to deadlocks.
- Example: Keyboard future

Complete Example

- Finished Example