Design Patterns for Asynchronous Code

- Compose `Future` values
- Avoid blocking I/O
- Avoid long-running computations

Compose `Future` Values

- `async` functions should be small, limited scope
- Perform a single well-defined task:
  - Read and parse a file
  - Read, process, and respond to a network request

- Rust provides combinators that can combine `Future` values, to produce a new `Future`:
  - `for_each()`, `and_then()`, `read_exact()`, `select()`
  - Can ease composition of asynchronous functions – but can also obfuscate

Avoid Blocking Operations

- Asynchronous code multiplexes I/O operations on single thread
  - Provides asynchronous aware versions of I/O operations
    - File I/O, network I/O (TCP, UDP, Unix sockets)
    - Non-blocking, return `Future` values that interact with the runtime
  - Does **not** interact well with blocking I/O
    - A `Future` that blocks on I/O will block **entire** runtime

```
Read  → AsyncRead
Write → AsyncWrite
```

- Programmer discipline required to ensure asynchronous and blocking I/O are not mixed within a code base
  - Including within library functions, etc.

Avoid Long-running Computations

- Control passing between **`Future`** values is explicit
  - **`await`** calls switch control back to the runtime
  - Next runnable **`Future`** is then scheduled
  - A **`Future`** that doesn't call **`await`**, and instead performs some long-running computation, will starve other tasks

- Programmer discipline required to spawn separate threads for long-running computations
  - Communicate with these via message passing – that can be scheduled within a **`Future`**

An Asynchronous Future?

When to use Asynchronous I/O?

- `async`/`await` restructure code to efficiently multiplex large numbers of I/O operations on a single thread
  - Gives a **very natural programming model when each task is I/O bound**
    - Code to perform asynchronous, non-blocking, I/O is structured very similarly to code that uses blocking I/O operations
    - Runtime can schedule many tasks can run concurrently on a single thread
    - Each task is largely blocked awaiting I/O – little overheads

When to use Asynchronous I/O?

- **`async`**/**`await`** restructure code to efficiently multiplex large numbers of I/O operations on a single thread
  - **Problematic** with blocking operations
    - If a task performs a blocking operation, it locks the entire runtime – all potentially blocking calls must be updated to use asynchronous I/O operations
    - Potential to fragment the library ecosystem
  - **Problematic** with long-running computations
    - Long-running computations starve other tasks of CPU time – runtime only switches between tasks when an asynchronous operation is started
    - Need to insert context switch calls – isn't this just **cooperative multitasking** reimagined?
      - Windows 3.1, MacOS System 7

Performance of Asynchronous I/O

- Do you **really** need asynchronous I/O?
  - Threads are more expensive than `async` functions and tasks in a runtime
  - But threads are not **that** expensive – a properly configured modern machine can run thousands of threads
    - ~2,200 threads running on the Core i5 laptop these slides were prepared on, in normal use
    - Varnish web cache (https://varnish-cache.org): “it’s common to operate with 500 to 1000 threads minimum” but they “rarely recommend running with more than 5000 threads”
    - Unless you’re doing something **very** unusual you can likely just spawn a thread, or use a pre-configured thread pool, and perform blocking I/O – and communicate using channels, **even if this means spawning thousands of threads**
- Asynchronous I/O **can** give a performance benefit
  - But this performance benefit will usually be small
  - Choose asynchronous programming because you prefer the programming style, accepting that it will often not significantly improve performance

Summary

- Blocking I/O
  - Multi-threading → overheads
  - `select()` → complex
  - Coroutines and asynchronous code
- Is it worth it?