Introduction in Asynchronous Computations

by Ruslan Ibragimov,

aka @IRus

Agenda

- Что такое асинхронное программирование
- Зачем нам оно
- Юзкейсы
- Akka, Fibers, Coroutines, Goroutines, и т.д.
- Kotlin 1.1 Coroutines

Асинхронное программирование это ...

This is Sync

```
fun doWork() {
    println("Start work")
    Thread.sleep(1)
    println("Complete work")
}
```

This is Async

```
fun doWork() {
    println("Start work")

    CompletableFuture.supplyAsync {
        println("Start async")
        Thread.sleep(1)
        println("Complete async")
    }

    println("Complete work")
}
```

This is Async

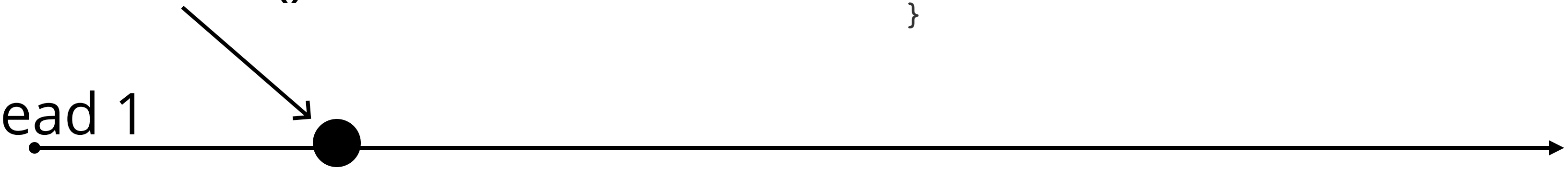
```
fun doWork() {
    println("Start work")

    CompletableFuture.supplyAsync {
        println("Start async")
        Thread.sleep(1)
        println("Complete async")
    }

    println("Complete work")
}
```

```
fun doWork() {
    println("Start work")

    CompletableFuture.supplyAsync {
        println("Start async")
        Thread.sleep(1)
        println("Complete async")
    }

    println("Complete work")
}
```

```
fun doWork() {
    println("Start work")

    CompletableFuture.supplyAsync {
        println("Start async")
        Thread.sleep(1)
        println("Complete async")
    }

    println("Complete work")
}
```

Thread 1

```
fun doWork() {
    println("Start work")

    CompletableFuture.supplyAsync {
        println("Start async")
        Thread.sleep(1)
        println("Complete async")
    }

    println("Complete work")
}
```

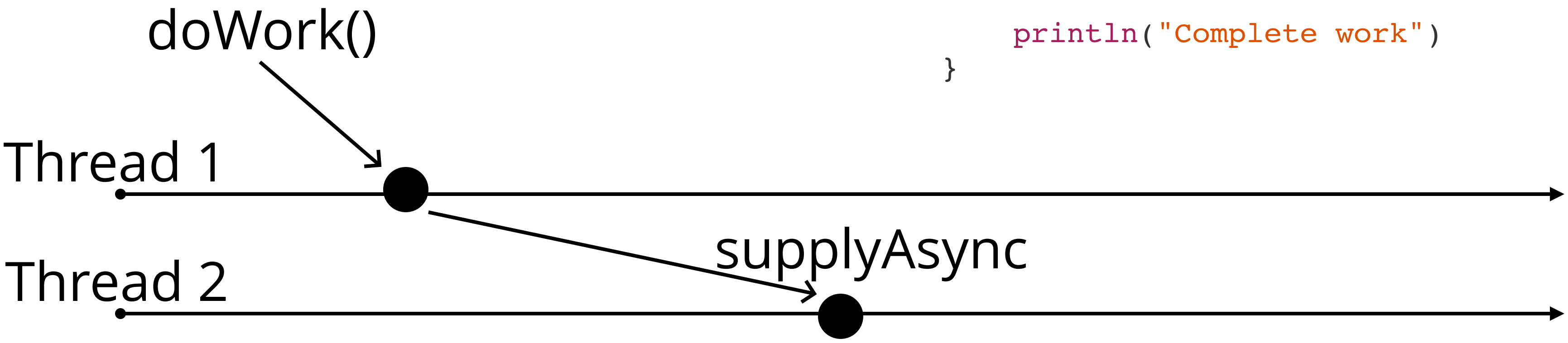


doWork()

Thread 1

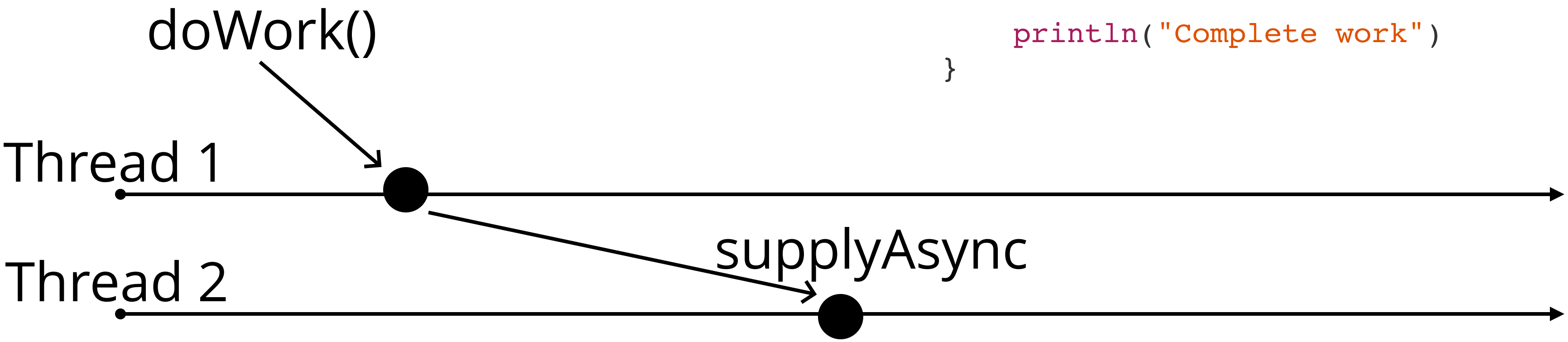
Start Work

```
fun doWork() {
    println("Start work")

    CompletableFuture.supplyAsync {
        println("Start async")
        Thread.sleep(1)
        println("Complete async")
    }

    println("Complete work")
}
```



Start Work

```
fun doWork() {
    println("Start work")

    CompletableFuture.supplyAsync {
        println("Start async")
        Thread.sleep(1)
        println("Complete async")
    }

    println("Complete work")
}
```

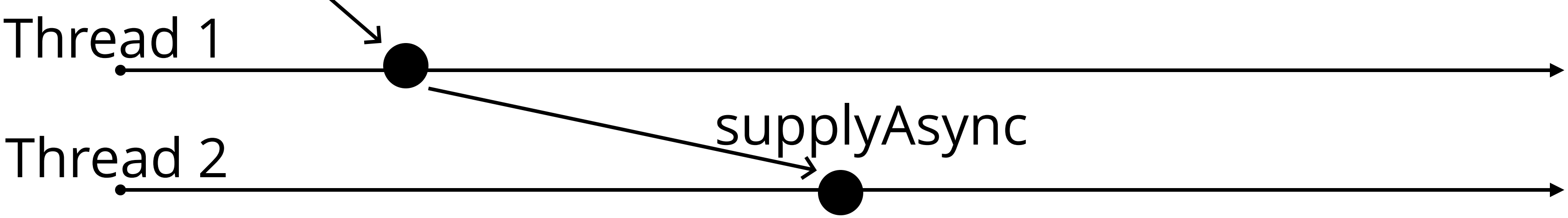


Start Work

Complete Work

```
fun doWork() {
    println("Start work")

    CompletableFuture.supplyAsync {
        println("Start async")
        Thread.sleep(1)
        println("Complete async")
    }

    println("Complete work")
}
```

doWork()



Thread 1

Thread 2

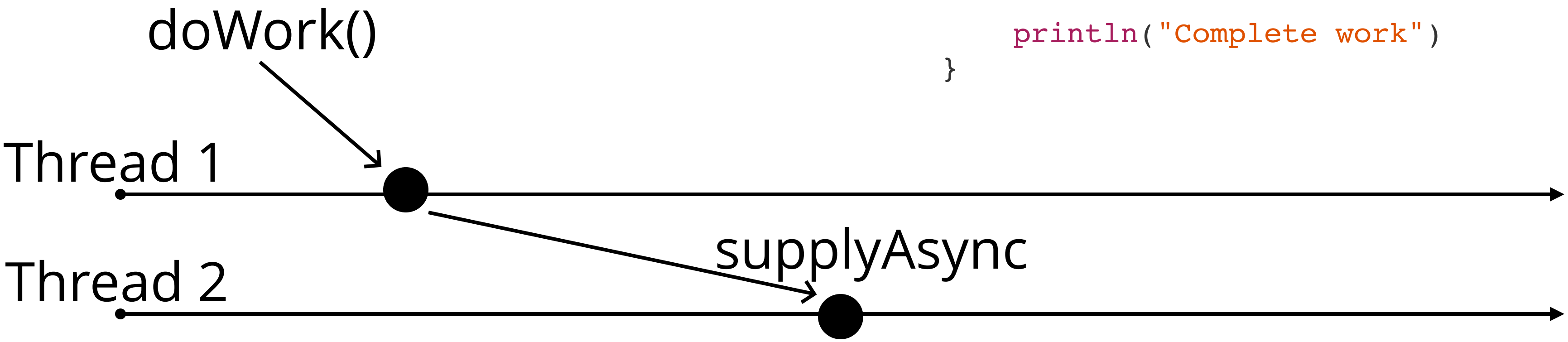
supplyAsync

Start Work

Complete Work

Start async

```
fun doWork() {
    println("Start work")

    CompletableFuture.supplyAsync {
        println("Start async")
        Thread.sleep(1)
        println("Complete async")
    }

    println("Complete work")
}
```



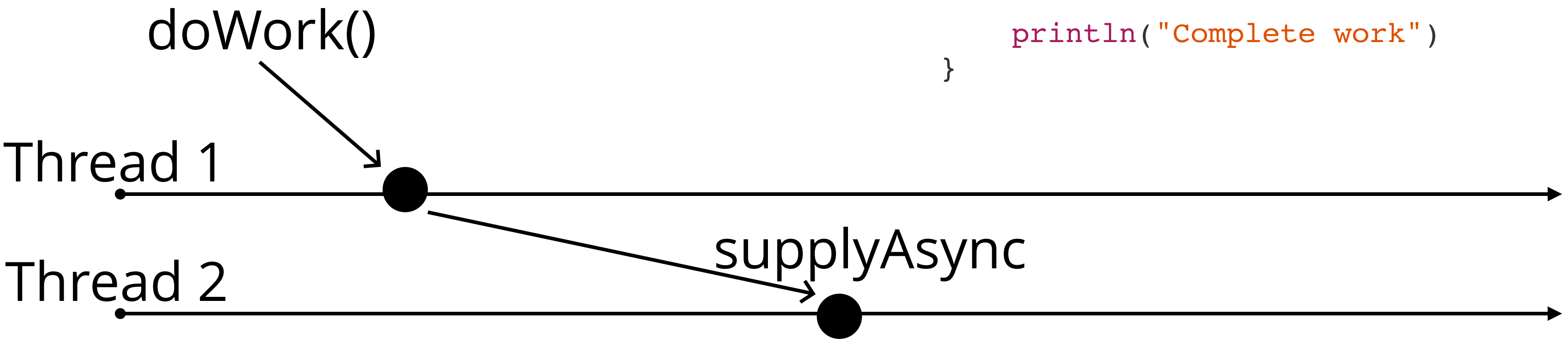
Start Work

Complete Work

Start async

Complete Async

```
fun doWork() {
    println("Start work")

    CompletableFuture.supplyAsync {
        println("Start async")
        Thread.sleep(1)
        println("Complete async")
    }

    println("Complete work")
}
```



doWork()

Thread 1

supplyAsync

Thread 2

Start Work

Complete Work

Start async

Complete Async

Асинхронные действия — действия, выполненные в неблокирующем режиме, что позволяет основному потоку программы продолжить обработку.

```kotlin
fun doWork() {
    println("Start work")

    // Мне все равно когда и как это выполнится,
    // просто выполни это
    CompletableFuture.supplyAsync {
        println("Start async")
        Thread.sleep(1)
        println("Complete async")
    }

    println("Complete work")
}
```

```
fun doWork() {
    println("Start work")

    // Мне все равно когда и как это выполнится,
    // просто выполни это
    CompletableFuture.supplyAsync {
        println("Start async")
        Thread.sleep(1)
        println("Complete async")
    }

    println("Complete work")
}
```

Не сильно то и полезно

```
typealias R = ResponseEntity<String> // Kotlin 1.1

fun doWork() {
    println("Start work")

    val response: ListenableFuture<R> = asyncClient.get()
    response.addCallback(object : ListenableFutureCallback<R> {
        override fun onSuccess(result: R) {
            println("Result: ${result.body}")
        }

        override fun onFailure(ex: Throwable) {
            logger.error("Exception during", ex);
        }
    })

    println("Complete work")
}
```

```
typealias R = ResponseEntity<String> // Kotlin 1.1

fun doWork() {
    println("Start work")

    val response: ListenableFuture<R> = asyncClient.get()
    response.addCallback(object : ListenableFutureCallback<R> {
        override fun onSuccess(result: R) {
            println("Result: ${result.body}")
        }

        override fun onFailure(ex: Throwable) {
            logger.error("Exception during", ex);
        }
    })

    println("Complete work")
}
```

```
typealias R = ResponseEntity<String> // Kotlin 1.1

fun doWork() {
    println("Start work")

    val response: ListenableFuture<R> = asyncClient.get()
    response.addCallback(object : ListenableFutureCallback<R> {
        override fun onSuccess(result: R) {
            println("Result: ${result.body}")
        }

        override fun onFailure(ex: Throwable) {
            logger.error("Exception during", ex);
        }
    })

    println("Complete work")
}
```

```
public interface ListenableFuture<T> extends Future<T> {
       void addCallback(ListenableFutureCallback<? super T> callback);

       void addCallback(
              SuccessCallback<? super T> successCallback,
              FailureCallback failureCallback
       );
}
```

```
typealias R = ResponseEntity<String> // Kotlin 1.1

fun doWork() {
    println("Start work")

    val response: ListenableFuture<R> = asyncClient.get()
    response.addCallback(object : ListenableFutureCallback<R> {
        override fun onSuccess(result: R) {
            println("Result: ${result.body}")
        }

        override fun onFailure(ex: Throwable) {
            logger.error("Exception during", ex);
        }
    })

    println("Complete work")
}
```

```
public interface ListenableFuture<T> extends Future<T> {
       void addCallback(ListenableFutureCallback<? super T> callback);

       void addCallback(
               SuccessCallback<? super T> successCallback,
               FailureCallback failureCallback
       );
}
```

```
typealias R = ResponseEntity<String> // Kotlin 1.1

fun doWork() {
    println("Start work")

    val response: ListenableFuture<R> = asyncClient.get()
    response.addCallback(object : ListenableFutureCallback<R> {
        override fun onSuccess(result: R) {
            println("Result: ${result.body}")
        }

        override fun onFailure(ex: Throwable) {
            logger.error("Exception during", ex);
        }
    })

    println("Complete work")
}
```

```
public interface ListenableFuture<T> extends Future<T> {
        void addCallback(ListenableFutureCallback<? super T> callback);

        void addCallback(
                SuccessCallback<? super T> successCallback,
                FailureCallback failureCallback
        );
}
```

```
function get(url, callback) {
    http.get(url, callback); // async client
}
```

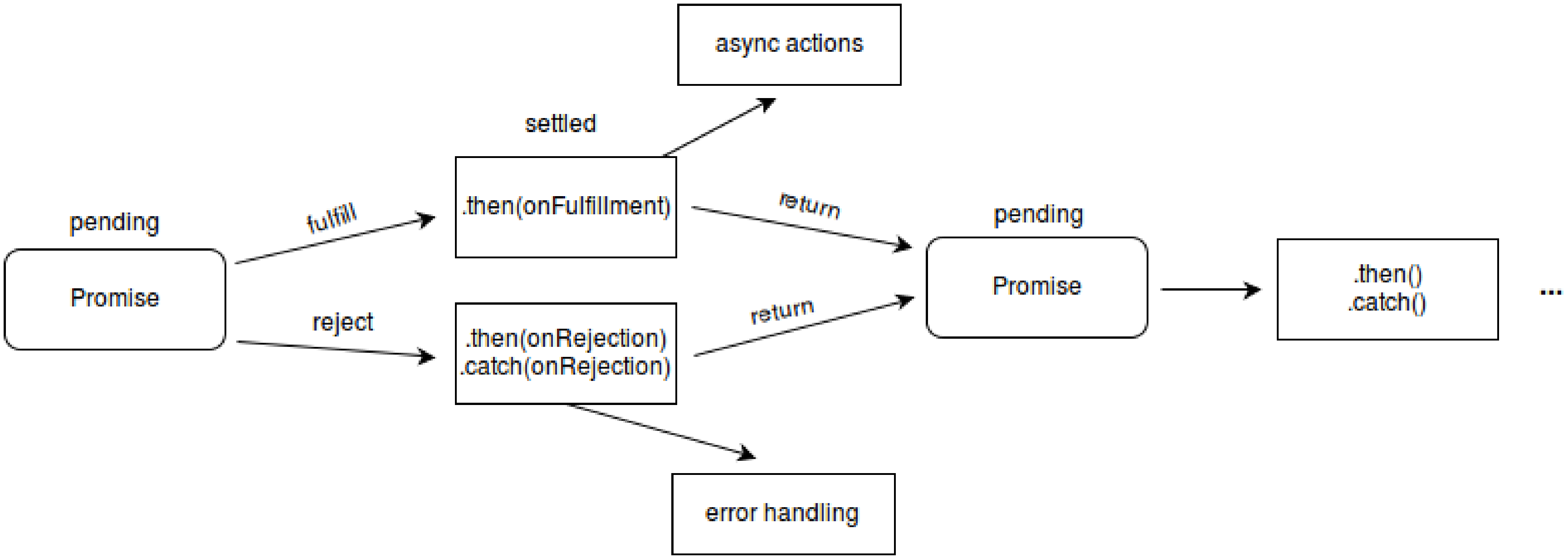
```
function get(url, callback) {
    http.get(url, callback); // async client
}


get('/foo', function (data1, error) {
    get(data1.url, function (data2, error) {
        get(data2.url, function (data3, error) {
            get(data3.url, function (data4, error) {
                console.log(data4);
            });
        });
    });
});
```

```
functi
    ht
}


get('/
    ge
                                             ) {
                                           error) {




    })
});
```

Callback Hell

Futures and promises

```
// Static Methods
Promise.all(iterable)

Promise.race(iterable)

Promise.reject(reason)

Promise.resolve(value)

// Instance Methods
promise.catch(onRejected)

promise.then(onFulfilled, onRejected)
```



```
get('/foo', function (data1, error) {
    get(data1.url, function (data2, error) {
        get(data2.url, function (data3, error) {
            get(data3.url, function (data4, error) {
                console.log(data4);
            });
        });
    });
});
```

```
get('/foo', function (data1, error) {
    get(data1.url, function (data2, error) {
        get(data2.url, function (data3, error) {
            get(data3.url, function (data4, error) {
                console.log(data4);
            });
        });
    });
});
```

```
get('/foo')
    .then(function (data1) {
      return get(data1);
    })
    .then(function (data2) {
      return get(data2);
    })
    .then(function (data3) {
      return get(data3);
    })
    .then(function (data4) {
      console.log(data4);
    });
```

```
get('/foo', function (data1, error) {
    get(data1.url, function (data2, error) {
        get(data2.url, function (data3, error) {
            get(data3.url, function (data4, error) {
                console.log(data4);
            });
        });
    });
});
```

```
get('/foo')
    .then(function (data1) {
      return get(data1);
    })
    .then(function (data2) {
      return get(data2);
    })
    .then(function (data3) {
      return get(data3);
    })
    .then(function (data4) {
      console.log(data4);
    });
```

Функциональная композиция

```
function callback(data, error) {
    ...
}
```

```
function callback(data, error) {
    ...
}
```

```
new Promise(function (resolve, reject) {
  get(url, function (data, error) {
    if (data) {
      resolve(data);
    } else {
      reject(error);
    }
  });
})
```

Problems

- Error Handling
- Control Flow
- Hard to learn

Problems

- Error Handling
- Control Flow
- Hard to learn

```
get('/foo')
    .then(function (data1) {
      return get(data1);
    })
    .then(function (data2) {
      return get(data2);
    })
    .then(function (data3) {
      // get here data1?
      // should introduce variable :(
      return get(data3);
    })
    .then(function (data4) {
      console.log(data4);
    });
```

Async/Await

- C#
- Scala
- JavaScript
- Python
- ...

```
get('/foo')
    .then(function (data1) {
      return get(data1);
    })
    .then(function (data2) {
      return get(data2);
    })
    .then(function (data3) {
      return get(data3);
    })
    .then(function (data4) {
      console.log(data4);
    });
```

```
get('/foo')
    .then(function (data1) {
      return get(data1);
    })
    .then(function (data2) {
      return get(data2);
    })
    .then(function (data3) {
      return get(data3);
    })
    .then(function (data4) {
      console.log(data4);
    });
```

```
async function doWork() {
    const data1 = await get('/foo');
    const data2 = await get(data1);
    const data3 = await get(data2);
    const data4 = await get(data3);
    console.log(data4);
}
```

```
get('/foo')
    .then(function (data1) {
      return get(data1);
    })
    .then(function (data2) {
      return get(data2);
    })
    .then(function (data3) {
      return get(data3);
    })
    .then(function (data4) {
      console.log(data4);
    });
```

```
async function doWork() {
    const data1 = await get('/foo');
    const data2 = await get(data1);
    const data3 = await get(data2);
    const data4 = await get(data3);
    console.log(data4);
}
```

```
get('/foo')
    .then(function (data1) {
      return get(data1);
    })
    .then(function (data2) {
      return get(data2);
    })
    .then(function (data3) {
      return get(data3);
    })
    .then(function (data4) {
      console.log(data4);
    });
```

```
async function doWork() {
    const data1 = await get('/foo');
    const data2 = await get(data1);
    const data3 = await get(data2);
    const data4 = await get(data3);
    console.log(data4);
}
```

Error handling?

Error handling?

```
async function doWork() {
    try {
        const data1 = await get('/foo');
        const data2 = await get(data2);
        const data4 = await get(data3);
        console.log(data4);
    } catch (e) {
      // deal with it
    }
}
```

Pros

- no explicit callbacks
- no future combinations
- looks like sync code

Should i write Async Code?

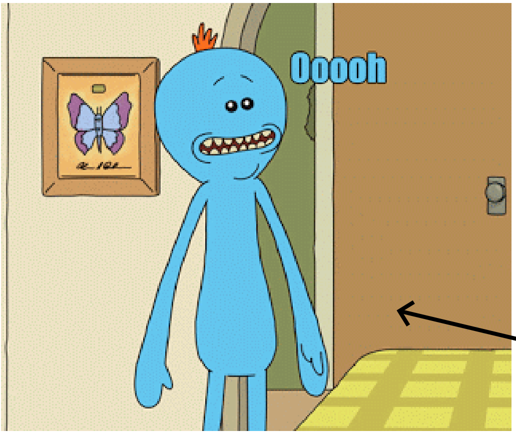
Use Cases

- UI
- Backend

User Interface

Browser:
1 Thread per Tab :(

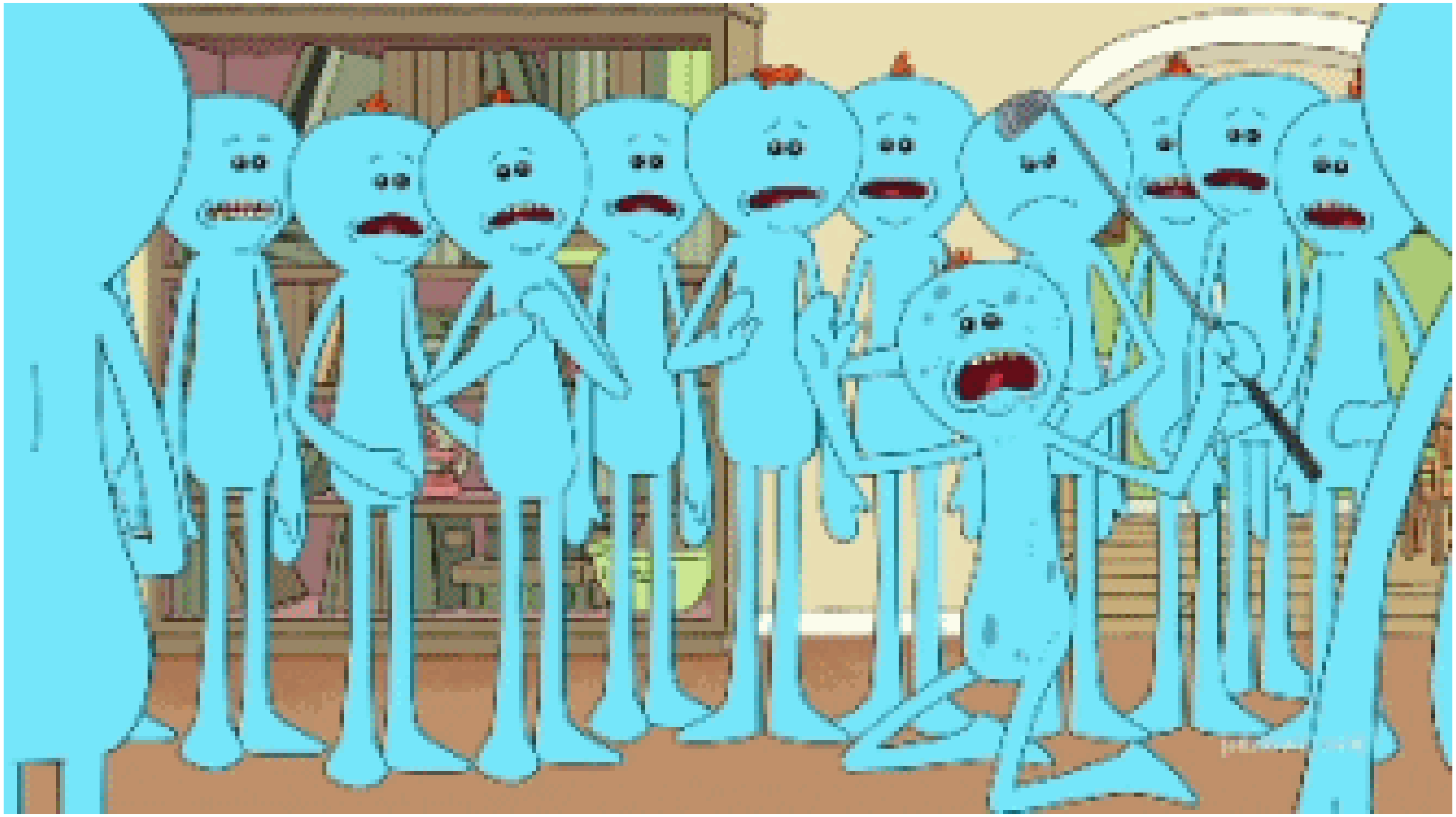
Any sync request/action = freeze of UI



Мисикс

Backend

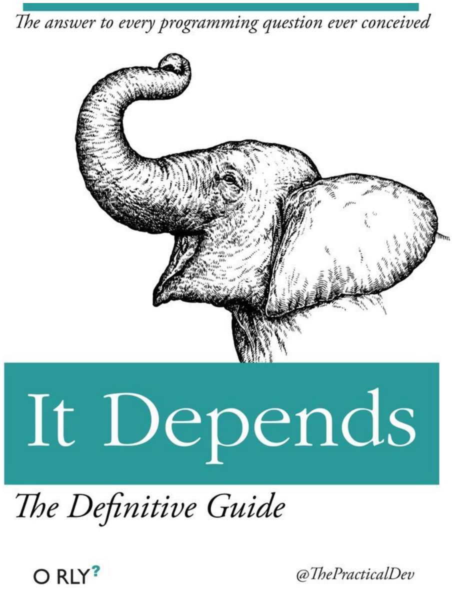
Async? Nah! I can create more threads!



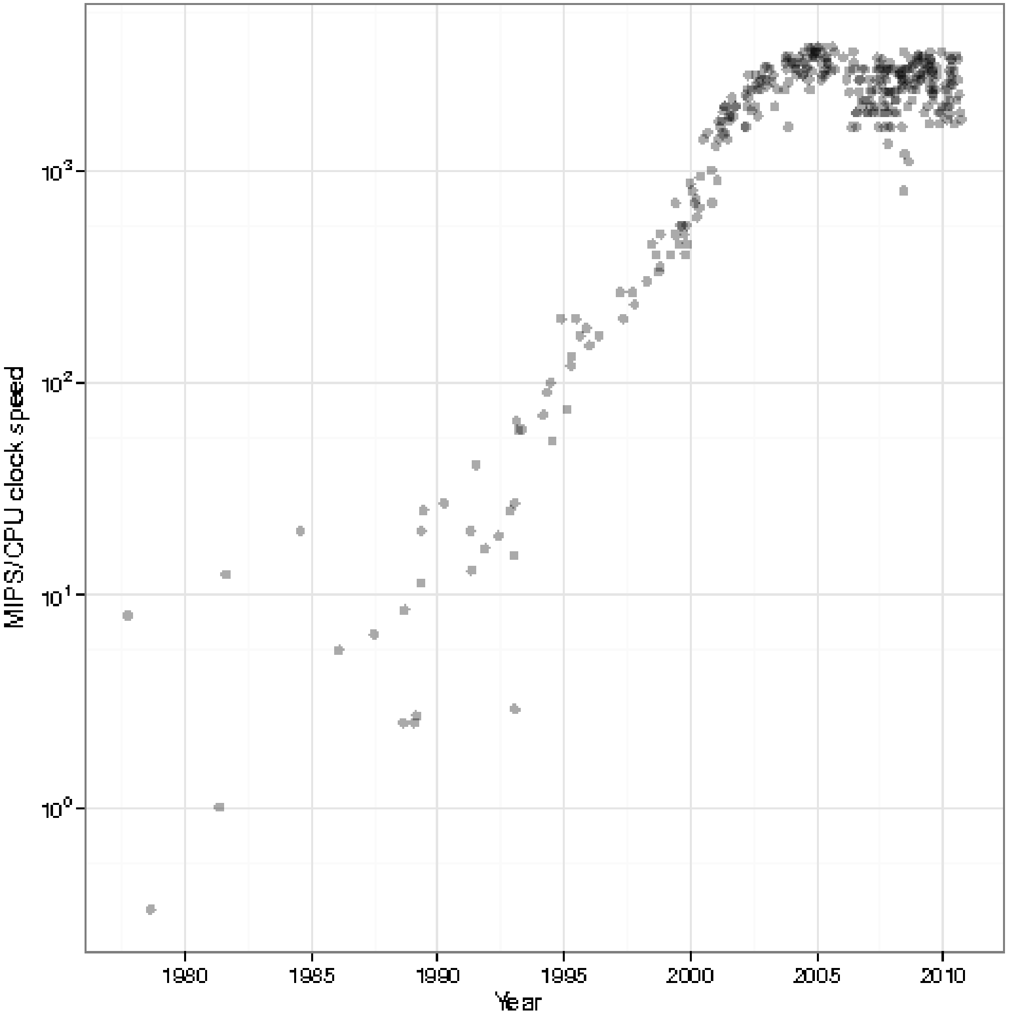
Thread per user not efficient*

Thread per user not efficient*

*It depends of course



CPUs

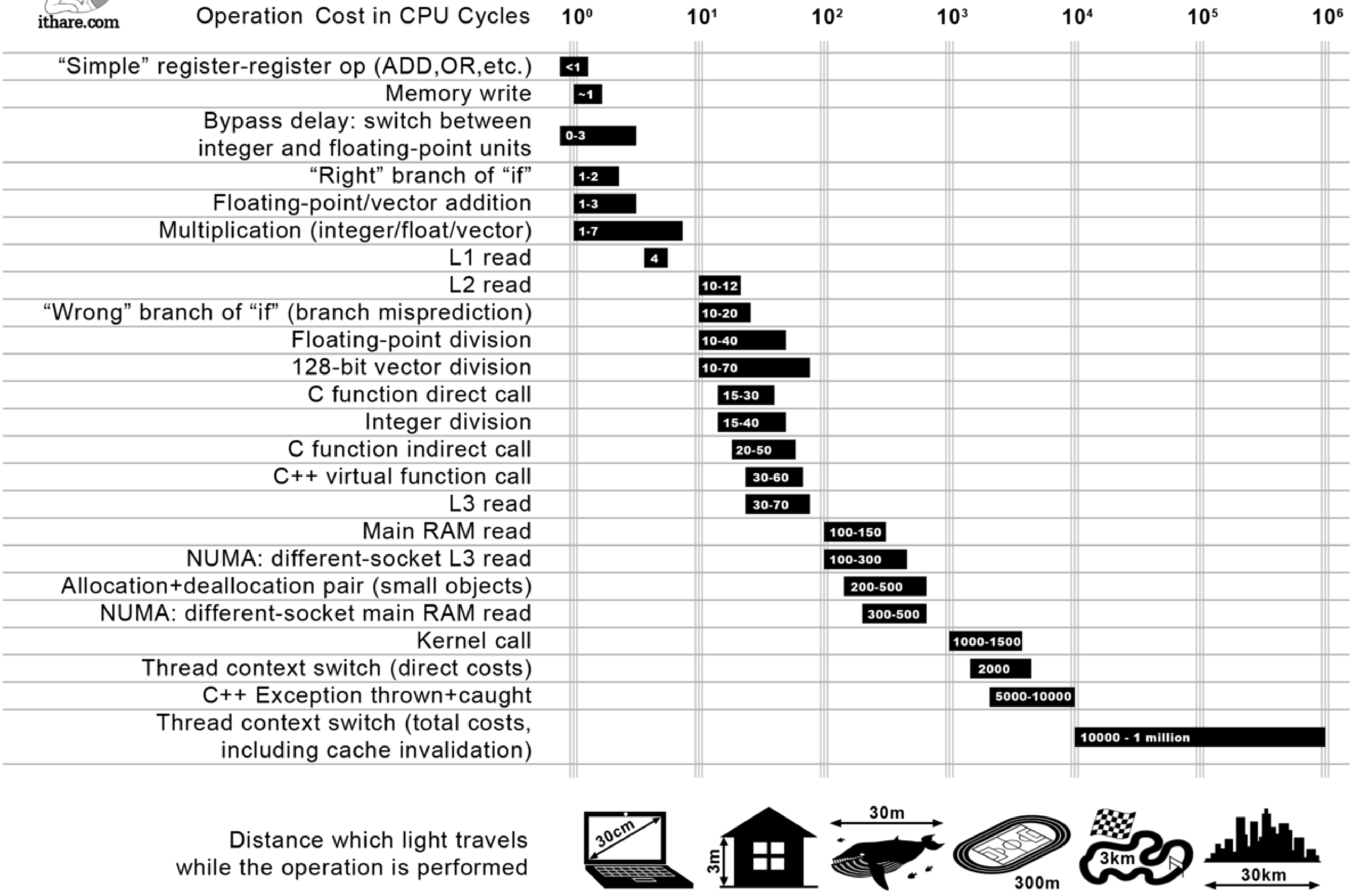


MIPS/CPU clock speed
10^3
10^2
10^1
10^0
1980
1985
1990
1995
2000
2005
2010
Year

Okay CPU speed is limited, and what?

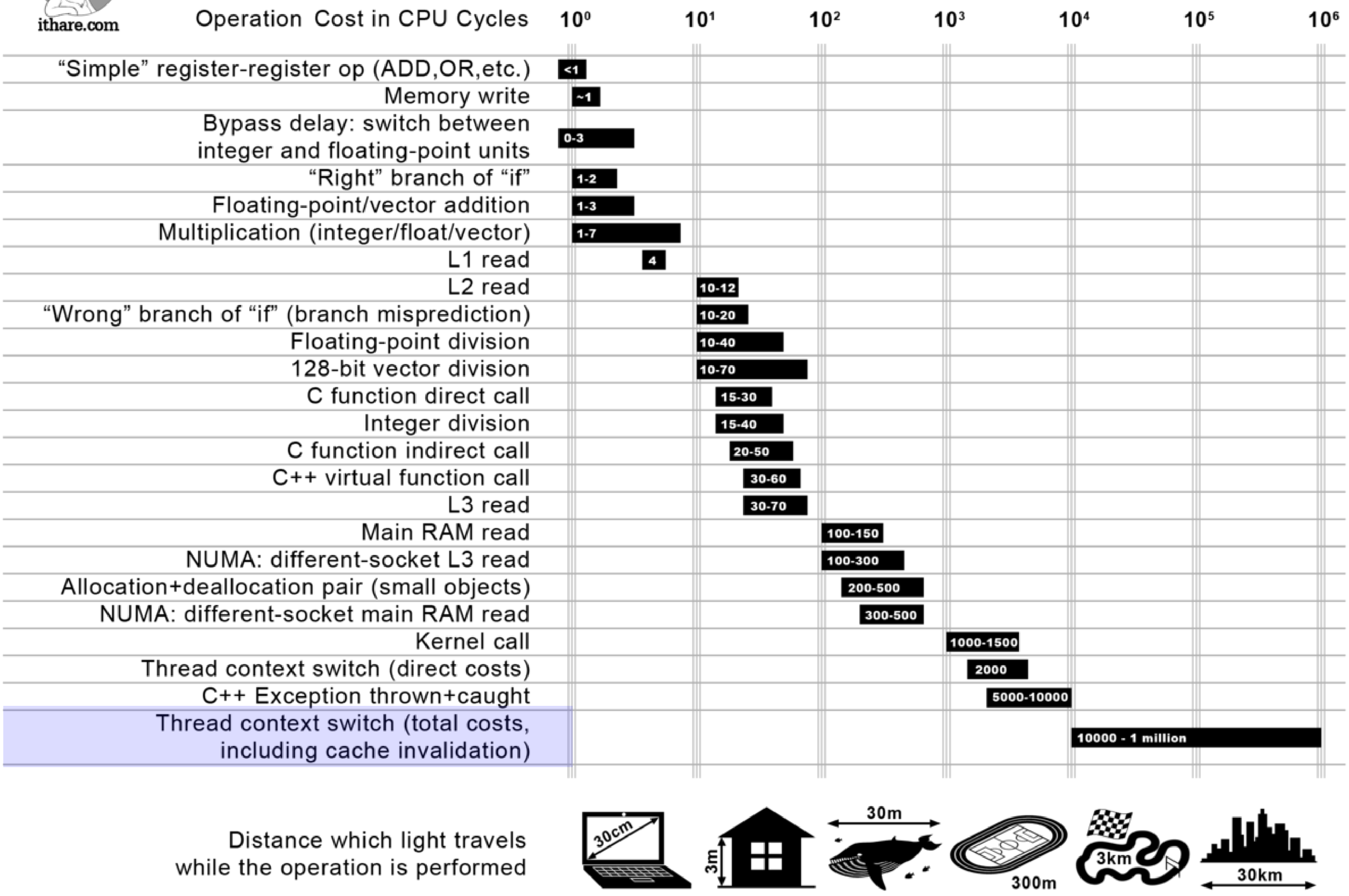
ithare.com

Not all CPU operations are created equal



ithare.com

Not all CPU operations are created equal

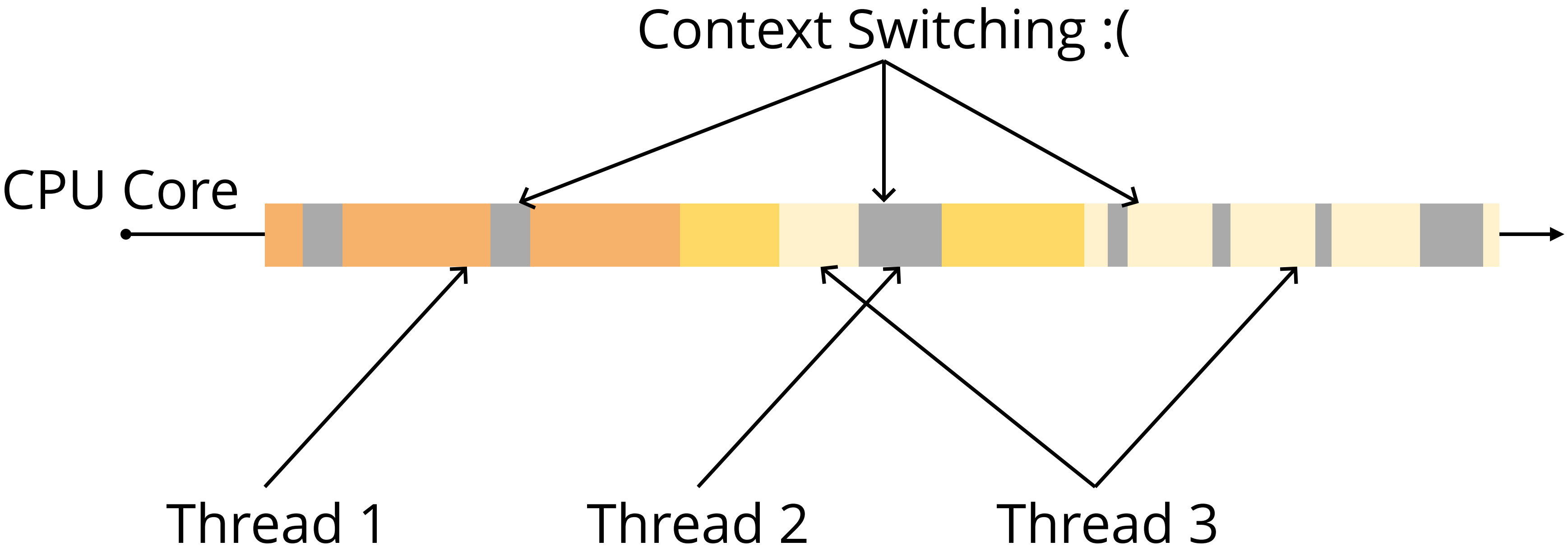


A lot threads = RAM and CPU consumption

What can we do?

Typical Web Server

OS: Preemption Multi Tasking



Alternative:

Non-Blocking APIs

Polling or Hardware support (interrupts and DMA)

Async in Java

Async in Java

- NIO (2011) - File, Networking

Async in Java

- NIO (2011) - File, Networking
- Servlet 3.0 - Networking

Async in Java

- NIO (2011) - File, Networking
- Servlet 3.0 - Networking
- Async JDBC (Java One: JDBC Next)

Async in Java

- NIO (2011) - File, Networking
- Servlet 3.0 - Networking
- Async JDBC (Java One: JDBC Next)
- Java 9 - Flow APIs :)

Best for:

- Latency (Network, Disk)
- Stateful Connections (Web Sockets Sample)
- No Choice (Go, Browser, Node.JS, etc)

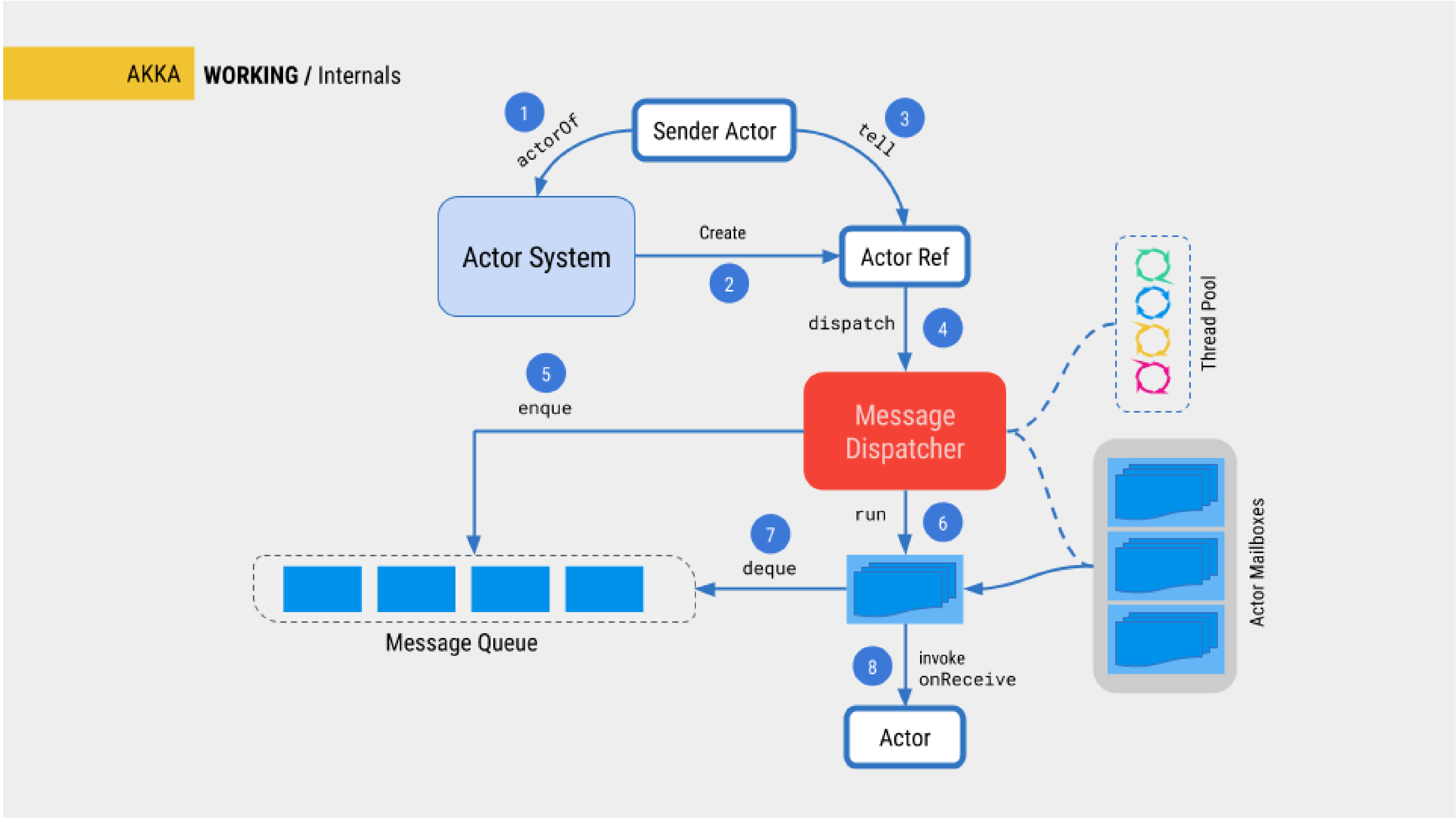
What's the difference?

- Akka
- Fibers
- Green Threads
- Java Flow
- Coroutines
- Goroutines
- Reactive Streams

Akka is

message-based and asynchronous

Akka

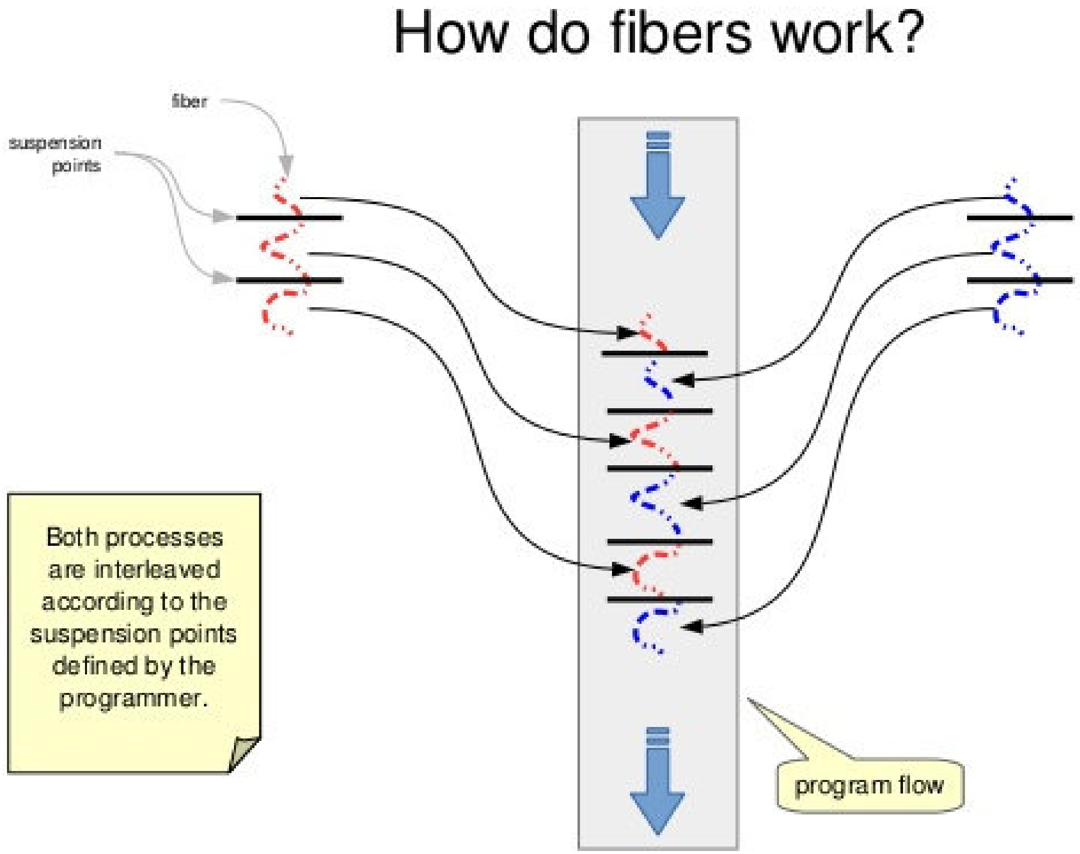

AKKA
WORKING / Internals
1
actorOf
Sender Actor
3
tell
Actor System
Create
2
Actor Ref
dispatch
4
Thread Pool
5
enque
Message Dispatcher
run
6
7
deque
Actor Mailboxes
Message Queue
8
invoke onReceive
Actor

Fibers (Green Threads) is lightweight threads

Cooperative multitasking, also known as non-preemptive multitasking

stackful

Fibers



How do fibers work?

Both processes are interleaved according to the suspension points defined by the programmer.

Goroutines

Basicly is Fibers in Go

Java Flow

Almost the same as Quasar Fiber's

Coroutines is

suspendable computations

Reactive Streams

On of key parts: Back pressure

*Kotlin Coroutines can be used with RS

Kotlin Coroutines

C# - Task

JS - Promise

Scala - Promise

Kotlin - Whatever

```
fun startLongAsyncOperation(v: Int) = CompletableFuture.supplyAsync {
    Thread.sleep(1000)
    "Result: $v"
}
```

Promise/Future

```kotlin
fun startLongAsyncOperation(v: Int) = CompletableFuture.supplyAsync {
    Thread.sleep(1000)
    "Result: $v"
}
```

```
val result = async {
    (1..5).map {
        await(startLongAsyncOperation(it))
    }.joinToString("\n")
}

println(result.get())
// Result: 1
// Result: 2
// Result: 3
// Result: 4
// Result: 5
```

```
val result = async {
    (1..5).map {
        await(startLongAsyncOperation(it))
    }.joinToString("\n")
}

println(result.get())
// Result: 1
// Result: 2
// Result: 3
// Result: 4
// Result: 5
```

```
val result = async {
    (1..5).map {
        await(startLongAsyncOperation(it))
    }.joinToString("\n")
}

println(result.get())
// Result: 1
// Result: 2
// Result: 3
// Result: 4
// Result: 5
```

```
val result = async {
    (1..5).map {
        await(startLongAsyncOperation(it))
    }.joinToString("\n")
}

println(result.get())
// Result: 1
// Result: 2
// Result: 3
// Result: 4
// Result: 5
```

```
fun <T> async(
   coroutine c: FutureController<T>.() -> Continuation<Unit>
): CompletableFuture<T> {
    val controller = FutureController<T>(continuationWrapper)
    c(controller).resume(Unit)
    return controller.future
}
```

```
async {
    (1..5).map {
        await(startLongAsyncOperation(it))
    }.joinToString("\n")
}
```

```
async {
    (1..5).map {
        await(startLongAsyncOperation(it))
    }.joinToString("\n")
}
```

```
suspend fun <V> await(f: CompletableFuture<V>, machine: Continuation<V>) {
    f.whenComplete { value, throwable ->
        wrapContinuationIfNeeded {
            if (throwable == null)
                machine.resume(value)
            else
                machine.resumeWithException(throwable)
        }
    }
}
```

```
suspend fun <T> FutureController<T>.await(
    future: ListenableFuture<T>,
    machine: Continuation<T>
) {
    future.addCallback(object : ListenableFutureCallback<T> {
        override fun onSuccess(result: T) {
            machine.resume(result)
        }

        override fun onFailure(ex: Throwable) {
            machine.resumeWithException(ex)
        }
    })
}
```

Future

- Spring 5 - mainstream meets Reactive Streams
- JDBC Next FTW!
- Java 9 Flow APIs (aka Reactive Streams)

References

- kotlin-coroutines
- SE-Radio Episode 267: Jürgen Höller on Reactive Spring and Spring 5.0
- Andrey Breslav: Kotlin Coroutines, JVMLS 2016
- reactive-streams-jvm

- Slides on https://bkug.by/
- Join Kotlin chat
- Join Kotlin Community chat
- Follow us on Twitter @BelarusKUG