Асинхронное программирование

with Monix

Контакты

Лекторы

- Платонова Наталья; **n.platonova@tinkoff.ru**
- Кобенко Михаил; **m.kobenko@tinkoff.ru**

Группа в Telegram

- https://goo.gl/Aq3Ntx

Цели занятия

- Термины и проблемы
- Scala Future для самых маленьких
- Monix протягивает руку помощи

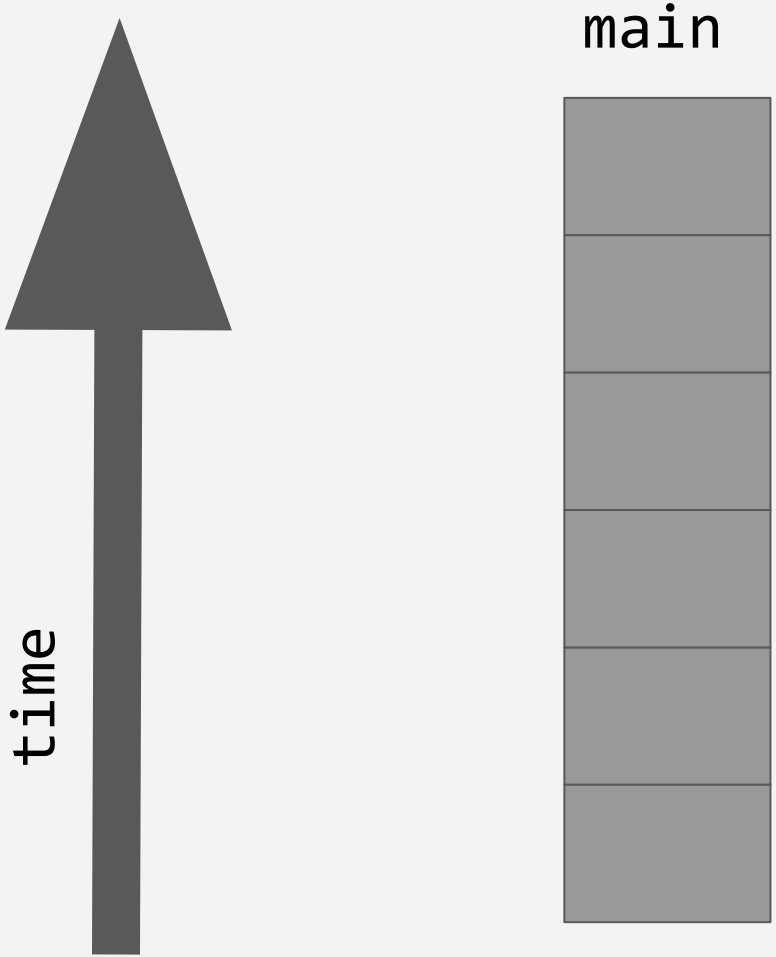
Concurrency

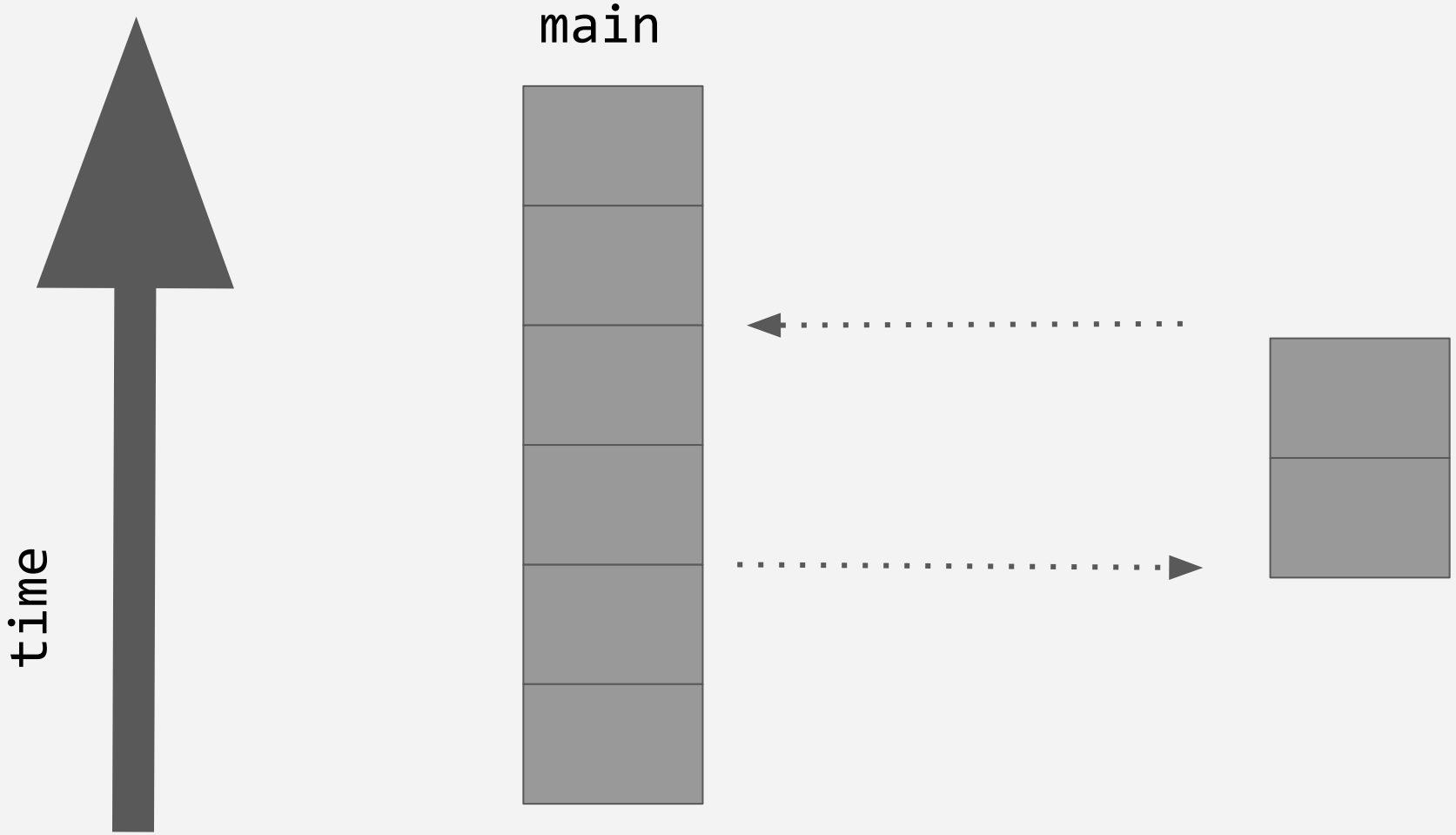
Parallelism

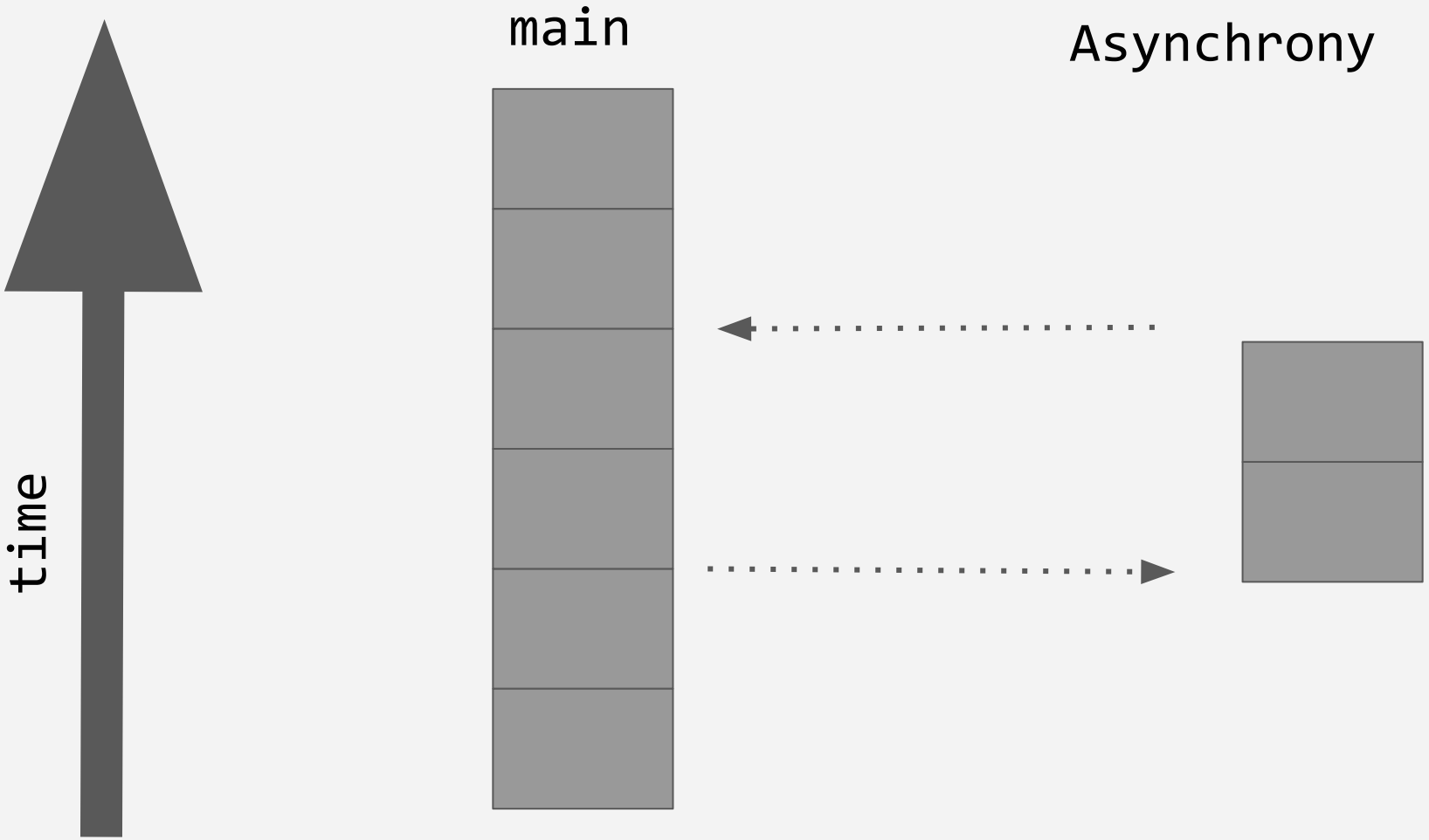
Multithreading

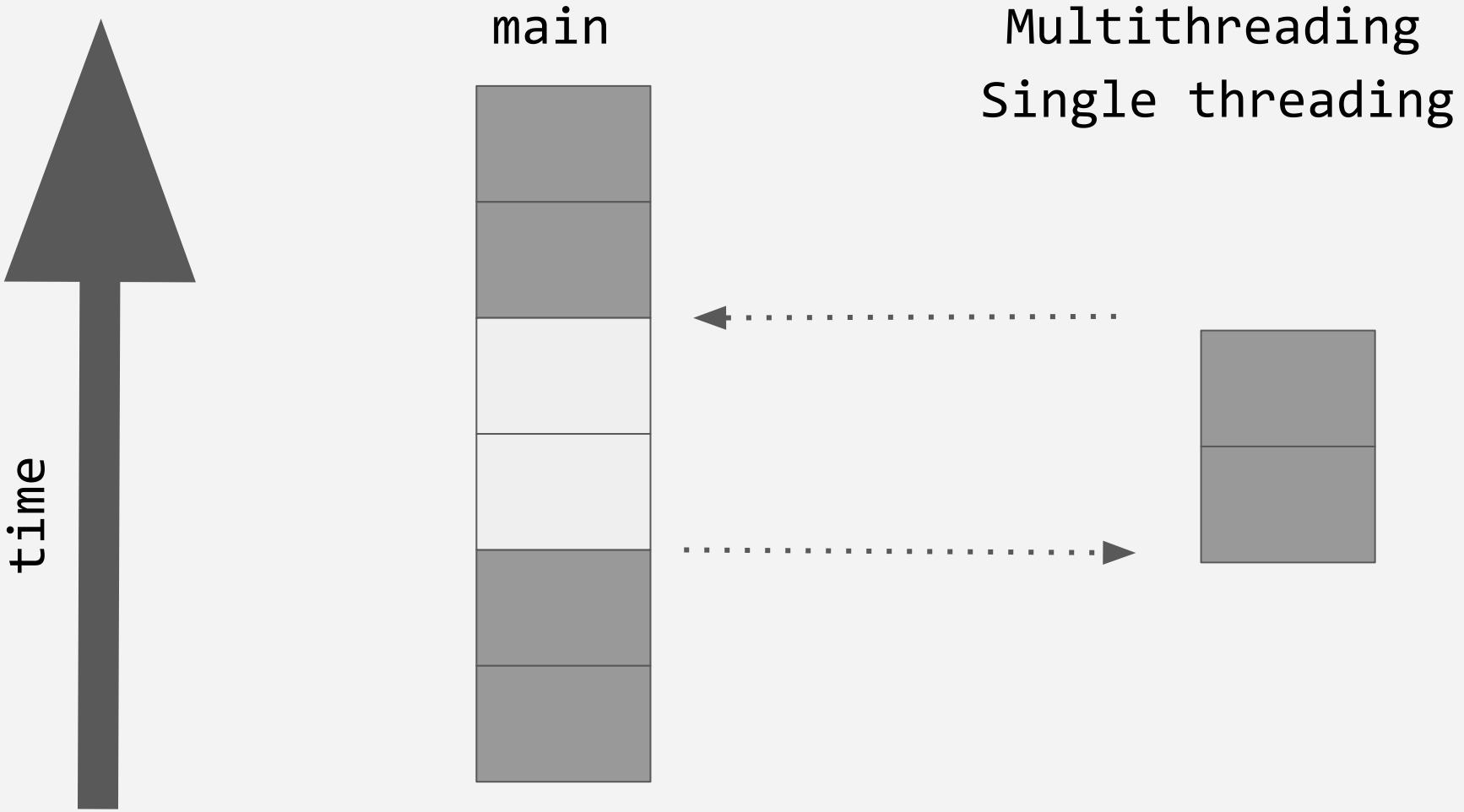
Asynchrony

Single threading

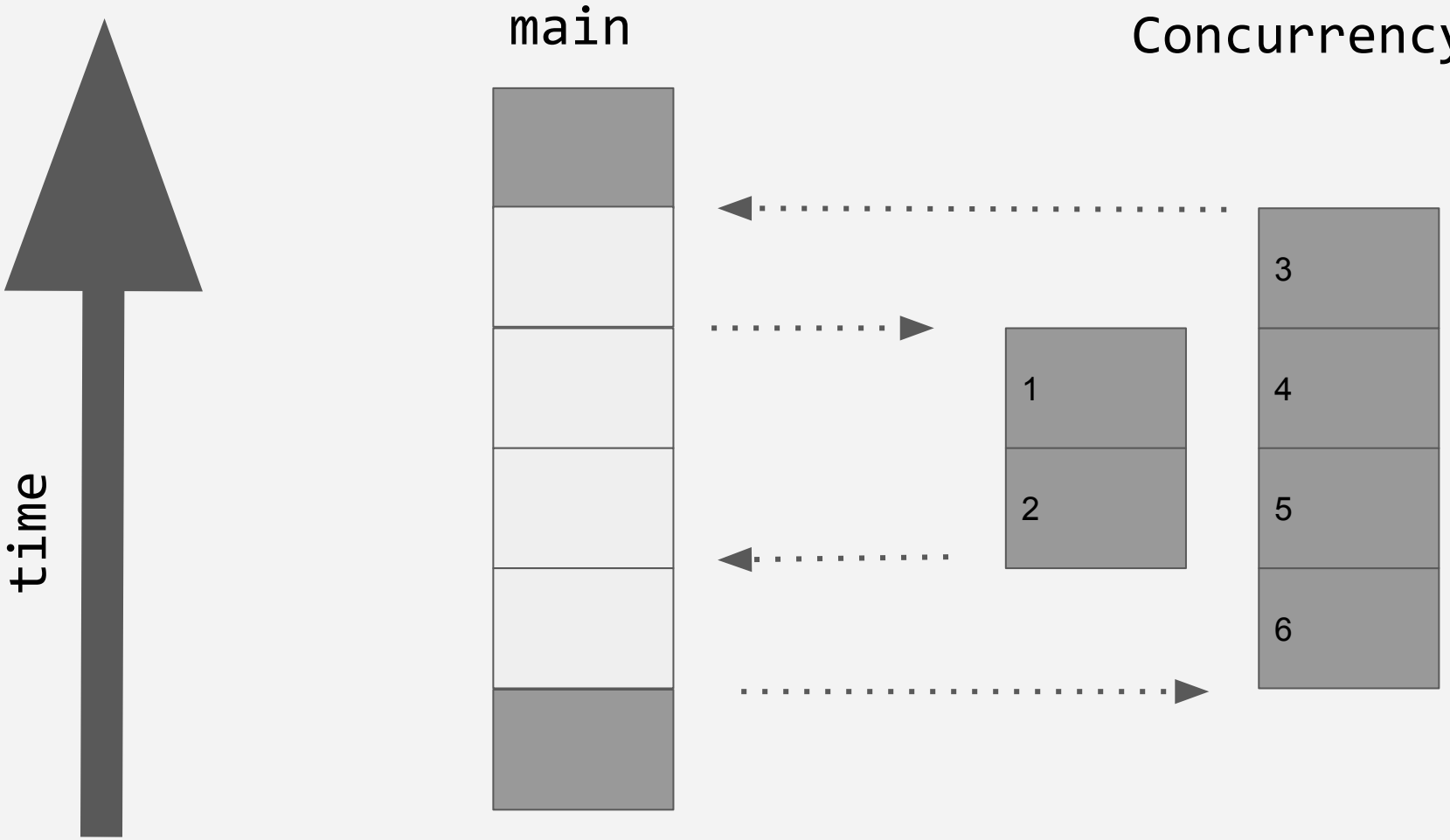

main
time

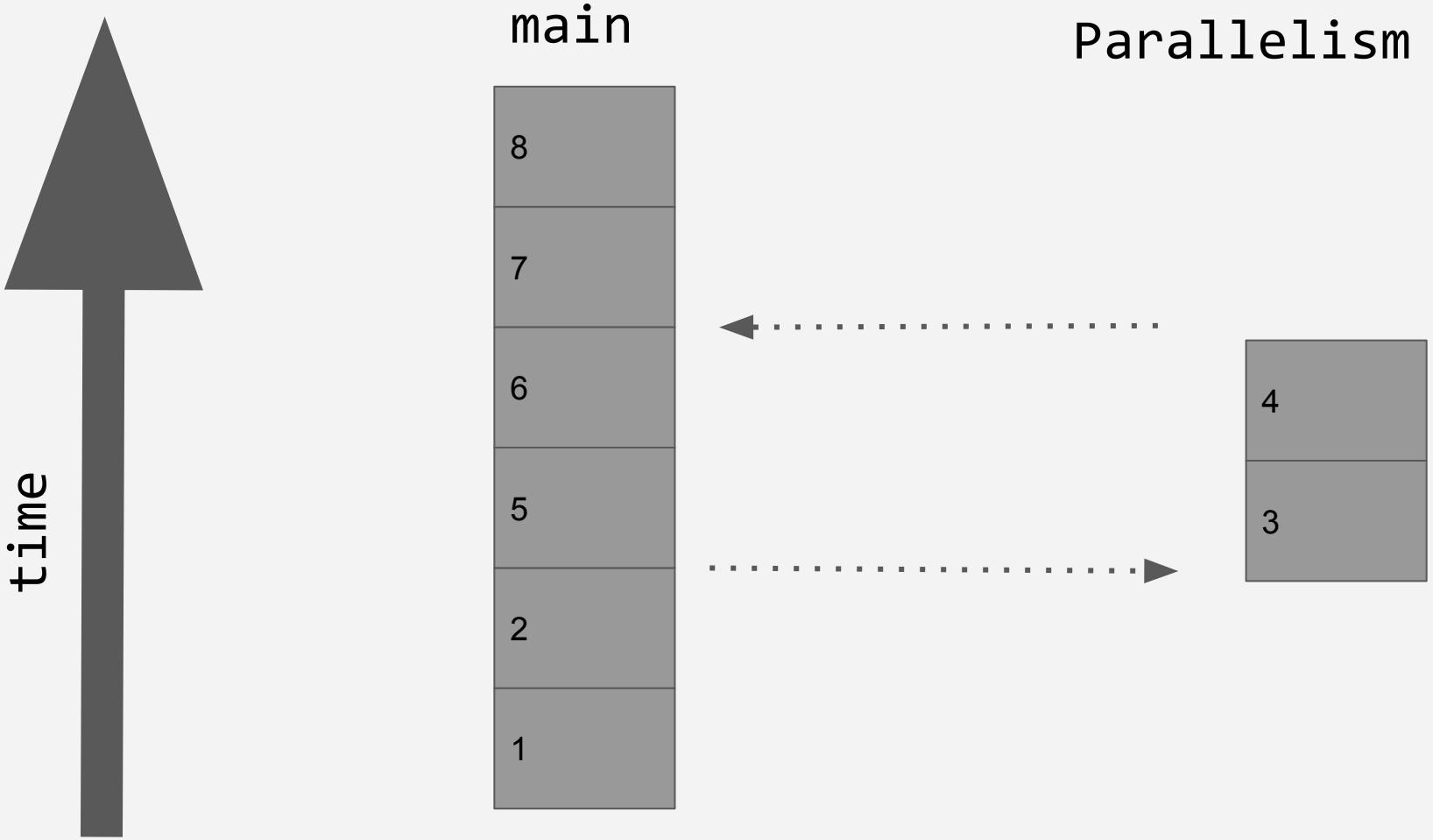

main
time

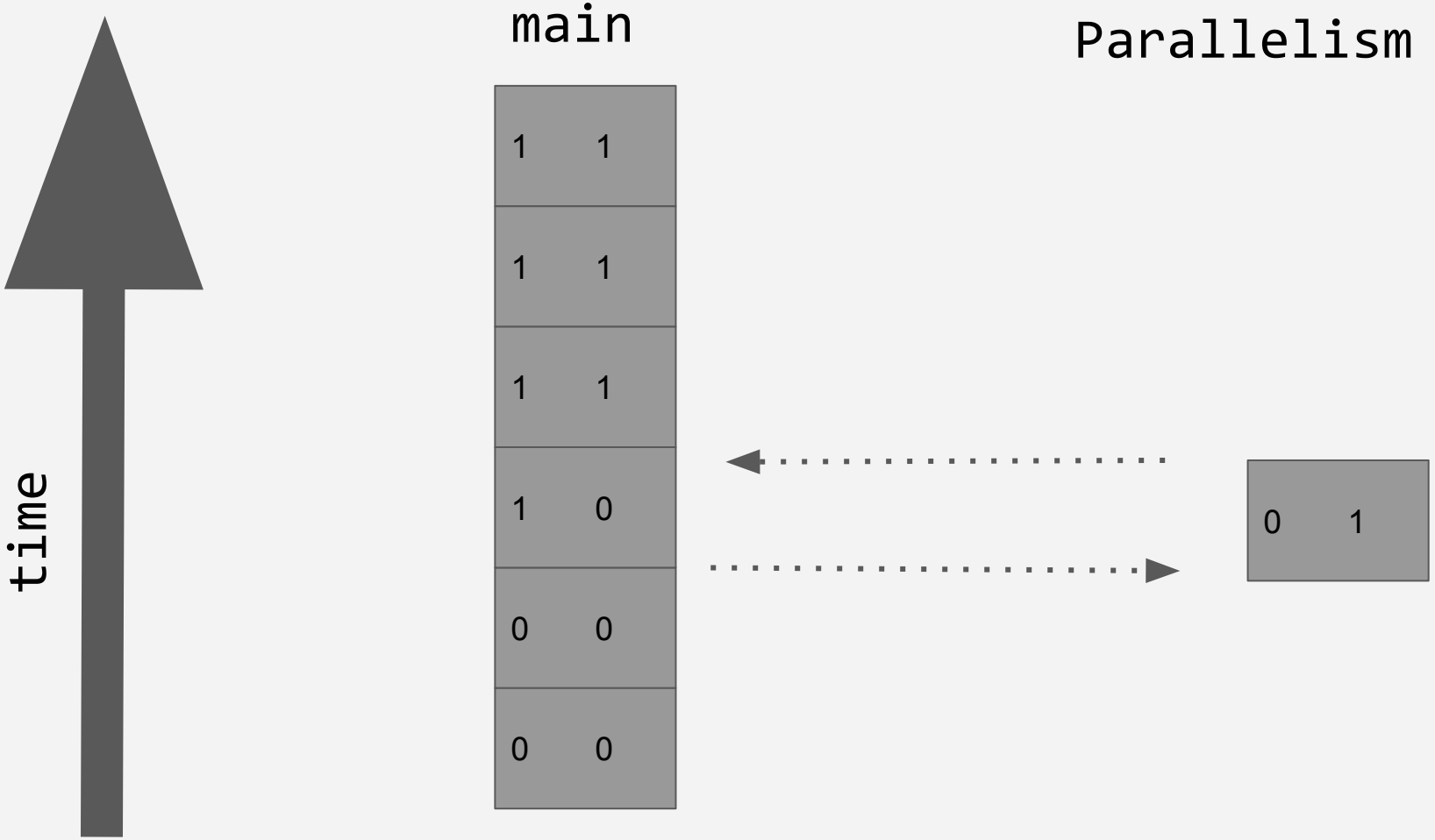

main
Asynchrony
time


main
Multithreading
Single threading
time


main
Cuncurrency
time
1
2
3


main
Concurrency
time
1
2
3
4
5
6


main
Parallelism
time
8
7
6
5
2
1
4
3


main
Parallelism
time
1 1
1 1
1 1
1 0
0 0
0 0
0 1

Асинхронность

1. Что хотим?
2. Чего не хотим?

```
def compute[A](): Async[A] = ???
```

```
type Callback[A] = (Try[A] => Unit)

type Async[A] = Callback[A] => Unit
```

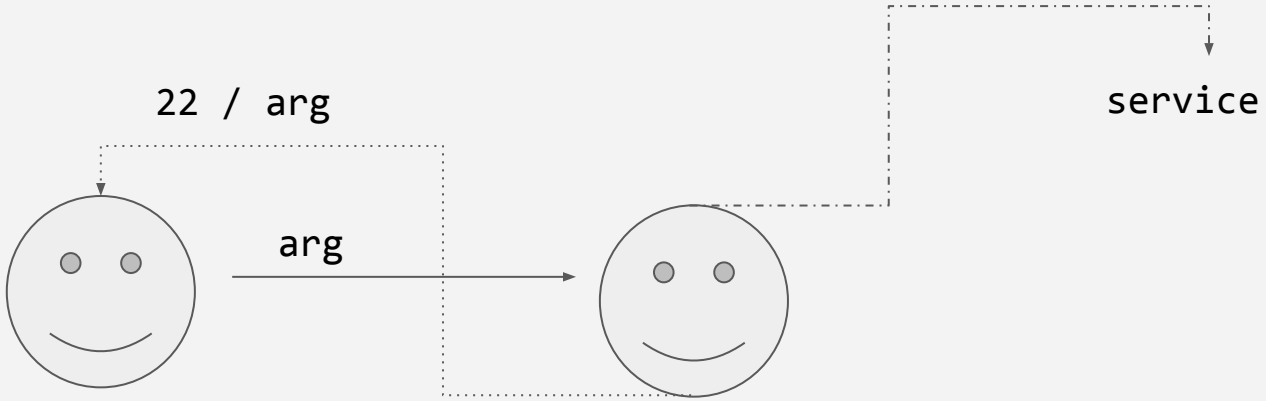
```
type Async[A] = (Try[A] => Unit) => Unit

def service(arg: Int): Async[Int] = cb => cb(Try(22 / arg))
```

```
type Callback[A] = (Try[A] => Unit) => Unit

def service(arg: Int): Async[Int] = cb => cb(Try(22 / arg))


service(44)(println)
```

```scala
type Callback[A] = (Try[A] => Unit) => Unit

def service(arg: Int): Async[Int] = cb => cb(Try(22 / arg))


service(44)(println)

def giveSomething(arg: Int): Unit = ???
```

```scala
type Callback[A] = (Try[A] => Unit) => Unit

def service(arg: Int): Async[Int] = cb => cb(Try(22 / arg))


service(44)(println)

def giveSomething(arg: Int): Unit = ???

service(33) {

 case Success(x) => giveSomething(x)

 case Failure(t) => throw new Exception(t)

}
```

```
def service(arg: Int): Async[Int] = cb => cb(Try(22 / arg))

Int => CallBack[Int] => Unit
```



Asynchrony illusion

```
def await[A](fa: Async[A]): A
```

make semicolon (;) great again

Проблемки Semicolon driven programming

1. Авейтится - это еще не значит что мы занимаемся распределенным программированием
2. Ваша платформа имеет меньше потоков, чем вы думаете
3. Вы убиваете параллелизм (Закон Амдала)

Закон Амдала

1. 20 часов вычислений
2. 1 час не может быть пропаралеллизирован
3. 19 часов могут быть пропаралеллизированы
4. Максимальным ускорением будет 20x, не имеет значения сколько ядер вы имеете
5. С await вы вообще теряете все

Execution platforms

1. 1:1 поток на уровне ядра (например в JVM) где блокировки проблематичны
2. M:N модель с зелеными потоками (Haskell, Golang, Erlang), где можно «блокироваться»
3. M:1 где у вас есть только один поток выполнения

```
def tweets(text: String): Int = ???

def fb(text: String): Int = ???

def totalMentions(text: String): Int = fb(text) + tweets(text)
```

```
def tweets(text: String): Async[Int] = ???

def fb(text: String): Async[Int] = ???

def totalMentions(text: String): Async[Int] = fb(text) + tweets(text)
```

```
def totalMentions(text: String): Async[Int] = {
cb => tweets(text) {
  tTry =>
    fb(text) {
      fTry =>
        (tTry, fTry) match {
          case (Success(tTry), Success(fTry)) => cb(Success(fTry + tTry))
          case _ => cb(Failure(new Exception("Failed")))
        }
    }
}
}
```

```
def tweets(text: String): Async[Int] = ???

def fb(text: String): Async[Int] = ???

def instagram(text: String): Async[Int] = ???

def vk(text: String): Async[Int] = ???

def reddit(text: String): Async[Int] = ???

(...)
```

Асинхронность + параллелизм

```
type Async[A] = (A => Unit) => Unit

def map2[A,B,C](x: Async[A], y: Async[B])(f: (A,B) => C): Async[C] = {
 cb =>
   var a: A = _
   var b: B = _
   x apply{a: A =>
     //(...) что-то веселое)
     }
   y apply{b: B =>
     //(...) что-то веселое)
   }
}
```

Проблема с stackoverflow

```
def map2[A,B,C](x: Async[A], y: Async[B])(f: (A,B) => C): Async[C]

def sequence[A](jobs: List[Async[A]]): Async[List[A]]
```

```
def async1: Async[Int] = cb =>
(Success(1))

async1 { x1 =>
   async1 { x2 =>
       async1 { x3 =>
     println(Thread.currentThread()
         .getStackTrace
         .length)
     }
   }
 }
}
```

```
async1 { x1 =>
   async1 { x2 =>
       async1 { x3 => {
         async1 { x4 =>
println(Thread.currentThread()
             .getStackTrace
             .length)
         }
       }
     }
   }
}
```

Пересечение асинхронных границ

```
import scala.concurrent.ExecutionContext.Implicits.global

def async1: Async[Int] = cb => global.execute() => cb(Success(1))

async1 { x1 =>
   async1 { x2 =>
     async1 { x3 => {
       println(Thread.currentThread()
         .getStackTrace
         .length)
     }
   }
 }
}
```

Scala Future

Nondeterminism

```
def getDataFromDB(id: Long): Future[CSV] //csv ~2GB
def aggregate(input: CSV): Future[Int]

val userList: List[Long] = ??? // ~100 clients
val storeJobs: List[Future[Int]] = userList.map { user => getDataFromDB(user)
 .flatMap(csvInput => aggregate(csvInput))
}

val batchedJob: Future[List[Int]] = Future.sequence(storeJobs)
Await.ready(batchedJob, 8.hours)
```

Jumping threads

```
def pureData1Transf(input: CSV): Seq[Seq[Int]] = ???
def pureData2Transf(intput: Seq[Seq[Int]]): Seq[Int] = ???
def pureData3Transf(input: Seq[Int]): Int

Future(pureData1Transf(csv))
 .map(pureData2Transf)
 .map(pureData3Transf)

Future{
 val pdr1 = pureData1Transf(csv)
 val pdr2 = pureData2Transf(pdr1)
 pureData3Transf(pdr2)
}
```

Правила которым лучше следовать

1. Возвращать чистые вычисления
2. Не загрязнять код интерфейсами с функциями, возвращающими Future
3. Избегать асинхронности, насколько это возможно
4. Использовать другие тулзы

Получаем high performance с параллелизмом

1. Желать параллелизма насколько это возможно
2. Помним о ядрах
3. Большое количество потоков приведет к частым переключениям контекста

Как масштабировать количество потоков?

Глядя на глобальный EC

- глобальный EC поддерживается ForkJoinPool
- построение потока контролируется специализированной ThreadFactory
- В основном, Runtime.getRuntime.availableProcessors покажет сколько тредов будет создано
- Возможно создание дополнительных потоков до 256

Параллелизм и блокировки

Алгоритм называется неблокирующим, если ошибки или прекращения работ некоторых тредов не могут вызвать ошибки или прекращения работ других тредов

Java concurrency in practice

Что есть блокировка?

1. Ожидание lock?
2. Выполнение Thread.sleep?
3. Выполнение чистого супер-длинно-дорогого вычисления?
4. Вызов IO API например JDBC?

Юзкейс scala.concurrent.blocking

```
import scala.concurrent.blocking

Future {
 blocking {
   Thread.sleep(999999)
 }
}
```

```scala
def blocking[T](body: => T): T =
BlockContext.current.blockOn(body)(scala.concurrent.AwaitPermission)

def newThread(fjp: ForkJoinPool): ForkJoinWorkerThread = {
 (...)
 new ForkJoinWorkerThread(fjp) with BlockContext
 (...)
}

Thread.currentThread match {
 case ctx: BlockContext => ctx
 case _ => DefaultBlockContext
}

private object DefaultBlockContext extends BlockContext {
 override def blocking[T](thunk: => T): T(implicit permission: CanAwait)
}
```

Monix

Task

```
import monix.execution.Scheduler.Implicits.global

def service(arg: Int) = Task(22 / arg)
val servTask = service(50) //я ленивый
servTask.runToFuture {
 case Success(x) => println(x)
 case Failure(t) => t.printStackTrace()
}
```

Task

```
import monix.execution.Scheduler.Implicits.global

def service(arg: Int) = Task(22 / arg)
val servTask = service(50) //я ленивый
servTask.runToFuture {
 case Success(x) => println(x)
 case Failure(t) => t.printStackTrace()
} // (implicit S: Scheduler)
```

Task

```
import monix.execution.Scheduler.Implicits.global

def service(arg: Int) = Task(22 / arg)
val servTask = service(50) //я ленивый
servTask.runToFuture {
 case Success(x) => println(x)
 case Failure(t) => t.printStackTrace()
} // (implicit S: Scheduler)

val taskMulti = servTask.map(_ * 2)
val runningTask: CancelableFuture[Int] = taskMulti.runToFuture
runningTask.map(_ * 4).onComplete(println)
```

Task

```
val p = for {
 i <- Task(1)
 j <- Task(2)
 _ <- Task(println("Hello world"))
} yield i + j
//p: monix.eval.Task[Int] = Task.FlatMap$1806091848

val x = p.runToFuture
//Hello world
//x: monix.execution.CancelableFuture[Int] =
monix.execution.CancelableFuture$Pure@76aa012d

x.value
//res0: Option[scala.util.Try[Int]] = Some(Success(3))
```

```scala
val now = Task.now { println("Hello"); 22 }
//Hello
//now: monix.eval.Task[Int] = Task.Now(22)

val eval = Task.eval { println("Hello"); 22 }
//eval: monix.eval.Task[Int] = Task.Eval$522226858

val once = Task.evalOnce { println("Hello"); 22 }
//once: monix.eval.Task[Int] = Task.Eval$1355512627
once.foreach(println)
//Hello
//22
once.foreach(println)
//22

val task = Task.defer { Task.now { println("Hello"); "Hello!" }}
//task: monix.eval.Task[String] = Task.Suspend$1926119441
```

```scala
val now = Task.now { println("Hello"); 22 }
//Hello
//now: monix.eval.Task[Int] = Task.Now(22)

val eval = Task.eval { println("Hello"); 22 }
//eval: monix.eval.Task[Int] = Task.Eval$522226858

val once = Task.evalOnce { println("Hello"); 22 }
//once: monix.eval.Task[Int] = Task.Eval$1355512627
once.foreach(println)
//Hello
//22
once.foreach(println)
//22

val task = Task.defer { Task.now { println("Hello"); "Hello!" }}
//task: monix.eval.Task[String] = Task.Suspend$1926119441
```

```
val now = Task.now { println("Hello"); 22 }
//Hello
//now: monix.eval.Task[Int] = Task.Now(22)

val eval = Task.eval { println("Hello"); 22 }
//eval: monix.eval.Task[Int] = Task.Eval$522226858

val once = Task.evalOnce { println("Hello"); 22 }
//once: monix.eval.Task[Int] = Task.Eval$1355512627
once.foreach(println)
//Hello
//22
once.foreach(println)
//22

val task = Task.defer { Task.now { println("Hello"); "Hello!" }}
//task: monix.eval.Task[String] = Task.Suspend$1926119441
```

```
val now = Task.now { println("Hello"); 22 }
//Hello
//now: monix.eval.Task[Int] = Task.Now(22)

val eval = Task.eval { println("Hello"); 22 }
//eval: monix.eval.Task[Int] = Task.Eval$522226858

val once = Task.evalOnce { println("Hello"); 22 }
//once: monix.eval.Task[Int] = Task.Eval$1355512627
once.foreach(println)
//Hello
//22
once.foreach(println)
//22

val task = Task.defer { Task.now { println("Hello"); "Hello!" }}
//task: monix.eval.Task[String] = Task.Suspend$1926119441
```

```
val forked = Task.fork {Task.now(1)}
forked.runSyncMaybe
//res0: Either[monix.execution.CancelableFuture[Int],Int] = Right(1)

lazy val io = Scheduler.io(name = "FILP2019")
val source = Task(println(s"Running on thread:${Thread.currentThread().getName}"))
val fork = Task.fork(source, io)
fork.foreach(println)
//Running on thread: FILP2019-11
```

```
val forked = Task.fork {Task.now(1)}
forked.runSyncMaybe
//res0: Either[monix.execution.CancelableFuture[Int],Int] = Right(1)

lazy val io = Scheduler.io(name = "FILP2019")
val source = Task(println(s"Running on thread:${Thread.currentThread().getName}"))
val fork = Task.fork(source, io)
fork.foreach(println)
//Running on thread: FILP2019-11
```

```
val danger = Task.delay {
 if (Random.nextInt % 2 == 0) throw new Exception("Boom") else 2
}.memoizeOnSuccess

danger.runOnComplete{ x => println(x)}
//Failure(java.lang.Exception: Boom)
//res2: monix.execution.Cancelable =
monix.execution.Cancelable$CancelableTask@6e0eeef8
//Success(2)
```

```scala
def task(taskName: String): Task[String] = Task{
 println(s"taskName at ${LocalDateTime.now()}"); taskName
}

val batch = for {
 t1 <- task("t1").delayExecution(10.seconds)
 t2 <- task("t2").delayExecution(5.seconds)
 t3 <- task("t3")
} yield s"$t1 $t2 $t3"
println(LocalDateTime.now)
batch.foreach(println)
```

```scala
def task(taskName: String): Task[String] = Task{
 println(s"$taskName at ${LocalDateTime.now()}"); taskName
}

val batch = Task.map3(
 task("t1").delayExecution(10.seconds),
 task("t2").delayExecution(5.seconds),
 task("t3")) {(t1, t2, t3) =>  s"$t1 $t2 $t3"}
println(LocalDateTime.now)
batch.foreach(println)
```

	Single	Multiple
Synchronous	A	Iterable[A]
Asynchronous	Future[A] / Task[A]	Observable[A]

```
val obs = Observable(1,2,3)

val plusOne = obs.map(_ + 1)
plusOne.foreach(println)
```

```
val infAsyncObs: Observable[Int] =
 Observable.repeatEval(Random.nextInt)
   .flatMap( x => Observable.fromFuture(Future(x * x * x)))
   .filter(_ % 2 ==0)

val task: Task[List[Int]] = infAsyncObs.take(1).toListL
task.runOnComplete(x => println(x))

//Success(List(-16117719296))
```

```
val await = Observable.zip2(
 Observable.intervalAtFixedRate(3.seconds),
 Observable.repeatEval(LocalDateTime.now())
)
 .map {
   case (_, t) => t
 }.take(3).foreach(println)
```

```
2019-04-24T23:44:25.855
2019-04-24T23:44:28.826
2019-04-24T23:44:31.826
```

```
val maxCuncurrentThreads = new AtomicInteger(0)
val allValuesOfmaxCuncurrentThreads = new AtomicReference[List[Int]](Nil)

val allJobs =
 Observable
   .repeatEval(Thread.currentThread().getId)
   .take(1000)
   .mapAsync(20)(threadId =>
     Task {
       blocking { //удаляем опционально
         val see = maxCuncurrentThreads.incrementAndGet()
         allValuesOfmaxCuncurrentThreads.updateAndGet { t =>
           see :: t
         }
         maxCuncurrentThreads.decrementAndGet()
         threadId
       }
   }).toListL.runAsync
```

```scala
val threadIdList: Seq[Long] = Await.result(allJobs, 5.minutes)

println(s"Highest observed no. of threads
${allValuesOfmaxCuncurrentThreads.get().max}")
println(s"Threads used ${threadIdList.distinct.length}")
```

//с блокировкой
Highest observed no. of threads 4
Threads used 9

//без блокировки
Highest observed no. of threads 4
Threads used 8

```
def getDataFromDB(id: Long): Future[CSV] = Future{
 blocking{
   println(s"Getting Data for $id")
   Thread.sleep(500)
   CSV(id.toString) }
}

def aggregate(input: CSV): Future[Int] = Future{
 println(s"Aggregating data for $input"); 1 }

val userList: List[Long] = (1L to 7L).toList
val storeJobs: List[Future[Int]] = userList.map { user => getDataFromDB(user)
 .flatMap(csvInput => aggregate(csvInput))
}

val batchedJob: Future[List[Int]] = Future.sequence(storeJobs)
Await.ready(batchedJob, 8.hours)
```

```
def getDataFromDB(id: Long): Task[CSV] = Task{
 blocking{
   println(s"Getting Data for $id")
   Thread.sleep(500)
   CSV(id.toString) }}

def aggregate(input: CSV): Task[Int] = Task{
 println(s"Aggregating data for $input"); 1}

val userList: List[Long] = (1L to 7L).toList
val storeJobs: List[Task[Int]] = userList.map { user => getDataFromDB(user)
 .flatMap(csvInput => aggregate(csvInput))
}

val batchedJob: Task[List[Int]] = Task.sequence(storeJobs)
Await.ready(batchedJob.runAsync, 8.hours)
```

Future

```
Getting data for 1
Getting data for 3
Getting data for 2
Getting data for 4
Getting data for 4
Getting data for 6
Getting data for 7
Aggregating data for 3
Aggregating data for 5
Aggregating data for 6
Aggregating data for 2
Aggregating data for 4
Aggregating data for 1
Aggregating data for 7
```

Task

```
Getting data for 1
Aggregating data for 1
Getting data for 2
Aggregating data for 2
Getting data for 3
Aggregating data for 3
Getting data for 4
Aggregating data for 4
Getting data for 5
Aggregating data for 5
Getting data for 6
Aggregating data for 6
Getting data for 7
Aggregating data for 7
```

```
def getDataFromDB(id: Long): Task[CSV] = Task{
 blocking{
   println(s"Getting Data for $id")
   Thread.sleep(500)
   CSV(id.toString) }}

def aggregate(input: CSV): Task[Int] = Task{
 println(s"Aggregating data for $input"); 1}

val userList: List[Long] = (1L to 7L).toList
val storeJobs: List[Task[Int]] = Observable.fromIterable(userList).mapAsync {
user => getDataFromDB(user)
 .flatMap(csvInput => aggregate(csvInput))
}

val batchedJob: Task[List[Int]] = storeJobs.toListL
Await.ready(batchedJob.runAsync, 8.hours)
```

Future

```
Getting data for 1
Getting data for 3
Getting data for 2
Getting data for 4
Getting data for 4
Getting data for 6
Getting data for 7
Aggregating data for 3
Aggregating data for 5
Aggregating data for 6
Aggregating data for 2
Aggregating data for 4
Aggregating data for 1
Aggregating data for 7
```

Task

```
Getting data for 1
Aggregating data for 1
Getting data for 2
Aggregating data for 2
Getting data for 3
Aggregating data for 3
Getting data for 4
Aggregating data for 4
Getting data for 5
Aggregating data for 5
Getting data for 6
Aggregating data for 6
Getting data for 7
Aggregating data for 7
```

Observable

```
Getting data for 2
Getting data for 3
Getting data for 1
Aggregating data for 3
Aggregating data for 1
Aggregating data for 2
Getting data for 4
Getting data for 5
Getting data for 6
Aggregating data for 4
Aggregating data for 6
Aggregating data for 5
Getting data for 7
Aggregating data for 7
```