Summary of the project

Introduction

Direct style concurrency in Scala

The async library (https://github.com/lampepfl/async) is an attempt at creating a library providing direct style concurrency on Scala. It exists in a similar space to ZIO (https://zio.dev/), Cats Effect (https://typelevel.org/cats-effect/) and Cats (https://typelevel.org/cats/), although none of these attempt direct style. A library also in the space and attempting direct style is Ox (https://github.com/softwaremill/ox), however Ox only aspires to be a thin wrapper around Project Loom and does not want to be a general Scala library.

The async library assumes some implementation of green threads / fibers / coroutines and (in the future) continuations. One such implementation is project Loom (details below).

The main goals of the library are

1. direct style,
2. structured concurrency, and
3. possibility of cancellation,
4. possibility of being used across different Scala runtimes.

Project Loom

The project focused on implementing and exploring the async library on the project-Loom-enabled JVM.

Project Loom (https://openjdk.org/projects/loom/) is an ongoing initiative to bring lightweight threads and related features to the Java runtime. Large part of it is already implemented and available as preview features in JVM starting from version 20. In order to utilize the new semantics one has to invoke the JVM with additional VM parameter `--enable-preview`.

The core semantic changes of project Loom are

1. Introduction of virtual threads (`Thread.startVirtualThread`), and
2. making most non-blocking operations blocking instead.

It is expected that Project Loom will be available by default in future JVM releases making relaying on it a future-proof choice.

While the work focused on implementation of the async library on the project-Loom-enabled JVM, the async library is also expected to be ported to other scala implementations, including Scala Native. Therefore, it is important not to limit the design to only project Loom.

Source Semantics

This project focused mostly on futures and channels. The rest of the semantics is explained in greater detail in the `README.md`.

`Async.Source`

Sources are objects that represent something that can be awaited (with the `await(Async.Source)` method of trait `Await`). Sources directly expose a polling interface and an interface to register and remove listeners.

```
def poll(k: Listener[T]): Boolean
def poll(): Option[T] // either return None if not done or Some(x) if x is ready
```

Polling represents an operation that ideally should complete almost immediately without blocking on anything external and either not call the listener at all if the source is not yet ready or, if it is ready, call it immediately with the ready value.

```
def onComplete(k: Listener[T]): Unit
def dropListener(k: Listener[T]): Unit
```

`onComplete` allows one to pass a callback listener that will be invoked once the source is ready with some value. If the source is already ready when the `onComplete` is invoked, the listener will be called immediately and not added to the subscriber list. `dropListener` called with the same listener reference will remove it from the subscriber list.

Listeners

Sources take listeners for `poll()` and `addListener()`. Listener of some value of type `T` is just a funciont taking that value and returning a boolean.

```
trait Listener[-T] extends (T => Boolean)
```

Each listener has to obey the following contract:

1. either consume the value and signal it by returning `true` as quickly as possible, or
2. determine that it is not interested in this value and return `false` as quickly as possible.

This mechanism was invented so that filers and other readers which can conditionally reject values do not consume the value despite rejecting it and deprive other listeners of consuming it.

`Async` trait

Instances of trait `Async` provide

1. a concrete implementation of `await(Async.Source[T]): T`,
2. an `ExecutionContext`, and

3. a completion group previously called cancellation group (details in the `README.md`).

In the project Loom version of the async library, execution context is never used. This is because virtual threads on JVM are not open to many user-provided settings, one cannot set their priority or change their scheduler type. Execution context is preserved however for the purpose of it being used in other async library implementations, like the Scala Native one.

The most common way to obtain an `Async` instance is to use an `Async.blocking` block:

```
Async.blocking:
  val f = Future { 10 }
  Async.await(f)
```

When the execution of an `Async.blocking` block is done, it cancels all cancellable instances created inside it.

`Cancellable` trait

This trait provides the ability to cancel execution:

```
  def cancel()(using Async): Unit
```

Implementations of cancellable sources have to provide this implementation by themselves and it usually involves shutting down pending actions/awaits, cleaning up resources and notifying listeners about the cancellation.

For the details of completion/cancellation groups see `REAMDE.md`.

`map` and `filter` on `Sources`

```
def map[U](f: T => U): Source[U]
def filter(p: T => Boolean): Source[T]
```

One can create another source from a source by invoking a `map` or `filter` on it. These operations work on a source and return a new source, which proxies polling to the original source with the map/filter on top of values returned by the original poll. This does not in any way destroy or consume the underlying sources.

`race` and `either` on `Sources`

```
def race[T](sources: Source[T]*): Source[T]
```

Racing returns a new source that represents awaiting on the first completed underlying source. This does not in any way destroy or consume the underlying sources.

```
def either[T1, T2](src1: Source[T1], src2: Source[T2]): Source[Either[T1, T2]] =
  race(src1.map(Left(_)), src2.map(Right(_)))
```

`either` is just a `race` which denotes which of the two sources was returned.

Future Semantics

Completable and runnable futures

```
trait Future[+T] extends Async.OriginalSource[Try[T]], Cancellable
```

A future is an abstraction of a computation that delivers result after some time. A future as a source represents awaiting the result of the thread's computation wrapped in `Success` or some exception (possibly `java.util.concurrent.CancellationException`) wrapped in `Failure`.

We distinguish completable (`CoreFuture[T]`) and runnable (`RunnableFuture[T]`) futures. Completable future's final value is set by their `complete()` method. Runnable future is essentially a function executed in a separate thread and the function's return value determines the future's final value. Runnable futures are the most common version and form the basis of most computations in the async library.

```
// example of runnable futures
Async.blocking:
  val f1 = Future { Thread.sleep(100); 10 }
  val f2 = Future { Thread.sleep(200); 40 }
  f1.value + f2.value
```

To obtain the result of a future `f`'s execution one can either

1. `Async.await(f)`, which awaits the future's completion and returns a `Try[T]`
2. `f.result`, which does the same thing, or
3. `f.value` which is the same as `f.result.get`.

A runnable future is essentially a thread that 1. returns some value (might be `Unit`), 2. is cancellable, and 3. conforms to structured concurrency by any uncaught exception raised inside the future's thread will be communicated to the outside via the `Try`.

`Future.now` allows to create a completable future that immediately sets its value to the given argument. The implementation also makes the use of completable future clear:

```
def now[T](result: Try[T]): Future[T] =
  val f = CoreFuture[T]()
  f.complete(result)
  f
```

`CoreFuture` is a private class, but its mechanism is exposed via the public class `Promise` which does the same thing:

```
class Promise[T]:
  private val myFuture = CoreFuture[T]()
  val future: Future[T] = myFuture
  def complete(result: Try[T]): Unit = myFuture.complete(result)
```

Cancellation

Cancelling a future with `future.cancel()` results in its value being immediately set to `Failure(CancellationException)` and, if it's a runnable future, its thread being `Thread.interrupt()`ed.

`uninterruptible` block inside a future `f` makes it so that if `f` is cancelled while the uninterruptible block executes, it's cancellation will be deferred until the block finishes, but it will immediately throw CancellatoinException inside `f` after the block. `uninterruptible` does not make the cancellation request disappear, it just defers it until after the block.

```
val f = Future {
  // this can be interrupted
  uninterruptible {
    Thread.sleep(300) // this cannot be interrupted *immediately*
  }
  // this can be interrupted
}
```

`Tasks`

```
class Task[+T](val body: Async ?=> T):
  def run(using Async) = Future(body)
  def schedule(s: TaskSchedule): Task[T]
```

Task is a template for making runnable futures and is therefore referentially transparent. `run` method instantiates the future and starts executing it immediately.

Tasks can be optionally given a schedule which describes how a task is to be repeated.

- `Task(g).schedule(TaskSchedule.Every(100)).run` will run every 100 milliseconds
- `Task(g).schedule(TaskSchedule.RepeatUntilSuccess()).run` will run until g succeeds
- `Task(g).schedule(TaskSchedule.RepeatUntilFailure()).run` will run until g fails
- `Task(g).schedule(TaskSchedule.ExponentialBackoff(100, maxRepetitions=10)).run` will run with exponentially increasing pauses, but only 10 times

`zip` and `alt`

```
extension [T](f1: Future[T])
  def zip[U](f2: Future[U])(using Async): Future[(T, U)]
  def alt(f2: Future[T])(using Async): Future[T]
  def altC(f2: Future[T])(using Async): Future[T]
```

`zip` takes two futures and represents a future of either a pair of both values taken from the underlying futures or, if at least one underlying future returns a `Failure`, the first returned error. Example usage: `f1.zip(f2)`. It can also be used with more than two futures like this: `f1 *: f2 *: f3.zip(f4)`.

`alt` takes two futures and represents a future with the value of the future which succeeds first. If all underlying futures fail, it passes the last error. It can be used like `f1.alt(f2)` or, with more than two futures, `alt(f1, f2, f3)`.

`altC` is a version of `alt` that `cancel`s all the other futures once the first successful one is ready. If none are successfull, it acts like `alt`.

Channel Semantics

```
trait SendableChannel[T]:
  def send(x: T)(using Async): Unit

trait ReadableChannel[T]:
  val canRead: Async.Source[Try[T]]
  def read()(using Async): Try[T] = await(canRead)

trait Channel[T] extends SendableChannel[T], ReadableChannel[T], java.io.Closeable
```

A channel is an object which one can write (send) to and read what was sent. A channel can also be closed. Closed channel raises `ChannelClosedException` when attempting to write to it and immediately returns `Failure(ChannelClosedException)` when attempting to read from it.

Since a channel's sender is usually different from its reader, Channel can be split into three references according to their function. One allowing only sending, one allowing only reading and one allowing only closing. This makes it possible to use channels in a similar manner to how POSIX pipes are created and used.

```
trait Channel[T] extends SendableChannel[T], ReadableChannel[T], java.io.Closeable:
  def asSendable(): SendableChannel[T] = this
  def asReadable(): ReadableChannel[T] = this
  def asCloseable(): java.io.Closeable = this
```

There are two main channel implementations:

```
trait SyncChannel[T] extends Channel[T]
trait BufferedChannel[T] extends Channel[T]
```

`SyncChannel`, sometimes called a rendez-vous channel has the following semantics:

- `send` to an unclosed channel blocks until a reader willing to accept this value (which is indicated by the reader's listener returning `true` after sampling the value) is found and this reader reads the value.
- reading is done via the `canRead` async source of potential values (wrapped in a `Try`). Note that only reading is represented as an async source, sending is a blocking operations that is implemented similarly to how `await` is implemented.

`BufferedChannel(size: Int)` is a version of a channel with an internal value buffer (represented internally as an array with positive `size`). It has the following semantics:

- `send` if the buffer is not full appends the value to the buffer and returns immediately.
- `send` if the buffer is full sleeps until some buffer slot is freed, then writes the value there and immediately returns.
- reading is done via the `canRead` async source that awaits the buffer being nonempty and the reader accepting the first value in the buffer. Because readers can refuse a value, it is possible that many readers await on `canRead` while the buffer is non-empty if all of them refused the first value in the buffer. At no point a reader is allowed to sample/read anything but the first entry in the buffer.

Channels were designed to be able to seamlessly have multiple readers and multiple writers at any time, coming from different threads. However, the readers in such situation are competing with each other for sent values. Each value sent to the channel gets delivered to exactly one reader. For situations where this is undesired, there exist `ChannelMultiplexer`.

ChannelMultiplexer

```
trait ChannelMultiplexer[T] extends java.io.Closeable:
  def addPublisher(c: ReadableChannel[T]): Unit
  def removePublisher(c: ReadableChannel[T]): Unit
  def addSubscriber(c: SendableChannel[Try[T]]): Unit
  def removeSubscriber(c: SendableChannel[Try[T]]): Unit
```

Channel multiplexer is an object where one can register publisher and subscriber channels. Internally a multiplexer has a thread that continuously races the set of publishers and once it reads a value, it sends a copy to each subscriber.

For an unchanging set of publishers and subscribers and assuming that the multiplexer is the only reader of the publisher channels, every subsciber will receive the same set of messages, in the same order and it will be exactly all messages sent by the publishers. The only guarantee on the order of the values the subscribers see is that values from the same publisher will arrive in order.

Channel multiplexer can also be closed, in that case all subscribers will receive `Failure(ChannelClosedException)` but no attempt at closing either publishers or subscribers will be made.

Example Uses

As an example of how the framework can be used, we present detailed tests of semantics in `src/test/scala` as well as the implementations of a custom source: `Timer` and `StartableTimer`; and the implementations of future-based IO integration: `PosixLikeIO.{SocketUDP, File}`.

Timers

```
type TimerRang = Boolean
class StartableTimer(val millis: Long) extends Async.OriginalSource[TimerRang], Cancellable
  def start(): Unit
class Timer(millis: Long) extends StartableTimer(millis):
  this.start()
```

`StartableTimer` is a timer that has to have its `start` method invoked for it to begin counting time. `Timer` is a subclass that invokes `start` immediately upon its creation and can be used like `Async.await(Timer(1000))`.

PosixLikeIO

`PosixLikeIO.File` and `PosixLikeIO.SocketUDP` were designed to mimic the POSIX interface of files and UDP sockets in a JVM-based language. There are two additional convenience methods `readString` and `writeString` mimicking similar methods from `java.nio.file.Files`. To integrate nicely with the environment the interface takes and returns types already present in the Java standard library (like `StandardOpenOption`, `DatagramPacket`) and uses nio ByteBuffers for zero-copy operations.

```
class File(val path: String):
  def isOpened: Boolean
  def open(options: StandardOpenOption*): File
  def close(): Unit
  def read(buffer: ByteBuffer): Future[Int]
  def readString(size: Int, charset: Charset = StandardCharsets.UTF_8): Future[String]
  def write(buffer: ByteBuffer): Future[Int]
  def writeString(s: String, charset: Charset = StandardCharsets.UTF_8): Future[Int]

class SocketUDP:
  def isOpened: Boolean
  def bindAndOpen(port: Int): SocketUDP
  def open(): SocketUDP
  def close(): Unit
  def send(data: ByteBuffer, address: String, port: Int): Future[Unit]
  def receive(): Future[DatagramPacket]
```

There are also methods allowing one to use files and sockets without having to explicitly open and close them. These methods are directly inspired by Python's `with open() as f` mechanism.

```
object PIOHelper:
  def withFile[T](path: String, options: StandardOpenOption*)(f: File => T): T
  def withSocketUDP[T]()(f: SocketUDP => T): T
  def withSocketUDP[T](port: Int)(f: SocketUDP => T): T
```

```
PIOHelper.withFile("x.txt", StandardOpenOption.WRITE): f =>
  Async.await(f.writeString("Hello world!"))
```

Performance

Machine details

Benchmarks in this section were performed on a ThinkPad L15 laptop with `11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz, family 6, model 140` processor on Linux and Windows operating systems.

The Linux system used was Ubuntu 22.04 with the output of `uname -a` being `Linux l15gen2 5.19.0-45-generic #46~22.04.1-Ubuntu SMP PREEMPT_DYNAMIC Wed Jun 7 15:06:04 UTC 20 x86_64 x86_64 x86_64 GNU/Linux`. The scala version used was `Scala 3.3.0-RC3 (20, Java OpenJDK 64-Bit Server VM)`. `java -version` output:

```
openjdk version "20" 2023-03-21
OpenJDK Runtime Environment (build 20+36-2344)
OpenJDK 64-Bit Server VM (build 20+36-2344, mixed mode, sharing)
```

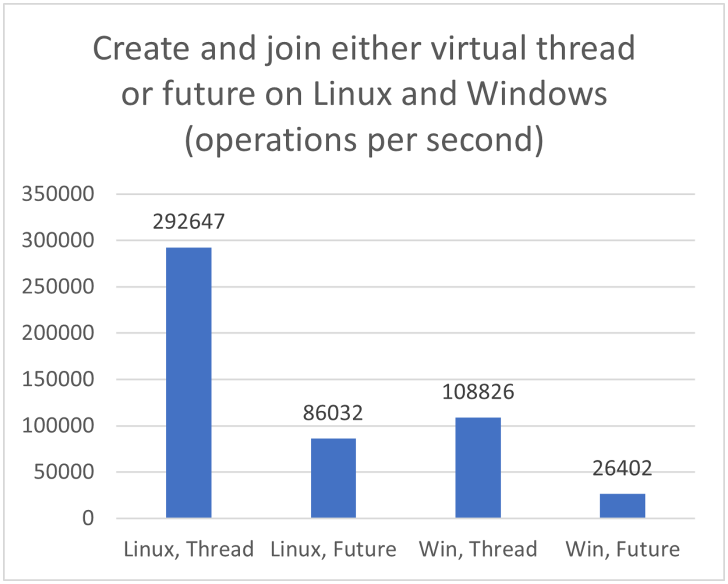
The Windows system used was Windows 11 22H2 Build No. 22621. The Scala version used was `Scala 3.3.0-RC3 (20.0.1, Java OpenJDK 64-Bit Server VM)`. `java -version` ouptut:

```
openjdk version "20.0.1" 2023-04-18
OpenJDK Runtime Environment (build 20.0.1+9-29)
OpenJDK 64-Bit Server VM (build 20.0.1+9-29, mixed mode, sharing)
```

Benchmarks were performed multiple times to verify reproducibility of the obtained values. Where appropriate, standard deviation of time measurements was computed.

Overhead of core primitives

Future The first measured primitive is the Future. Since futures are implemented essentially as virtual threads with some instrumentation in the Project Loop version of async, we compare creating of a Future and awaiting it to crating a virtual thread and joining it. We check how many such operations can be performed in a second.



In the following table overhead means the overhead versus the other operation on the same OS.

OS	Thread/Future	Operations per second	Overhead
Linux	Thread	292647	3.40x
Linux	Future	86032	0.29x
Windows	Thread	108826	4.12x
Windows	Future	26402	0.24x

As expected, futures do include overhead compared to just using virtual threads. On Linux it's 3.4 and on Windows 4.12 times as expensive (just in the creation/instrumentation/deletion aspect). It's worth noting that future/thread creation is around 3 times slower on Windows than on Linux.

Async.race The second benchmarked primitive is `Async.race`. Racing three futures is compared to two other operations.

1. First it is compared to just awaiting the future we know will be the first to finish.
2. Second, it is compared to replacing futures with virtual threads which upon completion set an atomic boolean to true. The awaiting thread spin locks on the boolean.

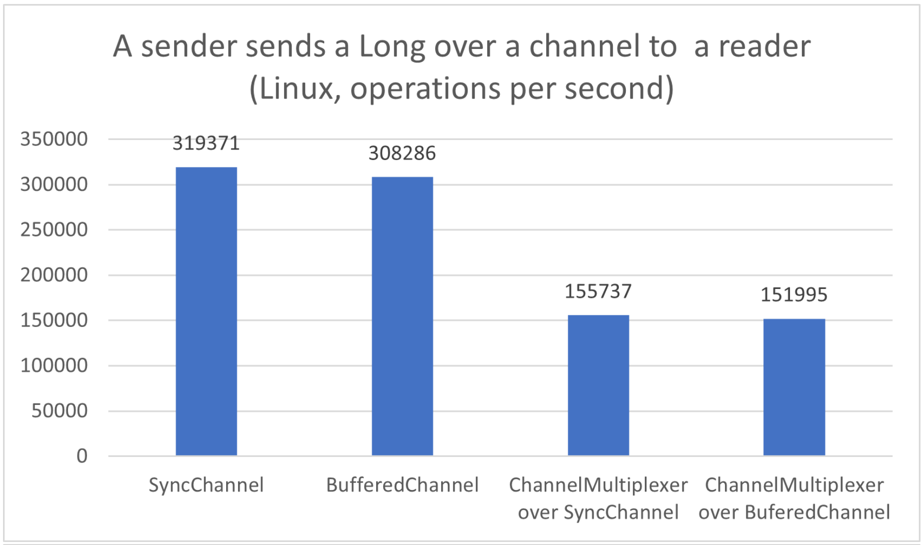
Racing method (Linux)	Operations per second	Overhead
Async.race	15.590	-
Await the fastest only	15.597	1.00
Spin lock	15.671	1.01

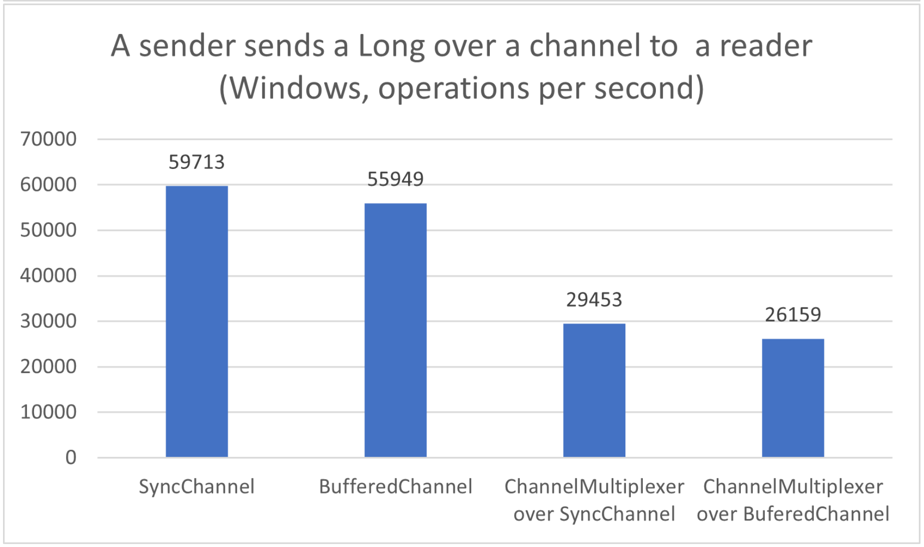
Racing method (Windows)	Operations per second	Overhead
Async.race	8.87	-
Await the fastest only	9.03	0.982
Spin lock	9.04	0.981

Surprisingly, the current implementation of Async.await matches the performance of alternative (expected to be faster) methods almost exactly. Again, operations on Linux are faster than on Windows.

Sending a value over a channel - comparison of channel types The first benchmarked core operation is sending a value over a channel. In this scenario there is one sender and one receiver. There are two types of channels synchronous (SyncChannel) and buffered (BufferedChannel). We also included a ChannelMultiplexer (with one publisher and one subscriber) where both the publisher and the subscriber are either Sync or Buffered channels. The goal is to check how many send/read operations can be performed in a second. The buffer size of the BufferedChannel was set to 1 since in this benchmark (where we send a number of elements much larger than expected buffer size) it is expected that at any point in time, the buffer will be either full or close to full so it's size should not make a difference.

OS	Channel type	Operations per second
Linux	SyncChannel	319371
Linux	BufferedChannel(1)	308286
Linux	ChannelMultiplexer over SyncChannel	155737
Linux	ChannelMultiplexer over BuferedChannel	151995
Windows	SyncChannel	59713
Windows	BufferedChannel(1)	55949
Windows	ChannelMultiplexer over SyncChannel	29453
Windows	ChannelMultiplexer over BuferedChannel	26159





As expected, ChannelMultiplexer makes things around two times slower since it essentially pipelines two channels together. Once again, in absolute terms, the Linux version is much faster than the Windows one, this time around 6 times.

Sending a value over a channel - comparison with ideal implementation
It is also worth estimating how much overhead in general channels have. In the scenario where there is one sender thread and one receiver thread, we can imagine a specialized implementation of the concept as follows:

```
@volatile var shared: Long = 0
@volatile var timeForWriting = true
```

```
val t1 = Thread.startVirtualThread: () =>
  var i: Long = 0
  while (true) {
    while (!timeForWriting) ()
    shared = i
    timeForWriting = false
    i += 1
  }

val t2 = Thread.startVirtualThread: () =>
  while (true) {
    while (timeForWriting) ()
    var z = shared
    timeForWriting = true
  }
```

There are two threads, the sender `t1` and the receiver `t2`. They have a shared variable for the communicated value and a shared atomic boolean lock, state of which signifies whether it is time for writing or reading from the shared value. Threads synchronize by spin locking on the shared boolean and writing/reading the shared variable.

In the following table overhead is taken versus the ideal implementation on the same OS.

OS	Method	Operations per second	Overhead
Linux	Ideal implementation	8691652	-
Linux	SyncChannel	319371	27.21x
Linux	BufferedChannel(1)	308286	28.19x
Windows	Ideal implementation	6859438	-
Windows	SyncChannel	59713	114.87x
Windows	BufferedChannel(1)	55949	122.60x

Compared to an ideal implementation, the overhead of channels on Linux is around 27 times and on Windows between 114 and 122 times. Such high overhead is expected, as the ideal code uses no functions, no lambdas, no additional threads or any instrumentation beyond two primitive memory cells, it is easy to imagine that for every primitive operation of the ideal code, 30 primitive operations in the channel version are performed.

Overhead of the example file IO implementation

In order to avoid the physical HDD/SSD hardware latency muddying the benchmarking results, we performed the benchmark on a RAM disk on both operating systems. RAM disk installs the file system entirely within the RAM memory of the computer, without touching any physical disk at all. On Linux, RAM disk was created by the command

```
mkdir -p /tmp/FIO && sudo mount -t tmpfs -o size=8g tmpfs /tmp/FIO
```

and on Windows the ImDisk Toolkit software (https://sourceforge.net/projects/imdisk-toolkit/) with the option 'Force physical memory' was used to create a virtual disk which was later formatted with NTFS with default parameters.

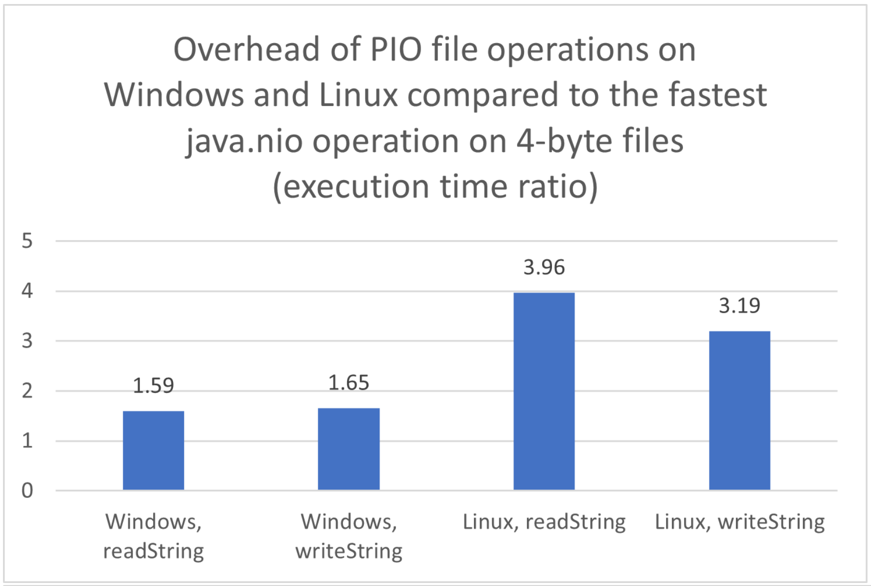
Writing and reading files of size either 4B or 40MB was benchmarked by three methods, the example file IO implementation included with the project, FileWriter/FileReader, and java.nio.Files utilities. In each case, java.nio.Files utilities turned out to be the fastest, so we do not include FileReader/FileWriter here.

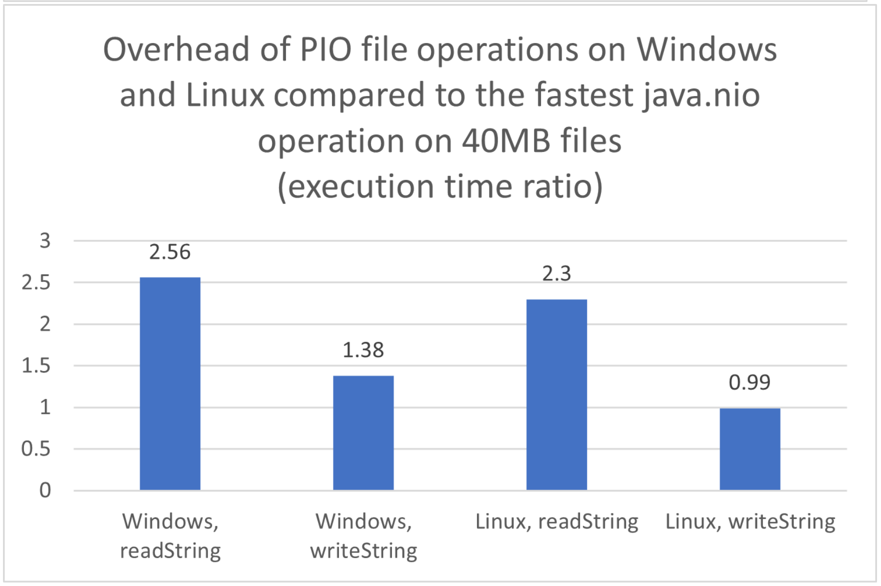
For 4B files:

OS	Method	Time in milliseconds	Standard deviation
Linux	PIO.readString	0.0171	0.04
Linux	Files.readString	0.004	0.00
Linux	PIO.writeString	0.019	0.03
Linux	Files.writeString	0.006	0.00
Windows	PIO.readString	0.125	0.021
Windows	Files.readString	0.079	0.014
Windows	PIO.writeString	0.266	0.07
Windows	Files.writeString	0.162	0.03

For 40MB files:

OS	Method	Time in milliseconds	Standard deviation
Linux	PIO.readString	28.739	2.37
Linux	Files.readString	12.522	2.10
Linux	PIO.writeString	17.027	0.75
Linux	Files.writeString	17.118	2.62
Windows	PIO.readString	71.356	2.37
Windows	Files.readString	27.917	1.56
Windows	PIO.writeString	72.695	9.40
Windows	Files.writeString	52.575	9.21





Overheads for the small files are around 1.6 on Windows and around 3.5 on Linux (the better performance of Windows can be explained by Windows handling file operations slower therefore async overhead mattering less). Overheads for the large files are surprisingly similar for the both operating systems, around 2.4 times for reading and almost no overhead for writing.

We expect the higher reading overhead for large files to be the result of suboptimal buffer allocation in the readString() method of the provided IO integration.

This shows that even with minimal optimizations, the async instrumentation does not impose a high overhead when wrapping operations in futures and promises.