

🐝 356.47
Rating

Conferences by Oleg Bunin (Ontico)

Professional conferences for IT developers

👤 hedgehog_on_rainbow 24 Feb 2021 at 15:05

Reactive programming in Java: how, why and is it worth it? Part I

🕒 12 min

👁 28K

Blog of Oleg Bunin Conference Company (Ontiko), High performance*, Programming*, Java*, Parallel programming*

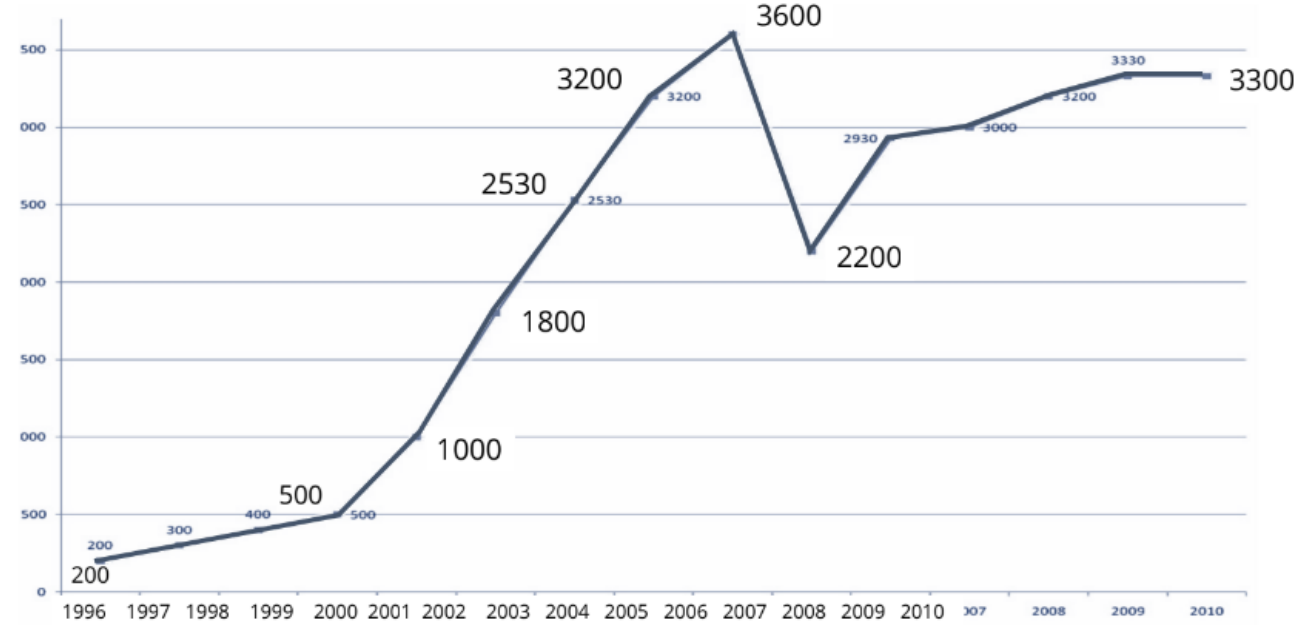
The idea of reactive programming appeared relatively recently, about 10 years ago. What caused the popularity of this relatively new approach and why it is now in trend, Vladimir Sonkin, an expert and trainer at Luxoft Training, told at the RIT ++ 2020 conference.

In the master class mode, he demonstrated why non-blocking I / O is so important, what are the disadvantages of classical multithreading, in what situations reactivity is needed, and what it can give. He also described the shortcomings of the reactive approach.

In this article, we will talk about what reactive programming is and why it is needed, discuss approaches and see examples.

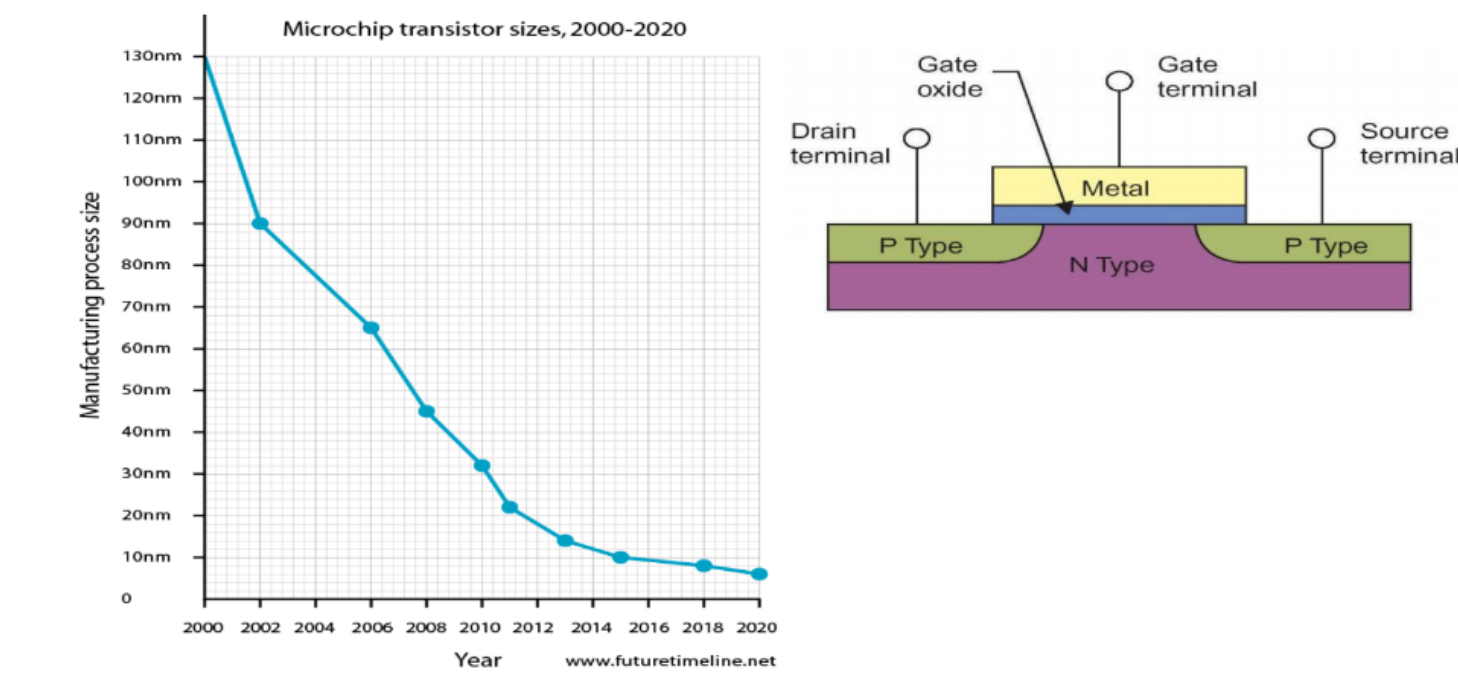
Why is reactive programming so popular? At some point, the speed of processors stopped growing, which means that developers no longer have to rely on the fact that the speed of their programs will increase on their own: now they need to be parallelized.

CPU frequency is not growing anymore



The figure shows that the graph of the frequency of processors grew in the 90s, and in the early 2000s the frequency increased sharply. It turned out that that it was the ceiling.

Why did the increase in frequency stop?



Transistors began to be made as small as possible. The PN junction that is used in them turned out to be as thin as possible. On the size chart of transistors inside the processor, we see: the size is getting smaller and smaller, there are more and more transistors in the processor.

Such miniaturization used to lead to the fact that the frequency grew. As the electrons travel at the speed of light, due to the reduction in size, the time it took for an electron to run all the way inside the processor was reduced. But technical processes hit the physical ceiling. I had to come up with something else.

Multithreading

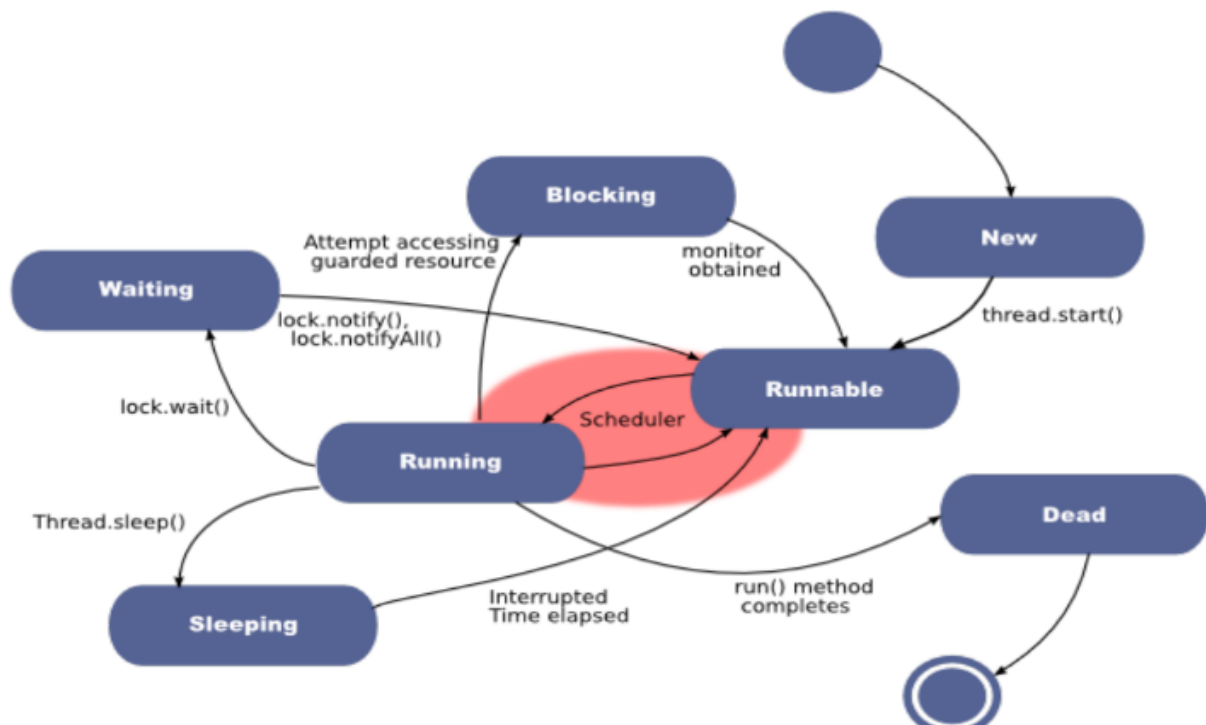
And we already know what we managed to come up with: we started making multi-core processors. Instead of relying on the fact that the performance of the processor will grow, they began to expect an increase in their number. But to effectively use multiple processors, you need multithreading.

The topic of multithreading is complex, but inevitable in the modern world. A typical modern computer has at least 4 cores and many threads. A modern powerful server can have 100 cores. If your program doesn't use multithreading, you don't get any benefit. Therefore, all global industries are gradually moving towards utilizing these opportunities.

At the same time, many dangers lie in wait for us. Programming with multithreading in mind is difficult: synchronization, races, difficult debugging, etc. ruined a lot of blood developers. In addition, the cost of such development becomes higher.

Multithreading has been around in Java for a long time, it has been around since the very first version.

It looks like this:



Writing a large system using the primitives of multithreading is, to put it mildly, difficult. Now nobody does that anymore. It's like coding in assembler.

In many cases, the effect that multithreading brings does not improve performance, but degrades it.

What to do with it?

Parallel and asynchronous programming

INFORMATION

Site	www.ontico.ru
Date of registration	February 25, 2010
Date grounds	January 1, 2008
population	31–50 people
Location	Russia

LINKS

- HighLoad++
highload.ru
- Team Lead Conf
teamleadconf.ru
- DevOpsConf
devopsconf.io
- TestDriven Conf
tdconf.ru
- TechLead Conf (conference dedicated to engineering processes and practices)
techleadconf.ru
- PHP Russia
phprussia.ru
- FrontendConf
frontendconf.ru
- GolangConf
golangconf.ru
- AppsConf
appsconf.ru
- Moscow Python Conf
conf.python.ru
- Knowledge Conf (knowledge management conference)
knowledgeconf.ru

HABR'S BLOG

10 hours ago

How to Optimize Latency in Cloud Gaming

👁 806

💬 10 +10

Feb 1 at 2:06 p.m.

SOAR in Kubernetes with little blood

👁 1.3K

💬 0

Jan 31 at 4:23 pm

Increasing the survivability of Raft in real conditions

👁 1.4K

💬 3 +3

Jan 26 at 2:32 pm

eBPF in production

👁 3.5K

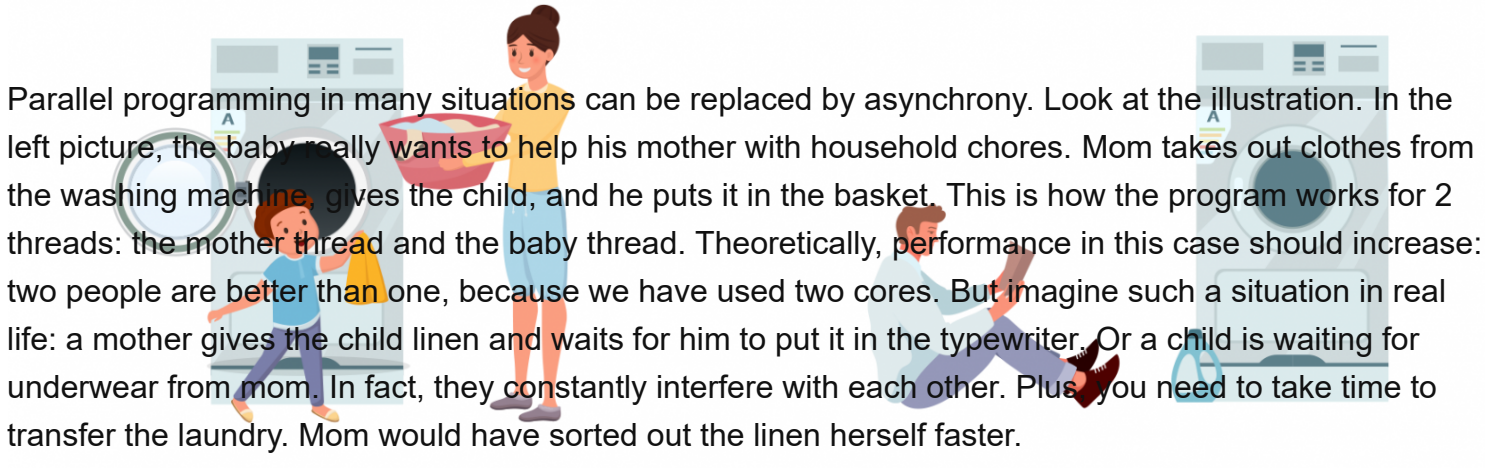
💬 2 +2

Jan 24 at 2:15 pm

Brain microservice. Quality Recipes

👁 6.5K

💬 1 +1



Approximately the same situation occurs in the computer. Therefore, working with parallelism is not at all as simple as it seems. All synchronization between execution threads actually takes a lot of time.

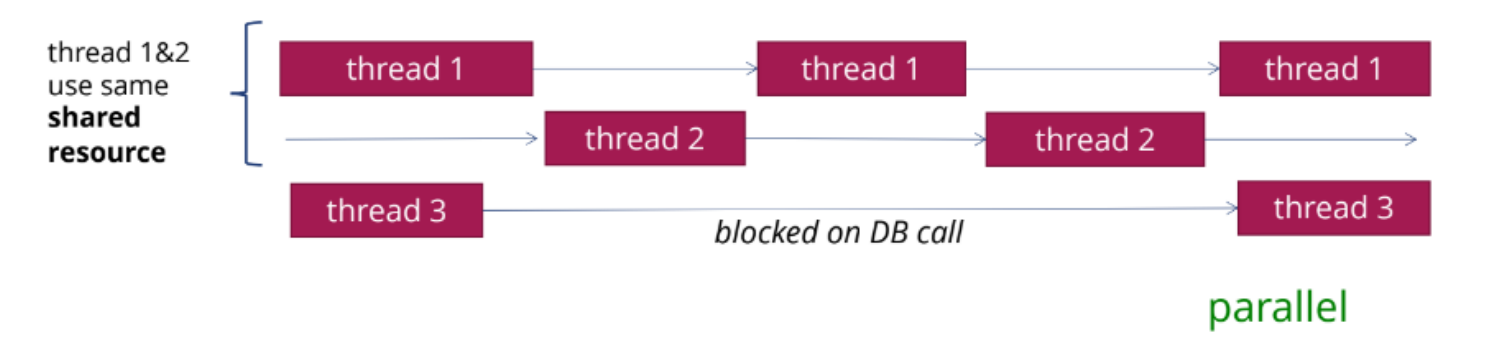
In the picture on the right is a lonely guy who bought himself an automatic typewriter. She erases, and at this time he can read a book. There is definitely an advantage here for the young man, because he is minding his own business and at the same time does not monitor whether the laundry is completed. When the wash is completed, it will hear a beep and respond to it. That is, there is parallelism, but there is no synchronization. Therefore, there is no time wasting on synchronization, a solid benefit!

This is the asynchronous approach. There is a separate executor, and we gave him not part of our task, but our own. On the left picture, the mother and the boy are doing a common task, and on the right, the washing machine and the guy are doing different things, each with his own. At some point, they will connect: when the washing machine finishes washing, the young man will put his book aside. But all 1.5 hours, while the laundry was being washed, he felt great, read and did not think about anything.

Examples of Parallel and Asynchronous Approaches

Consider 2 options for executing threads: parallel and asynchronous.

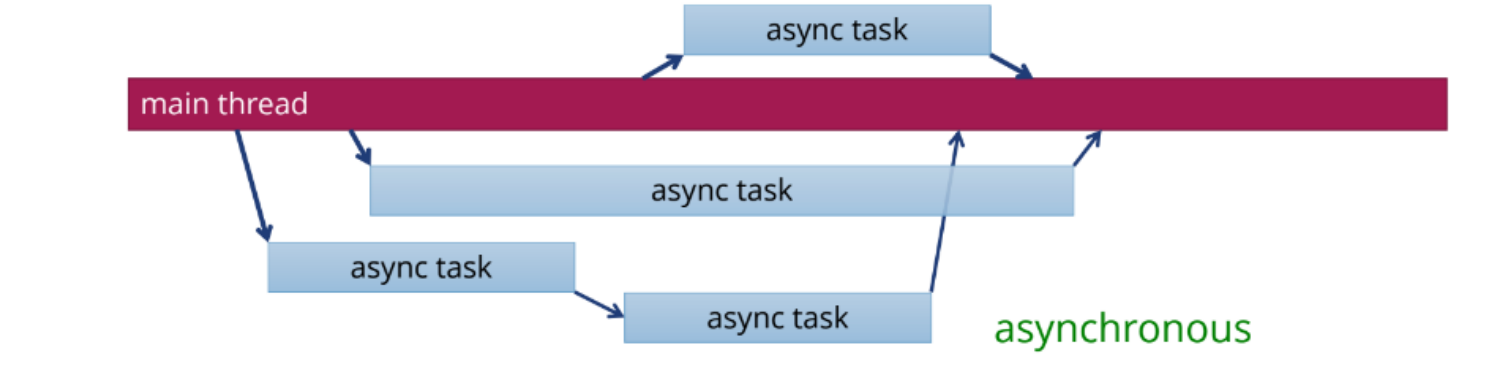
- Threads run in parallel;



Threads 1 and 2 need to access the same shared resource. Let's say it's some kind of database and it doesn't allow threads to connect to it at the same time. Or it allows, but this immediately reduces the speed of its work, so it is better for threads to access it in turn. There is no parallelism here: the threads have to work in turn. And the third thread is waiting for a response from the database, and is also blocked - such a system is inefficient.

It seems that there is parallelism, but there are not so many advantages from it.

- Threads run asynchronously.



If we use asynchrony, we set a task, and it is executed somewhere in another thread. For example, a different processor core or a different processor. We have set a task and are doing other things, and then at some point, when this task is completed, we will get the results. This can be illustrated by the work of the organization. The boss, the main thread, sets a task for Petya and says: "As soon as you complete it, pass it on to Kolya, and he, after completing work on the task, let him report to me. As a result, Petya and Kolya are busy with work, and the boss can set new tasks for other employees."

Another example: contention and concurrency.

Imagine an office, morning, everyone wants to drink coffee. Concurrency (competition) is when a queue is lined up for one on all coffee machines. People compete: "Hey, I was the first one here!" - "No, I!". They interfere with each other.

In parallelism, there are two coffee machines and two queues: each stands in its own. But still, employees spend time standing there.

How to find the right solution for this scenario if using asynchrony?

Delivery of coffee directly to the table is a good option, you do not stand in line, but you will have to hire a waiter to serve drinks.

Another possible option is a fixed schedule. For example, one employee comes for coffee at 11:10, the next at 11:20, and so on. But it's not asynchronous. Downtime will occur, which means that this is not a full load of the coffee machine. Someone didn't make it in time, and someone didn't have 10 minutes to make coffee for themselves, and as a result, the whole schedule shifts. And if you do

large gaps, the coffee machine will be underloaded. And then, everyone wants to come at 10 in the morning and drink coffee, and this stretches for 2 hours, and someone will get his cup only at 12.

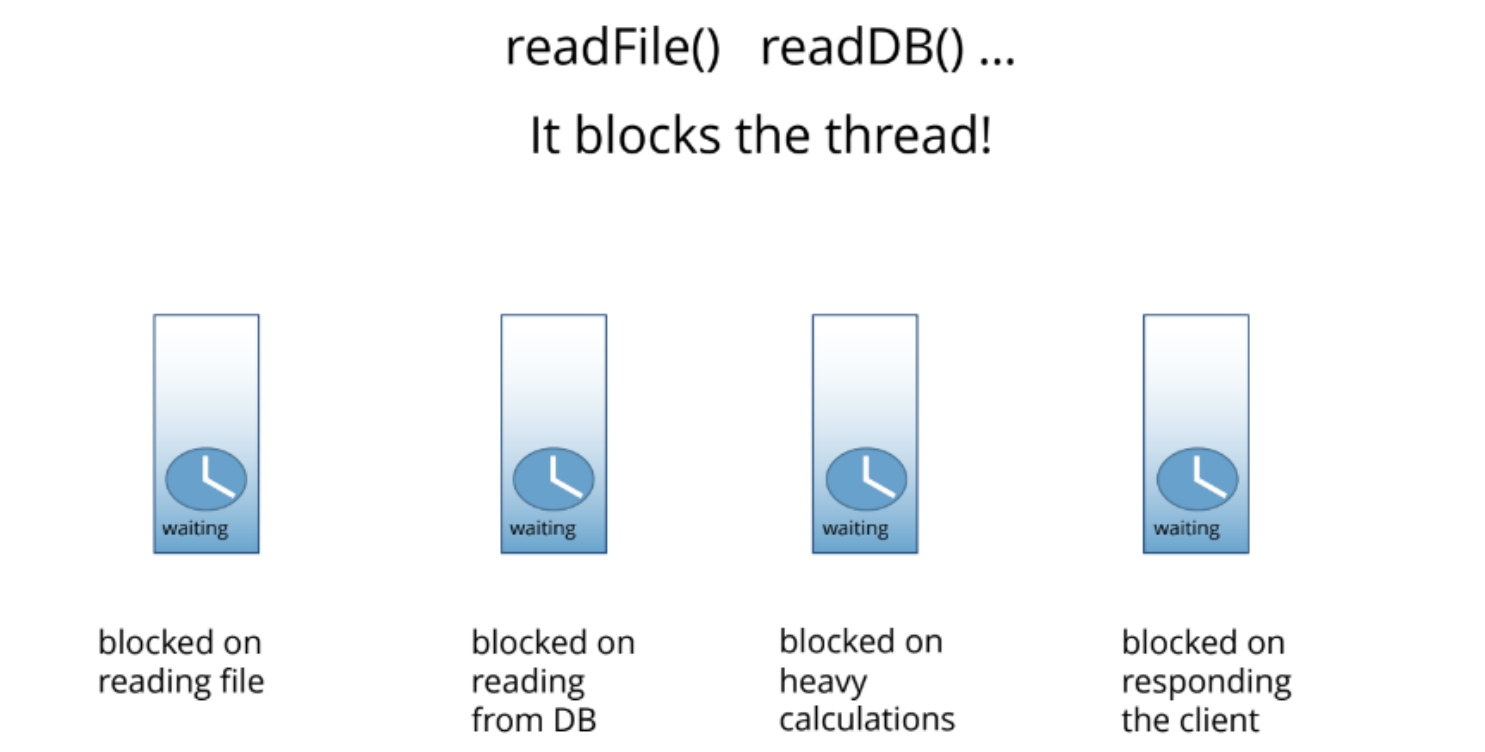
Another option is to enroll everyone in a "virtual queue". When the coffee machine is free from previous coffee drinkers, the person is notified and goes to the coffee machine without a queue. This is what many organizations do now. For example, in online stores with self-delivery. You take a ticket and go about your business, and when the time comes, you come up and get the goods. This is what asynchrony is all about: no one is waiting for anyone, everyone is working and getting their coffee as fast as possible. And the coffee machine is also not idle.

Dealt with asynchrony. But there is another important issue: blocking I/O.

Blocking I/O

Traditional I/O is blocking. What is blocking I/O?

Let's say you are reading a file or a database. The method that does this is called, and it blocks the thread: we don't do anything else, we wait. For example, you have called readFile() and are waiting for it to finally happen. The thread is blocked and does not progress: it is waiting. But in fact the processor is not busy.



In this example, threads are blocked:

- Blocked on reading file;

- On reading from the database (blocked on reading from DB);
- On complex calculations (blocked on heavy calculations);
- Blocked on responding the client.

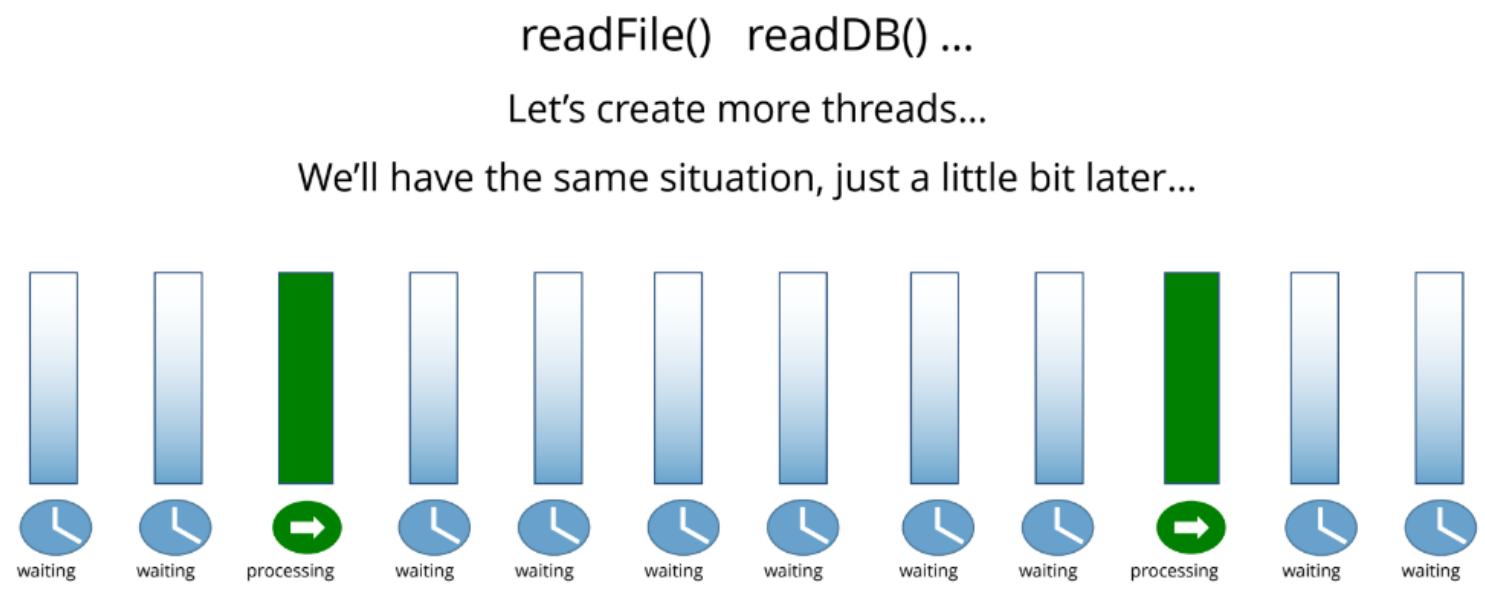
This situation is similar to when you came to the supermarket, there are four checkouts, but the checkout system slows down. The cashiers sit, do nothing and just wait for the button to be pressed at the checkout.

What if all threads are blocked? How such problems are solved in the supermarket?

Synchronous I/O

The usual option is synchronous I/O. There is little good, in this option, queues are formed at the box office.

What can be done to avoid huge queues near the cash registers? For example, you can open more checkouts, or create more flows.



More streams - more cash registers. This is a working option. But the load is uneven.

We opened a lot of cash desks (created a lot of flows), and it turns out that someone is idle. In fact, it's not just simple: when we have a lot of threads, there is an additional resource consumption. Memory consumption increases. In addition, the processor needs to switch between threads.

...and we will have context switch overhead



For sure, we have instruments to deal with these issues:

- BlockingQueue
- ThreadPool

The more threads, the more often you need to switch between them. It turns out that we have much more threads than cores. Let's say we have 4 cores, and we created a hundred threads, because all the others were blocked by reading data. Accordingly, there is a switch, the so-called context switching, so that different threads receive their portion of machine time.

But this approach has its downsides. Context switching is not free. He takes time. Procreate an unlimited number of threads would be a good option in theory. But in practice, we get a drop in speed and an increase in memory consumption.

In Java, there are different approaches that allow you to deal with this - these are blocking queues and thread pools (ThreadPool). You can limit the number of threads, and then all other clients queue up. At the start, we can have a minimum number of threads, then their number grows.

Let's return to the example of a store: if there are no people, two cash desks are open in the store, and ten are open during rush hour. But we can't open more, because then we would have to rent additional space and hire people. And it will hurt business.

Now let's talk about more modern approaches: self-service checkouts, pre-orders, and so on. So, we are getting closer to the asynchronous approach.

Asynchronous I/O

Asynchronous I/O has been known for a long time, because asynchrony is necessary when we work with the slowest I/O tool. And the slowest input-output device that you constantly have to work with is not a console or keyboard, but a person. In systems that work with a person, asynchrony appeared a long time ago.

Blocking interfaces were once used when working with a person. For example, the old DOS command line interface. And now there are such utilities that ask a question, block and do nothing else, but wait for the person to answer. Since window interfaces began to appear, asynchronous I / O has appeared. Currently, most interfaces are asynchronous.

How does asynchrony work?

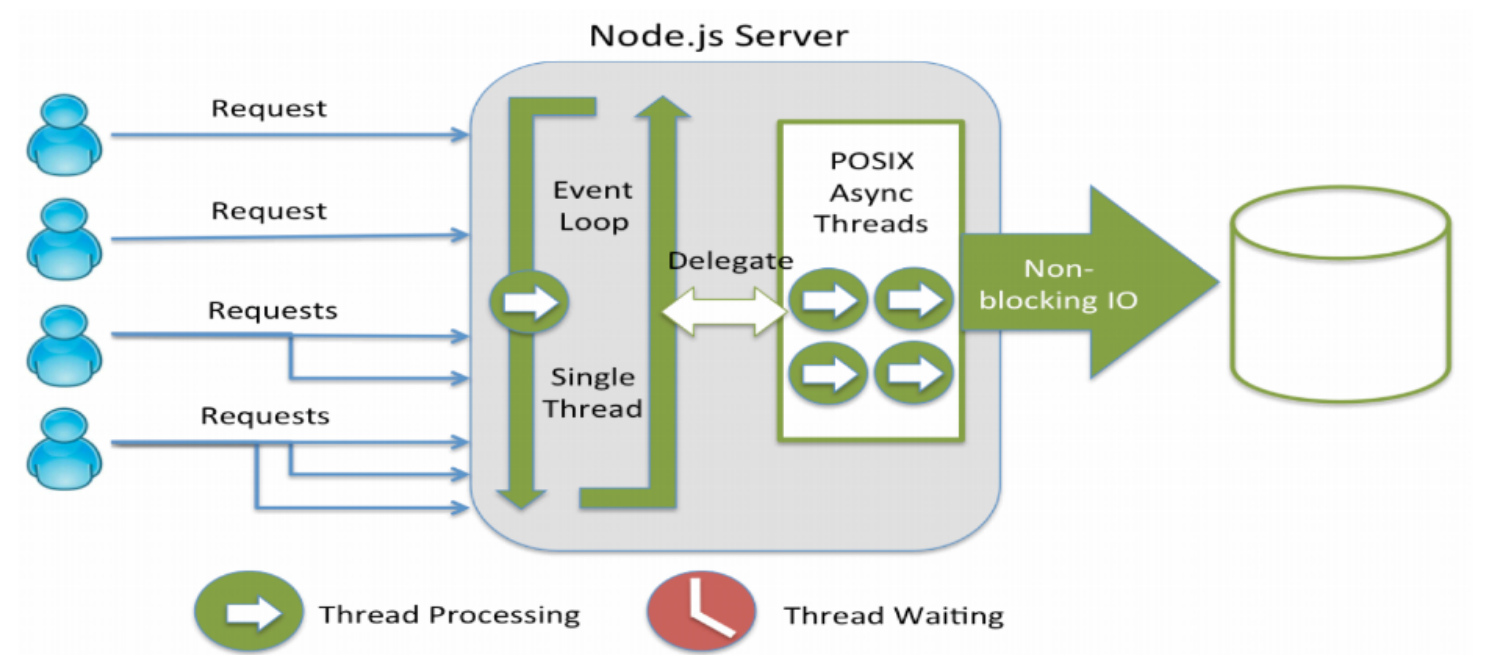
We register a callback function, but this time we don't say: "Man, enter the data, and I'll be waiting." It sounds different: "When a person enters data, please call this function - callback." This approach is used in any user interface libraries. But it was originally in JavaScript. In 2009, when the JavaScript engine started to run much faster, the smart guys decided to use it on the server and made a tool called Node.js.

Node.js

The idea of Node.js is that JavaScript is transferred to the server side, and all I / O becomes asynchronous. That is, instead of the thread blocking, for example, when accessing a file, we get asynchronous I/O. Accessing the file also becomes asynchronous. For example, if a thread needs to get the contents of a file, it says: "Please give me the contents of the file, and when it is read, call this function." We set a goal and go about our business.

This asynchronous approach proved to be very effective, and Node.js quickly gained popularity.

How does Node.js work?

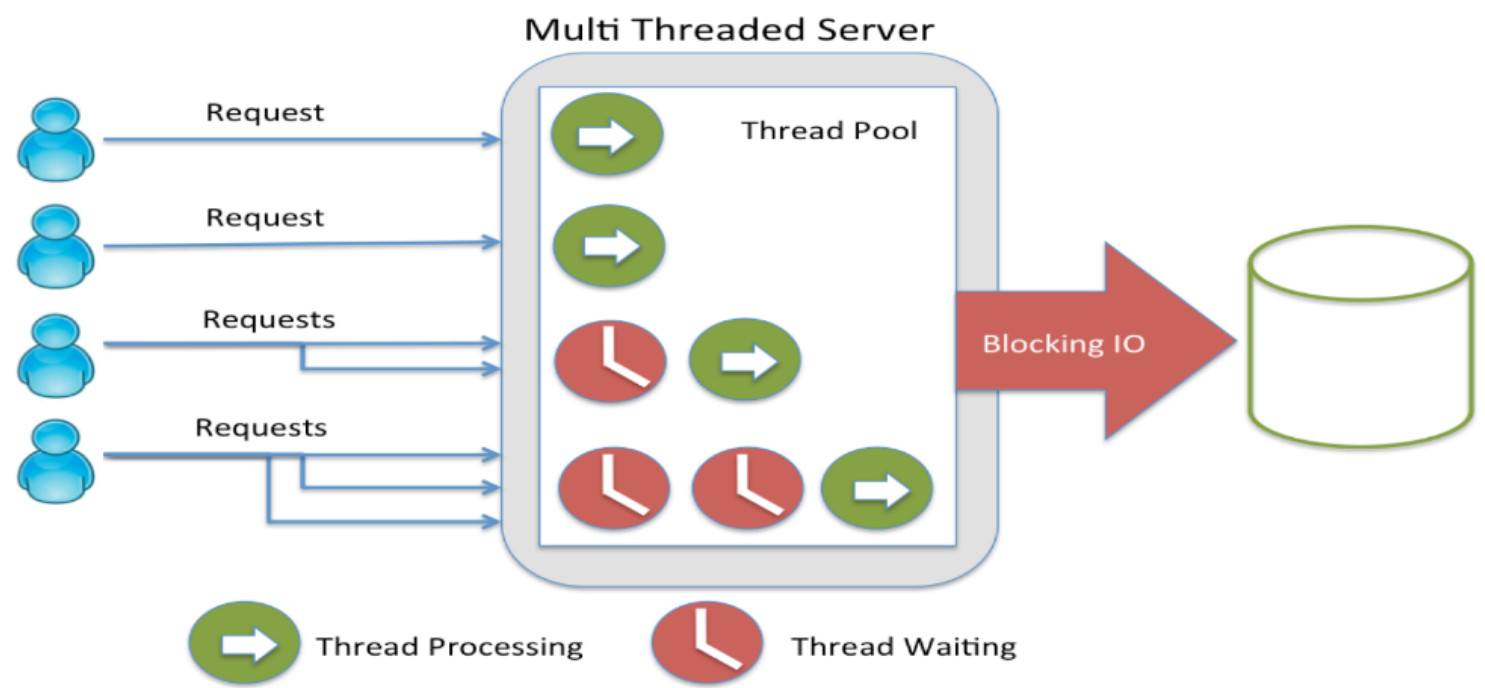


There is a receiver at the input - this is a cycle. JavaScript is a single threaded language. But this does not mean that nothing can be done there in other threads. It supports threading through Web Workers and so on. But there is one thread at the input.

Computational tasks for Node.js are usually very small. The main work is with input-output (to the database, to the file system, to third-party services, etc.). The calculations themselves take little time. When the data is received from the database or from the file system, a callback is called, that is, some kind of function to which the data is passed.

But in this scheme there is no expectation. Let's compare it with the traditional multi-threaded server model in Java.

What happens in Java?



There is a thread pool here. First, the call hits the first thread, then some thread blocked, and we created another one. It is also blocked, create the next one. And they are blocked because they refer to blocking I / O operations. For example, a thread has requested a file or data from the database and is waiting for this data to arrive.

The Node.js model quickly became popular. Naturally, at this point, people began to rewrite it in other languages. Node.js at one point took the lead in I/O-heavy systems. But it is not suitable for all systems. If you have a lot of calculations or a small number of requests, then you will not see much benefit. Accordingly, similar solutions began to appear in Java, including the platform for working with asynchronous I / O Vert.x. The Vert.x server is built on the same principle as Node.js.

The Node.js solution is interesting. It really helps to improve performance. When reactivity came, they began to use a server called Netty. This approach has proven to be very beneficial.

Multithreading history

How does multithreading work in Java? Good old Java multithreading is the basic multithreading primitives:

- Threads (threads);
- Synchronization (synchronization);
- Wait/notify (wait/notify).

Difficult to write, difficult to debug, difficult to test.

Java 5

- Future interface:
- V.get()
- boolean cancel()
- boolean isCancelled()
- boolean isDone()
- Executors
- Callable interface
- BlockingQueue

In the fifth version, the Future interface appeared. Future is a promise, or future; something that we can get from a function, the result of some asynchronous work that has not yet been completed.

The Future interface now has a get method. It blocks the call until the calculation completes. For example, we have a Future that returns data from the database, and we call the get method:

```
Future getDBData();
```

```
getDBData().get();
```

This is where the blockage occurs. In fact, there is no advantage from the fact that we used Future. When can you take advantage? For example, we set some task, perform it, access the get method, and at this moment we block:

```
Future f = getDBData();
```

```
doOutJob();
```

```
f.get();
```

If we return to the metaphor of the boss and subordinates, we set a task, did some work, and then we wait until the task is completed. And if the result of this work needs to be transferred to some other person? Previously, we considered this scenario: do the task, submit the results, and then come back. In the case of parallelism, there is no such possibility.

The capabilities of the Future interface are very limited. For example, you can find out if this task has completed:

```
Future f = getDBData();
```

```
doOutJob();
```

```
if (f.isDone) doOtherJob();
```

```
f.get();
```

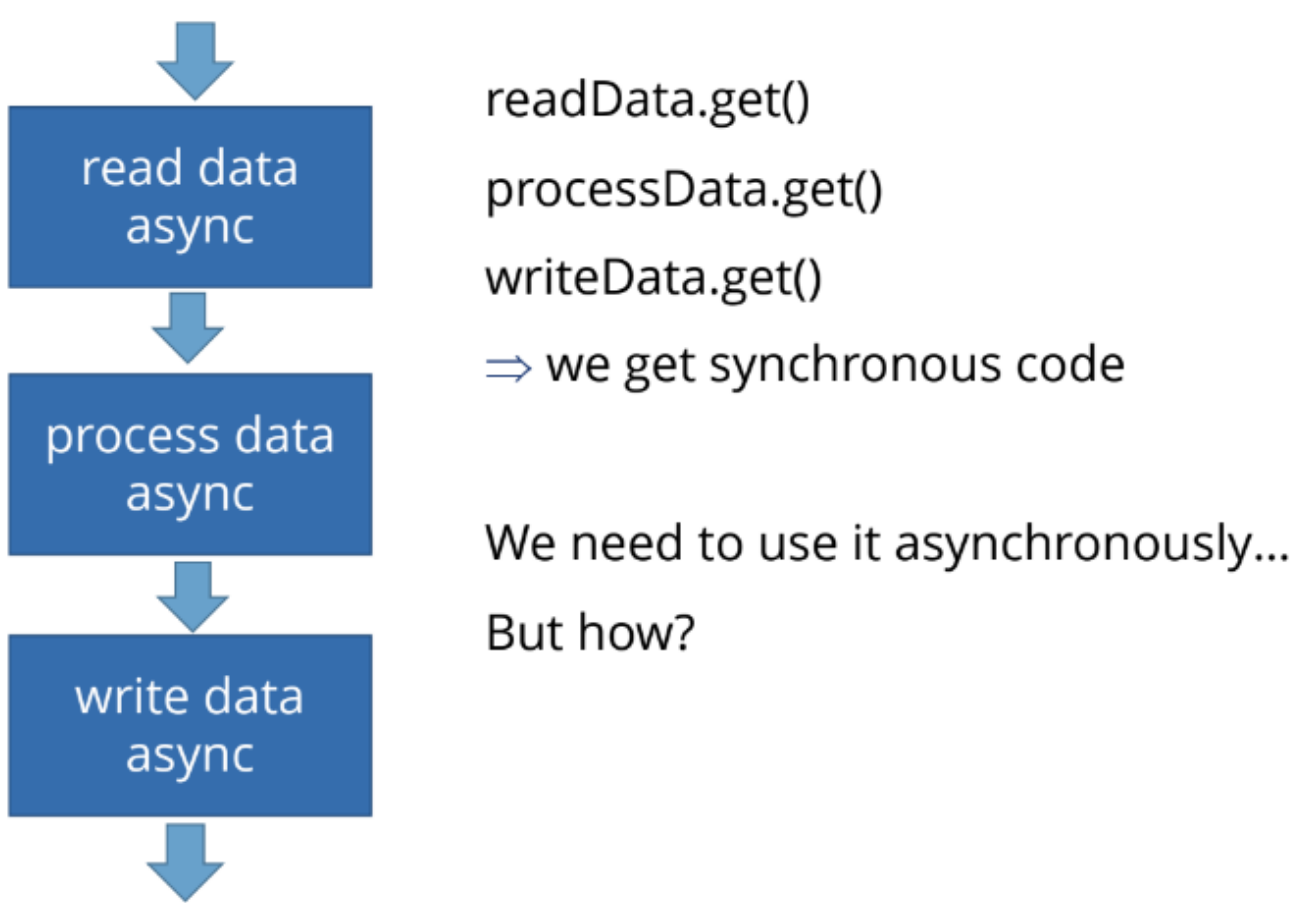
If the task is not yet completed, we can do something else. But in any case, let's skip this moment: either the task will not have time to complete again, or, conversely, it has already been completed a long time ago, and we were doing other work. The moments of synchronization here are not very clear.

The Future interface was very limited in Java 5, so it was awkward to use.

Let's think about what business challenges a typical application faces?

Data flow

Typically, the typical task of an application is to read, process, and write data. If we want to do this asynchronously, we need to use asynchronous reading, processing and writing.



For example, if an async function is blocking, we've written some great code:

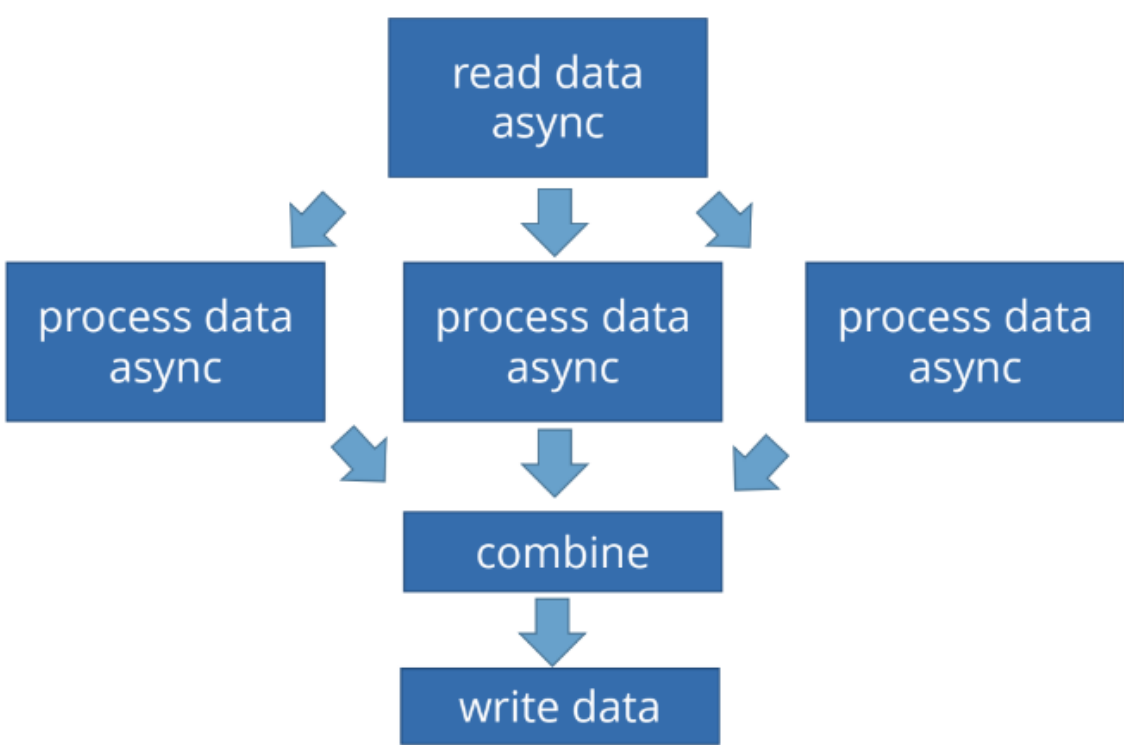
readData.get() and blocked,

processData.get() and blocked,

writeData.get() blocked here too.

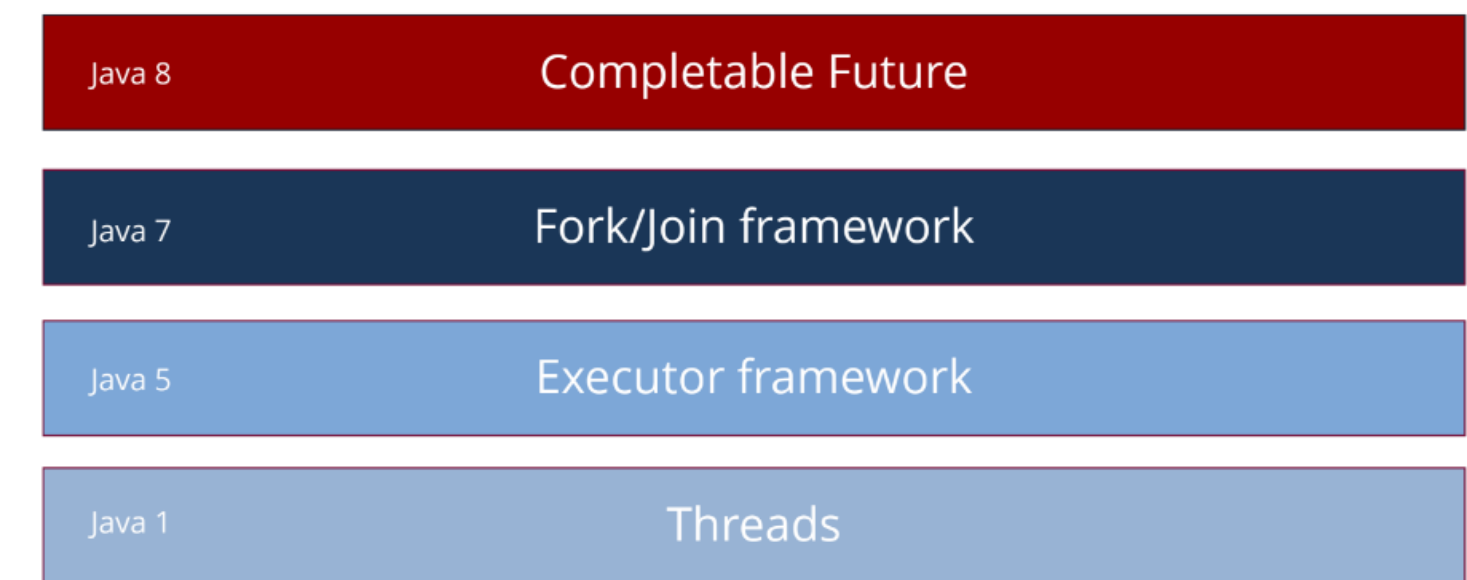
We get a synchronous code at the output. There is no asynchrony here, it is inconvenient to use it.

Lef's consider a typical task when there is an asynchronous reading of data, and then we want to process it "in three throats":



In order to wait for the result of reading, many threads are not needed. We just have to get the data. Then they need to be processed, and processing is a resource-intensive task from the point of view of the processor, and it would be nice to parallelize it. We say, "Read the data. When you have done this, process them in three threads, then combine the results of the execution and write the data. I would like it to be done asynchronously.

CompletableFuture brings us to the Async world



Java 8 introduced CompletableFuture. It is based on the Fork/Join framework. The same way, by the way, as the parallelization of threads. Fork/Join framework was introduced in Java 7, but it was hard to use. In version 8, CompletableFuture was a step forward: towards an asynchronous world.

Lef's consider a simple example.

In the code, the CompletableFuture methods from the standard JDK are highlighted in orange.

```
// API
public CompletableFuture<Data> readData(Source source);
public Data processData1(Data data);
public Data processData2(Data data);
public Data mergeData(Data a, Data b);
public void writeData(Data data, Destination dest);

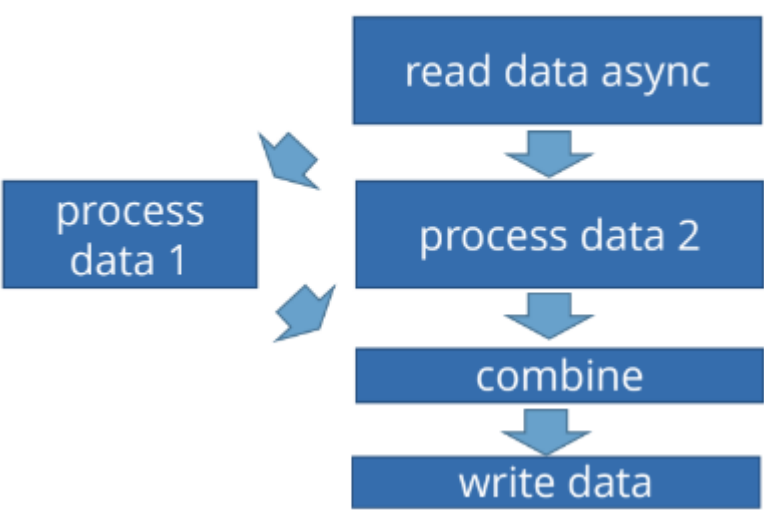
CompletableFuture<Data> data = readData(source);
CompletableFuture<Data> processData1 =
    data.thenApplyAsync(this::processData1);
CompletableFuture<Data> processData2 =
    data.thenApplyAsync(this::processData2);

processData1
    .thenCombine(processData2,
        (a, b)->mergeData(a,b))
    .thenAccept(
        data->writeData(data, dest));
```

Lef's say we have an API that allows us to:

- Read data (readData) from source and return CompletableFuture because it is asynchronous;
- Process data, for which there are two handlers: processData1 and processData2;
- Merge data (mergeData) after they are processed;
- Write data (writeData) to the receiver (Destination).

This is a typical task - to read the data, process it "in two throats", then combine the results of this processing and write it down somewhere.



We have read the data:
CompletableFuture data = readData(source);

Then we say: when we read the data, we need to send it for processing:

CompletableFuture processData1 = data.thenApplyAsync(this::processData1);

This means that you need to run their processing in a separate thread. Since we are using Async postfix here, processing starts in two different threads:

CompletableFuture processData2 = data.thenApplyAsync(this::processData2);

That is, the functions this::processData1 and this::processData2 will run on two different threads and will be executed in parallel. But after running in parallel, their results should merge. This is what thenCombine does.

We started two threads of execution here, and when they finished, we combined them. thenCombine works like this: it waits for both processData1 and processData2 to complete, and then calls the data combine function:

thenCombine(processData2,

(a, b)->mergeData(a,b))

That is, we combine the results of the first and second processing, and after that we write the data:

thenAccept(

data->writeData(data, dest));

Here we get a chain, which is essentially a business process. We seem to be saying: "Tanya, take the data from the archive, give it to Lena and Grisha for processing. When both Lenya and Grisha bring results, pass them on to Vera so that she connects them, and then give them to Vita to write a report on these data.

We don't have a clear timetable here, which we talked about at the beginning: there is an opportunity to transfer data as soon as we can. The only one waiting here is thenCombine. It waits until both processes, the result of which it combines, have completed.

CompletableFuture is a really cool approach that helps build asynchronous systems.

In the next part of our article, we will talk about higher-level approaches to asynchrony and analyze reactive programming operators.

The HighLoad++ 2020 conference will take place on May 20 and 21, 2021. [You can buy tickets](#) now.

Do you want to receive the Saint HighLoad++ 2020 mini-conference conference materials for free? [Subscribe](#) to our newsletter.

Tags:

java, asynchronous programming, reactive programming, upload, conferences

Hub:

Blog of Oleg Bunin Conference Company (Ontiko), High performance, Programming, Java, Parallel programming


✕

Editorial digest


We send the best articles once a month

Electric mail

→



Conferences by Oleg Bunin (Ontiko)
Professional conferences for IT developers
[Twitter](#) [Telegram](#) [In contact with](#)



22
Karma


0
Rating

Alena Mironova

@hedgehog_on_rainbow

Content editor

Comments 11




jugard 02/24/2021 at 16:45

I read about electrons moving at the speed of light, I write a comment before closing the article. And why are these physicists building accelerators, can they learn how to make transistors?

−3

Answer



alkimlenko 25.02.2021 at 18:44


Everything about that offer is great.

As the electrons travel at the speed of light, due to the reduction in size, the time it took for an electron to run all the way inside the processor was reduced.

It turns out that the frequency of processors is growing, because the electrons need to run less) We have already mentioned the light speed of electrons. In reality, it is in conductors of the order of a millimeter per second

0

Answer




sergey-gornostaev 02/24/2021 at 19:31

From the text, you may get the misconception that Node.js appeared before Netty and asynchrony was not used in Java before it, but this is not so.

+2

Answer




grobitto 25.02.2021 at 13:02

> Hard to write, hard to debug, hard to test.

For the first time I see these epithets in relation to synchronous / multi-threaded code when compared with asynchronous. Debugging it is just a pleasure compared to a hodgepodge of callbacks. As soon as project Loom with its lightweight threads (Fibers) goes into production, it will be great, simple understandable synchronous code without performance problems

+1

Answer




Kirillchug 02/25/2021 at 18:42

Asynchronous code is very convenient to debug. We at the company use Project Reactor. It is written as a regular Stream API, and if necessary, you can safely "fall through" into one of its stages. And in multi-threaded, the minus is that you can not wait for the result at which you had an error, because threads are not predictable. In general, they perform the task correctly, but at a specific level, it is not clear how they can work with data.

0

Answer



grobitto 27.02.2021 at 03:15

In general, asynchrony is not the antonym of multithreading, but of synchronism. Asynchronous code works in different threads in the same way, and requires synchronization when accessing resources, etc. Its advantage is in speed, since it is impossible to create millions of heavy threads for modern tasks, but this advantage is achieved at the cost of complicating development and debugging, and this is a generally recognized fact.

As soon as fibers are brought into java, the performance problem will be solved by itself, and externally

synchronous code will work under the hood as asynchronous, and fibers can be produced by the millions

0

Answer

+++

sergey-gornostaev

02/27/2021 at 09:23

^

Asynchronous code works exactly the same in different threads

Not necessarily, the event loop can run on a single thread.

As soon as fibers are brought into java, the performance problem will be solved by itself

There is no silver bullet . Read for example a series of articles "[Do Looms Claims Stack Up](#)".

0

Answer

+++

tonhead

02/25/2021 at 18:42

Only after reading the article, I realized that there is nothing about reactive programming in java, which is a pity

+5

Answer

+++

OlegRazgul

03/02/2021 at 14:52

When will the next part come out?

0

Answer

+++

hedgehog_on_rainbow

03/02/2021 at 14:52

^

The article is tentatively due out on Wednesday, March 10th.

0

Answer

+++

zesetup

03/08/2021 at 03:02

↗

But in this scheme there is no expectation. Let's compare it with the traditional multi-threaded server model in Java.

What are the tricks? There is an expectation there. There's just an easier way to call the handler on completion(calback, promise). In Java, this must be done "manually", that is, checking the status of the stream and calling some code when receiving its result.

0

Answer

+++

Only authorized users can post comments. [Sign in](#) please.

Publications

[BEST OF THE DAY](#) [SIMILAR](#)

- amazing_mike

13 hours ago

Robot-not-vacuum cleaner with a knife or how we made smart scissors on wheels

7 min

3.6K

case

+49

16

30 +30
- oldadmin

10 hours ago

How to use GitLab under sanctions?

Average

5 min

6.1K

+31

34

19 +19
- T1_Consulting

12 hours ago

How Taras became Senior+ due to CS 1.6 and grandfather with CHP

Simple

5 min

6.7K

case

+24

13

11 +11
- Kudessnick33

11 hours ago

Registers vs libraries on the example of hearts

15 min

2.1K

case

+22

26

20 +20
- monreva

17 hours ago

Tiny11: thinner Windows 11. Can it really run on older PCs? Part 1

4 min

24K

+18

42

77 +77

[show more](#)

Your account

To come in
Registration

Sections

Publications
news
Hub
Companies
The author
Sandbox

Information

Device site
For authors
For companies
Documentation
Agreement
Confidentiality

Services

Corporate blog
Media advertising
Native projects
Educational programs
I'm starting up
Megaprojects