How does non-blocking IO work under the hood?

Hielke de Vries · Follow
Published in ING Blog
8 min read · Nov 20, 2019

Listen Share More

At ING, we host several applications that have to handle 10000s of requests per second. For instance, we run a cluster of Nginx web servers and reverse proxies that serve all our customer-facing applications. With millions of customers using our applications every day, peak loads come up to 20K requests per second. Some of our internal applications have to handle even higher loads. An ING squad in Friesland has built the application *TracING*, that uses Vert.x to handle more than 50K requests per second during peaks.

how do we handle lots of traffic?

To deal with these amounts of concurrent connections, *thread-per-connection* models do not suffice. Nginx and Vert.x instead use *non-blocking* models to achieve such high levels of concurrency. As someone who likes to figure out how things work under the hood, this made me want to understand the following:

How does the caller of a non-blocking/asynchronous API gets notified when data is ready? How does this work under the hood, at low level?

After calling a non-blocking API, at some point later in time the caller must be notified that the data is ready. But *how* does the caller know when the data is ready? Does it get signalled? Is there some mechanism that constantly checks if the data is ready? When you think about the fact that a CPU can only execute instructions sequentially, how could it be possible that software or hardware can "listen" for an event or notification when the data is ready?

The only "listener" that is natively available in most computers is the hardware interrupt processor. But hardware interrupts do not scale well and lack flexibility. And what happens when there are 100k events per second, will there be 100k hardware interrupts per second? This also seems unlikely.

My first guess was that there should be some kind of constant checking, like an infinite loop, that polls to see if data is ready. But that also seems inefficient at first sight. An infinite loop must be taking quite some CPU time, right? First let's understand some basics about IO and blocking.

Blocking IO

Input/output (IO) refers to interaction with devices such as a hard drive, network or database. Generally anything that is not happening in the CPU is called IO. When you call an API that requests data from IO, you will not get a response instantly, but with some *delay*. This delay can be very small for requesting a file on a hard drive, and much longer when requesting data from a network. This is because the data you request from IO devices has to travel longer to the caller. For instance:

- A file stored on a **hard drive** must be transferred through SATA cables and main board buses to the CPU
- The data from a **network** resource located on a server far away must travel through network cables, routers and eventually the network interface card (NIC) in your computer to the CPU

Calling an API that requests data from IO will cause the running thread to “block”, i.e. it is waiting until the requested data has returned to the caller. When a thread is blocked in Linux, it will be put in a Sleep state by the kernel until data has returned to the caller. Threads in sleep state immediately give up its access to the CPU, so to not waste CPU time. After IO is ready, the thread is taken out of the Sleep state and put in Runnable state. Threads in this state are eligible to be executed on the CPU again. The thread scheduler will put the thread on a CPU when one is available. The

> *are only two states: running and not running. Java threads have six different states.*

Why non-blocking IO?

The main benefit of non-blocking IO is that we need less threads to handle the same amount of IO requests. When multiple calls to IO are done using *blocking IO*, for each call a new thread is created. A thread costs around 1MB, and there are some costs due to context switching. If you have a web server that handles 50k connections per second, a thread per connection can be quite expensive.

Types of blocking

There are actually two types of thread blocking:

- CPU-bound blocking
- IO-bound blocking

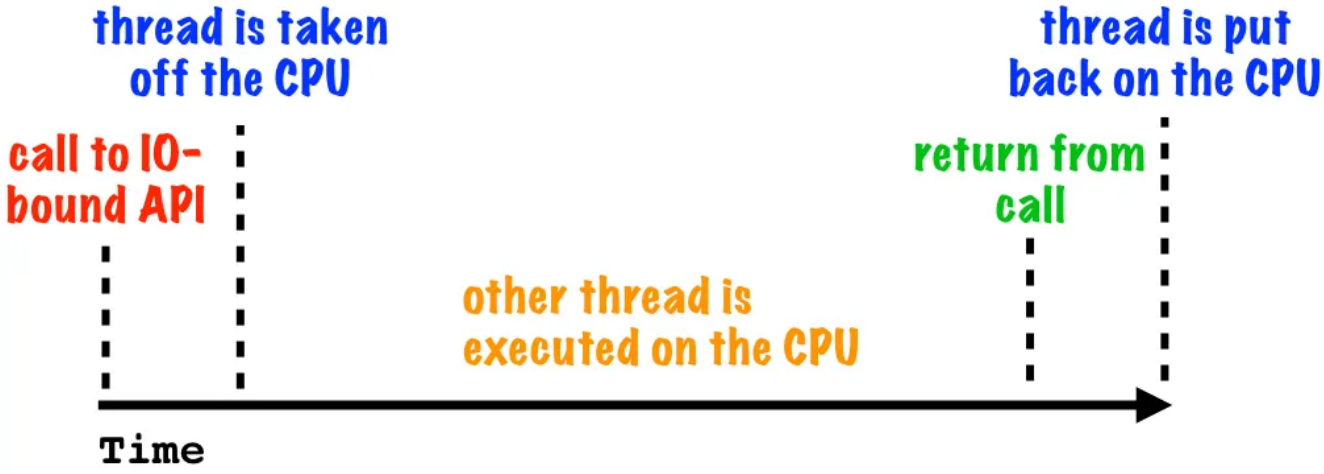
CPU-bound blocking

In this case the thread gets blocked because of some CPU intensive task it performs takes more time than “instantly”. For example when generating a bunch of prime numbers or rendering a 3d model. With CPU-bound blocking the thread is blocked because it’s actively being executed on the processor.

CPU-bound blocking

IO-bound blocking

Here, the thread gets blocked because it has to wait for data to return from an IO source, such as a network or a hard drive. The kernel will notice that there is no data available from IO and will therefore put the thread in some "sleep" state. Hence, with IO-bound blocking the thread is *not*[1] actively being executed on the processor.



IO-bound blocking

Non-blocking IO

APIs that use blocking IO will block the thread until data from IO has returned. So what happens when you call a non-blocking API? Very well, it returns instantly and will *not* block the thread. This means the thread can immediately continue executing the code that comes after calling the API.

When data has returned from IO, the caller will be notified that the data is ready. This is generally done with a *callback* function that has access to the returned data.

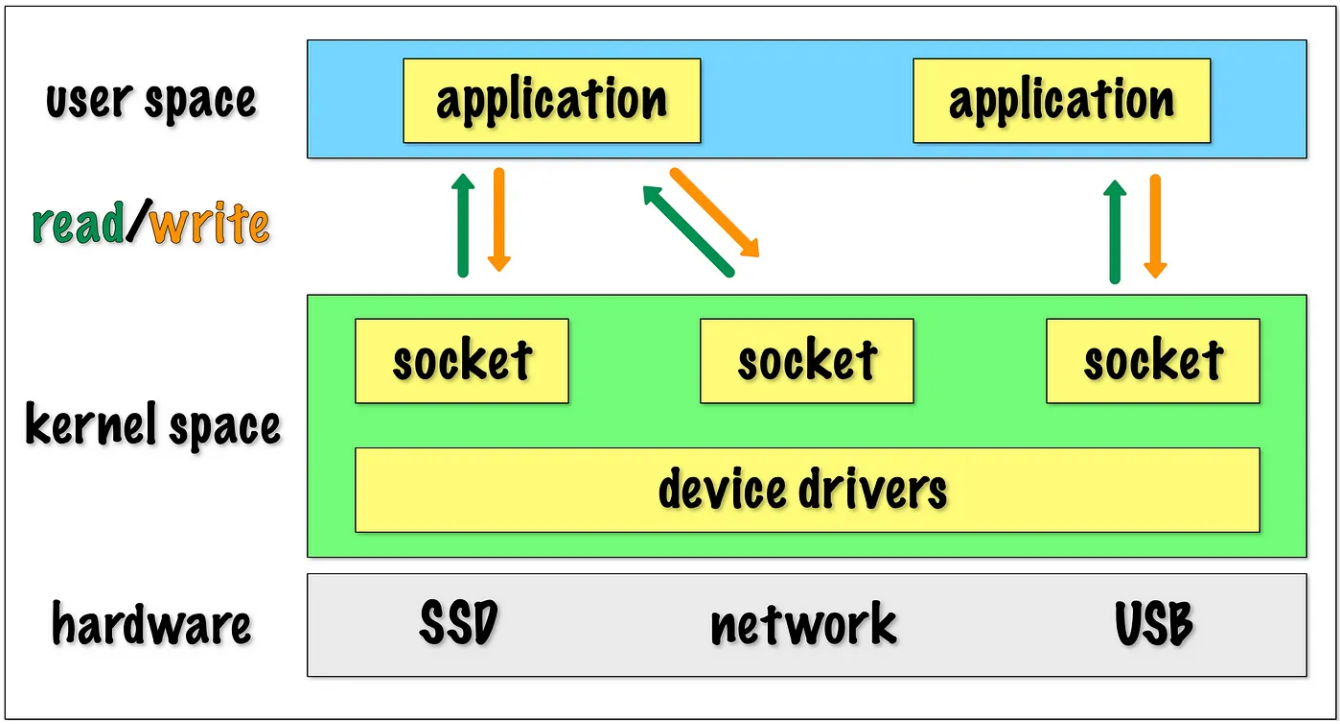
callback

> *There are other ways to express a non-blocking or asynchronous action with e.g. futures, promises or coroutines. These constructs are only syntactically different. Under the hood they are all based on a routine (function) that is called the moment data has returned from IO.*

Network IO and sockets

To understand how non-blocking IO works under the hood we first need some understanding of how IO works at low level. A common use case for non-blocking IO is network IO, so it is best explained in this context. At kernel level a *socket* is used as an abstraction to communicate with a NIC. This socket takes care of reading and writing data to/from the NIC, which in turn sends the data over the UTP cable on its way to the internet. For example, if you go to a URL in your browser; at low level the data in your HTTP request is written to a socket using the `send(2)` system call. When a response is returned, the response data can be read from that socket using

the `recv(2)` system call. The important thing to understand here is that when data has returned from network IO, it is ready to be read from the socket.



Non-blocking IO under the hood

Most non-blocking frameworks use an infinite loop that constantly checks (polls) if data is returned from IO. This is often called the *event loop*. An event loop is literally a `while(true)` loop that in each iteration will check if data is ready to read from a network socket. Technically, sockets are implemented as *file descriptors* (FD) on UNIX systems. It is therefore better to say that a FD is checked for ready data. The list of FDs that you want to check for ready data is generally called the *interest list*.

> *"event", "FD readiness" and "FD is ready for data" are synonyms in this text*

You might think that an event loop can be expensive on the CPU if it's endlessly running, but there are some optimizations to make them more efficient.

Let's zoom a bit in on the event loop to see how these optimizations work. Each (major) operating system provides kernel level APIs to help create an event loop. In Linux there is `epoll` or `io_uring`, BSD uses `kqueue` and Windows has `IOCP`. Each of these APIs is able to check FDs for ready data with a computational complexity of around O(number_of_events_occurred). In other words, you can monitor 100.000s of FDs, but the API's execution speed only depends on the amount of events that occur in the current iteration of the event loop. Compared to the older `poll` and `select` APIs this is a huge improvement:

```
operations    |  poll  |  select   | epoll
10            |   0.61 |    0.73   | 0.41
100           |   2.9  |    3.0    | 0.42
1000          |  35    |   35      | 0.53
10000         | 990    |  930      | 0.66
```

From: The Linux programming interface (Michael Kerrisk), table 63–9

Another important “optimization” is that a call to `kqueue(..)` or `epoll(..)` blocks if there is no ready data in the FDs of the interest list. This means that there are no unnecessary iterations in the event loop when there is no ready FD data.

Further optimization of the event loop can be done with what `epoll` calls “edge triggered mode” and `kqueue` calls `EV_CLEAR`. When there is new data available in at least one of the FDs in the interest list, and when this data is *not* (completely) read/emptied from the FDs, then in the next iteration of the event loop this FD will not be seen as having new data ready. This will save unnecessary iterations in the case that a FD is not completely read when the next iteration of the event loop is executed.

Here is an outline of how an event loop using `kqueue` would look like:

event loop outline using `kqueue`

> *Here is a complete TCP echo server example using a* `kqueue` *event loop*

IO_uring

From kernel 5.1 Linux offers `io_uring`, which can even further optimize an event loop. It does so by minimizing expensive system calls and minimize copying of data.

Other ways to achieve non-blocking IO

We have only looked at event loops as a way to achieve non-blocking behavior. It is possible to implement non-blocking IO in other ways. For instance, *hardware interrupts* are used to stop the current executing thread (interrupt it), execute some code and return to the thread. Hardware interrupts are not ideal for applications that have high event rates such as high-performance web servers. For each new request the CPU is interrupted and this will slow down the CPU too much.

Another possibility is using *signals*, which are used for inter-process communication. A signal could also be used as an "event" to signal ready IO data. Catching signals is expensive and therefore also not preferred over event loops in high event rate applications.

Hardware interrupts and signals do not offer the flexibility and scalability that an event loops offer. Of course, it this all depends on your needs. E.g. in embedded systems, hardware interrupts are often preferred over event loops.

Conclusion

Applications that need to handle high event rates mostly use non-blocking IO models that are implemented with event loops. For best performance, the event loop is built using kernel APIs such as `kqueue`, `io_uring`, `epoll` and `IOCP`. Hardware interrupts and Signals are less suited for non-blocking when handling large amounts of events per second.

. . .

[1] depending on the load and the thread scheduling algorithm.

Thanks to Bart Warmerdam for reviewing this text.

Non Blocking Io | Asynchronous | Event Loop | Io Uring | Epoll

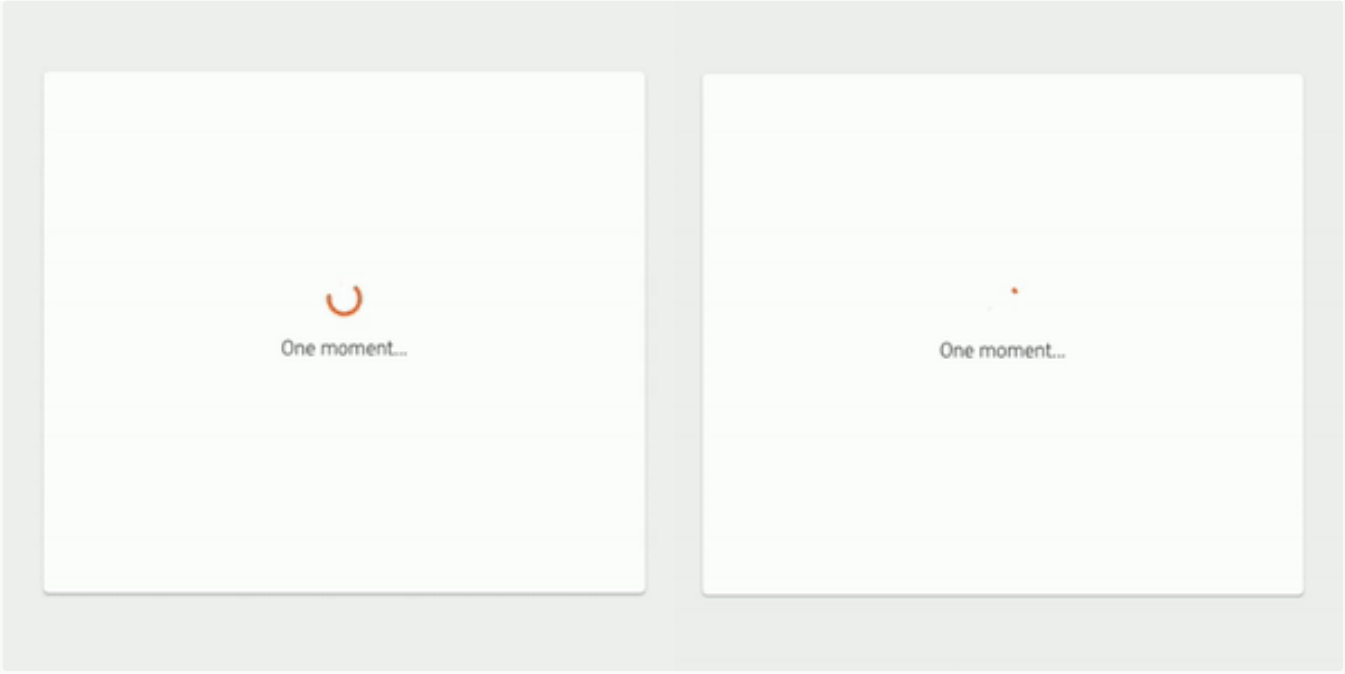
Follow

Written by Hielke de Vries

82 Followers · Writer for ING Blog

Software Developer and Security Engineer

More from Hielke de Vries and ING Blog



Pavlik Kiselev in ING Blog

Shared element transitions or how to fight Cumulative Layout Shift

A concise story about how to make a smooth transition between pages if you don't know the exact size of the coming elements with a new...

5 min read · Nov 3, 2022

Carlos Muñoz in ING Blog

Best practices to build Web Components

What are the key things that you should take into account when you start building a new Web Component?

9 min read · Mar 23, 2021

41

Mateusz Garbacz in ING Blog

Open-sourcing ShapRFECV — Improved feature selection powered by SHAP

This post introduces ShapRFECV, a new method for feature selection in decision-tree-based binary classification models.

9 min read · Dec 15, 2020

192 6

Maxime Gerbe in ING Blog

React Native share PDF

TL;DR: Use react-native-share-pdf library to share base64 PDF on Android/iOS. Disclaimer: Intended for React developers.

5 min read · Jun 5, 2019

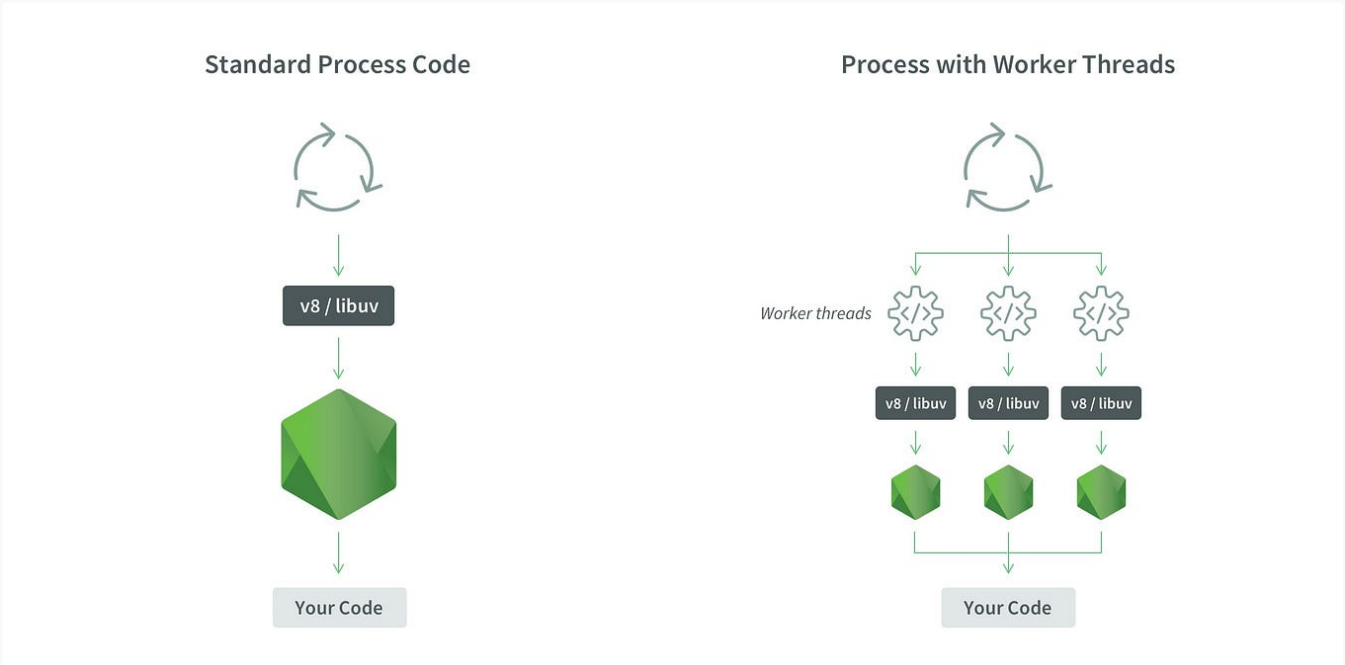
Arslan Ahmad in Level Up Coding

Don’t Just LeetCode; Follow the Coding Patterns Instead

What if you don’t like to practice 100s of coding questions before the interview?

5 min read · Sep 15, 2022

4.7K 26



Pablo Curty in Better Programming

How to Use Promises and “worker_threads” in Nodes to Parallelize CPU Usage

Discover a new way to work with asynchronous operations in JavaScript more easily and intuitively

14 min read · Apr 25

138 4

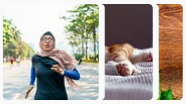
Lists

Staff Picks

467 stories · 332 saves

Stories to Help You Level-Up at Work

19 stories · 237 saves

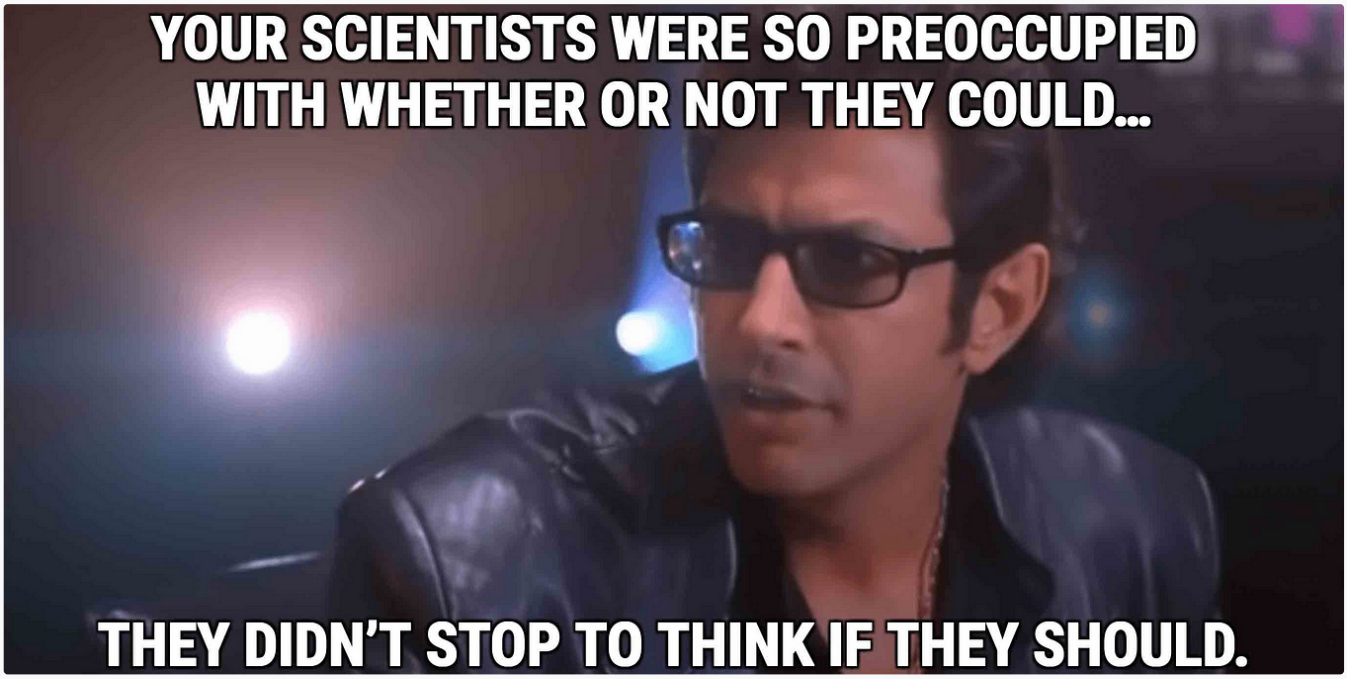


Self-Improvement 101

20 stories · 670 saves

Productivity 101

20 stories · 616 saves



Paul Folbrecht

The Happy (Hopeful) Death of OOP

I am certainly not unique as a developer in coming to eschew “object-oriented programming” in the vein of the typical C++/Java/.Net...

7 min read · Jun 21

123 12

Full Stack from Full-Stack

Top 25 Kafka interview questions

1. What is Apache Kafka and why is it used?

5 min read · Aug 27

15

Tuhin Banerjee in Bits and Pieces

A Comprehensive Guide to Redis Cluster

How Redis Cluster works—communication, management, and more and learn how to set up a Redis Cluster locally.

15 min read · Aug 1

Naveen

Single-Threaded vs. Multi-Threaded Programs in Java: A Comprehensive Comparison

Introduction Java is a versatile programming language that supports concurrent programming through threads. Threads allow programs to...

4 min read · Jul 22