**Your own table factory**

**Your first employee and work bench**

Imagine you are starting a business which produces tables. You are renting a small building and buying a single work bench because you only have one employee, let’s call him George. In the morning, George enters the building, goes to the work bench, and picks a new order from the inbox.

Tables vary in size and color. The respective resources and supplies are available in the store room. However sometimes the store room does not have the required materials, e.g. a color is missing, so George has to order new supplies. Because George likes to finish one thing before he starts another, he will simply wait at the work bench until the new supplies are delivered.

In this analogy, the factory represents a computer system, the work bench represents your CPU, and George is a working thread. Ordering new supplies corresponds to I/O operations and you can be seen as the operating system, coordinating all the interactions. The CPU has no multi-tasking capabilities and every operation is not only blocking a thread but the whole CPU and thus the whole computer.

**Multiple employees, single workbench**

You are wondering if you could increase the productivity by convincing George to work on something else while the supplies are being delivered. It can take multiple days before a new delivery arrives and George will just stand there doing nothing. You confront him with your new plan but he replies: “I’m really bad at context switching, boss. But I’d be happy to go home and do nothing there so I’m at least not blocking the work bench!”.

You realize that this is not what you had hoped for but at least you can hire another employee to work at the bench while George is at home, waiting for the delivery. You are hiring Gina and she is assembling another table while George is at home. Sometimes George has to wait for Gina to finish a table before he can continue his work but the productivity is almost doubled nevertheless, because George’s waiting time is utilized much better.

By having multiple employees sharing the same workbench we introduced a form of multi-tasking. There are different multi-tasking techniques and here we have a very basic one: As soon as a thread is blocked waiting for I/O it can be parked and another thread can use the CPU. In I/O heavy applications this approach however requires us to hire more employees (spawn more threads) that will be waiting. Hiring workers is expensive. Is there another way to increase productivity?

**Multitasking, non-blocking employees**

In her second week, Gina also ran out of supplies. She realized that it is actually not that bad to simply work on another table while waiting for the delivery so she asks you to send her a text message when the delivery arrived so she can continue working on that table as soon as she finishes her current work or is waiting for another delivery.

Now Gina is utilizing the work bench from 9 to 5 and George realizes that she is way more productive than him. He decides to change jobs, but luckily Gina has a friend who is as flexible as her and thanks to all the tables you sold you can afford a second work bench. Now each work bench has an employee working the whole day, utilizing waiting time for supply deliveries to work on another order in the meantime. Thanks to your notification on arrived deliveries they can focus on their work and do not have to check the delivery status on a regular basis.

After changing the working mode to no longer idle when waiting for deliveries, your employees are perfoming I/O in a non-blocking way. Although George was also no longer blocking the CPU after he started waiting for the delivery at home, he was still waiting and thus blocked. Gina and her friend are simply working on something else, suspending the assembly of the table which requires supplies to be delivered, waiting for the operating system to signal them that the I/O result is ready.

**Benefits of non-blocking I/O**

I hope the previous analogy made it clear what the basic idea of non-blocking I/O is. But when is it useful? Generally one can say that the benefit starts kicking in once your workload is heavily I/O bound. Meaning your CPU would spend a lot of time waiting for your network interfaces, for example.

Using non-blocking I/O in the right situation will improve throughput, latency, and/or responsiveness of your application. It also allows you to work with a single thread, potentially getting rid of synchronization between threads and all the problems associated with it. Node.js is single-threaded, yet can handle [millions of connections](http://blog.caustik.com/2012/08/19/node-js-w1m-concurrent-connections/) with a couple of GB RAM without problems.

A common misconception lies in the fact that non-blocking I/O means fast I/O. Just because your I/O is not blocking your thread it does not get executed faster. As usual there is no silver bullet but only trade-offs. There is a nice blog post on [TheTechSolo](https://thetechsolo.wordpress.com/2016/02/29/scalable-io-events-vs-multithreading-based/) discussing advantages and disadvantages of different concepts around this topic.

**Implementations**

There are many different forms and implementations of non-blocking I/O. However all major operation systems have built-in kernel functions that can be used to perform non-blocking I/O. [epoll](http://man7.org/linux/man-pages/man7/epoll.7.html) is commonly used on Linux systems and it was inspired by [kqueue](https://www.freebsd.org/cgi/man.cgi?query=kqueue&sektion=2) ([research paper](https://people.freebsd.org/~jlemon/papers/kqueue.pdf)) which is available in BSD based systems (e.g. Mac OS X).

When using Java, the developer can rely on Java NIO. In most JVM implementations you can expect Java NIO to use those kernel functions if applicable. However there are some subtleties when it comes to the details. As the Java NIO API is generic enough to work on all operating systems, it cannot utilize some of the advanced features that individual implementations like epoll or kqueue provide. It resembles very basic poll semantics.

Thus if you are looking for a little bit of extra flexibility or performance you might want to switch to native transports directly. [Netty](https://netty.io/), one of the best network application framework on the JVM, supports both Java NIO transports as well as [native libraries](https://netty.io/wiki/native-transports.html) for Linux and Mac OS X.

Of course most of the time you are not going to work with Java NIO or Netty directly but use some web application framework. Some frameworks will allow you to configure your network layer to some extend. In [Vert.x](https://vertx.io/), for example, you can choose whether you want to use native transports if applicable and it offers

* [EpollTransport](https://github.com/eclipse-vertx/vert.x/blob/master/src/main/java/io/vertx/core/net/impl/transport/EpollTransport.java) based on Netties [EpollEventLoopGroup](https://netty.io/4.1/api/io/netty/channel/epoll/EpollEventLoopGroup.html),
* [KQueueTransport](https://github.com/eclipse-vertx/vert.x/blob/master/src/main/java/io/vertx/core/net/impl/transport/KQueueTransport.java) based on [KQueueEventLoopGroup](https://netty.io/4.1/api/io/netty/channel/kqueue/KQueueEventLoopGroup.html), and
* [Transport](https://github.com/eclipse-vertx/vert.x/blob/master/src/main/java/io/vertx/core/net/impl/transport/Transport.java) based on [NioEventLoopGroup](https://netty.io/4.1/api/io/netty/channel/nio/NioEventLoopGroup.html).

**Final thoughts**

The term non-blocking is used in many different ways and contexts. In this post we were focusing on non-blocking I/O which refers to threads not waiting for I/O operations to finish. However sometimes people refer to APIs as non-blocking only because they do not block the current thread. But that doesn’t necessarily mean they perform non-blocking I/O.

Take JDBC as an example. JDBC is blocking by definition. However there is a [JDBC client](https://vertx.io/docs/vertx-jdbc-client/java/) out there which has an asynchronous API. Does it block your thread while waiting for the response of the database? No! But as I mentioned earlier, JDBC is blocking by definition so who is blocking? The trick here is simply to have a second thread pool that will take the JDBC requests and block instead of your main thread.

Why is that helpful? It allows you to keep doing your main work, e.g. answering to HTTP requests. If not every requests needs a JDBC connection you can still answer those with your main thread while your thread pool is blocked. This is nice but still blocking I/O and you will run into bottlenecks as soon as your work becomes bound by the JDBC communication.

The field is very broad and there are many more details to explore. I believe however that with a basic understanding of blocking vs. non-blocking I/O you should be able to ask the right questions when you run into performance problems. Did you ever use native transports in your application? Did you do it because you could or because you were fighting with performance issues? Let me know in the comments!