

# What the heck is Asynchronous Non-blocking I/O !!!



Nayanava De · [Follow](#)

5 min read · Jul 31, 2021



294



3



## What to expect from this article

This article covers the basics of asynchronous non-blocking I/O and how it is beneficial for applications which need to cater to large number of concurrent requests in a cost effective manner.

## Back in the day

Sometime ago when the number of consumers of a service were relatively less, a system built using a language which supported a thread per request model was sufficient to meet the throughput requirements. But as the volume started to increase over the years and almost what exponentially!!! this model of request execution could only scale so much to the requirements in terms of the number of concurrent requests it could handle, beyond which the only way forward was to scale out horizontally by adding more hardware instances of the same service to be able to cope with the load which was an expensive solution.

## Thread per request Model

Before we move ahead with Asynchronous Non-blocking I/O lets dig a little deeper into what a **thread per request** or a **thread per connection** model is. A thread per request model is one in which a single thread is coupled to the lifecycle of a request. In such a scenario the thread goes into a blocked state as soon as the request starts to perform an I/O operation and is ready for execution once the I/O operation completes.

This approach has two drawbacks.

1. Since the majority of the time of a request is spent in performing an I/O operation, the thread spends most of the time in a blocked state during the request life-cycle, which reduces the thread efficiency because it cannot do anything else till the I/O operation completes.
2. There starts to be a trade-off between the number of threads that we would want to have in the thread-pool which can handle an appreciable number of concurrent requests and also not pin down the CPU because of the overhead added because of the context switches.

Both 1 and 2 indicate that the CPU remains under-utilised during the whole process because they introduce bottlenecks in the form of thread blocking which requires an application to have more number of threads initialised to handle higher number of requests which also introduces the overhead of context switches.

### **Behaviour of a blocked thread**

Just to give a feel, imagine a piece of code for the UI which blocks the thread on an I/O operation which takes a few seconds to complete. The entire UI would just freeze and become unresponsive because the thread has been blocked by design.

### **Async Non-Blocking I/O**

The efficiency of a thread is maximised when it spends majority of the time running on a CPU.

Asynchronous non-blocking I/O takes advantage of exactly the same fact. An async non-blocking I/O is implemented in such a way that the application thread is returned to the thread-pool instead of being blocked on the request, which lets the thread serve new requests waiting in the request queue and start processing them on the CPU. No single thread in the thread-pool is tied to a request. Once the I/O operation completes, one of the threads from the thread-pool can be handed the job of carrying the task further.

### **Flow of Execution of Async Non-Blocking I/O**

One interesting point to notice here is that no single thread is tied to the request life-cycle. In order to maintain all relevant information about the execution of the request, for example: the call stack till the point of an I/O operation, the context of execution of the request is captured and saved in the form of a state machine before it is handed over to a non-application thread for carrying out the I/O. Once the I/O operation completes and the task is assigned to a thread from the application thread-pool, the execution picks up from the point right after the network call.

### **Advantages of Async Non-Blocking I/O**

As we can see, this model of request execution decouples the application thread-pool threads from the I/O operations, providing us with two important advantages

1. Allowing for more efficient use of the request threads and an improved utilisation of the CPU, by maximising the thread CPU time of every thread.
2. Minimised number of threads required to serve higher number of concurrent requests, reducing the overall context switching overhead.

### **When to use Async Non-Blocking I/O?**

1. From a UI standpoint, we would want to provide the users with a seamless experience and hence async non-blocking I/O is an ideal choice. Javascript is an asynchronous programming language which supports non-blocking I/O under the hood.
2. From a backend service standpoint, if we have to build a high throughput system which is I/O intensive in nature, this would be an ideal choice as this model of request execution provides better CPU utilisation thus reducing the overall hardware requirements. C#, Node.js supports async non-blocking I/O out of the box.

### When can it not be a value add?

We will not be able to see much of a difference or in fact hardly any difference when our applications are CPU intensive, because in such cases the threads would spend most of the time on the CPU providing little to no benefit when it comes to using Async non-blocking I/O.

### Personal Experience with Async non-blocking I/O

While working on a very high throughput IO intensive service at Microsoft, we figured that it was **not able to handle around 260k requests per minute with 75 instances** of the service running. On looking at the codebase we managed to figure that the I/O calls were all blocking calls which caused a very high increase in the number of thread-pool threads spawned to serve the requests (a behaviour which causes a new thread to be spawned in the application thread-pool as soon as a the windows OS indicates that a thread has entered into a blocked state), ultimately pinning down the CPU over context switches.

Once the code was fixed by converting all such blocking calls to non-blocking ones, we started seeing almost **10x improvement in throughput**. On further tweaking of the application threads and dedicated OS threads we managed to finally achieve an **18x improvement in throughput by serving 250k requests per minute with only 5 instances** of the service!! And more so, the performance has been consistent while having to handle more than 400K rps of peak traffic.

Asynchronous

Backend

Scalability

Java

Software Development



Written by Nayanava De

90 Followers · 23 Following

Software Engineer II at Microsoft working on highly scalable distributed systems

Follow

Responses (3)



What are your thoughts?

Respond



Ashish Prasad  
Jul 31, 2021



Good article Nayanava. You can add references for Moore's and Amdahl's law for efficiently using multi cores as advance reads as part of next steps for readers.

4 [Reply](#)



Vijitha Za Gunta  
Aug 7, 2021



Really nice article Nayanava! And very thorough by even explaining when it won't be a value add.

2 [Reply](#)



Ahmad Usama  
Jan 15, 2022



Great ground-up clarifications of the concept, Nayanava.

Hey, which framework you used to implement this non-blocking io, this would be helpful in connecting the concepts end to end.

1 reply [Reply](#)

More from Nayanava De



Nayanava De

Controlling Degree Of Parallelism with Task.WhenAll() in C#

Background

Jul 21, 2021 22 2




Nayanava De

Relational Databases vs NoSQL

This discussion here will give an eagle's eye view of the difference between Relational...

Jul 5, 2020 68 3



 Nayanava De

## Weight training before or after cardio?? The explanation is quite...

Two things to know before we jump on to answering that question

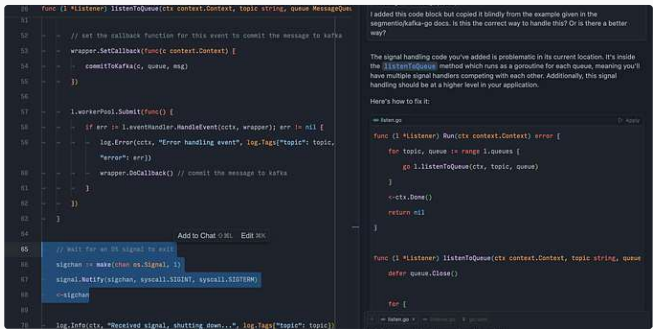
Nov 10, 2022


 3



See all from Nayanava De

## Recommended from Medium



 In Level Up Coding by Jacob Bennett

## The 5 paid subscriptions I actually use in 2025 as a Staff Software...

Tools I use that are cheaper than Netflix



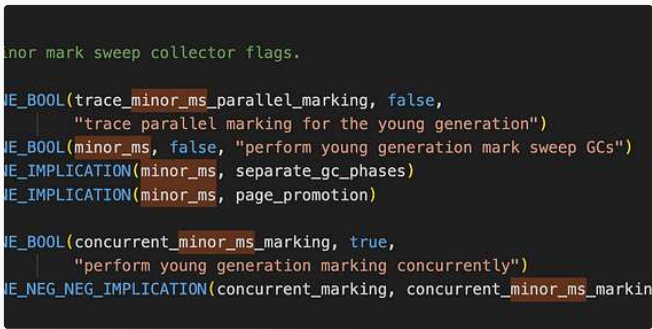
Jan 7



3.7K



91



 In JavaScript in Plain English by Peter K

## Uncovering Node.js Internals: Minor Garbage Collection...

In Node.js, memory management is a critical aspect of performance. The V8 engine, which...

Nov 5, 2024



10



### Lists



#### General Coding Knowledge

20 stories · 1874 saves



#### Coding & Development

11 stories · 975 saves



#### Stories to Help You Grow as a Software Developer

19 stories · 1560 saves




#### Good Product Thinking

13 stories · 797 saves





 In Towards Data Science by Subha Ganapathi

## Maximizing Python Code Efficiency: Strategies to Overcom...

Navigating Nested Loops and Memory Challenges for Seamless Performance using...

★ Jan 11, 2024 🖱️ 257 📌+ ⋮




 In Javarevisited by Dylan Smith

## Because I Didn't Know the Difference Between Exception an...

My articles are open to everyone; non-member readers can read the full article by...

★ Dec 9, 2024 🖱️ 898 💬 22 📌+ ⋮




 Ajay Gurav

## Zero to Hero: Mastering Rust for Data Science and Machine...

Rust, known for its speed and memory safety, is emerging as a valuable tool in the data...

★ Nov 28, 2024 📌+ ⋮



 In DevOps.dev by Gaddam.Naveen

## How the Garbage collection work in java? but why so many...

Java's garbage collection (GC) is an automatic memory management process. It helps...

★ 5d ago 🖱️ 69 📌+ ⋮

See more recommendations