**Multi-threading is always the wrong design**

“We’ll just do that on a background thread”

[[](https://unetworkingab.medium.com/?source=post_page-----a227be57f107--------------------------------)](https://unetworkingab.medium.com/?source=post_page-----a227be57f107--------------------------------)

[uNetworking AB](https://unetworkingab.medium.com/?source=post_page-----a227be57f107--------------------------------)

·

Follow

4 min read

·

Nov 29

73

8

Say what you want about Node.js. It sucks, a lot. But it was made with one very accurate observation: multithreading sucks even more.

A CPU with 4 cores doesn’t work like you are taught from entry level computer science. There is no “shared memory” with “random time access”. That’s a lie, it’s not how a CPU works. It’s not even how RAM works.

A CPU with 4 cores is going to have the capacity of executing 4 seconds of CPU-time per second. It does not matter how much “background idle threading” you do or don’t. The CPU doesn’t care. You always have 4 seconds of CPU-time per second. That’s an important concept to understand.

If you write a program in the design of Node.js — isolating a portion of the problem, pinning it to 1 thread on one CPU core, letting it access an isolated portion of RAM with no data sharing, then you have a design that is making as optimal use of CPU-time as possible. It is how you optimize for NUMA systems and CPU cache locality. Even a SMP system is going to perform better if treated as NUMA.

A CPU does not see RAM as some “shared random access memory”. Most of the time you aren’t even touching RAM at all. The CPU operates in an address space that is cached in SRAM in different layers of locality and size. As soon as you have multiple threads access the same memory, either you have cache coherence, threading bugs (which all companies have plenty of, even FAANG companies), or you need synchronization primitives that involve memory barriers that will cause shared cache lines to be sent back and forth as copies between the CPU cores, or caches to be committed to slow DRAM (the exact details depend on CPU).

In other words, isolating the problem at a high level, tackling it with single-threaded simple code is always going to be a lot faster than having a pool of threads bounce between cores, taking turn handling a shared pool of tasks. What I am saying is that designs like those in Golang, Scala and similar Actor designs are the least optimal for a modern CPU — even if the ones writing such code think of themselves as superior beings. Hint: they aren’t.

Not only is multithreading detrimental for CPU-time usage efficiency, it also brings tons of complexity very few developers (really) understand. In fact, multithreading is such a leaky abstraction that you really must study your exact model of CPU to really understand how it works. So exposing threads to some high level [in terms of abstraction] developer is opening up pandoras box for seriously complex and hard to trigger bugs. These bugs do not belong in abstract business logic. You aren’t supposed to write business logic that depend on the details of your exact CPU.

Coming back to the idea of 4 seconds of CPU-time per second. The irony is that, since you are splitting the problem in a way that requires synchronization between cores, you are actually introducing more work to be executed in the same CPU-time budget. So you are spending more time on overhead due to synchronization, which does the opposite of what you probably hoped for — it makes your code even slower, not faster. Even if you think you don’t need synchronization because you are “clearly” mutating a different part of DRAM — you can still have complex bugs due to false sharing where a cache line spans across the addressed memory of two (“clearly isolated”) threads.

And since you have threads with their own stack, things like zero-copy are practically impossible between threads since, well they stand at different depths in the stack with different registers. Zero-copy, zero-allocation flows are possible and very easy in single threaded isolated code, duplicated as many times there are CPU-cores. So if you have 4 CPU cores, you duplicate your entire single threaded code 4 times. This will utilize all CPU-time efficiently, given that the bigger problem can be reasonably cut into isolated parts (which is incredibly easy if you have a significant flow of users). And if you don’t have such a flow of users, well then you don’t care about the performance aspect either way.

I’ve seen this mistake done at every possible company you can imagine — from unknown domestic ones to global FAANG ones. It’s always a matter of pride and thinking that, we, we can manage. We are better. No. It always ends with a wall of text of threading issues once you enable ThreadSanitizer and it always leads to poor CPU-time usage, complex getter functions with return by dynamic copy, and it blows the complexity out of proportions.

The best design is the one where complexity is kept minimal, and where locality is kept maximum. That is where you get to write code that is easy to understand without having these bottomless holes of mindbogglingly complex CPU-dependent memory barrier behaviors. These designs are the easiest to deploy and write. You just make your load balancer cut the problem in isolated sections and spawn as many threads or processes of your entire single threaded program as needed.

Again, say what you want about Node.js, but it does have this thing right. Especially in comparison with legacy languages like C, Java, C++ where threading is “everything goes” and all kinds of projects do all kinds of crazy threading (and most of them are incredibly error prone). Rust is better here, but still causes the same overhead as discussed above. So while Rust is easier to get bug-free, it still becomes a bad solution.

I hear so often — “just throw it on a thread and forget about it”. That is simply the worst use of threading imaginable. You are adding complexity and overhead by making multiple CPU cores cause invalidation of their caches. This thinking often leads to having 30-something threads just do their own thing, sharing inputs and outputs via some shared object. It’s terrible in terms of usage of CPU-time and like playing with a loaded revolver.

Rant: over.