Cooperative vs. Preemptive: a quest to maximize concurrency power

Brief report on our research in trying to get the most out of our highly concurrent systems at scale.

Bobby Priambodo
Sep 3, 2019 · 16 min read

Cog wheels. [source]

Motivation

Imagine if Traveloka worked like a supermarket that had only one cashier, and we handled our customers one by one. There would be a very long queue, and even the 5th customer would probably leave as soon as she felt that it was taking too long. She would then go to another supermarket. We wouldn't want that.

A slightly better approach would be to have the cashier work on, say, five customers at a time, switching customers for every barcode scanned. If the cashier were as fast as Barry Allen, then all five customers would feel that they are being serviced individually with no interruption. In this case, we say the cashier works **concurrently**.

What if one cashier still served customers one by one, but now we had five cashiers? Of course, the service time would also speed up. For this one, we say that our cashiers work **in parallel**. We can also say that, from the customers' perspective, the cashiers work concurrently: there are more than one customer being serviced at one time.

Now, we understand that there is probably no human on Earth who is as fast as The Flash, let alone one conveniently being a cashier, but switching tasks in an ultra high speed is an easy feat for computers. With the advent of multicore processors, computers can even combine the two: having multiple cashiers switching customers. Imagine the velocity!

. . .

At Traveloka's scale, it is a necessity for our systems to handle requests from our users concurrently. Every user should be able to do what they need, such as to book a flight or hotel, discover exciting things to do for the weekend, pay their bills, without needing to wait for other users to finish what *they* need to do. Indeed, our services have been concurrent since day one.

My team in particular use Java heavily. To build our systems, we use battle-tested open source Java libraries for many things, such as web servers and database drivers, which have built-in concurrency capabilities through threads. In implementing our business specific logic, we also utilize Java's threading primitives to achieve concurrency, running Runnables through Executors for tasks that can be done independently.

It Works™.

But over time, we started to see **performance issues**. We had one service that hogged all of its host's RAM because the Java app had a total of 10 thousand threads running. With the default JVM thread stack size for a 64bit VMs which was 1024 bytes, it meant that we allocated around 10GB memory for thread stack alone, even when most of it perhaps were unused. The CPU utilization was also considerably high because of all the context switches.

One solution would be to scale up the host, but that's just throwing money to the problem. We finally mitigated the issue by auditing the thread pool usages and eventually decoupled the service to smaller ones anyway (because it was already too

big), but this particular problem made us think: **is there a better way for us to do concurrency more efficiently, without inflicting damage to our hardware costs?**

This rest of this article is the result of our (non-academic) research on this problem. We will talk about three important aspects of concurrency: task scheduling, threading model, and memory model. We'll also see how popular technologies (i.e. programming languages or libraries) approach concurrency in practice.

Now, let's talk about scheduling.

. . .

Cooperative vs. Preemptive scheduling

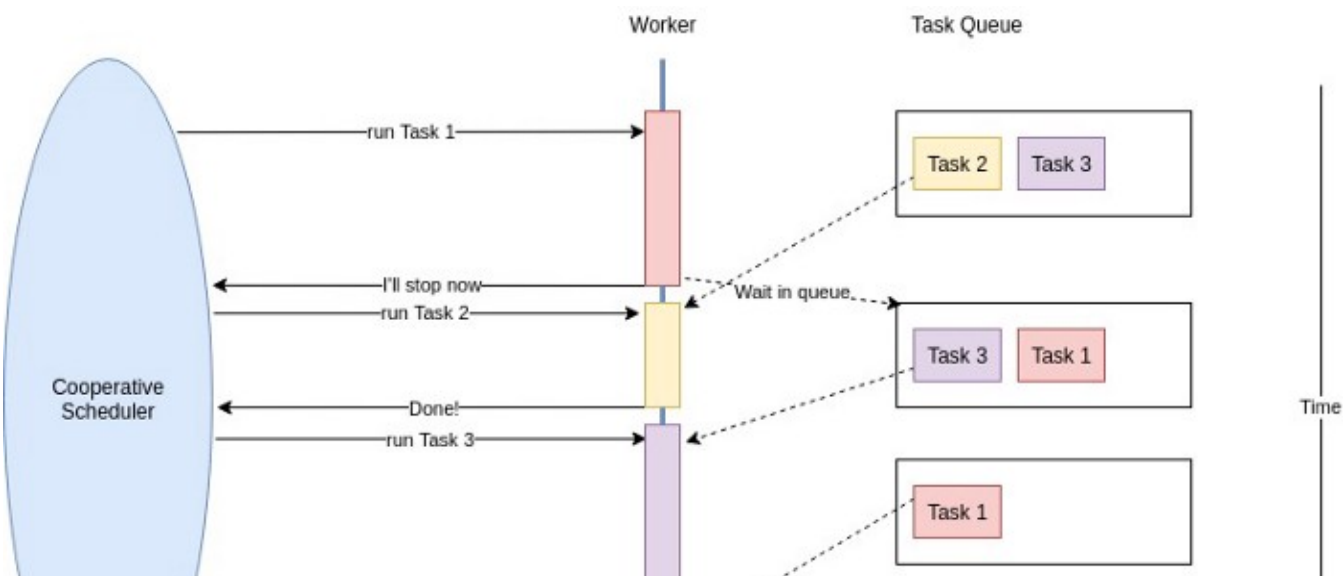
Scheduling is a mechanism to assign tasks to workers. The one who assign tasks is often called a **scheduler**. In an operating system (OS), usually *tasks* translates to *threads*, and *workers* translates to *CPU cores*.

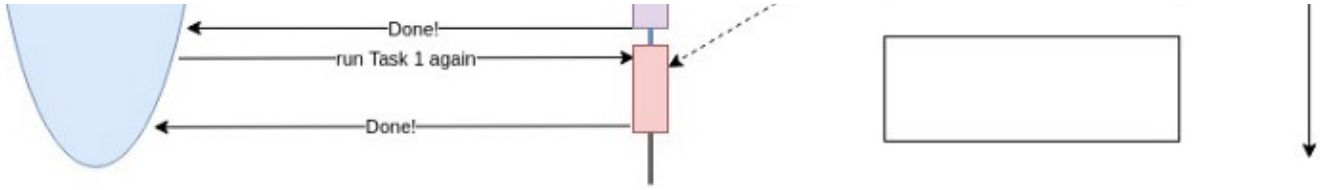
That said, I will be speaking in the general term of tasks and workers because this concept is not necessarily tied to operating systems only.

There are two styles of scheduling, **cooperative** and **preemptive**.

1. Cooperative

In a cooperative scheduling style, the tasks manage their own lifecycle. After being assigned to a worker, it is up to the task whether or not it should be released from a worker (i.e. yield control back to scheduler). The scheduler's job is only to assign tasks to any worker that is free.

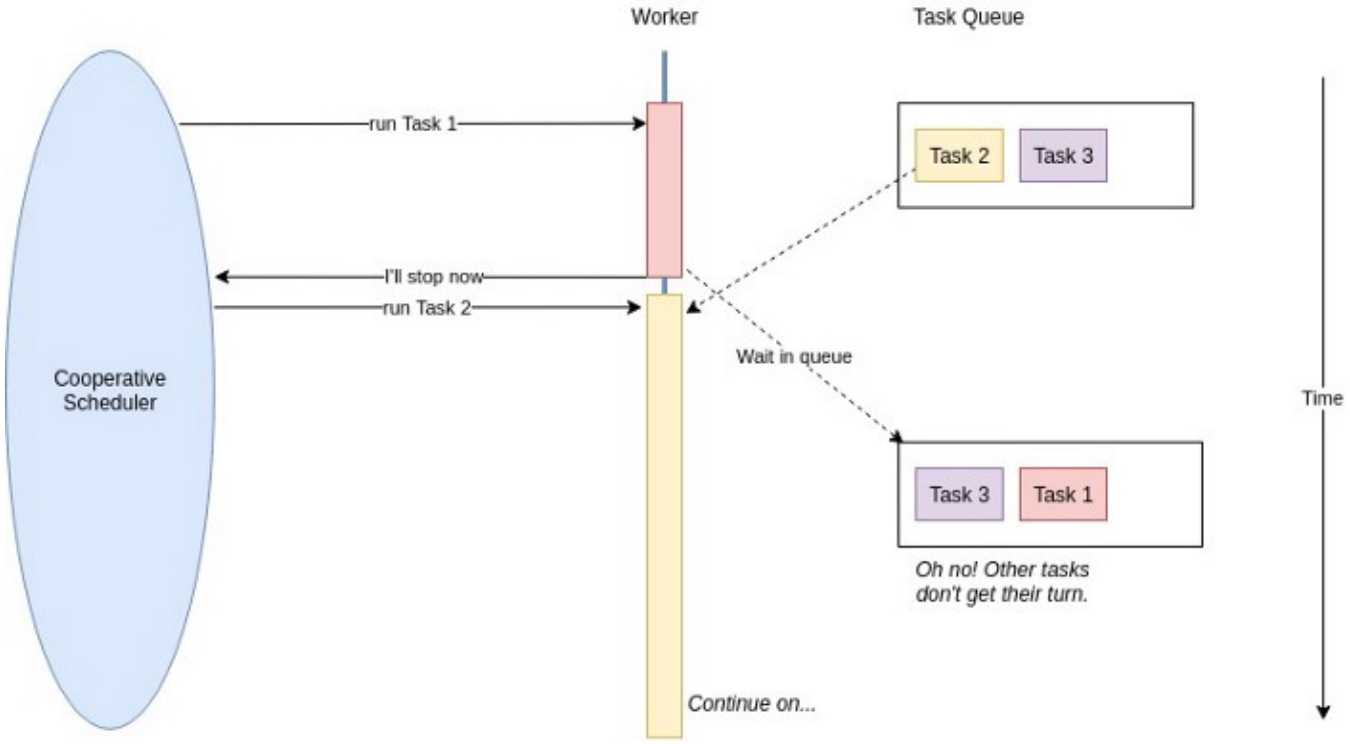




Cooperative scheduling illustration with 1 worker and 3 tasks.

If there is no free worker, then there's nothing the scheduler could do other than waiting for one. That is why it's called cooperative: the tasks should cooperate with each other to ensure everyone gets a fair share of workers.

You can probably see how this could cause problem: if a task is being evil and hogs the only worker available for a long time, all other tasks wouldn't get scheduled and the process can hang.



Cooperative scheduling issue: stray task (Task 2) can hog the worker.

Back in the day, Windows 3.1x and and MacOS 9 used a cooperative scheduler for its concurrency. Say a developer writes a program poorly and forgot to yield control, the entire OS could stop working.

Cooperative concurrency works well for tasks that are designed to work closely, such as in one isolated program or an OS for embedded systems. It does not work for multi-purpose OS which may let third party programs be installed on it.

But even in isolated programs, it still requires extra care from the engineer not to introduce stray functions that may block the CPU.

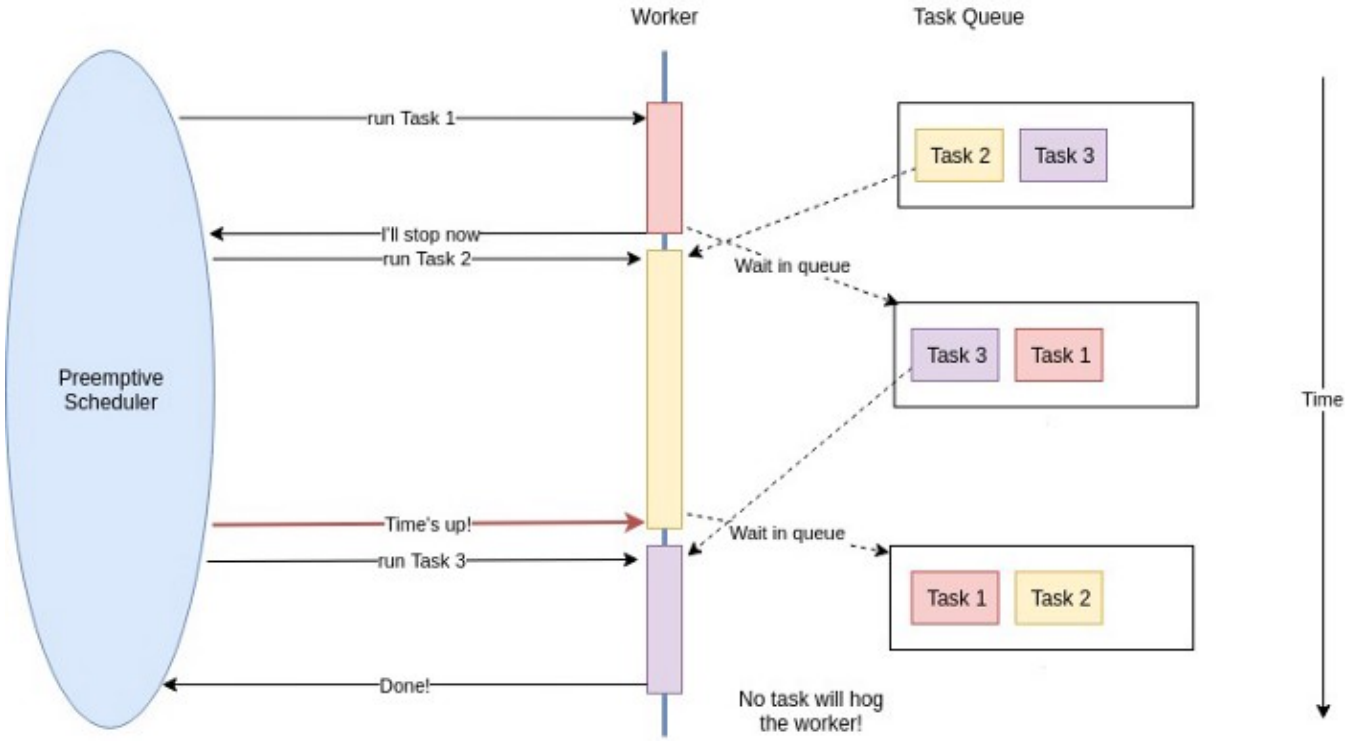
So then, why would one use cooperative scheduling? The reason is that **it is generally much cheaper than the alternative**. Before we get into that, let's talk about preemptive scheduling next.

2. Preemptive

In preemptive scheduling style, the scheduler has more control over tasks. Aside of assigning a task to a worker, the scheduler also assigns a *time slice* for the task.

A task gets removed from the worker when it yields (i.e. done or blocking for I/O) or when it has used up its time slice.

When the time is up, the scheduler interrupts (preempts) the task and let another task run in its place, and the original task waits for its turn again.



Preemptive scheduling illustration with 1 worker and 3 tasks.

With this approach, the scheduler can maintain a degree of fairness to all tasks. One stray task couldn't hog the worker, and prioritized tasks could be assigned more time in workers than others with less priority.

Most modern multi-purpose operating systems right now have a preemptive scheduler for scheduling program threads.

For the particular case of OS, preemptive works better since even when one program hangs, other programs can still work normally since they have their own time slices. That is something a cooperative scheduler could not guarantee.

However, preemptive scheduling is not without drawbacks. Since tasks can get interrupted midway, there is a necessity to store and restore the state that the interrupted task had. When a task run again after being interrupted, it should continue where it left off, not starting from scratch again.

This process of storing and restoring the state of tasks is called a **context switch**. In operating systems, a context switch is computationally expensive. Switching from one thread to another involves a degree of administration, such as switching registers, switching stack pointer, updating various tables, and others. This takes time. We usually call this the context switch overhead.

This is where cooperative scheduling wins over preemptive; in cooperative scheduling, as the tasks maintain their own lifecycle, the scheduler doesn't have to take note of each task's state, hence switching tasks is much cheaper and faster.

This duality of fairness guarantee and overhead cost is the tradeoff one would have to decide when choosing a scheduling mechanism for their use case.

. . .

Next up, we'll talk about threading models.

Threading model (kernel-level vs. user-level)

From the perspective of an OS, there are two types of threads.

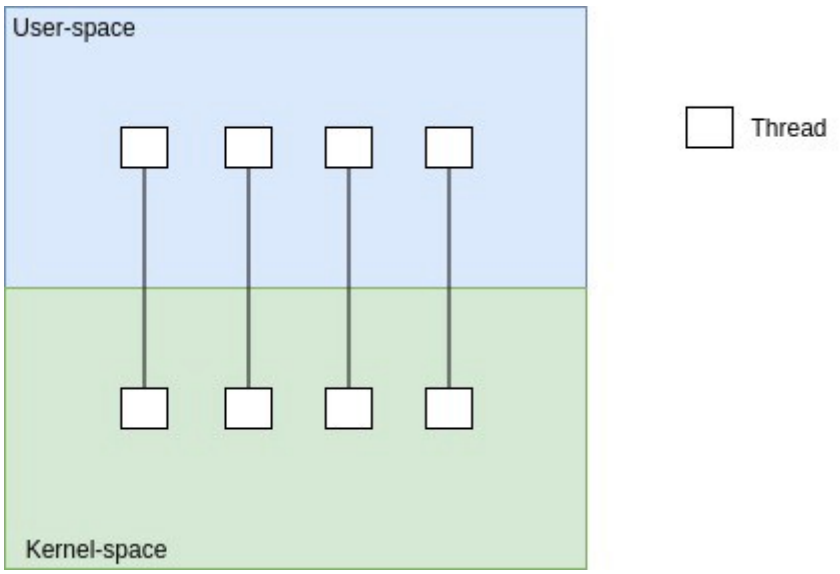
Kernel-level threads are threads managed by the OS itself. The OS performs the creation, scheduling, and bookkeeping of the threads.

User-level threads, in contrast, are threads that are managed by programs in the user space. User-level threads are invisible to the OS, and doesn't need OS threading support. They are also often called *green threads,lightweight threads*, or *coroutines*.

Based on these two types, we can categorize three threading models that are used by most programs: kernel-level threading, user-level threading, and hybrid threading.

1. Kernel-level threading (1:1 model)

In this threading model, one program thread translates to one OS thread. This means that whenever we *spawn* a thread in the program, it will invoke a system call to create an OS thread.

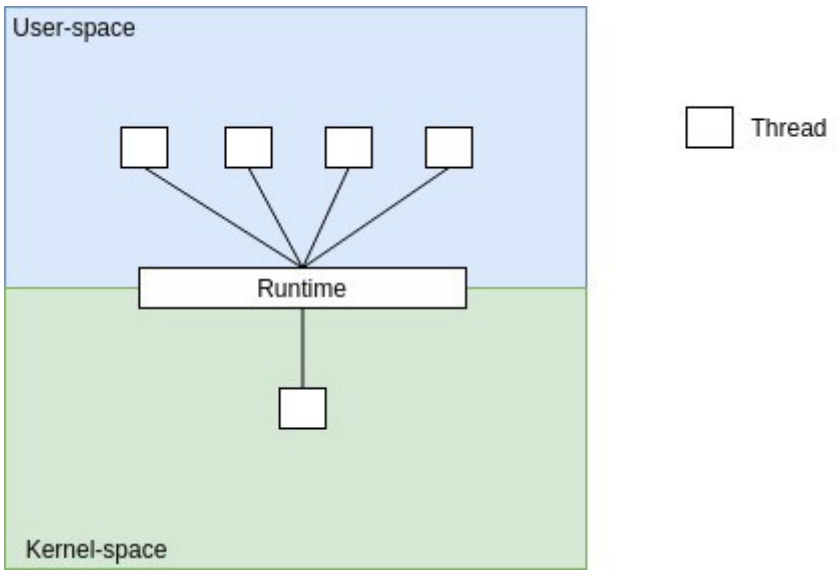


Kernel-level threading model Illustration.

As such, this threading model will have the same scheduler (often preemptive) as the OS, as well as its drawbacks such as the context switch overhead.

2. User-level threading (N:1 model)

The user-level threading model uses multiple user-level threads that are mapped to one OS thread. Programs that use this threading model usually need to include a runtime that has its own scheduler to manage the execution of threads. This scheduler could be either preemptive or cooperative.

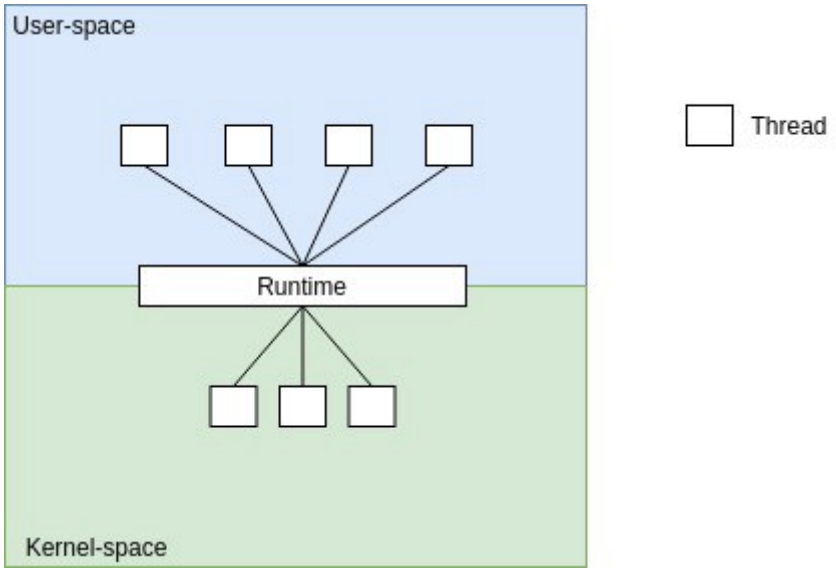


User-level threading model Illustration.

The drawback of this threading model is that, since the program essentially runs in a single OS thread, it can't really leverage multicore processors to achieve parallelism, or at least not as straightforward as kernel-level threading.

3. Hybrid threading (M:N model)

This threading model combines the best of both kernel-level and user-level threading model. It makes use of kernel-level threads for parallelism, but also maintain user-level threads that are used by the program to run tasks concurrently.



Hybrid threading model Illustration.

I understand that these threading models might be a bit abstract right now, but we will try to make them more concrete towards the end of this post.

Before that, I need to tackle one more important aspect of concurrency: the memory model.

. . .

Memory model

Shamelessly copy-pasting from Wikipedia:

> *a **memory model** describes the interactions of threads through memory and their shared use of the data.*

In a multithreaded program, we have a possibly large number of threads running in one process. These threads share the same *address space*; this means that the threads

have the same visibility of the memory, e.g. data structures and variables.

In preemptive scheduling, threads can interleave at any time. Those threads might also be accessing and mutating the same variable at the same time, and these kind of scenarios might introduce concurrency problems such as race conditions.

A program that potentially demonstrates the issue is said to be not **thread-safe**. To prevent such issues and maintain thread safety, programs are required to define a memory model.

A memory model describes how shared-memory concurrency works, and usually involves some kind of synchronization primitives, such as locks and mutexes, that can ensure the consistency and atomicity of all the data in memory even when concurrently accessed.

That said, synchronizations are not without problems too. A synchronized block will ensure that only one thread can run it at one time, and you can see how it can be a performance bottleneck in a multithreaded program with many threads.

There are also numerous well-known classic synchronization problems such as the dining philosophers problem, readers-writers problem, producer-consumer problem, and others. I won't go into detail, but I encourage curious readers to go read up on them later.

Coming up with a robust memory model is not easy, and that's why some programs (i.e. programming language runtimes) skip them altogether. Using a cooperative scheduler mitigates this problem somewhat, because the threads stop at a known point in time and hence will have no unwanted access, but if the scheduler is run on top of preemptive scheduler (e.g. in hybrid threading model) the issue will still come up.

. . .

Concurrency approach in practice

Now that we have all the theoretical foundations laid down, let's see how popular technologies approach concurrency in practice. Specifically, we'll see how they combine the the various kinds of scheduling, threading model, and memory model to provide concurrency.

1. Programs with kernel-level threading model, with preemptive scheduler

Example: Java (JVM), C, Rust

This family of programs directly utilize the OS's threading capability (1:1). As such, the scheduler used for these programs are preemptive; that is why it is okay (in the scheduling sense) for you to write or use blocking code, because the scheduler will ensure that it will not hog the CPU.

Java (through the JVM threading capability) and C (with pthreads) have synchronization primitives for ensuring the consistent memory model.

Rust is an interesting player in this field: thanks to its concept of value ownership which guarantees memory safety without GC, they don't have to deal with many common problems related to unrestricted concurrent memory access. Any potential concurrency issue can then be detected at compile-time. They dubbed this feature Fearless Concurrency (which I think is pretty rad!).

The drawbacks of preemptive and kernel-level threading also apply to these programs, such as the context switch overhead and the memory allocation. That's where we got our performance problem in the first place.

2. Programs with kernel-level threading model, but with GIL

Example: Python, Ruby, OCaml

These programs can utilize kernel-level threading during their lifetime. However, those threads are constrained by a GIL. GIL stands for *Global Interpreter Lock*. With GIL, there can only be one thread executing at a time within a process. This happens to languages such as Python, Ruby, and OCaml.

Why GIL? The use of GIL greatly simplifies the concurrency implementation: since it is guaranteed that only one thread can run at one time, there's no need to define a memory model, because no two threads can access the same part of memory at the same time. You can think of it as an invisible synchronization primitive.

The disadvantage of GIL-powered concurrency is that even though you use OS threads with its preemptive scheduler, you can't really have parallelism (except for IO tasks), because eventually the current running thread will only be executed in one CPU core.

With this kind of programs, we can usually achieve the need of parallelism through running multiple processes. However, processes are heavy and communicating between processes (IPC) are more expensive than threads sharing memory, so people

usually rely on user-level libraries to improve concurrency (which we will cover in the next section).

Note: Ruby has "vague plans" to remove GIL, Python actually has solved the GIL problem if you used Jython instead of CPython (and PyPy has plans for it) , while OCaml's ongoing multicore project is progressing rapidly. It's interesting how these languages will turn out in the future.

3. Programs with user-level threading model, cooperative scheduling

Example: Node.js, Twisted, EventMachine, Lwt

Programs that use this approach have their own runtime to manage the lightweight threads. As we have discussed, implementing cooperative scheduling is easier than preemptive scheduling because we don't need to define a memory model.

One of the key points of this family of programs is that they usually have an event loop. The event loop is essentially the scheduler for the lightweight threads.

Node.js is one of the biggest players in this field. The other three I mentioned in the example above are actually user-level libraries for GIL-powered languages, e.g. Twisted for Python, EventMachine for Ruby, and Lwt for OCaml.

Programs with this approach usually boasts about scalability even when being single-threaded (in the OS sense), and it's not without reason: the nature of lightweight user-level threads allows maximum throughput without the drawbacks of preemptive scheduler.

Since it's cheap to spawn threads, it's not uncommon to have hundreds of thousands of threads living in the application. If you try to have hundreds of thousands of kernel-level threads, you'd need a "good" enough specs for your machine (and it will cost a LOT of money).

However, it's also the case that it works best if your operations are commonly non-blocking. Performing heavy computations with these approach is not recommended because they could hog the event loop (which runs on a single thread), ultimately breaking concurrency.

4. Best of both worlds: Hybrid threading model, preemptive + cooperative scheduling

Example: RxJava, Kotlin coroutines, Akka, uThreads, Go (goroutines)

This approach combines both preemptive OS threads with cooperative user-level threads. The advantage should be apparent: we can achieve the high throughput and efficiency of the user-level threads, while still maintaining the parallelism from the kernel-level threads.

Often, the use of the kernel-level threads are invisible from the user, in that the user-level runtime that manages the lightweight threads also manages assigning them to OS threads under the hood.

RxJava, Kotlin coroutines, and Akka are all user-level libraries on top of the JVM, hence they can assign those threads to JVM threads. The uThreads library is one example implementation of cooperative user-level thread for C and C++.

Go natively has goroutines, and is an interesting one in this category. It is very similar with Kotlin's coroutines, but in Kotlin you would need to be very careful in writing code, because you have to explicitly be cooperative by creating coroutines as well as using yields and suspending functions.

In Go, it's just a matter of invoking any normal function with the `go` keyword.

Goroutines don't need to explicitly yield control, because the scheduler will yield control for every function invocation. This makes cooperative concurrency really easy because we will less likely forget to yield and accidentally hogs the scheduler.

However, if we run a tight long-running loop with no function calls, we can still block the underlying thread. There was a proposal of adding a scheduler calls in loops, but it might cause slow downs in tight loops.

(There's also an ongoing design proposal of making the scheduler preemptive instead of cooperative.)

In general, this approach still has the drawbacks of concurrency data-race and synchronization issues because of the use of OS threads. With Kotlin, Akka, and Go we can avoid this by using *actors* or *channels* to communicate between threads with message passing instead of sharing memory, however it is still possible to accidentally write non-thread-safe code.

Blocking code will also still block the underlying OS thread that the lightweight thread was assigned to.

But even with those disadvantages, the proposed advantages of this solution should be adequate most of the time, as it is a major efficiency improvement from the standard kernel-level threading or user-level threading.

5. Bonus: Hybrid threading model, preemptive + preemptive

Example: Erlang, Elixir, Haskell

I initially thought that it was not possible (or at least feasible) to implement a preemptive scheduler for user-level threading, but it turned out that such thing does exist!

From my research, I came to know that Haskell, Erlang, and Elixir (which uses Erlang runtime) use this very approach for their concurrency.

I said before that implementing a preemptive scheduler is hard because we need to take into account context switches and synchronization primitives, so how could Haskell and Erlang do it?

Haskell
Haskell benefits from the fact that it's a **pure functional language**. That means side effects such as IO and mutating variables or global states are strictly controlled. It by default enforces immutable values. As such, there's no need for a complicated memory model.

From the documentation, Haskell (actually, GHC, its most popular compiler implementation) claims that it uses a preemptive scheduler. I would argue that it's actually a cooperative scheduler which yields every time there's a memory allocation. Quoting the docs:

> *More specifically, a thread may be pre-empted whenever it allocates some memory, which unfortunately means that tight loops which do no allocation tend to lock out other threads.*

But since memory allocation usually happens all the time in normal programs (unless you've done low-level optimizations), I think it can pass just fine as a preemptive scheduler.

Erlang/Elixir
Erlang's approach, on the other hand, is arguably more superb. Erlang threads are called "processes", as they mimic an OS process in which multiple processes don't share

memory. Like Akka (in fact, Erlang is the inspiration of Akka), **Erlang processes communicate through passing messages between them**. This is all built-in in the language runtime.

This is also true for Elixir, which is a relatively new language that runs on top of Erlang VM (think Scala or Kotlin on JVM).

Because processes are the building blocks of Erlang, **there is simply no way for processes to share memory**, and hence there is also no need for a complicated memory model. Erlang can then easily use a preemptive scheduling mechanism to schedule the processes. By running one scheduler per CPU cores, the Erlang VM can then maintain parallelism.

Because Erlang VM uses preemptive scheduler, it's perfectly okay to write blocking code, because the VM will happily share the available resources by interrupting long running process. This results in a very efficient concurrent program that is straightforward to write!

. . .

So, it seems that Haskell and Erlang/Elixir are the epitome of the best concurrency approach, doesn't it? Why do I put it as a bonus?

The reason is that because they both have significantly different programming paradigms from other languages. They are both functional instead of imperative. They both utilize immutable values very heavily and don't rely on mutating objects or variables.

There is a huge learning curve for programmers coming from popular languages such as Java, Python, and C to learn these languages.

If programmers of the former languages look for concurrency-first language, it is very likely that they will try Go before anything else. It provides the similar almost-preemptive cooperative concurrency, but with the familiar imperative paradigm like the others.

. . .

So, what's next?

How did this research benefit my team?

Since my team works heavily with Java, we're currently adopting Kotlin and coroutines in our code. The reason we picked Kotlin was that, after considering the pros and cons, we thought Kotlin coroutines would provide the best value for effort, because it has the hybrid preemptive + cooperative concurrency approach while having 100% interop with our Java codebase and in-house libraries.

That said, personally I would love to try a PoC of either Go, Haskell, or Erlang in one of our services, to see how they compare with our current solution.

I'm also keen to try Rust for things that require more raw machine power but with better memory safety guarantee than C or C++. But that's for another story :)

. . .

If you're reading this far, congrats! You have the curiosity and eagerness to learn expected of Traveloka engineers.

I acquired all this knowledge as I'm dealing with real-life engineering problems with my team while working at Traveloka, one of the largest online travel companies in Southeast Asia. If you're a software engineer interested in tackling real scalability and reliability issues in large distributed systems which allow users to create moments with their loved ones, have a look at the opportunities on Traveloka's careers page!

Suggestions and corrections for this article are welcome.

Programming | Concurrency | Software Engineering | Threads | Engineering