On the Performance of User-Mode Threads and Coroutines

Ron Pressler on August 7, 2020

Discussions of coroutines and user-mode threads — like Project Loom's virtual threads or Go's goroutines — frequently turn to the subject of performance. The question I'll try answering here is, how do user-mode threads offer better application performance than OS threads?

One common assumption is that this has to do with task-switching costs, and that the performance benefit of user-mode threads and coroutines — the two are similar enough in the aspects relevant for this discussion, so I will treat them interchangeably — is due to their low task-switching overheads compared to OS threads. As a result, some further ask how much task-switching overhead, if any, user-mode threads add over asynchronous programming (using callbacks or some asynchronous composition, like that offered by `CompletableFuture` or `Flow`/reactive streams), and whether the task-switching costs harm or improve performance compared to asynchronous programming. But as we'll see, in their most common, most useful use-case, the performance benefit of user-mode threads (or coroutines) has little to do with task-switching costs. Their power comes from elsewhere.

Performance is a complex subject, so first — a few definitions. *Throughput* is a performance measure defined as number of operations accomplished per time-unit; *latency*, another performance measure, is the duration some operation takes to complete. For example, sending data from London to New York by putting it on flash drives and shipping them on a 747 will suffer from bad (high) latency but can have a pretty good (high) throughput.

Another important concept is what I will call *impact*. It is the contribution of an operation to the overall performance, and is a measure of how important it is to optimize the operation. If some operation comprises 1% of the total time spent in the application, making that work infinitely faster would only improve the application's total performance by 1%, so this operation would have a low impact. Impact is highly dependent on the application, but we will talk about common types of applications and classify impact by use-case.

Consider one use-case for coroutines: generators — a convenient way for writing iterators. Suppose a generator emits a sequence of incrementing integers to a consumer that sums them; task switching occurs between the generator and the consumer with each number produced. Crucially, this scenario involves nothing but pure computation, and very short-lived at that. We'll call this the "pure computation" use-case. The throughput is the number of integers summed per second. In this scenario, the task-switching operation has a very high impact. If we take the processing time (of incrementing the numbers and summing them) to be zero, the impact of the task-switching overhead is 100%.

Now consider another use-case that we'll call "transaction processing": A server waits for and responds to requests arriving over the network. When a request arrives, the server processes it by doing some computation as well as contacting other auxiliary services, perhaps a database, over the network; it sends requests to those other services, collects their responses, and finally replies to the client with some aggregate result. This system's throughput is the number of incoming requests the server processes per second.

Our server is said to be *stable* if requests don't pile up indefinitely, and so the response rate is equal to the request rate. To analyze its throughput we turn to Little's law, which says that in a stable system the average *level of concurrency*, L — the number of requests concurrently processed by the server — is equal to the average rate of requests, λ, times the average duration of processing each request, W:

$$L = \lambda W$$

Because the system is stable, its throughput is equal to λ and the maximal achievable λ is the system's capacity. The law is simple, but is, in fact, remarkable because the result does not depend on the distribution of the requests' arrival.

To simplify matters, let's assume, as before, that all computation takes zero time and only consider the time it takes to contact the auxiliary services over the wire, as we expect that cost to dominate the latency, W. There are two nuances to dispense with: First, if our server employs multiple cores that can do processing in parallel, we can consider each of them a separate server; so without loss of generality, we assume our server is single-core (true, the cores share a network interface, but we'll ignore this complication). Second, if, in the course of processing the request, we must employ, say, three services over the network, each responding in 1ms, we could reduce the total latency of collecting their responses from 3ms to 1ms by performing them in parallel. This reduces W by a factor of three, but the interaction with the remote services also adds three more sub-operations in-flight, increasing our level of concurrency, L, also by a factor of three, and the two cancel out. Therefore, to employ Little's law we should consider W to be the sum total of the durations of all outgoing requests. And so, our throughput is:

$$\lambda = L/W$$

We are now ready to calculate the impact of task-switching. W is the sum of latencies of our service requests, and because we're talking of averages, it is the average number of service calls per transaction, which we'll call n, times the average service call latency, w, so $W = nw$.

To process multiple requests simultaneously on a single core, when we wait for a response from a service, we switch to another transaction, so every outgoing call is accompanied by one task-switch. If the average task-switching latency is t and the average wait-time for a service response is μ, then $w = \mu + t$ and $W = n(\mu + t)$. The impact of task-switching, t, on our throughput, λ — how much capacity we'd gain if task-switching cost absolutely nothing — is, then,

$$(L/n(\mu + 0)) / (L/n(\mu + t)) = n(\mu + t)/n\mu = (\mu + t)/\mu = 1 + t/\mu$$

If we choose to process each transaction on a single OS thread that blocks every time we wait for a service, task-switching through the kernel — a slow operation that takes, say t = 1us — and even if our services and network are very fast and give us an average service call latency, μ, of 20us, the impact of task-switching is 1/20. The best we can hope for by optimizing task-switching is to increase our capacity by 5%! More generally, the impact of task-switching is *its average latency divided by the average wait time*. If waiting for network IO is involved, that ratio can be quite low even if task-switching is relatively inefficient.

Clearly, to significantly increase the capacity of such a system we shouldn't focus on lowering t, which would only modestly reduce the latency W and increase the throughput λ, but instead focus on increasing L, the number of transactions we can process concurrently. If we keep the simple thread-per-request programming model and process each transaction on a single thread, L would be the number of active threads we need. And *that* is how user-mode threads help: they increase L by orders of magnitude with potentially millions of user-mode threads instead of the meager thousands the OS can support (but don't expect a 1000x increase in capacity; we've neglected computation costs and are bound to hit bottlenecks in the auxiliary services). Asynchronous programming also improves performance in the same way: not by reducing task-switching costs, but by increasing L, the number of transactions concurrently processed, only it doesn't represent each transaction with a thread but with a different construct.

Nevertheless, task-switching overhead can still matter. If N is the number of task switches per transaction, then $W = n\mu + Nt$, and the impact of task-switching on throughput is really:

$$Nt / n\mu$$

We assumed one task-switch per service call, so had $N = n$, but when we have so many threads at our disposal, it is convenient to employ several, communicating with each other by passing messages, to process the transaction. This can increase the number of task-switches per transaction well beyond one per service call, so keeping the cost of task switching low is still a good idea. It's important, just not as much as we'd naively think.

We believe that transaction processing is a more important use case, certainly in Java, for user-mode threads or coroutines than the pure-computation one, and so while Loom strives to keep the task-switching costs for virtual threads low, that is not its primary contribution to the performance of the applications we have in mind, and not is main source of its power. If ever a conflict arises between the convenience of targeting the transaction processing use-case and task-switching overhead, we prioritize the former, for reasons that I hope are now clear.

Lastly, a few words about implementation: very little can be extrapolated from the qualities of an implementation of coroutines or user-mode threads in one language to another. Some languages allow pointers to local variables while other do not. In some languages allocation is cheap and can be hidden while in others it might be costly and/or require explicit management. And, as we’ve seen, the target use case can be different. A design that, say, optimizes for cases where all coroutines participating in some job may fit in the CPU cache (like a generator) might not be best for a use case where a great many tasks are involved and task switching invariably incurs a cache miss. Judging the merits of an implementation requires evaluating it in the context of the use-cases it seeks to optimize as well as the idiosyncratic constraints and strengths of the language/runtime it targets. But that is the subject of another discussion.