12th January

How Erlang does scheduling

In this, I describe why Erlang is different from most other language runtimes. I also describe why it often forgoes throughput for lower latency.

TL;DR - Erlang is different from most other language runtimes in that it targets different values. This describes why it often seem to perform worse if you have few processes, but well if you have many.

From time to time the question of Erlang scheduling gets asked by different people. While this is an abridged version of the real thing, it can act as a way to describe how Erlang operates its processes. Do note that I am taking Erlang R15 as the base point here. If you are a reader from the future, things might have changed quite a lot—though it is usually fair to assume things only got better, in Erlang and other systems.

Toward the operating system, Erlang usually has a thread per core you have in the machine. Each of these threads runs what is known as a scheduler. This is to make sure all cores of the machine can potentially do work for the Erlang system. The cores may be bound to schedulers, through the +sbt flag, which means the schedulers will not "jump around" between cores. It only works on modern operating systems, so OSX can't do it, naturally. It means that the Erlang system knows about processor layout and associated affinities which is important due to caches, migration times and so on. Often the +sbt flag can speed up your system. And at times by quite a lot.

The +A flag defines a number of async threads for the async thread pool. This pool can be used by drivers to block an operation, such that the schedulers can still do useful work while one of the pool-threads are blocked. Most notably the thread pool is used by the file driver to speed up file I/O - but not network I/O.

While the above describes a rough layout towards the OS kernel, we still need to address the concept of an Erlang (userland) process. When you call spawn(fun worker/0) a new process is constructed, by allocating its process control block in userland. This usually amounts to some 600+ bytes and it varies from 32 to 64 bit architectures. Runnable processes are placed in the run-queue of a scheduler and will thus be run later when they get a time-slice.

Before diving into a single scheduler, I want to describe a little bit about how migration works. Every once in a while, processes are migrated between schedulers according to a quite intricate process. The aim of the heuristic is to balance load over multiple schedulers so all cores get utilized fully. But the algorithm also considers if there is enough work to warrant starting up new schedulers. If not, it is better to keep the scheduler turned off as this means the thread has nothing to do. And in turn this means the core can enter power save mode and get turned off. Yes, Erlang conserves power if possible. Schedulers can also work-steal if they are out of work. For the details of this, see [1].

IMPORTANT: In R15, schedulers are started and stopped in a "lagged" fashion. What this means is that Erlang/OTP recognizes that starting a scheduler or stopping one is rather expensive so it only does this if really needed. Suppose there is no work for a scheduler. Rather than immediately taking it to sleep, it will spin for a little while in the hope that work arrives soon. If work arrives, it can be handled immediately with low latency. On the other hand, this means you cannot use tools like top(1) or the OS kernel to measure how efficient your system is executing. You must use the internal calls in the Erlang system. Many people were incorrectly assuming that R15 was worse than R14 for exactly this reason.

Each scheduler runs two types of jobs: process jobs and port jobs. These are run with priorities like in an operating system kernel and is subject to the same worries and heuristics. You can flag processes to be high-priority, low-priority and so on. A process job executes a process for a little while. A port job considers ports. To the uninformed, a "port" in Erlang is a mechanism for communicating with the outside world. Files, network sockets, pipes to other programs are all ports. Programmers can add "port drivers" to the Erlang system in order to support new types of ports, but that does require writing C code. One scheduler will also run polling on network sockets to read in new data from those.

Both processes and ports have a "reduction budget" of 2000 reductions. Any operation in the system costs reductions. This includes function calls in loops, calling built-in-functions (BIFs), garbage collecting heaps of that process[n1], storing/reading from ETS, sending messages (The size of the recipients mailbox counts, large mailboxes are more expensive to send to). This is quite pervasive, by the way. The Erlang regular expression library has been modified and instrumented even if it is written in C code. So when you have a long-running regular expression, you will be counted against it and preempted several times while it runs. Ports as well! Doing I/O on a port costs reductions, sending distributed messages has a cost, and so on. Much time has been spent to ensure that any kind of progress in the system has a reduction cost[n2].

In effect, this is what makes me say that Erlang is one of a few languages that actually does preemptive multitasking and gets soft-realtime right. Also it values low latency over raw throughput, which is not common in programming language runtimes.

To be precise, preemption[2] means that the scheduler can force a task off execution. Everything based on cooperation cannot do this: Python twisted, Node.js, LWT (Ocaml) and so on. But more interestingly, neither Go (golang.org) nor Haskell (GHC) is fully preemptive. Go only switches context on communication, so a tight loop can hog a core. GHC switches upon memory allocation (which admittedly is a very common occurrence in Haskell programs). The problem in these systems are that hogging a core for a while—one might imagine doing an array-operation in both languages—will affect the latency of the system.

This leads to soft-realtime[3] which means that the system will degrade if we fail to meet a timing deadline. Say we have 100 processes on our run-queue. The first one is doing an array-operation which takes 50ms. Now, in Go or Haskell/GHC[n3] this means that tasks 2-100 will take at least 50ms. In Erlang, on the other hand, task 1 would get 2000 reductions, which is sub 1ms. Then it would be put in the back of the queue and tasks 2-100 would be allowed to run. Naturally this means that all tasks are given a fair share.

Erlang is meticously built around ensuring low-latency soft-realtime properties. The reduction count of 2000 is quite low and forces many small context switches. It is quite expensive to break up long-running BIFs so they can be preempted mid-computation. But this also ensures an Erlang system tend to degrade in a graceful manner when loaded with more work. It also means that for a company like Ericsson, where low latency matters, there is no other alternative out there. You can't magically take another throughput-oriented language and obtain low latency. You will have to work for it. And if low latency matters to you, then frankly not picking Erlang is in many cases an odd choice.

[1] "Characterizing the Scalability of Erlang VM on Many-core Processors" http://kth.diva-portal.org/smash/record.jsf?searchId=2&pid=diva2:392243 [http://kth.diva-portal.org/smash/record.jsf?searchId=2&pid=diva2:392243]
[2] http://en.wikipedia.org/wiki/Preemption_(computing) [http://en.wikipedia.org/wiki/Preemption_(computing)]

[3] http://en.wikipedia.org/wiki/Real-time_computing [http://en.wikipedia.org/wiki/Real-time_computing]

[n1] Process heaps are per-process so one process can't affect the GC time of other processes too much.
[n2] This section is also why one must beware of long-running NIFs. They do not per default preempt, nor do they bump the reduction counter. So they can introduce latency in your system.
[n3] Imagine a single core here, multicore sort of "absorbs" this problem up to core-count, but the problem still persists.

(Smaller edits made to the document at Mon 14th Jan 2013)

Posted 12th January by Jesper Louis Andersen