WORKING *with* RUBY THREADS

Jesse Storimer

This book is dedicated to Sara, Inara, and Ora, who make it all worthwhile.

Chapter 4

Concurrent != Parallel

I hinted in previous chapters that threads provide a concurrency mechanism, one capable of utilizing multi-core systems.

A sensible question following this might be:

> *So multiple threads will be running my code in parallel, right?*

Before I can answer that question, I need to clear up a common misunderstanding: **concurrent and parallel are not the same thing.**

Keeping that in mind, I can rephrase your question into two more sensible questions:

1. Do multiple threads run your code concurrently? Yes.
2. Do multiple threads run your code in parallel? Maybe.

Now I'll attempt to explain the difference. Since we're all programmers here, I'll use a programmer at work as an example.

An illustrative example

Imagine you're a programmer working for an agency. They have two projects that need to be completed. Both will require one full day of programmer time. There are (at least) three ways that this can be accomplished.

1. You could complete Project A today, then complete Project B tomorrow.
2. You could work on Project A for a few hours this morning, then switch to Project B for a few hours this afternoon, and then do the same thing tomorrow. Both projects will be finished at the end of the second day.
3. Your agency could hire another programmer. He could work on Project B and you could work on Project A. Both projects will be finished at the end of the first day.

This example is a bit contrived. It doesn't take into account the time required to switch projects, ramp-up time, inevitable delays, etc. But let's pretend things work this way just for the sake of example.

What do these three ways of working represent?

The first way represents working serially. This is the normal path of single-threaded code. Given two tasks, they will be performed in order, one after another. Very organized and easy to follow.

The second way represents working concurrently. This represents the path of multi-threaded code running on a single CPU core. Given two tasks, they will each be

performed at the same time, inching forward bit by bit. In this case there's just one programmer, or CPU, and the tasks compete to get access to this valuable resource. Otherwise, their work won't progress.

This way nicely illustrates that concurrent, multi-threaded code doesn't necessarily run faster than single-threaded code. In this case the tasks aren't being accomplished any quicker; they're just being organized differently.

The third way represents working in parallel. This represents the path of multi-threaded code running on a multi-core CPU. Given two tasks, they will be performed simultaneously, completing in half the time. Notice the subtle difference between #2 and #3. Both are concurrent, but only #3 is parallel.

The way I've explained it, #3 almost looks like two instances of #1 progressing side-by-side. This is one possible configuration, but it's also possible that as one programmer gets stuck on an issue, a context switch takes place. In this example, another programmer might come in to take his place. This preserves the 'inching forward bit by bit' idea, except now there are sufficient resources to keep the process inching forward continually.

Notice that more resources are required in order to work in parallel. The idea of one programmer working on two projects simultaneously, with one hand on each of two keyboards, for instance, is just as absurd as one CPU core executing two instructions simultaneously.

You can't guarantee anything will be parallel

This last example illustrated that your code can be concurrent without being parallel. Indeed, all you can do is to organize your code to be concurrent, using multiple threads, for instance. But **making it execute in parallel is out of your hands. That responsibility is left to the underlying thread scheduler.**

For instance, if you're running on a 4-core CPU, and you spawn 4 threads, it's possible, but unlikely, that your code will all be executed on just one CPU core. That's ultimately the choice of the thread scheduler. In practice, thread schedulers employ fair queueing so that all threads are given more-or-less equal access to available resources, but that cannot be controlled by your code.

The point of all this is to say that when you write multi-threaded code, you have no guarantee as to the parallelism of the environment that your code will be executed in. In practice, you should assume that your concurrent code *will* be running in parallel because it typically will, but you can't guarantee it!

Given this, multi-threaded code should be referred to as concurrent, rather than parallel. There's very little you can do from your side of the keyboard to guarantee that your code will run in parallel. However, **by making it concurrent, you enable it to be parallelized when the underlying system allows it.**

This is an important topic to grasp, so if you're still not 100% clear on concurrency versus parallelism, I highly recommend these two other explanations of the same conflation:

1. Rob Pike describes how concurrency enables parallelism, but they're not the same thing. Lots of simple diagrams to explain key concepts. link
2. Evan Phoenix describes the difference between concurrency and parallelism and how it relates to existing Ruby implementations. link

The relevance

I've been saying this is an important concept to grasp. The first reason is so that you can use the right terms when you're talking about it.

The second reason is that this lays a foundation for understanding the effects of thread synchronization and locking, which will get more coverage in coming chapters. Before that, this knowledge will drive your understanding of MRI's infamous GIL.