Introduction

This lesson introduces the concept of asynchronous programming as an important programming paradigm.

Introduction

Concurrency can be defined as dealing with multiple things at once. You can concurrently run several processes or threads on a machine with a single CPU but you'll not be parallel when doing so. Concurrency allows us to create an illusion of parallel execution even though the single CPU machine runs one thread or process at a time.

Parallelism is when we execute multiple things at once. True parallelism can only be achieved with multiple CPUs.

A machine with four cores and running one hundred processes is both parallel and concurrent but mostly concurrent because it can only do four things at a time.

Another paradigm to achieve concurrency on a single core-machine is via cooperative multitasking, which we discuss next.

Cooperative Multitasking

Multithreading on a single core machine achieves concurrency by context switching threads for CPU time. Threads can be taken off of the CPU if they engage in an I/O call or exhaust their time slice. The OS *preempts* a thread forcing it to give up the use of the CPU. Cooperative Multitasking, on the other hand, takes a different approach in which the running process voluntarily gives up the CPU to other processes. A process may do so when it is logically blocked, say while waiting for user input or when it has initiated a network request and will be idle for a while.

The process scheduler relies on the processes to *cooperate* amongst

themselves while using the CPU and is the reason why this paradigm of multitasking is called cooperative multitasking or non-preemptive multitasking.

Fibers in Ruby enable cooperative multitasking. The onus of scheduling tasks reliably for CPU time falls on the programmer. A misbehaving task can hog the CPU without giving other tasks a chance to execute and cause starvation.

A Ruby fiber comes with its stack and can be paused and resumed again. It is conceptually similar to a coroutine. Let's disambiguate fibers and related concepts first.

Coroutine

A coroutine is a general programming concept also found in various programming languages. A coroutine can be defined as a special function that can give up control to its caller without losing its state. The methods or functions that we are used to, the ones that conclusively return a value and don't remember state between invocations, can be thought of as a specialization of a coroutine, also known as **subroutines**.

Some languagues such as Python also have a related concept called **generators** which are essentially iterators and a subset of coroutines.

Fiber

Fibers are essentially the same concept as coroutines. Python, for instance, has coroutines which behave similar to Ruby fibers. Wikipedia says the only difference between them is fibers are a system level construct whereas coroutines are a language level construct. Coroutines may be thought of as implementation of fibers. From a practical standpoint, there is no difference when writing a coroutine or a fiber.

Note that another benefit of fibers is that they are lightweight in comparison to threads and don't suffer from thread-safety issues. We don't have to use any expensive locking or synchronization when working with fibers.

As an example, we can rewrite our ping-pong example from the threading lessons using fibers. For now, don't fret about the syntax and note that we

don't have any synchronization code in our solution below.

```
pingFib = Fiber.new do
  while true do
    puts "ping"
    Fiber.yield()
  end
end

pongFib = Fiber.new do
  while true do
    puts "pong"
    Fiber.yield()
  end
end


10.times do
  pingFib.resume()
  pongFib.resume()
end
```

Asynchronous Programming

So why are fiber important? Because they enable cooperative multitasking and why is cooperative multitasking important? Because cooperative multitasking allows us to program asynchronously. Let's work through an analogy to understand asynchronous programming first.

Consider a restaurant with a single waiter. Suddenly, three customers, Kohli, Amir and John show up. The three of them take a varying amount of time to decide what to eat once they receive the menu from the waiter. Let's assume Kohli takes 5 minutes, Amir 10 minutes and John 1 minute to decide. If the single waiter starts with Amir first and takes his order in 10 minutes, next he serves Kohli and spends 5 minutes on noting down his order and finally spends 1 minute to know what John wants to eat then in total, he spends 10 + 5 + 1 = 16 minutes to take down their orders. However, notice in this sequence of events, John ends up waiting 15 minutes before the waiter gets to him, Kohli waits 10 minutes and Amir waits 0 minutes.

Now consider if the waiter knew the time each customer would take to decide. He can start with John first, then get to Amir and finally to Kohli. This way each customer would experience a 0 minute wait. An illusion of three waiters, one dedicated to each customer is created even though there's only one. Lastly, the total time it takes for the waiter to take all three orders is 10 minutes, much less than the 16 minutes in the other scenario.

The single waiter scenario can be mapped to a single thread juggling various items and whenever an item gets blocked on an operation that requires time to receive a result externally, the thread shifts to executing another ready item. This is an example of asynchrony using a single thread.

Going back to our restaurant analogy, we can increase the number of waiters to serve an increasing number of patrons at the restaurant. This would constitute an example of asynchrony with multiple threads, which can't be achieved in MRI Ruby because only a single thread can execute at a time.

Finally, it should be obvious that dedicating one waiter for each customer is an example of multithreading with no asynchrony and wasteful. Tying up one thread for each work item that blocks the thread waiting for results from an external source limits the number of work items a system can service.

Also, note that when we say that a thread waits for a result to arrive from an external source, it means hardware components such as the disk controller, network card, etc far below the level of OS threads are to bring back a result for the waiting thread. The waiting thread sits completely idle using up memory and other resources while the external source is working on a result for the waiting thread. Remember, there's no other hidden thread that is working to produce a result for the blocked thread.

We can achieve asynchrony using a single thread or multiple threads but multithreading doesn't imply asynchrony. We can have a multithreaded system with no asynchrony at all.

Incorporating asynchrony improves the throughput of a system - more work gets done with the same resources. The same number of waiters can serve more patrons. But asynchrony doesn't speed up computation. A waiter can't take an order faster than the time a customer takes to decide for an order.