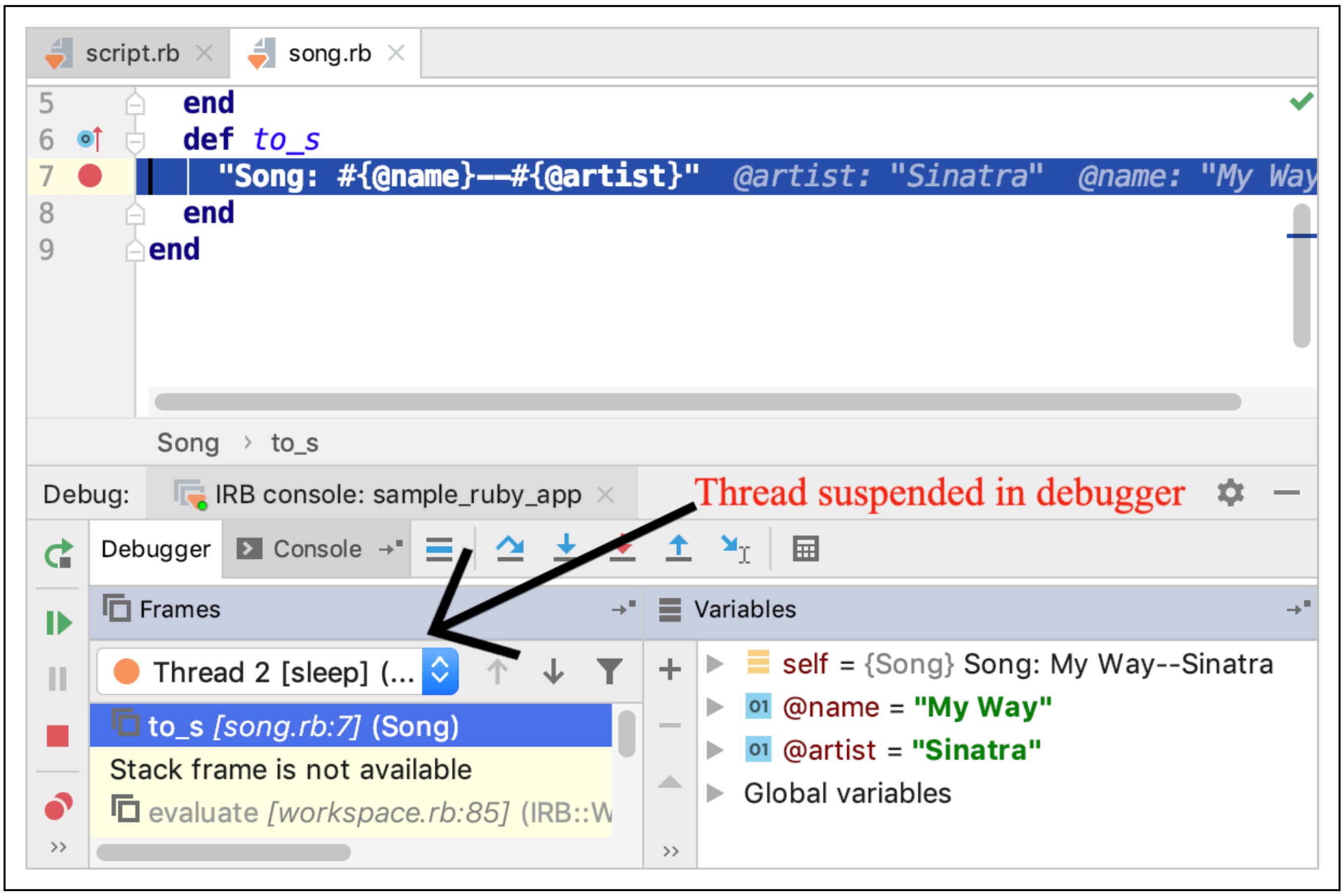
Introduction

This lesson introduces concurrency and provides motivational examples to further our understanding of concurrent systems.

Introduction

Understanding of concurrency and its implementation models using either threads or coroutines exhibits maturity and technical depth in a candidate, which can be an important differentiator in landing a higher leveling offer at a company. More importantly, grasp over concurrency lets us write performant and efficient programs. You'll inevitably run into the guts of the topic when working on any sizeable projects.

We'll start with threads as they are the most well-known and familiar concepts in the context of concurrency. Threads, like most computer science concepts, aren't physical objects. The closest tangible manifestation of a thread can be seen in a debugger. The screen-shot below shows the threads of an example program suspended in the debugger.



```
def to_s
  "Song: #{@name}--#{@artist}"  @artist: "Sinatra"  @name: "My Way
end
end
```

Thread suspended in debugger

Debug: IRB console: sample_ruby_app

Thread 2 [sleep] (...

to_s [song.rb:7] (Song)
Stack frame is not available
evaluate [workspace.rb:85] (IRB::W

```
self = {Song} Song: My Way--Sinatra
@name = "My Way"
@artist = "Sinatra"
Global variables
```

The simplest example of a concurrent system is a single-processor machine running your favorite IDE. Say you edit one of your code files and click save. That clicking of the button will initiate a workflow which will cause bytes to be written out to the underlying physical disk. However, IO is an expensive operation, and the CPU will be idle while bytes are being written out to the disk.

While IO takes place; the idle CPU could work on something useful. Here is where threads come in - the IO thread is ***switched out*** and the UI thread gets scheduled on the CPU so that if you click elsewhere on the screen, your IDE is still responsive and does not appear hung or frozen.

Threads can give the illusion of multitasking even though the CPU is executing only one thread at any given point in time. Each thread gets a slice of time on the CPU and then gets switched out either because it initiates a task that requires waiting and not utilizing the CPU, or it completes its time slot on the CPU. There are many nuances and intricacies on how thread scheduling works, but what we just described forms the basis of it.

With advances in hardware technology, it is now common to have multi-core machines. Applications can take advantage of these architectures and have a dedicated CPU run each thread.

If you code in Ruby, you would already have had worked with threads, albeit unknowingly. In the absence of any explicitly instantiated threads, one's code runs in the main thread. Consider the snippet below, which prints the id of the current thread in which the code executes.

```
# Prints id of main thread
puts Thread.current.__id__
```

Benefits of Threads

1. **Higher throughput**, though in some pathetic scenarios, it is possible to have the overhead of context switching among threads steal away any throughput gains and result in worse performance than a single-threaded scenario. However, such cases are unlikely and are exceptions rather than the norm. Also, multithreaded programs can utilize multiple cores in a machine but a single-threaded program limits itself to a single core even if multiple are available.

2. **Responsive applications** that give the illusion of multitasking on single core machines.

3. **Efficient utilization of resources**. Note that thread creation is lightweight in comparison to spawning a brand new process. Web servers that use threads instead of creating new processes when fielding web requests consume far fewer resources. Moreover, because all threads within a process share the same address space, they need not use shared memory, unlike processes.

All other benefits of multi-threading are extensions of or indirect benefits of the above.

Performance Gains via Multi-Threading

As a concrete example, consider the example code below. The task is to ***compute the sum of all the integers from 0 to 300000000 (30 million)***. In the first scenario, we have a single thread doing the summation while in the second scenario, we split the range into two parts and have one thread sum for each range. Once both the threads are complete, we add the two half sums to get the combined sum. Finally, we repeat the previous exercise using processes instead of threads. We measure the time taken for each scenario and print it.

```
MAX_NUM = 30000000

class SumUpExample

  def initialize(startRange, endRange)
    @counter = 0
    @startRange = startRange
    @endRange = endRange
  end

  def add()
    for i in @startRange..@endRange do
      @counter += i
    end
  end

  def getCounter
    return @counter
  end
end

def singleThread()
  s = SumUpExample.new(1, MAX_NUM)

  thread = Thread.new do
    s.add()
  end

  thread.join()
  return s.getCounter()

end

def runTwoThreads()
  s1 = SumUpExample.new(1, MAX_NUM / 2)
  s2 = SumUpExample.new(1 + (MAX_NUM / 2), MAX_NUM)

  thread1 = Thread.new do
    s1.add()
  end

  thread2 = Thread.new do

    s2.add()
  end

  thread1.join()
  thread2.join()
  return s1.getCounter() + s2.getCounter()

end

def runTwoProcesses()

  read1, write1 = IO.pipe
  read2, write2 = IO.pipe

  pid1 = Process.fork do
    s = SumUpExample.new(1, MAX_NUM / 2)
    s.add()
    write1.puts s.getCounter()
  end

  pid2 = Process.fork do
    s = SumUpExample.new(1 + (MAX_NUM / 2), MAX_NUM)
    s.add()
    write2.puts s.getCounter()
  end

  Process.wait pid1
  Process.wait pid2

  write1.close()
  write2.close()

  return (read1.read().to_i) + (read2.read().to_i)

end


starting = Process.clock_gettime(Process::CLOCK_MONOTONIC)
sum = singleThread()
ending = Process.clock_gettime(Process::CLOCK_MONOTONIC)
puts "Single thread took : #{ending - starting} to sum to #{sum}"

starting = Process.clock_gettime(Process::CLOCK_MONOTONIC)
sum = runTwoThreads()
ending = Process.clock_gettime(Process::CLOCK_MONOTONIC)
puts "Two Threads took : #{ending - starting} to sum to #{sum}"

starting = Process.clock_gettime(Process::CLOCK_MONOTONIC)
sum = runTwoProcesses()
ending = Process.clock_gettime(Process::CLOCK_MONOTONIC)
puts "Two Processes took : #{ending - starting} to sum to #{sum}"
```

The table below presents the time in seconds it takes for various scenarios to sum up integers from 1 to 30 million in one of the runs of the above code widget.

Setup	**Time (secs)**
Single Thread	3.97
Multiple Threads	3.98
Multiple Processes	3.97

The results are interesting and counterintuitive. We would expect the multithreaded scenario to perform better than the single-threaded one since two threads can work in parallel if the system has at least two CPUs. The relatively poor performance in comparison to the single-threaded scenario can be explained by (MRI) Ruby's Achilles' heel, the **Global Interpreter Lock**, an entity within the Ruby interpreter that allows a single thread to execute even in the presence of more than one idle CPUs. We'll have more to say on that subject in later sections. Multithreaded scenarios may not experience any performance gains in case of CPU-intensive tasks such as the one in our example because threads don't execute in parallel and incur an additional cost of management and coalescing partial results.

With multiple processes, we can expect each process to be scheduled onto a separate CPU and work in parallel. However, there's the additional overhead of creating and tearing down processes, which is higher than doing the same for threads. Additionally, communication between processes is more expensive than between threads as threads share the same address space, but processes don't. Collectively these headwinds explain why the multiprocessing scenario performs no better than a explain why the multiprocessing scenario performs no better than a single-threaded scenario.

Although, these numbers aren't definite and may vary depending on the hardware configurations and the version of Ruby you run. For instance, on my Macbook, the runs for the three scenarios take roughly one second to complete and the multiprocessing scenario performs the best by a significant margin. The takeaway is that there are several variables at play that determine the running time of a program and one can't assume performance improvement for a program just because it is multithreaded or runs using multiple processes.

However, in general, tasks involving blocking operations such as network and disk I/O, can see significant performance gains when migrated to a multithreaded or multiprocessor architecture.

As a demonstration, we present the same summing-up program written in Java in the code widget below. Java is a truly multithreaded language i.e. it doesn't have the GIL restriction and can run multiple threads in parallel using multiple cores. You aren't required to understand the syntax or the program. If you run the code widget below, you'll observe that the two threads run faster than a single thread when we sum up to 3 billion. If you remove two zeroes on **line#12** and change the variable `MAX_NUM` to 30 million, the single thread runs faster! This example demonstrates that there's a tipping point after which multiple threads solve a computationally-intensive problem faster than a single thread does but before that tipping point the single thread runs faster because it doesn't deal with context-switches, shared memory accesses and other managerial overhead associated with multiple threads or multiple processes.

```
class Demonstration {
    public static void main( String args[] ) throws InterruptedException {
        SumUpExample.runTest();
    }
}

class SumUpExample {

    long startRange;
    long endRange;
    long counter = 0;
    static long MAX_NUM = 3000000000L;

    public SumUpExample(long startRange, long endRange) {

        this.startRange = startRange;
        this.endRange = endRange;
    }

    public void add() {

        for (long i = startRange; i <= endRange; i++) {

            counter += i;
        }
    }

    static public void twoThreads() throws InterruptedException {

        long start = System.currentTimeMillis();
        SumUpExample s1 = new SumUpExample(1, MAX_NUM / 2);
        SumUpExample s2 = new SumUpExample(1 + (MAX_NUM / 2), MAX_NUM);

        Thread t1 = new Thread(() -> {
            s1.add();
        });

        Thread t2 = new Thread(() -> {
            s2.add();
        });

        t1.start();
        t1.join();

        t2.start();
        t2.join();

        long finalCount = s1.counter + s2.counter;

        System.out.println("Two threads final count = " + finalCount + " took " + (System.cur
    }

    static public void oneThread() {

        long start = System.currentTimeMillis();
        SumUpExample s = new SumUpExample(1, MAX_NUM );
        s.add();
        System.out.println("Single thread final count = " + s.counter + " took " + (System.cu
    }


    public static void runTest() throws InterruptedException {

        oneThread();
        twoThreads();

    }
}
```

Problems with Threads

There's no free lunch in life. The premium for using threads manifests in the following forms:

1. ***It's usually tough to find bugs***, some that may only rear their heads in production environments.
2. ***Higher cost of code maintenance*** since the code inherently becomes harder to reason about.
3. ***Increased utilization of system resources***. The creation of each thread consumes additional memory, CPU cycles for book-keeping, and a waste of time in context switches. Multithreaded scenarios may not experience any performance gains in case of CPU-intensive tasks such as the one in our example because threads don't execute in parallel and incur an additional cost of management and coalescing partial results.
4. ***Programs may experience slowdown*** as coordination amongst threads comes at a price. Acquiring and releasing locks adds to program execution time. Threads fighting over acquiring locks cause lock contention.

With this backdrop, let's delve into more details of concurrent programming, which you are likely to be quizzed about in an interview.