**MULTITHREADING: THREAD SCHEDULING IN JAVA AND RUBY**

Bhavani Prasad Rao Ejanthkar,

***ABSTRACT***

*The concept of multithreading was first researched and implemented by IBM in the year 1967. Multithreading allows us to execute the task in parallel thereby maximizing the utilization of the core. Proper thread scheduling is mandatory when executing a multithreaded application. Though the overall concept of multithreading is similar its implementation may vary among different types of languages. Since Java was the first programming language to implement threading in the language itself and Ruby is the present trending language a lot of syntactic and semantic differences can be observed between these two languages over the concept of in multithreading. Similarly, these differences can be observed among the other concepts (thread scheduling) of multithreading.*

***Keywords:*** *multithreading, core, Java, Ruby, syntactic,**semantic, thread scheduling*

**Table of Contents**

**Introduction** **1**

**Problem Statement** **1**

**Java** **2**

Introduction 2

Multithreading in Java 2

Thread Creation 2

Race Conditions and Critical Sections 2

Deadlock 2

Context Switching Overhead 3

Parallelism and Synchronization 3

Thread Scheduling in Java 3

Thread Priority 3

User Level Threads 3

Kernel Level Threads 3

Green Thread Scheduling Model 3

Native OS Level Thread Scheduling 3

Preemptive Scheduling 3

Non-Preemptive Scheduling 3

Equal Priority Threads 4

JRuby 6

**Test Cases** **6**

Fibonacci Series 6

Count Million 7

Merge Sort 7

**Syntactic and Semantic Analysis …………………. 8**

**Conclusion** **9**

**Critique** **9**

**References** **10**

**INTRODUCTION**

Multithreading is the ability to break a task into several subtasks and perform execution of those subtasks in parallel on single or multiple cores of CPU. Multithreading increases the utilization of one core by using instruction level and thread-level parallelism. Multiprocessing is different from multithreading, in multithreading the threads and processes share the resource of multiple cores or a single core, whereas in multiprocessing multiple processing units are involved. To perform multithreading and execute the code in the right manner, threads must be scheduled appropriately. Scheduling of the threads is done based on some algorithms. Though the concept of thread scheduling remains same, its implementation varies among different languages. In this paper, we tried to compare the concept of thread scheduling between Java and Ruby. The reason for us to select Java is, Java was the first language to implement the concept of threads in the language itself and the reason to select Ruby as our second language is, it offers better flexibility, readability and has fewer lines of code. The inferences obtained after comparing the concept of thread scheduling in multithreading are provided below in detail.

**PROBLEM STATEMENT**

Implementing the concept of thread scheduling in two different languages in order to differentiate their behavior. Among the selected languages one was statically typed and the other dynamic typed language.

Static Typed Language: Java

**Ruby** **4**

Introduction 4

Multithreading in Ruby 4

Ruby 1.8 4

Ruby 1.9 4

Dynamic Typed: Ruby

This paper focuses on above-mentioned languages and explores the syntactic and semantic differences between these languages. It also explains the functionality of threading and thread scheduling in these languages.

CSCI 56500|Multithreading: Thread Scheduling 1

**1. JAVA**

1.1 Introduction:

Java was developed at Sun Microsystems by James Gosling and released in the year 1995 as a core component of ‘Sun Microsystems’ Java platform. Java is general purpose programming language that is class-based, concurrent, statically typed and object-oriented language. Java is platform independent i.e. a compiled Java code can run on any platform that supports Java without a need for recompilation. Initially, Java code is converted to bytecode, and that bytecode can run on any JVM (Java Virtual Machine) irrespective of the computer architecture. The Syntax of Java is derived from C and C++. Initially programming in Java was slower and required more memory when compared with C++, however, execution speed of Java has been improved with the introduction of JIT (Just-in-time compilation). Java does automatic garbage collection to manage memory in object lifecycle.

1.2 Multithreading in Java:

A thread is a lightweight process which can run concurrently. A thread allows execution of two or more sections of a program at the same time. Execution of threads is independent of each other as each thread has different execution path, so exception in one thread does not affect the execution of the other thread. Executing of multiple threads in parallel is called multithreading. Purpose of multithreading is to utilize the available CPU resources in an effective way. Every Java program has a minimum of one thread. Types of threads in Java:

* User Thread: These are user created threads and have higher priority. JVM will not exit until all the user threads finish their execution. These are foreground threads.
* Daemon Threads: These are JVM created threads, they always run in the background. These threads are used to perform JVM activities like garbage collection, house-keeping tasks etc. These are low priority threads. JVM does not wait for these threads to finish their execution, it exits immediately after user threads are executed.

1.2.1 Thread Creation:

In Java, threads can be created in two ways – by extending Thread class and the other by implementing a Runnable interface.

* Extending Thread Class: By extending the thread class each thread will have a unique object associated with it. Extending the thread class is preferred only when the class that needs to be executed as thread need not be executed from

other class, this approach provides more flexibility in handling the threads.

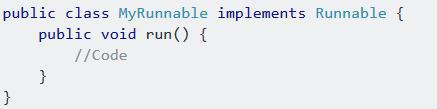
Thread t = new Thread();  Thread creation t.start();  start the thread



* Implementing Runnable Interface: Methods in the Runnable interface are abstract methods. In this approach, only one instance of a class is created and all the threads will share the object instance of that specific class where an interface or more than one interfaces can be extended.

MyRunnable m = new MyRunnable();

Thread t = new Thread(m);  to start the thread.



It is preferred to create a class by implementing a Runnable interface because we can implement more than one interface at once (Multiple Inheritance) whereas multiple classes cannot be extended in Java. When we implement Runnable interface, the same object is shared to multiple threads, whereas by extending thread class each thread creates unique object and associates with it.

1.2.2 Race Conditions and Critical Sections:

The situation where more than one thread competes for the same resource, where the sequence of thread execution changes the output then such a situation is called race condition. A code section that leads to race condition is known as the critical section. To avoid race conditions Java provides **synchronized** blocks or methods another way to prevent race condition is by using **locks** or **atomic variables**.

1.2.3 Deadlock:

A situation where two or more threads are waiting for each other. Deadlock occurs when more than one thread needs the same locks but obtain them in a different order. Improper use of synchronize keyword

CSCI 56500|Multithreading: Thread Scheduling 2

can lead to deadlock. To prevent deadlock unnecessary locks must be avoided in the code.

1.2.4 Context Switching Overhead:

Switching the execution from one thread to another without completing the execution of the current thread is known as context switching. Whenever context switch happens CPU must save the environment (variable values, program pointer etc.) of the current thread and load the environment of the new incoming thread. Context switching isn’t preferred until it is required.

1.2.5 Parallelism and Synchronization:

Dividing a task into several subtasks and executing those subtasks in parallel is called parallelism. Threads when executing in parallel communicate by sharing access to fields and objects reference fields refer to. Synchronization is important in Java since multiple threads will run in parallel to complete program execution. Synchronization in Java can be done using **synchronize** and **volatile** keywords.

1.3 Thread Scheduling in Java:

1.3.1 Thread Priority:

Every Java thread has a priority that helps the OS to determine the order in which threads are scheduled. By default, a Java thread inherits the priority from its parent. In Java, there are 3 thread priorities,

1. MIN\_PRIORITY: priority value is 1. o MAX\_PRIORITY: priority value is 10. o NORM\_PRIORITY: priority value is 5.

We can modify the priority of the thread at any time after its creation using the **setPriority** method which is a method of the thread class, **getPriority()** is used to know the priority of the thread.

1.3.2 User Level Threads:

The user level threads lie in user space and the scheduling of these threads are based on the application. These threads are user manageable threads. User level threads follow deterministic thread scheduling algorithm. These are fast because thread creation, scheduling, and termination are managed by the user. Scheduler schedules the threads based on their priority (higher priority threads are scheduled first)

1.3.3 Kernel Level Threads:

The kernel level threads lie in the kernel space and are managed by the OS. These threads are slower because here the thread creation, scheduling is managed by the kernel. Context switching between kernel level threads is slow.

1.3.4 Green Thread Scheduling Model:

In this model, the user level threads (JVM) are directly given to underlying hardware for its execution. Moreover, in this model neither kernel space nor OS scheduler exists. Thread scheduling is independent of OS. Scheduling of the threads is done by Virtual Machine (VM)/runtime library. JVM manages the threads and the execution of the threads is deterministic and is handled by the application in the JVM.

1. No OS scheduler.
2. Provides multithreaded environment.
3. Even if a single thread in a process is blocked, the entire process will be blocked.

1.3.5 Native OS Level Thread Scheduling:

In this model, the user level threads (JVM) are directly given to kernel. The kernel maps these native level threads to kernel level threads (OS) which are given to underlying hardware for their execution. Here, the scheduling of the thread is done by both thread library (JVM) and the OS scheduler (Kernel). The thread is dependent on JVM, OS and underlying hardware. Since the scheduling of the thread is dependent on OS and the hardware, these threads will be executed in a non-deterministic way. This scheduling model is provided by Windows OS. This model is provided by JVM 1.1 later versions.

1. User threads make use of multi-core processor efficiently.
2. If one thread in a process is blocked, other threads are executed.

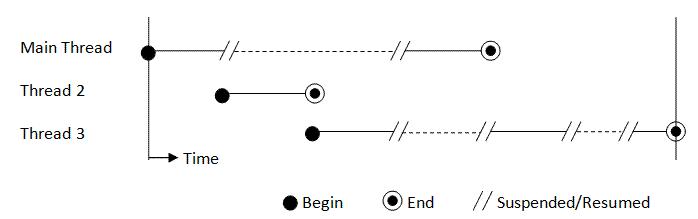
1.3.6 Preemptive Scheduling:

In the preemptive model, the JVM doesn’t wait for the thread to complete its execution. The context switch can happen if any thread with the higher priority comes into the runnable state. If a thread with low priority is being executed and another thread with the higher priority comes into the runnable state, then JVM switches the control from the currently executing thread to the thread with higher priority. Java follows priority-based preemptive thread scheduling i.e. a thread with high priority will be executed first. JVM executes the thread with lower priority only if, thread with high priority goes to the sleep state or it blocks, or it completes its execution. However, if the high priority thread unblocks or wakes up then JVM switches the control to high priority thread and the current execution thread will be blocked.

1.3.7 Non-Preemptive Scheduling:

In the non-preemptive model, JVM waits till thread completes its execution or thread yields control or thread goes to block state.

CSCI 56500|Multithreading: Thread Scheduling 3



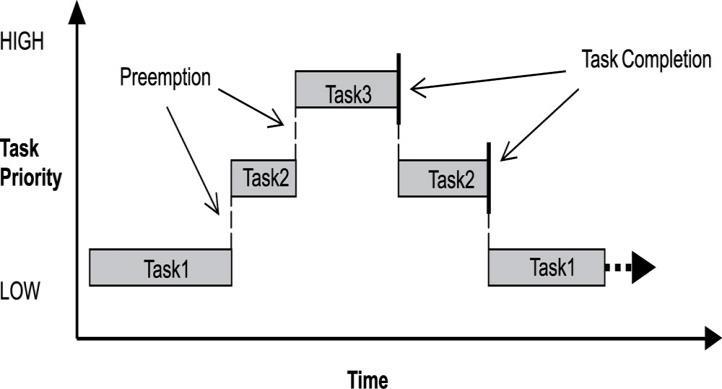
1.3.8 Equal Priority Threads:

If we have a scenario where a thread is being executed by the JVM and another thread with the same priority as the currently executing thread changes its state to runnable, then which thread should finish its execution first?

Case 1, threads are executed in time slicing manner. In this method CPU will allocate some fixed time for execution of each thread, the current thread is executed till the time allotted then context switch happens to the next thread of same priority. Likewise, context switch happens between the threads till they finish their execution.

Case 2, this follows non-preemptive scheduling algorithm i.e., if two threads of same priority move to runnable state then schedule randomly schedules one thread among them and keep the other thread in the wait state until at the scheduled thread finishes its execution.

Preemptive scheduling



**2. RUBY**

2.1 Introduction:

Ruby was created by Yukihiro Matsumoto, or “Matz”, in Japan in the mid-1990. It is a dynamic, reflective,

object-oriented, general-purpose programming language. It has dynamic type system and automatic memory management. Ruby is pure object-oriented i.e. every value is an object, including classes and type instances (which are considered as primitives like int, bool etc. in other languages). Ruby derives its syntax from Perl and Python. One difference when Ruby compared to Python and Perl is Ruby keeps all of its instance variables private to the class and exposes them to ancestor methods (attr\_writer, attr\_reader, etc.). The official Ruby Interpreter is named as Matz’s

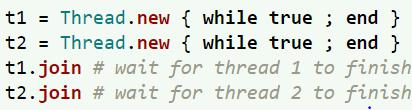
Ruby Interpreter or MRI. Its implementation is written in C language and uses its own Ruby-specific virtual machine. The standardized Ruby 1.8 interpreter was written as single-pass interpreted language. From Ruby 1.9 and continuing with Ruby 2.x, the official Ruby interpreter is YARV (Yet Another Ruby VM).

2.2 Multithreading in Ruby:

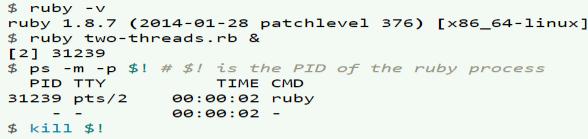
The two common terms that we come across in multithreading are concurrency and parallelism, Ruby concurrency is when two tasks start, run and complete in overlapping time periods. It doesn’t mean that they’ll be running at the same instant. Ruby parallelism is when two tasks literally run at same time (multiple threads on multiple cores execute at an instant).

2.2.1 Ruby 1.8:

Ruby v1.8 supports only green threads, the difference between the green threads and native threads is, green threads are not visible to the kernel and native threads are visible to the kernel. In Ruby v1.8 all the thread creation, scheduling, and destruction takes place in the user space and is hidden from the kernel. Now, let us check this with a test program (two-threads.rb): Code: Two-threads.rb



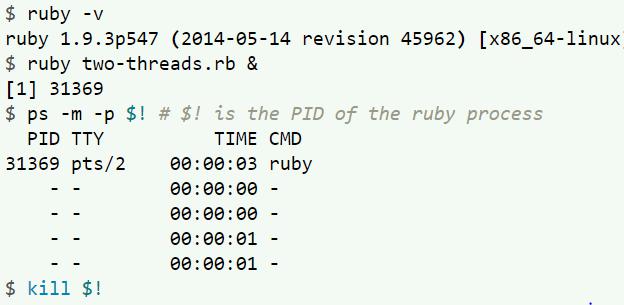
Output:



As we can see here ps command tells us that process is single threaded in Ruby v1.8.

2.2.2 Ruby 1.9:

Ruby v1.9 supports native threads. Native threads are superior to green threads because they are the `only schedule the threads on multiple processors. To achieve concurrency green threads are sufficient but parallelism requires native threads. Now, let us check the same two-threads.rb program in Ruby v1.9 Output:



Here we were expecting to see 3 threads but surprisingly we got 4 threads, in order to explain the extra thread, first we must know about GIL which is a feature in Ruby v1.9.

CSCI 56500|Multithreading: Thread Scheduling 4

2.2.2.1 GIL:

GIL stands for Global Interpreter Lock, According to Dr. Ilya Grigorik (web engineer at Google, author of High-Performance Browser Networking (O’Reilly) book) – “Parallelism is a Myth in Ruby” – because GIL doesn’t allow threads to be executed in parallel. Before going to the concept of GIL, let us check a sample code which is executed in Ruby v1.9 and Ruby on other VM’s like JVM (Ruby on JVM is called JRuby), LLVM (Ruby on LLVM is Rubinius).

Here the sample code uses 5 different threads where each thread pushes nil into the array 1000 times and we are returning the size of the array. The expected output is 5000 by the Ruby code when executed on all the VM’s.

Code: Output:

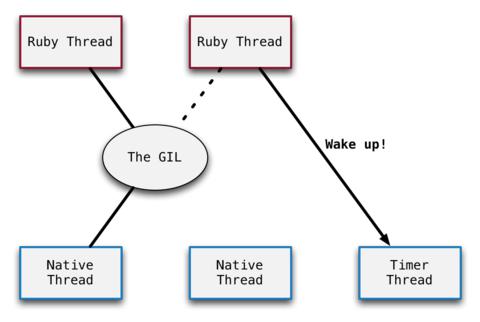


Here, we can notice that MRI produced the correct result, 5000. Both JRuby and Rubinius produced an incorrect result. These inconsistent results are because of context switching and race conditions. Because MRI has GIL, even when there are 5 threads running at once, only one thread will be active at a time. JRuby and Rubinius don’t have GIL, so when we have 5 threads, all the 5 threads will be running in parallel across all the cores. Therefore on parallel implementation of Ruby, all the 5 threads enter the code that is not thread-safe thereby corrupting the data because of context switching. When we are using MRI, JRuby or Rubinius, Ruby language is implemented on the top of another language like in MRI (the Ruby code is implemented in C), in JRuby (Ruby code will be implemented in Java), Rubinius (Ruby implemented in a mixture of C++ and Ruby). As MRI is the standard interpreter of Ruby all the inbuilt API’s of MRI are atomic thereby making the Ruby code (which uses the inbuilt API’s) on MRI thread-safe, but when the same Ruby code is implemented on other VM’s they may not be atomic because core collections differ from language to language, example consider the above code where the array is an atomic operation in C, if we perform any other operation on an atomic function then we cannot say this function is thread-safe and atomic i,e. if we insert elements in the array and try to find any element from it then this additional function will access the ruby code which is not atomic as we have written it.So, the MRI cant guarantee the thread safety for all

the codes written or any functions which are not the C libraries. So, Ruby on JVM (JRuby) and Ruby on LLVM (Rubinius) do not produce the correct result as the core collections vary from language to language. So, in order to produce the correct result in JRuby or Rubinius or in some cases of MRI as well, we have to make the operations atomic by making them atomic we can ensure that it can’t be interrupted until its execution is finished. The simplest way to make an operation atomic is, use a lock. So in the above code if we use mutex lock then the code is guaranteed to be correct. Now, in MRI we said that only one thread will be executed on one core in an instant and this is because of GIL, the functionality of GIL is explained below in detail

* As we know that an MRI thread will be backed by a native thread for its execution.
* When a Ruby thread wants to execute code in its native thread, it must first acquire the GIL, GIL mediates access between Ruby thread and its underlying native thread, severely reducing parallelism. Only one thread can acquire GIL at any given time, so parallel execution of code is disabled.
* GIL protects the internal state of the system. With

GIL, we don’t require any locks or synchronization. If 2 threads can’t be mutating internal data structures at same time, then no race condition can occur.



* The timer thread – The timer thread is a native thread that exists internally in MRI, no Ruby thread will be backed to the timer thread. The timer thread is started up when MRI starts up. When MRI boots up and the only main thread is running the timer thread sleeps, but when a Ruby thread is waiting for the GIL then that Ruby thread wakes up the timer thread. The timer thread sets a boolean flag to true if context-switch is required, while executing the code the interpreter checks for the boolean flag if the value of the flag is true then it halts the execution of the current thread and releases the GIL for the next waiting thread (the GIL will be assigned to next thread which is scheduled by the scheduler). The thread scheduling is done by the scheduler (Scheduler

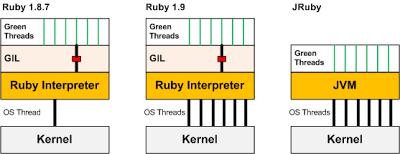
CSCI 56500|Multithreading: Thread Scheduling 5

uses scheduling algorithms to schedule the threads for execution) in Ruby. The functionality of timer thread is simple

1. Wait for a fixed period of time (100ms)
2. Set the flag to true.
3. Start over with step 1.

2.3 JRUBY:

JRuby is an implementation of Ruby code on JVM. JRuby removes GIL feature that CRuby (Ruby on MRI) has thereby allowed us to execute truly code in parallel. JRuby is the first Ruby implementation to offer thread-level parallelism. In JRuby, Ruby threads are backed by Java threads which are backed by the native threads. So the thread scheduling is JRuby is similar to the thread scheduling in Java, as JRuby thread is mapped on to a Java thread. JRuby provides concurrency primitives as in standard Ruby in thread library. Queue, Mutex, ConditionVariable etc. work as they do in MRI, they are useful in writing thread-safe code in JRuby.



JRuby reuses the standard library functions and does not alter those libraries, but all the standard libraries are not thread-safe except for some libraries like, generator, thread, timeout, weakref, and fcntl are expected to be thread-safe.

As JRuby is run on JVM, it also has access to all JVM offers related to concurrency. The package java.util.concurrent contains many thread-safe collections like sets, queues and other data structures. JRuby provides a non-standard way to convert class or object to fully synchronized data structure: the module

JRuby:: Synchronized. We need ‘require jRuby/synchronized’ and then include ‘include JRuby:: Synchronized’ for any class or ‘obj.extend(JRuby: :Synchronized)’ for any object.

Performance is guaranteed to be better than CRuby in terms of multithreading. As we don’t have GIL in JRuby, it is the programmer’s responsibility to make the code thread-safe. The best part about JRuby is, it runs most of the CRuby code without any syntactical changes.

Pros of JRuby:

* Ruby on JVM means it can run anywhere similar to Java (like on Android).
* Multithreading in JRuby will be nearly 2 to 5 times faster than multithreading in CRuby.
* JRuby can make use of Java platform which includes both standard and third-party libraries.

Cons of JRuby:

* JRuby has standard JVM related issues like the slow startup of Ruby interpreter.
* Debugging classpath issues if we are using third party libraries of Java.
* Larger memory usage.
* Code must be written thread-safe by the programmer.
* Features of Ruby i.e. C API’s are not implemented

in JRuby.

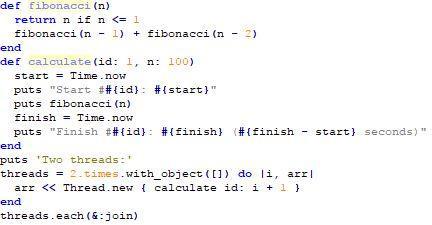
Performance wise JRuby tends to be slower than Java. In cases where the code involves more IO operations, more user interaction is required etc., Java tends to perform better than JRuby. Another reason for Java to perform better than JRuby is, Java is statically typed language whereas Ruby is dynamically typed language.

**3. TEST CASES**

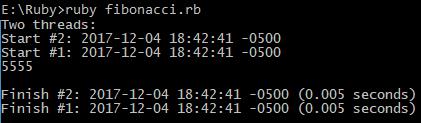
3.1 FIBONACCI SERIES:

3.1.1 RUBY:

Code: fibRuby.rb



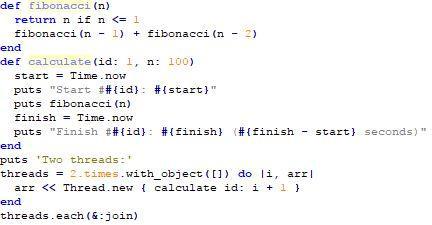
Output:



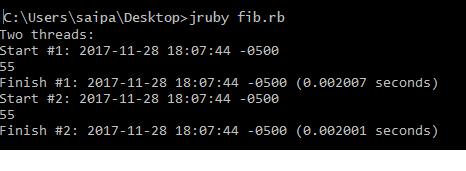
3.1.2 JRUBY:

Code: fibJRuby.rb

CSCI 56500|Multithreading: Thread Scheduling 6

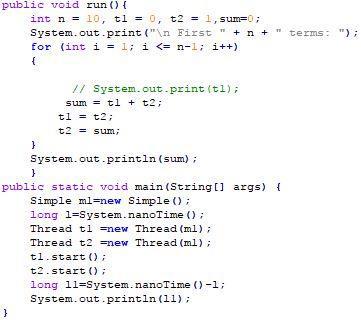


Output:

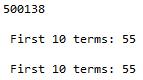


3.1.3 JAVA:

Code: fib.java



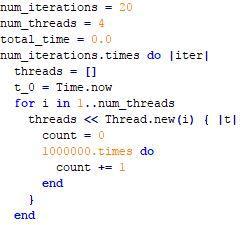
Output:



3.2 COUNT MILLION:

3.2.1 RUBY:

Code: countRuby.rb



Output:



3.2.2 JRUBY:

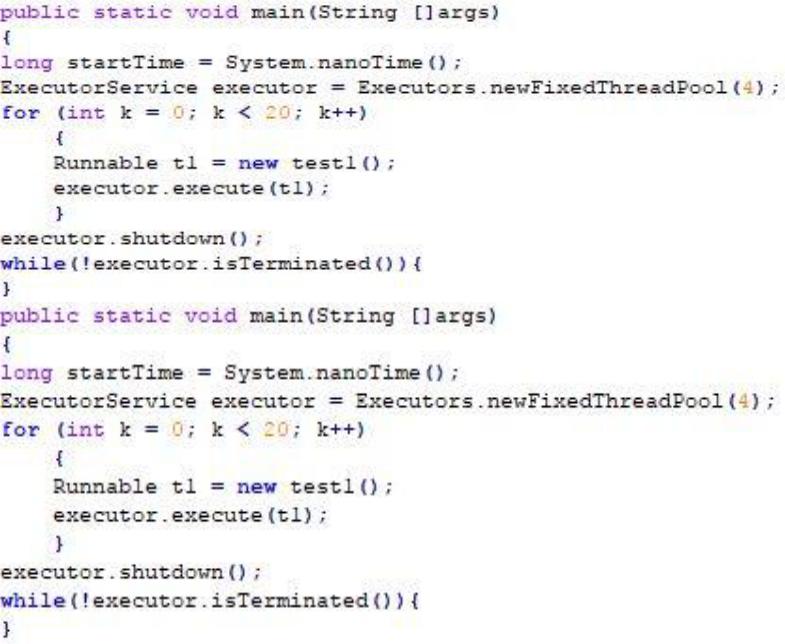
Code: countJRuby.rb

Output:



3.2.3 JAVA:

Code: count.java



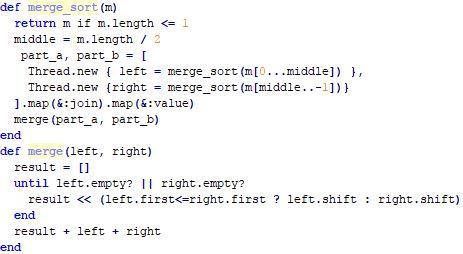
Output:



3.3 MERGE SORT:

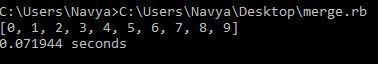
3.3.1 RUBY:

Code Snippet: mergeRuby.rb



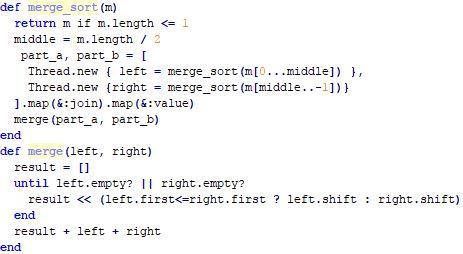
CSCI 56500|Multithreading: Thread Scheduling 7

Output:



3.3.2 JRUBY:

Code: mergeJRuby.rb

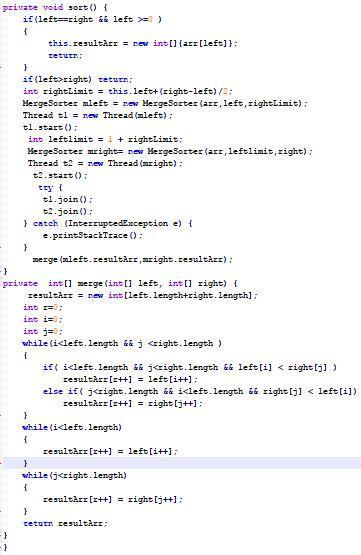


Output:



3.3.3 JAVA:

Code: merge.java



Output:



For both Fibonacci series and merge sort algorithm, the same code is used in both the languages Ruby and JRuby to check the concept of parallelism and to also know whether Ruby is supported on JRuby. Whereas for the program count up to million separate code is implemented in Ruby and JRuby to check whether JRuby supports Java and use its libraries.

From the use cases, it is observed that Java performs way faster than JRuby and JRuby run faster than Ruby. JRuby is expected to work as fast as Java but during the implementation of these use cases its noticed that JRuby is slower than Java. This is because of the package *require ‘java’* which (used to call Java from JRuby) takes more time to load Java Libraries into JRuby.

**SYNTACTIC and SEMANTIC ANALYSIS**

|  |  |
| --- | --- |
| RUBY | JAVA |
|  |  |
| Interpreter based | Compiler-based language |
| language |  |
|  |  |
| Dynamically typed | Statically typed |
|  |  |
| Pure object-oriented | Not purely object-oriented |
| language | language |
|  |  |
| Syntax is terse | Syntax is not terse |
|  |  |
| Type declaration not | Type declarations are |
| required | required |
|  |  |
| It is run directly without | The code is compiled in Java |
| compilation. | and bytecode is generated |
|  | for it. |
|  |  |
| require is used to import | import keyword is used to |
| package or module | import package or module. |
|  |  |
| Exceptions are handled | Exceptions are handled |
| using begin-rescue- | using try-catch-finally |
| ensure-end construct |  |
|  |  |
| Multiple inheritance can | Multiple inheritance is |
| be implemented by | implemented by using |
| modules and mixins | interfaces. |
|  |  |
| Public, private and | Only public and protected |
| protected methods can be | methods can be inherited |
| inherited |  |
|  |  |

CSCI 56500|Multithreading: Thread Scheduling 8

|  |  |
| --- | --- |
| Focused on the behavior | Focused on type of objects |
| of objects |  |
|  |  |
| Null value is declared | Null value is declared using |
| using nil | null |
|  |  |
| Duck typing is supported | Duck typing is not supported |
|  |  |
| All the members are | Default access specifier is |
| private | public |
| Method overloading is | Method overloading is |
| not supported | supported |
| Some syntax is optional | Syntax is not optional |
| unless result is ambiguous |  |
| Operator overloading is | Operator overloading is not |
| supported | supported |
|  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Readability | Easy to read | Not so easy | Easy when |
|  | and | as Ruby | compared |
|  | understand |  | with Java |
|  |  |  |  |

**CONCLUSION**

Finally, this paper mainly focused on the concepts of thread scheduling which is a feature of multithreading. Two programming languages opted are Java and Ruby and the following factors were achieved: syntactic and

semantic analysis between the languages, implementation of thread scheduling in multithreading is explained in both the languages, the reason we selected the third language (JRuby) has been justified. The runtime behavior of three different algorithms in all the languages has been studied. Theoretically, we reached the conclusion as thread scheduling in Java and Ruby was explained.

|  |  |  |  |
| --- | --- | --- | --- |
| PARAMETERS | RUBY | JAVA | JRUBY |
|  |  |  |  |
| Parallelism | Does not | Supports | Supports |
|  | support | parallelism | parallelism |
|  | parallelism |  |  |
|  | because of |  |  |
|  | GIL |  |  |
|  |  |  |  |
| Concurrency | Implicitly | We have to | We must |
|  | concurrency | use locking | use locking |
|  | is achieved | mechanism | mechanism |
|  | by GIL |  |  |
|  |  |  |  |
| Simplicity | Ruby is | Little | Same as |
|  | much | complex | Ruby |
|  | simpler | when |  |
|  |  | compared |  |
|  |  | with Ruby |  |
|  |  |  |  |
| Portability | Can run on | Platform | Platform |
|  | any | independent | independent |
|  | platform | (as | (as |
|  |  | bytecode is | bytecode is |
|  |  | generated) | generated) |
|  |  |  |  |
| Cost- | Ruby is | Java is not | Cost- |
| effectiveness | cost- | cost- | effective |
|  | effective (it | effective | because |
|  | requires |  | code will be |
|  | less lines of |  | written in |
|  | code, less |  | Ruby |
|  | memory |  |  |
|  | etc.) |  |  |
|  |  |  |  |
| Extensibility | Not strictly | Extensible | Not strictly |
|  | extensible |  | extensible |
|  |  |  |  |

**CRITIQUE**

We have focused on the concept of thread scheduling and its implementation in Java and Ruby languages.

Initially as Ruby was purely object-oriented and dynamic typed we assumed that Ruby supports parallelism in a better way when compared with Java, but later when we further continued with the research we came across a new concept called GIL, which is responsible in restricting the feature of parallelism in Ruby and we came across the implementation of Ruby on JVM, implementation of Ruby on LLVM and other similar concepts. So, to implement parallelism in Ruby we moved to JVM instead of MRI. The Ruby implementation on JVM was quite similar to Java and we expected the threads to behave same as Java threads, after implementing some test cases we understood the reason for JRuby to be slower when compared with Java.

In the future, we would like to master the concept of multithreading in Ruby implementation on JVM, LLVM (Rubinius). We were unable to do it in this project because of the limited time constraint. We enjoyed working on implementation of parallel programs in Ruby as it was pretty fun, and we enjoyed studying the new language, but we would have done much better if we had focused mainly on JRuby rather than on Ruby. We also studied much more about the Ruby interpreter versions and also tried to implement some algorithms in Ruby 2.3 but because of the page limit, we were unable to add them to the report. Due to the limited resources available on JRuby, it consumed lot of time to gain proper knowledge on this concept.

We have done all the theoretical research required that has helped us in providing the conclusions which are presented in this paper.

CSCI 56500|Multithreading: Thread Scheduling 9

**REFERENCES**

1. [https://www.igvita.com/2008/11/13/concurren](https://www.igvita.com/2008/11/13/concurrency-is-a-myth-in-ruby/) [cy-is-a-myth-in-ruby/](https://www.igvita.com/2008/11/13/concurrency-is-a-myth-in-ruby/)
2. [https://en.wikipedia.org/wiki/Ruby\_(program](https://en.wikipedia.org/wiki/Ruby_(programming_language)#Syntax) [ming\_language)#Syntax](https://en.wikipedia.org/wiki/Ruby_(programming_language)#Syntax)
3. [https://csinaction.com/2014/10/10/multithreadi](https://csinaction.com/2014/10/10/multithreading-in-the-mri-ruby-interpreter/) [ng-in-the-mri-ruby-interpreter/](https://csinaction.com/2014/10/10/multithreading-in-the-mri-ruby-interpreter/)
4. [https://www.jstorimer.com/blogs/workingwith](https://www.jstorimer.com/blogs/workingwithcode/8085491-nobody-understands-the-gil) [code/8085491-nobody-understands-the-gil](https://www.jstorimer.com/blogs/workingwithcode/8085491-nobody-understands-the-gil)
5. [https://www.jstorimer.com/blogs/workingwith](https://www.jstorimer.com/blogs/workingwithcode/8100871-nobody-understands-the-gil-part-2-implementation) [code/8100871-nobody-understands-the-gil-part-2-implementation](https://www.jstorimer.com/blogs/workingwithcode/8100871-nobody-understands-the-gil-part-2-implementation)
6. <http://www.rubyinside.com/does-the-gil-make-your-ruby-code-thread-safe-6051.html>
7. [https://www.mutuallyhuman.com/blog/2012/1](https://www.mutuallyhuman.com/blog/2012/11/26/speeding-things-up-with-jruby/) [1/26/speeding-things-up-with-jruby/](https://www.mutuallyhuman.com/blog/2012/11/26/speeding-things-up-with-jruby/)
8. <https://www.toptal.com/ruby/the-many-shades-of-the-ruby-programming-language>
9. <https://en.wikipedia.org/wiki/Java_compiler>
10. [https://docs.oracle.com/javase/tutorial/essentia](https://docs.oracle.com/javase/tutorial/essential/concurrency/pools.html)

[l/concurrency/pools.html](https://docs.oracle.com/javase/tutorial/essential/concurrency/pools.html)

1. [http://web.cs.ucdavis.edu/~pandey/Teaching/E](http://web.cs.ucdavis.edu/~pandey/Teaching/ECS150/Lects/05scheduling.pdf)

[CS150/Lects/05scheduling.pdf](http://web.cs.ucdavis.edu/~pandey/Teaching/ECS150/Lects/05scheduling.pdf)

1. [https://computersciencesource.wordpress.com/](https://computersciencesource.wordpress.com/2009/11/21/year-2-operating-systems-process-thread-scheduling/) [2009/11/21/year-2-operating-systems-process-thread-scheduling/](https://computersciencesource.wordpress.com/2009/11/21/year-2-operating-systems-process-thread-scheduling/)
2. [https://en.wikipedia.org/wiki/Thread\_(computi](https://en.wikipedia.org/wiki/Thread_(computing)) [ng)](https://en.wikipedia.org/wiki/Thread_(computing))
3. [https://www.google.com/imgres?imgurl=https](https://www.google.com/imgres?imgurl=https%3A%2F%2Fwww.ntu.edu.sg%2Fhome%2Fehchua%2Fprogramming%2Fjava%2Fimages%2FMultithread.gif&imgrefurl=https%3A%2F%2Fwww.ntu.edu.sg%2Fhome%2Fehchua%2Fprogramming%2Fjava%2FJ5e_multithreading.html&docid=OX3_7bZKSHk5oM&tbnid=w4LE9m17C6lgpM%3A&vet=10ahUKEwjeyrignoHYAhUo4IMKHWN7CiMQMwhIKAkwCQ..i&w=698&h=219&client=firefox-b-1&bih=753&biw=1536&q=preemptive%20%20thread%20in%20java%20diagram&ved=0ahUKEwjeyrignoHYAhUo4IMKHWN7CiMQMwhIKAkwCQ&iact=mrc&uact=8)

[%3A%2F%2Fwww.ntu.edu.sg%2Fhome%2Fe](https://www.google.com/imgres?imgurl=https%3A%2F%2Fwww.ntu.edu.sg%2Fhome%2Fehchua%2Fprogramming%2Fjava%2Fimages%2FMultithread.gif&imgrefurl=https%3A%2F%2Fwww.ntu.edu.sg%2Fhome%2Fehchua%2Fprogramming%2Fjava%2FJ5e_multithreading.html&docid=OX3_7bZKSHk5oM&tbnid=w4LE9m17C6lgpM%3A&vet=10ahUKEwjeyrignoHYAhUo4IMKHWN7CiMQMwhIKAkwCQ..i&w=698&h=219&client=firefox-b-1&bih=753&biw=1536&q=preemptive%20%20thread%20in%20java%20diagram&ved=0ahUKEwjeyrignoHYAhUo4IMKHWN7CiMQMwhIKAkwCQ&iact=mrc&uact=8)

[hchua%2Fprogramming%2Fjava%2Fimages%](https://www.google.com/imgres?imgurl=https%3A%2F%2Fwww.ntu.edu.sg%2Fhome%2Fehchua%2Fprogramming%2Fjava%2Fimages%2FMultithread.gif&imgrefurl=https%3A%2F%2Fwww.ntu.edu.sg%2Fhome%2Fehchua%2Fprogramming%2Fjava%2FJ5e_multithreading.html&docid=OX3_7bZKSHk5oM&tbnid=w4LE9m17C6lgpM%3A&vet=10ahUKEwjeyrignoHYAhUo4IMKHWN7CiMQMwhIKAkwCQ..i&w=698&h=219&client=firefox-b-1&bih=753&biw=1536&q=preemptive%20%20thread%20in%20java%20diagram&ved=0ahUKEwjeyrignoHYAhUo4IMKHWN7CiMQMwhIKAkwCQ&iact=mrc&uact=8)

[2FMultithread.gif&imgrefurl=https%3A%2F](https://www.google.com/imgres?imgurl=https%3A%2F%2Fwww.ntu.edu.sg%2Fhome%2Fehchua%2Fprogramming%2Fjava%2Fimages%2FMultithread.gif&imgrefurl=https%3A%2F%2Fwww.ntu.edu.sg%2Fhome%2Fehchua%2Fprogramming%2Fjava%2FJ5e_multithreading.html&docid=OX3_7bZKSHk5oM&tbnid=w4LE9m17C6lgpM%3A&vet=10ahUKEwjeyrignoHYAhUo4IMKHWN7CiMQMwhIKAkwCQ..i&w=698&h=219&client=firefox-b-1&bih=753&biw=1536&q=preemptive%20%20thread%20in%20java%20diagram&ved=0ahUKEwjeyrignoHYAhUo4IMKHWN7CiMQMwhIKAkwCQ&iact=mrc&uact=8)

[%2Fwww.ntu.edu.sg%2Fhome%2Fehchua%2](https://www.google.com/imgres?imgurl=https%3A%2F%2Fwww.ntu.edu.sg%2Fhome%2Fehchua%2Fprogramming%2Fjava%2Fimages%2FMultithread.gif&imgrefurl=https%3A%2F%2Fwww.ntu.edu.sg%2Fhome%2Fehchua%2Fprogramming%2Fjava%2FJ5e_multithreading.html&docid=OX3_7bZKSHk5oM&tbnid=w4LE9m17C6lgpM%3A&vet=10ahUKEwjeyrignoHYAhUo4IMKHWN7CiMQMwhIKAkwCQ..i&w=698&h=219&client=firefox-b-1&bih=753&biw=1536&q=preemptive%20%20thread%20in%20java%20diagram&ved=0ahUKEwjeyrignoHYAhUo4IMKHWN7CiMQMwhIKAkwCQ&iact=mrc&uact=8) [Fprogramming%2Fjava%2FJ5e\_multithreadin](https://www.google.com/imgres?imgurl=https%3A%2F%2Fwww.ntu.edu.sg%2Fhome%2Fehchua%2Fprogramming%2Fjava%2Fimages%2FMultithread.gif&imgrefurl=https%3A%2F%2Fwww.ntu.edu.sg%2Fhome%2Fehchua%2Fprogramming%2Fjava%2FJ5e_multithreading.html&docid=OX3_7bZKSHk5oM&tbnid=w4LE9m17C6lgpM%3A&vet=10ahUKEwjeyrignoHYAhUo4IMKHWN7CiMQMwhIKAkwCQ..i&w=698&h=219&client=firefox-b-1&bih=753&biw=1536&q=preemptive%20%20thread%20in%20java%20diagram&ved=0ahUKEwjeyrignoHYAhUo4IMKHWN7CiMQMwhIKAkwCQ&iact=mrc&uact=8) [g.html&docid=OX3\_7bZKSHk5oM&tbnid=w](https://www.google.com/imgres?imgurl=https%3A%2F%2Fwww.ntu.edu.sg%2Fhome%2Fehchua%2Fprogramming%2Fjava%2Fimages%2FMultithread.gif&imgrefurl=https%3A%2F%2Fwww.ntu.edu.sg%2Fhome%2Fehchua%2Fprogramming%2Fjava%2FJ5e_multithreading.html&docid=OX3_7bZKSHk5oM&tbnid=w4LE9m17C6lgpM%3A&vet=10ahUKEwjeyrignoHYAhUo4IMKHWN7CiMQMwhIKAkwCQ..i&w=698&h=219&client=firefox-b-1&bih=753&biw=1536&q=preemptive%20%20thread%20in%20java%20diagram&ved=0ahUKEwjeyrignoHYAhUo4IMKHWN7CiMQMwhIKAkwCQ&iact=mrc&uact=8) [4LE9m17C6lgpM%3A&vet=10ahUKEwjeyri](https://www.google.com/imgres?imgurl=https%3A%2F%2Fwww.ntu.edu.sg%2Fhome%2Fehchua%2Fprogramming%2Fjava%2Fimages%2FMultithread.gif&imgrefurl=https%3A%2F%2Fwww.ntu.edu.sg%2Fhome%2Fehchua%2Fprogramming%2Fjava%2FJ5e_multithreading.html&docid=OX3_7bZKSHk5oM&tbnid=w4LE9m17C6lgpM%3A&vet=10ahUKEwjeyrignoHYAhUo4IMKHWN7CiMQMwhIKAkwCQ..i&w=698&h=219&client=firefox-b-1&bih=753&biw=1536&q=preemptive%20%20thread%20in%20java%20diagram&ved=0ahUKEwjeyrignoHYAhUo4IMKHWN7CiMQMwhIKAkwCQ&iact=mrc&uact=8) [gnoHYAhUo4IMKHWN7CiMQMwhIKAkw](https://www.google.com/imgres?imgurl=https%3A%2F%2Fwww.ntu.edu.sg%2Fhome%2Fehchua%2Fprogramming%2Fjava%2Fimages%2FMultithread.gif&imgrefurl=https%3A%2F%2Fwww.ntu.edu.sg%2Fhome%2Fehchua%2Fprogramming%2Fjava%2FJ5e_multithreading.html&docid=OX3_7bZKSHk5oM&tbnid=w4LE9m17C6lgpM%3A&vet=10ahUKEwjeyrignoHYAhUo4IMKHWN7CiMQMwhIKAkwCQ..i&w=698&h=219&client=firefox-b-1&bih=753&biw=1536&q=preemptive%20%20thread%20in%20java%20diagram&ved=0ahUKEwjeyrignoHYAhUo4IMKHWN7CiMQMwhIKAkwCQ&iact=mrc&uact=8) [CQ..i&w=698&h=219&client=firefox-b-1&bih=753&biw=1536&q=preemptive%20%](https://www.google.com/imgres?imgurl=https%3A%2F%2Fwww.ntu.edu.sg%2Fhome%2Fehchua%2Fprogramming%2Fjava%2Fimages%2FMultithread.gif&imgrefurl=https%3A%2F%2Fwww.ntu.edu.sg%2Fhome%2Fehchua%2Fprogramming%2Fjava%2FJ5e_multithreading.html&docid=OX3_7bZKSHk5oM&tbnid=w4LE9m17C6lgpM%3A&vet=10ahUKEwjeyrignoHYAhUo4IMKHWN7CiMQMwhIKAkwCQ..i&w=698&h=219&client=firefox-b-1&bih=753&biw=1536&q=preemptive%20%20thread%20in%20java%20diagram&ved=0ahUKEwjeyrignoHYAhUo4IMKHWN7CiMQMwhIKAkwCQ&iact=mrc&uact=8) [20thread%20in%20java%20diagram&ved=0a](https://www.google.com/imgres?imgurl=https%3A%2F%2Fwww.ntu.edu.sg%2Fhome%2Fehchua%2Fprogramming%2Fjava%2Fimages%2FMultithread.gif&imgrefurl=https%3A%2F%2Fwww.ntu.edu.sg%2Fhome%2Fehchua%2Fprogramming%2Fjava%2FJ5e_multithreading.html&docid=OX3_7bZKSHk5oM&tbnid=w4LE9m17C6lgpM%3A&vet=10ahUKEwjeyrignoHYAhUo4IMKHWN7CiMQMwhIKAkwCQ..i&w=698&h=219&client=firefox-b-1&bih=753&biw=1536&q=preemptive%20%20thread%20in%20java%20diagram&ved=0ahUKEwjeyrignoHYAhUo4IMKHWN7CiMQMwhIKAkwCQ&iact=mrc&uact=8) [hUKEwjeyrignoHYAhUo4IMKHWN7CiMQ](https://www.google.com/imgres?imgurl=https%3A%2F%2Fwww.ntu.edu.sg%2Fhome%2Fehchua%2Fprogramming%2Fjava%2Fimages%2FMultithread.gif&imgrefurl=https%3A%2F%2Fwww.ntu.edu.sg%2Fhome%2Fehchua%2Fprogramming%2Fjava%2FJ5e_multithreading.html&docid=OX3_7bZKSHk5oM&tbnid=w4LE9m17C6lgpM%3A&vet=10ahUKEwjeyrignoHYAhUo4IMKHWN7CiMQMwhIKAkwCQ..i&w=698&h=219&client=firefox-b-1&bih=753&biw=1536&q=preemptive%20%20thread%20in%20java%20diagram&ved=0ahUKEwjeyrignoHYAhUo4IMKHWN7CiMQMwhIKAkwCQ&iact=mrc&uact=8) [MwhIKAkwCQ&iact=mrc&uact=8](https://www.google.com/imgres?imgurl=https%3A%2F%2Fwww.ntu.edu.sg%2Fhome%2Fehchua%2Fprogramming%2Fjava%2Fimages%2FMultithread.gif&imgrefurl=https%3A%2F%2Fwww.ntu.edu.sg%2Fhome%2Fehchua%2Fprogramming%2Fjava%2FJ5e_multithreading.html&docid=OX3_7bZKSHk5oM&tbnid=w4LE9m17C6lgpM%3A&vet=10ahUKEwjeyrignoHYAhUo4IMKHWN7CiMQMwhIKAkwCQ..i&w=698&h=219&client=firefox-b-1&bih=753&biw=1536&q=preemptive%20%20thread%20in%20java%20diagram&ved=0ahUKEwjeyrignoHYAhUo4IMKHWN7CiMQMwhIKAkwCQ&iact=mrc&uact=8)

1. [https://github.com/jruby/jruby/wiki/Concurren](https://github.com/jruby/jruby/wiki/Concurrency-in-jruby#thread_safety) [cy-in-jruby#thread\_safety](https://github.com/jruby/jruby/wiki/Concurrency-in-jruby#thread_safety)
2. [http://www.restlessprogrammer.com/2013/02/](http://www.restlessprogrammer.com/2013/02/multi-threading-in-jruby.html) [multi-threading-in-jruby.html](http://www.restlessprogrammer.com/2013/02/multi-threading-in-jruby.html)
3. [https://rosettacode.org/wiki/Sorting\_algorithm](https://rosettacode.org/wiki/Sorting_algorithms/Merge_sort#Ruby) [s/Merge\_sort#Ruby](https://rosettacode.org/wiki/Sorting_algorithms/Merge_sort#Ruby)
4. [http://www.cs.trinity.edu/~bmassing/Classes/C](http://www.cs.trinity.edu/~bmassing/Classes/CS3366_2015fall/SamplePrograms/Java/csci3366sample/mergesort/MergeSortPar.java) [S3366\_2015fall/SamplePrograms/Java/csci33](http://www.cs.trinity.edu/~bmassing/Classes/CS3366_2015fall/SamplePrograms/Java/csci3366sample/mergesort/MergeSortPar.java) [66sample/mergesort/MergeSortPar.java](http://www.cs.trinity.edu/~bmassing/Classes/CS3366_2015fall/SamplePrograms/Java/csci3366sample/mergesort/MergeSortPar.java)

CSCI 56500|Multithreading: Thread Scheduling 10