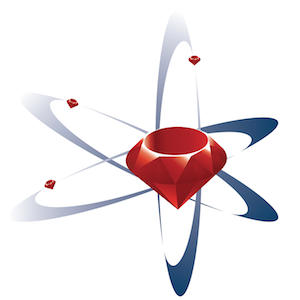
Language Features **Concurrency in Ruby**

****

**COS40003 – Concurrent Programming**

**Semester 1 2015**

**Reuben Wilson**

**9988289**

**Bachelor of Professional Software Development**

1. Overview

Ruby is an open source, object-oriented programming language, which saw its initial release in the year of 1995. Ruby has seen continual development over it’s twenty-year existence from the open source community, which has made it a highly supported development language.

There are many libraries that are openly available that offer functionality for an array of different programming concepts. One such library is Ruby’s official, community developed concurrency library. Ruby’s concurrent library aims to offer modern concurrency tools to aid in the implementation of concurrency in programming, much like the utilities offered in the concurrency library developed during the course of COS40003 Concurrent Programming.

The aim of this report is to explore the concurrent utilities provided within Ruby’s concurrent library and outline comparisons based on implementation and functional similarities and how both libraries are geared towards providing functional, thread safe and lean utilities that support concurrency in programming.

1. Concurrent Utilities in Ruby’s Concurrent Library
   1. Atomic Variables

Thread safe variables are provided in Ruby’s concurrent library. They are objects that represent a single value that provides guaranteed thread safety when it comes to reading and writing to the value that a thread safe variable represents. No explicit locking is required while accessing the value that a thread safe variable represents, rather, read and write permissions to the value are controlled with a concurrent Ruby Mutex object.

* + 1. Atomic variables in Concurrent Programming library

The concurrent programming library does not provide support for thread safe variables. During development of concurrent programs using the concurrent programming library, any values that were not considered as thread safe and could potentially be exposed to race conditions were explicitly locked. Thread safe variables could be implemented to eliminate the use of explicit locking by providing a generic thread safe variable template. Access to the value that the variable represents would be controlled by a Mutex through the implementation of a turnstile system. A generic template for a thread safe variable would ensure that it could be instantiated with a value of any type.

* 1. Semaphore & Mutex

Provided within the Ruby concurrent library are both the Semaphore and Mutex concurrency utilities. Mutex is the base class and Semaphore inherits from Mutex. The Mutex utility provides as a thread activity permission mechanism that implements a permit system. A single thread that relies upon the Mutex for activity permission will need to acquire a permit from the Mutex in order to commence with activity. Mutex implements an acquiring system where a thread can be told to wait infinitely to acquire or it can be told the maximum time it should wait in milliseconds to acquire from the Mutex. If the Mutex does not have any available permits, the thread will be blocked until such time as the Mutex has an available permit to offer. Permits may be released into the Mutex to allow blocked threads the ability to attempt to acquire a permit. A Mutex may be asked to return the number of permits that it possesses as well as being told to reduce the number of permits that it has available by an indicated value. Another functional aspect of the Mutex is that it may be told to drain all of its permits, which sets the number of available permits in the Mutex to zero.

* + 1. Concurrent Programming: Semaphore & Mutex

The concurrent programming library also offers both the Semaphore and Mutex concurrency utilities.

* + 1. Functional Similarities & Differences

The Semaphore and Mutex concurrency utilities offered in both libraries function in an almost identical fashion in order to ensure the implementation of a thread activity permission mechanism. The key differences are that, instead of Semaphore inheriting from Mutex as it is defined in the Ruby concurrent library, in the concurrent programming library, the Mutex inherits from the Semaphore. This aside, the likeness of the two stems from the fact that a Mutex, as defined in both libraries, may only ever contain a maximum of one permit, or token as they are called in the concurrent programming implementation. A Semaphore object in both libraries may contain more than a maximum of one token at any given time. Acquiring and releasing in both are performed in identical fashion, although concurrent Ruby’s utilities do not have a force release method that may be invoked on them.

Ruby’s concurrent Mutex and Semaphore may be asked for the number of permits that they possess, as well as being able to be told to reduce their permits by an indicated value or drain all of their permits. Concurrent programming’s Semaphore and Mutex contain no such functionality, but it could be implemented if required.

* 1. Read Write Lock

Concurrent Ruby offers a ReadWriteLock utility that provides for controlled read and write thread access. The implementation dictates that any number of readers may be reading at a given point in time while only one writer may be writing. A writer may not write if readers are reading; it must wait for all readers to complete reading before it can attempt to write. Similarly, readers may not read if a single writer is writing; they must wait for the writer to finish writing before they may attempt to commence reading. A concurrent Ruby ReadWriteLock can be acquired by a reader and released by a reader. Both methods of interaction return a Boolean, representing whether or not the operation was completed successfully. The same kind of interactions may occur for a writer, where a writer can acquire and release the writer lock. The utility can also be asked if it has any threads waiting on it, i.e. waiting to acquire from the ReadWriteLock. Additionally, it may be asked if a writer currently posses the writer lock. Finally, the Ruby ReadWriteLock can be told to execute code blocks while acquiring either the read or write lock.

* + 1. Concurrent Programming: ReaderWriterLock

The concurrent programming library includes a concurrent utility that functions in much the same way as the Concurrent Ruby’s ReadWriteLock; it is called the ReaderWriterLock.

* + 1. Functional Similarities & Differences

The ReaderWriterLock from the concurrent programming library implements the ability to acquire and release the reader lock as well as acquire and release the writer lock. If readers are reading, a writer may not acquire until the reader lock is released, at which point it may attempt to writer. Similarly, readers may not read if a writer is writing and may only attempt to read once the writer has released the writer lock. ReaderWriterLock does not support being queried in regards to whether it has threads waiting on it to acquire or whether the writer lock has been acquired, like Ruby’s ReadWriteLock does. ReaderWriterLock does not support the ability to be passed blocks of code to be executed under either the read or write lock. It is important to note that in both implementations, the write lock must not be acquired if you are already holding the read lock or the read lock must not be acquired if you already hold the write lock, otherwise a deadlock will occur.

* 1. WaitableList

Concurrent Ruby implements a thread safe list called WaitableList. An instance of WaitableList functions in a fashion that you would expect, it may have items removed from it as well as have items added to it. It may be asked for a size representation of the number of elements in it and asked whether or not it is empty. All of these operations are performed with thread safety through the implementation of a Mutex turnstile system.

* + 1. Concurrent Programming: List

The concurrent programming library offers a thread safe List which can be manipulated concurrently.

* + 1. Functional Similarities & Differences

Originally, there was no implementation of a thread safe list in the concurrent programming library. Instead, a queue object implemented by .NET’s collections library was utilised and thread safety was ensured by locking the queue every time it were to be de-queued or en-queued. It was discovered that performance could be increased by implementing lock splitting in a custom, thread safe list. Concurrent programming’s thread safe list differs from Ruby’s concurrent list in the respect that instead of using a Mutex turnstile system, lock splitting is implemented and distributed across the top and bottom nodes in the list.

* 1. BufferedChannel & UnBufferedChannel

Concurrent Ruby offers thread safe message passing pipelines through the implementation of BufferedChannel and UnBufferedChannel. The underlying list that they represent is a WaitableList (as previously defined). Both BufferedChannel and UnBufferedChannel may have data pushed to their list as well as being able to pop data from their list. Each implement the functionality to have their list replaced by a new list object (WaitableList) or removed. The difference between the two is that BufferedChannel works on a set list size (size of the WaitableList) where as UnBufferedChannel has no size restrictions. It is interesting to note that BufferedChannel does not inherit from UnBufferedChannel as would be expected due to the functional nature of both.

* + 1. Concurrent Programming: Channel & BoundedChannel

The concurrent programming library includes concurrent utilities called Channel and BoundedChannel that provide for similar functionality to Ruby’s concurrent BufferedChannel and UnBufferedChannel.

* + 1. Functional Similarities & Differences

Channel and BoundedChannel may both have data en-queued and de-queued from them. Their underlying list object is an instance of concurrent programming’s thread safe list. The differences are highlighted by the fact that Channel and BoundedChannel may not have their underlying list objects instantiated to new list objects; they are set to the list objects that they represent upon initialisation. Also worth noting is that Channel and BoundelChannel implement timeouts, where a single thread can be instructed to wait a maximum amount of time to en-queue or de-queue before continuing on with another task. Based on Ruby’s concurrent documentation, it was not evident that this kind of functionality was present within their utilities. BoundedChannel inherits from Channel, as it functions in the same manner except for being restricted to a maximum list size. Ruby’s concurrent channels do not share inheritance and are implemented as separate utilities.

* 1. Agents

Agents are a concurrent utility that can be identified by a single atomic value that they are assigned upon instantiation. Agents operate within a thread pool and receive working requests through a queueing system. The queueing system is either a BufferedChannel or UnBufferedChannel.

Functions can be posted to an agent’s working queue. The agent will execute the instructions on its queue, using the identity value that it was instantiated with as the parameter value for the function that it processes. Upon completing a posted function, the identity value that the agent represents is reassigned to the value that the function returns. For reasons of data validation, when instantiating an agent, there is an option to request that the new agent object contains a validation value. The validation value is passed to the agent with the posted instructions and represents what the resulting value of the posted instructions should be.

As the agent implements a working queue system, the functions posted to the agent’s queue are executed in a first in first out manner.

* + 1. ConcurrentProgramming: ActiveObject & ChannelBasedActiveObject

ActiveObject is an abstract class, that is designed to control the lifecycle and operation of a single thread, existing on the thread pool. ActiveObjects can be inherited from, in order to implement functional execution of instructions that are required by the thread that the ActiveObject represents. ActiveObject implements a single, abstract run method, in which overriding classes specify the instructions that the ActiveObject’s thread will execute.

ChannelBasedActiveObject implements a layer of functionality on top of ActiveObject, which sees for the inclusion of a working queue (Channel or BoundedChannel). Data can be posted to a ChannelBasedActiveObject’s queuing system. ChannelBasedActiveObject may be overridden in order to implement the functional requirements for the manner in which the data that it receives is processed. The ChannelBasedActiveObject processes the data on its queue in the order in which it is received.

* + 1. Functional Similarities & Differences

The agent utility offered within Ruby’s concurrent library implements similar functionality to ChannelBasedActiveObject. An agent is passed the instructions that it is required to execute, and upon completion, the Agent may be asked for it’s current working value. ChannelBasedActiveObject implements a system where, instead of passing it the instructions, it is overridden to implement functionality suitable to process the data that is passed. The similarities stem from the queueing system that both Agent and ChannelBasedActiveObject implement in order to receive instructions to process.