ECE 421: ASSIGNMENT 3 - A Multi-Threaded Sort

By: Reem Maarouf, Calvin Ho, and Aaron Philips

- What is your definition of an object?
    - Objects represent classes created in ruby. Everything in ruby is essentially considered an object. Objects have their own type and contain methods that reflect their behaviours. An object has its own state that represents the identity of an object at a given time. Objects of the same type can contain different information from each other. For example, two string objects can have different characters.
- What strategies should be deployed in terms of accepting input (i.e. the large number of objects.)?
    - We will be iterating through the command line arguments that are passed into the multithreaded sort. A custom iterator should be made to loop through the objects passed in.
- Is your sort generic? What sorting criterion does your system use? Is this criterion flexible; i.e. changeable by the user at the point of execution? What limitations have you placed upon legal sorting criteria? To make it reusable to other people, how should we include it into the Ruby Class hierarchy?
    - Yes, the sorting algorithm will be generic to ensure all objects can be passed in and sorted. The system is would be reusable and robust. The user would be able to specify a body of how to sort the objects (ascending or descending). The objects to be sorting will be sorted with the comparable module. Some of the limitations of our sorting program would be that it must sort the same time of object. It will be generic but will need to sort based on the same type. We will be using parallel merge sort. To make it reusable, we will be overriding a class array.
- In reality we have a uniprocessor system, describe “equationally” what is happening to your solution as you increase the concurrency (e.g. produce a regression model (remember regression from Statistics) of processing time against the number of threads. Your solution can be modeled as a program which: (1) has a component which produces threads; and (2) a set of threads which undertake the same (small) task. This is in essence the basis of stress testing (discussed in ECE 322)
    - In a uniprocessor system, concurrency increases when more threads are introduced to the system. The time required to run a multithreaded program would therefore increase. This is because all the processes require the same shared resource (processor). A thread would be blocked until the resource frees up. As more threads are introduced to the system they would be blocked for a longer amount of time. The time would increase linearly when more threads are used. The equation that would demonstrate this is a linear line equation: $Y = mx + b$.

- Concurrent systems tend to crash frequently – what approach to exception-handling have you devised? Consider the content of the library at: http://c2.com/cgi/wiki?ExceptionPatterns; which are applicable to this problem? Is Module Errno useful in this problem? What components of the Ruby exception hierarchy are applicable to this problem? Discuss in detail your strategy for exception-handling.
    - A couple of approaches are applicable for this problem from the exception patterns website. One of the exception patterns that will be used in this problem is ExceptionsTidyThread. If a thread dies or gets interrupted, the exception handler will deallocate its space in memory and free up the resources that its been using. Another exception pattern that can be used in this problem is the Security Door Pattern. The Security Door Pattern ensures that only one thread will be accesses and run at a given time. Only authorized threads will be able to access the resources. Handling interrupts exceptions will also be useful for this application. If a thread is interrupted from an external source, an interrupt exception will be raised and will automatically terminate the program. The user can then input objects again to sort.
- What differences exist between thread-based and process-based solutions? How has this impacted the design of your solution?
    - Thread-based uses less resources than process-based. Threads uses less expensive methods to communicate, shares address spaces, and uses less overhead. Process-based is better for larger solutions where it is the main program that is being ran. While threads are better when a solution is part of a larger program.
- Do you have any race-condition or task synchronization concerns about your solution? How do we tidy-up a multi-threaded program, if stopped mid-execution?
    - If we decide to go with a merge sort, there might be race-conditions or task synchronization. One thread could be sorting two pairs and finish before its sister thread is done with their threads. Resulting in their parent thread to sort them before both are done. To prevent this, the parent thread needs to wait for both threads to finish sorting before it goes. To clean up the system. We might join all the threads together and terminate the top thread.
- As discussed in CMPUT 274/5: What is configuration management? What is version control? Are you using either concept? If “yes”, describe your process and any tool support what you utilize – illustrate your process with regard to Assignments 1 and 2; if “no”, justify your decision.
    - Configuration management is the processes of saving and tracking information in the program. Version control is considered a configuration management technique. We use GIT to track and control the revisions in our software. We found this to be very useful, as we can all modify and work on the code on our own branches. In Assignment 1 and 2, we also used GIT for version control. Each member on the group would contribute to the code and would be pushed on to github. Eventually, we merged our work together and tested it all together. If

an error occurs, we would revert back to a previous commit without losing any information or code.

- Briefly Explain:
  - a. What is refactoring (as discussed in CMPUT 301)?
    - Refactoring refers to the restructuring of code designs into more maintainable designs. The restructuring of code does not change its behaviour or results. It is to ensure that the code looks cleaner and clearer to the developer. Refactoring code is also done to avoid redundancy of source code in the program.
  - b. Are you using refactoring in your development process? Justify your answer?
    - Yes, we will be using refactoring in the development process. We will ensure that refracting patterns will be present in the code to avoid consistency and readable code.
  - c. If "yes", give examples, minimum of 2, of the refactoring "patterns" that you used in Assignment 1
    - One of the refactoring methods that was used in assignment one is the Long Method. We avoided using long methods in the throughout the code to ensure that each method has a clear task. We extracted code from the long methods and created new smaller methods. Another refactoring method that was used in Assignment 1 is Hide Delegate pattern. We created delegating methods in our NDimentionalMatrix class that delegates its methods to Ruby's Matrix class.
  - If "no", give examples of where your solution to Assignment 1 would be improved by applying refactoring patterns. Supply a minimum of two different (i.e. different refactoring patterns) as examples
    - Some refactoring patterns that could've been applied are the Large Class refactoring pattern. Our class was fairly large and could've been separated into smaller classes. We also had slightly high coupling in the code. This could have also mean separated to two or more classes in order to decrease coupling.

Contract:

```
Def sort_postcondition()
        Assert checksum
        Assert same size
        End

Def copy_postcondition()
        Assert arrays same size
        End

Def partition_postcondition()
        Assert split happened in right spot
        Assert right size == left size or right size == left size +1
        End

Def thread_sort_postcondition()
        Assert both child threads done
        End

Def merge_postcondition()
        Assert size = sum of merge array size
        End
```

Precondition:

```
Def sort_precondition()
        Assert array is an array
        End

Def copy_precondition()
        Assert array is an array
        End

Def partition_precondition()
        Assert array size > 1
        End

Def thread_sort_precondition()
        Assert array size > 1
        End
```

```
Def merge_precondition()
        Assert both arrays being merged are arrays
        End
```

Invariants:

```
def invariants()
        Size >= 0
   end
```