ASSIGNMENT 3 PART 3 REPORT

By: Reem Maarouf, Aaron Philips, Calvin Ho

Changes Made to proposed design:

a) Users are able to input a large amount of arguments through either command line or in the driver program. A random generator in the driver program is also implemented. This allows the user to determine how many random integers to generate and sort. The user is also allowed to input any object as an argument, as long as they provide the comparator for the object for sorting. The large amount of objects that need to be sorted must be placed in an Array.

b) We use the join method to handle race conditions. The main thread will not continue executing until the merge threads are done executing. The parent thread will wait until all other threads, besides the timertask thread, are done.

c) Various refactoring patterns are used in this assignment. The long method pattern was originally used in the mergesort. We used the long method refactoring pattern to split the original long method in two smaller and more meaningful functions. Instead of creating a new class Array, we refined the Array class in Ruby to ensure that we reuse existing code.

d) The list of objects that are to be sorted can be a list of any objects. They need not be the same object. The default comparator will compare the values based on their ASCII values.

Deviations in contract:

A. We took out invariants as it is designed as a module instead of a class

B. Changed sort function to msort.
Added pre_condtion that array can get size and timeout is a positive integer
Post_condition checks size is consistent, timeout thread is dead and time elapsed is less than given timeout time

C. Removed copy function

D. Changed partition function to split
Pre_condition now checks if array passed can get size
post _condition now just checks size of result instead of checksum if list is not just integers

E. Thread_sort changed to merge_sort
Pre_condition checks if array can call size and block passed can be called
Post_condition checks size is consistent and that children threads are done/killed

F. Add merge function. Puts two arrays in order
Pre_condtion left array and right array can get size
Post_condition checks if size is consistent

G. Added run concurrently. Creates a thread to merge sort a part of the array
Pre_condition checks if passed array can get size and block is callable
Post_condition checks if thread created is alive and can be killed

H. Added post_condition to when timeout occurs
   Checks if the timeout thread is alive

Copy of Code:
A copy of the code is provided with this email.

Additional Tests:
Tested with list of integers generated by a random number generator
Tested with a list of up to 1000 numbers
Tested with a list of a string list of numbers and characters
Tested if it can't take in negative or non integers for timeout time

Missing Functionality and Errors:

a) The Timer in the mergesort does not stop and terminate the program if the duration limit is reached. Instead, it would continue sorting and then raise an timeout exception.
b) We were planning on making a thread pool to ensure that only a limited amount of threads can be executed at once. Due to stress testing, we realized that the thread count increases indefinitely. Although the assignment assumes that we have unlimited amount of resources, we realize that this is not practical in a real system. We have reached a point where the thread count is increased tremendously, even though there is no need for that large amount.