Yilin Yang

CSC345

Project 2 Report

Program requirements (100 pts):

- *[40 pts] 10 points each for correct implementation of each of the four requested tasks. To demonstrate this, your program should offer a command line option such as*

    $ ./main single 1

that runs the application in single-thread mode, and performs the first task. This way, you can show that your implementation of each task is correct, regardless of multi-thread part.

For this project, all of the four tasks are contained within their own functions. Individual functions may be run separately by entering "single 1", "single 2", etc., when executing the program through the command line. Depending on which number is entered, the corresponding task is performed, allowing for verification of each task. Further elaboration on how the tasks were implemented can be found further below.

- *[10 pts] Your program should be able to yield the same result in multi-thread mode.*

Running the program through the use of any of the commands detailed further below should yield the same results as those outputted by the single-thread mode mentioned above.

- *[10 pts] Support different scheduling algorithm for the worker threads. Pthread library supports two options: FIFO and RR. Provide a command line option to demonstrate this such as:* ***$ ./main RR*** *or* ***$ ./main FIFO****. Your program must provide a measure (e.g., total elapsed time), at the end, to see the difference between scheduling schemes.*

The keywords "RR" or "FIFO" may be specified when executing the program through the command line to specify the scheduling policy. This is accomplished through the use of pthread_attr_set_schedpolicy.

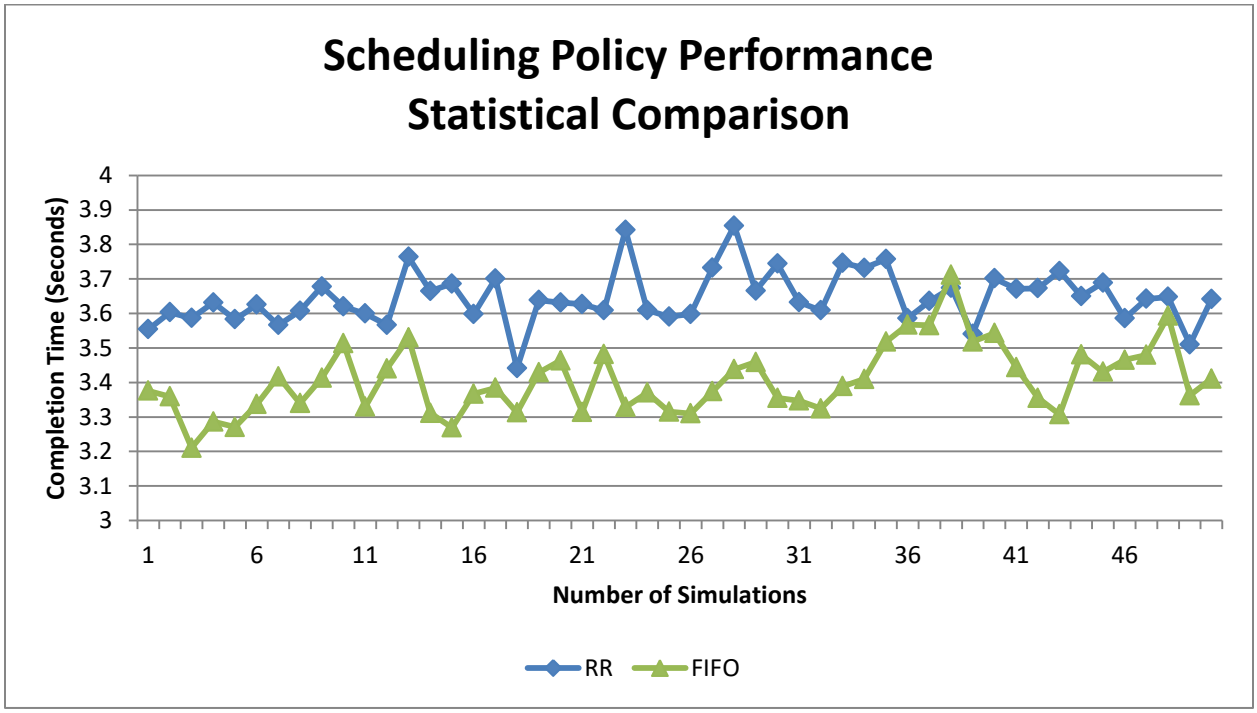
• *[10 pts] Support priority setting for your program. User should be able to set the priority of one task higher or lower than the other. For example,*

$ ./main priority 1 low

will set the priority of task 1 lower than the other, thus in the end, user should see that task 1 completed after the other three were finished. [high] option for opposite case.

Priority of specific tasks can be specified exactly like the example shown above. A series of if statements check to see which task is specified and whether to raise or lower the urgency of the task's completion. This is accomplished through the use of pthread_attr_setschedparam. The FIFO scheduling policy is used by default for this.

• *[10 pts] Devise a statistical experiment that compares RR and FIFO. For example, run 50 independent runs, average out the total run time, and provide analytical conclusion of your experiments. Graphical plots describing the results would be expected.*



FIFO		
Min: 3.2103	Max: 3.7117	Avg: 3.4066
RR		
Min: 3.4409	Max: 3.8540	Avg: 3.6452

In this experiment, 100 trials were performed in total, 50 consisting of the RR scheduling policy and 50 consisting of the FIFO scheduling policy. For each trial, the four tasks were executed using the scheduling policy dictated and the overall completion time was recorded. These data points were then plotted into the line graph shown above. For both scheduling policies, the average total time was calculated, which can be seen displayed in the table above along with minimum and maximum time recorded. From these results, it can be seen that FIFO demonstrates a faster execution time compared to RR by a margin of approximately 0.2386 seconds. This may be attributed to the implementation of the scheduling policies. RR policy must allot miniscule, yet noticeable, time to switch between each task once the quantum is reached whereas FIFO simply performs the task to completion. Over the course of numerous trials, this difference is compounded and becomes distinctly observable from the standard negligible variation in performance. As seen from the graph, there are isolated occurrences where RR matched or even surpassed FIFO in execution time. However, from the cumulative experiments performed, the final average suggests that these cases are exceptional rather than typical.

- *[20 pts] Resolve synchronization issue explained in the attached pages.*

Results of the tasks may be output to the console or to an “output.txt” file. Console printing is by default, however a user can specify to output to a text file by entering “file” on the command line when executing the program. Doing so triggers the modification of a condition variable. If statements consult this variable to determine the output method. Regardless of the output method, synchronization is maintained through the use of semaphores to lock the printing process. This prevents tasks from printing their reports if another task is already in the middle of doing so. The only process that is locked is the printing process; tasks do not halt one another when performing their jobs.

Task Description Details

Your program should be implemented in C (not C++) language, run on Linux machines, and should use pthread (POSIX thread) library.

Given 84 data files in CSV format (comma separated data; each line contains a record in the dataset and a record contains multiple attributes of the record separated by comma), your tasks are like following:

The 84 data files are saved in a directory named “analcatdata”. This directory is saved in the same location as the executable file for this project. The contents of the analcatdata folder are exactly as extracted from the zip file provided on canvas; this includes miscellaneous files such as README. Each tasks searches the analcatdata folder for data files, ignoring files that are not .csv types. Files that are .csv types have their names stored into an array of strings. Their contents are then read and interpreted based on the specific task.

- *T1: Find total number of unique words among all non-numeric string literals in each file*

The task1 function identifies non-numeric strings by attempting to perform a string to integer conversion. If the conversion fails, presumably the string is not comprised of numbers and therefore stored into an array holding words. A while loop examines all data points in the file performing this check to collect all non-numeric strings in the .csv file. A series of nested for loops compares each element in the words array with every other element in the array to identify unique words, incrementing a counter when a new word is found. This counter is not incremented if duplicates of a word are found, thereby counting all unique words found.

- *T2: Find maximum, minimum, average, and variance of numbers in each file*

The task2 function identifies numbers similarly to task1 by attempting to perform a string to integer conversion. If the conversion does not fail, then the string consists of numbers and a series of calculations are performed to determine the maximum, minimum, average, and variance of the numerical contents of the .csv file. Four arrays are declared for each statistic, one index

per file. Three double variables are also declared: max, min, and sum (variance can be calculated through existing variables). atof conversions turn all identified numbers from strings into doubles. Both max and min are set equal to the first double identified. An if statement checks whether every double afterwards is greater/lesser than the current max/min variable and if so, sets the variable equal to the greater/lesser double. The sum variable adds all numerical data points together. Two counters track the number of rows and columns in the .csv file. The number of rows multiplied by columns yields the total number of data points, which is used to divide the sum to find the average. As a result of this method, NULL values are accounted for in the calculation, effectively treated as zeroes in the summation. After reading the contents of the file, the max, min, and sum variables are stored into their respective arrays. A for loop calculates the variance, the result of which is also stored into an array.

- *T3: Find maximum, minimum, average of the numbers of rows and columns of all files*

Similar to what was described in the Task 2 implementation, the task3 function has several counter variables counting the number of rows and columns found within the file. Two integers, one for maximum and minimum, and one double, for average, are declared for rows and columns each for a total of six variables. Both max and min are set equal to the first number of rows or columns counted. An if statement checks whether every number afterwards is greater/lesser than the current max/min variable and if so, sets the variable equal to the greater/lesser number. The total variable adds all counted rows/columns and is then divided by the number of files, tracked by a separate counter, to determine the average.

- *T4: Count ratio of missing or zero values in each file*

Like the implementation for Task 1 and Task 2, the task4 function uses a string to integer function to perform its job. A counter variable is declared to count the number of nonzero elements (i.e. not zero or NULL). If the string to integer conversion fails, the data point is clearly not zero, therefore the counter is incremented. If the conversion does succeed, another check is

performed to determine whether the converted value is equal to zero or not, and the counter variable is incremented appropriately. The counter logically tallies all strings and nonzero numbers. Zero values are ignored and NULL values are skipped by the parsing procedure when identifying data points. Like Task 3, counter variables were also included to count the number of rows and columns. The number of rows multiplied by the number of columns yields the total number of data points, including NULL values. Subtracting the number of nonzero values and dividing by the total number of data points yields the ratio of zero or missing values.