The German University in Cairo

Operating Systems

Threads and Scheduling

A report submitted by:

Names of students:

1. Abdelrahman Mohamed Gamal 55-3901 T - 17
2. Mohamed Amr Shaker 55-0903 T - 14
3. Ahmed Amr Aboelfadl 55-1221 T - 9
4. Mariem Hatem Mobarak 55-1178 T - 14
5. Adam Adham Abdelfattah 55-3076 T - 9
6. Engy Sameh Abdelaziz 55-1361 T - 15

Supervised by: Dr. Catherine M. Elias

April 2024

Description:

This milestone mainly focused on implementing four threads in C using the pthreads library as the goal of this milestone was to recreate a part of the operating system functionalities which is the scheduling in executing different needed operations, which was represented using the four created threads being executed using different scheduling algorithms. Each thread has a minimum of three print statements, and it outputs its thread ID to identify itself. Furthermore, different scheduling algorithms are tested to observe how they affect the thread execution.

The predefined scheduling algorithms are FIFO(First in, First Out), RR(Round Robin) , and OTHERS.

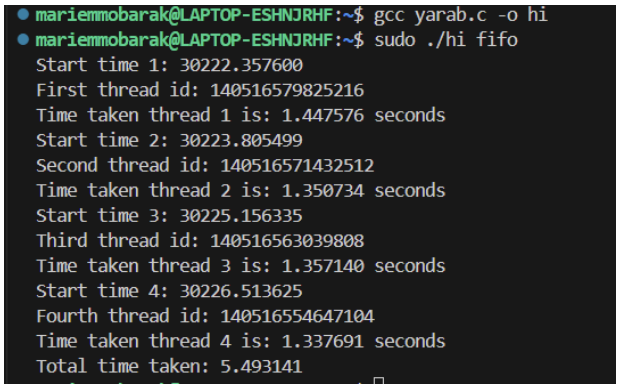
What we faced in this milestone and how we resolved it:

During the execution of our code, we encountered inconsistencies in the output across different devices, operating systems, and system loads.
This made it extremely difficult for our software to have the required functionality and dependability. At first, we tried applying scheduling policies without identifying thread affinities or specifying CPU sets in an attempt to reduce these inconsistencies. However, despite these efforts, the output remained inconsistent, leading us to explore alternative solutions. After some research, we discovered that the threads needed to execute on a single CPU core to maintain the correct order of execution. To implement this, we assigned all threads to a single CPU core using CPU set affinity. We continued investigating the issue's underlying causes because the program was still unable to generate the desired results after this modification. After further research, we discovered that running the program with elevated privileges, using the sudo command, was necessary. This gave us the necessary privileges to set the thread scheduling policies and CPU affinities effectively, resolving the inconsistencies in the output of our code. By implementing CPU set affinity and running the program with elevated privileges, we successfully fixed the inconsistencies in our code's output. This experience highlights the importance of considering system-level factors and privileges when optimizing code performance across different environments.

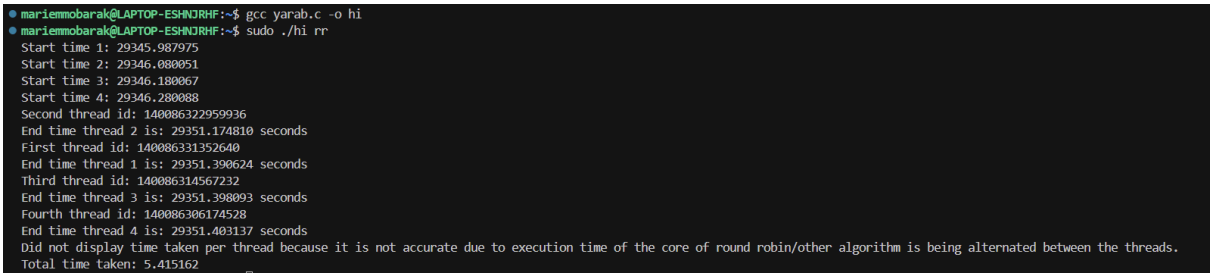
Summary of Milestone 1 steps:

1. We created functions for each thread we need where each function has 3 print statements (total of 4 functions + main).
2. We generated thread ids for each of our four threads in the main procedure.
3. Using the CPU_ZERO and CPU_SET functions, we then managed which CPU cores are used by our threads and assigned them a single core.
4. Establish and set the pthread properties.
5. Use the sched_setscheduler function to establish the desired scheduling policy.
6. In the main method, we used pthread_setaffinity_np to ensure the main thread is running on the same CPU core as the other 4 threads, and then create each thread.
7. We calculated the start time of and the end time of each thread using the clock_gettime function
8. Next, we used the pthread_join function to wait for the threads to finish in order to synchronize our program.
9. Elapsed time = End time – Start time
10. Finally, we cleaned up and released the resources associated with the thread attributes object using the pthread_attr_destroy function (Note: this does not affect any threads that were created using this attributes object. It only releases the resources associated with the attributes object itself).

Results of our code:

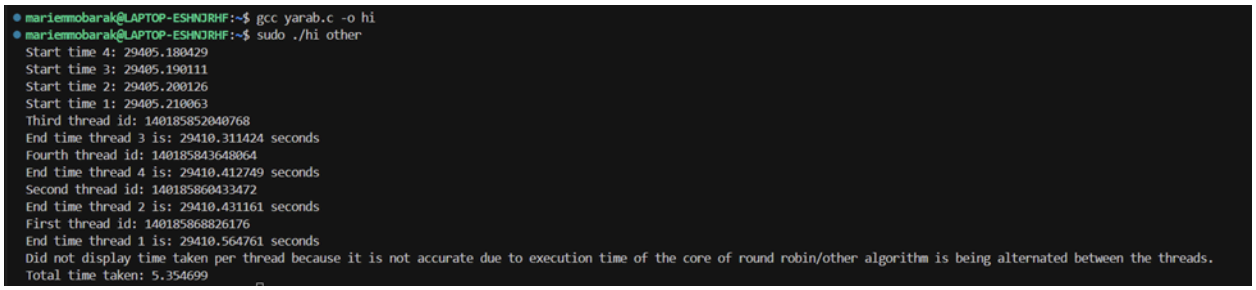


1. Using fifo: each thread executed in the order it was created, which indicates that FIFO is working correctly as shown in the image above.



2. Here we ran our code using round robin, where all threads started simultaneously, the end times were not in the same order due to the quantum/processing privileges given to each thread which indicates that our scheduling policy was working correctly.

3. Lastly, we tested the other scheduling policy, where it uses the default system's scheduling algorithm and no strict policy or time slicing rules are used, which indicates that our output was the expected output.



Kpi Findings:

The following shows the total time taken in seconds for all four threads to finish executing and the average time in seconds of the 10 runs.

Scheduling Policy	1st run	2nd run	3rd run	4th run	5th run	6th run	7th run	8th run	9th run	10th run	Average
FIFO	0.512923	0.506132	0.509329	0.509522	0.505969	0.510597	0.506632	0.512724	0.509242	0.510201	0.5093271
RR	0.517149	0.516208	0.513934	0.521542	0.520792	0.517316	0.514846	0.513889	0.513790	0.513521	0.5162987
OTHER	0.480934	0.471103	0.482200	0.469143	0.490503	0.483146	0.473997	0.470836	0.473080	0.483826	0.4778768

We calculated the total time taken by the threads using the start time which is the time the first thread starts executing and the end time which is the time the last remaining thread finishes executing. Based on the averages, the other scheduling algorithm proved to be the most efficient since it resulted in the shortest average execution time among the three scheduling algorithms.