CS 342 Operating Systems

Spring 2021

Project 2

Multi-threaded Scheduling Simulator

Name: Zeynep Cankara

Id: 21703381

Section: 1

Instructor: İbrahim Körpeoğlu

Contents

1 Introduction

Threads are being used to run multiple tasks concurrently. This property of threads can help to speed up the program execution by splitting up the program into tasks which we can assign to the separate threads. Threads can access shared data portions of the program which needs to be taken care of via *synchronization*. Accessing the shared data concurrently can lead to inconsistent behavior of the program.

There exist operating systems libraries such as **`pthreads`** for POSIX threads to create threads and handle synchronisation. In order to benefit from multi-threaded execution of programs we need to have *scheduling* policy to control the order of the tasks execution. Tasks that are not ready to execute yet are waiting within a queue and thread can choose which task to run first using the scheduling algorithm policy. The following section of the report explains project's implementation details of the project where I create a *multithreaded scheduling simulator* to experiment with performance of various scheduling algorithms.

2 Implementation Details

Since every thread has access to the ready queue it will impose a critical section problem. Thus, I have used **`pthread_mutex`** locks to control mutually exclusive access to the shared data portion in the ready queue.

The *ready queue* implementation done by a *Linked list*. The ready queue has operations *pushBurst* to add the CPU Burst and *getBurst* to retrieve the CPU burst waiting for execution according to the scheduling algorithm. Four non-preemptive scheduling algorithms implemented are FCFS, SJF, PRIO, VRUNTIME.

3 Experiments and Results

3.1 How average waiting time (ms) for a thread changes with different thread counts (N) and algorithms (ALG)?

```
Fixed variables:
```

- `(Bcount = 75, minA = 1000, avgA =1500 , minB = 100, avgB = 200)`

```
Independent variables:
```

- `(N, ALG)`

N/ALG	FCFS	SJF	PRIO	VRUNTIME
3	55.33	55.33	55.66	55.33
5	116.20	115.40	114.60	116.20
10	486.70	398.70	455.60	485.70

It can be seen from the table above that in general Short Job First (SJF) performs generally slightly better than the other non-preemptive scheduling algorithms. Since the CPU burst lengths and interarrival time between CPU bursts chosen randomly from exponential distribution it is hard to draw direct conclusions from the results but we can hypothesize that the heuristic of executing the shortest job first in the queue helps to distribute the workload in between threads better. By using SJF algorithm we are imposing a prioritisation scheme in between CPU burst lengths and not the thread ids. It is logical that the average waiting time for a thread increase as we increase number of threads since we are processing `N * Bcount` number of CPU bursts which increases togather with the number of threads.

3.2 How average interarrival time (avgA) and minimum interarrival time (minA) affects the average waiting time (ms) for a thread?

```
Fixed variables:
```

- `(N = 3, Bcount = 20, minB = 100, avgB = 200, ALG=SJF)`

```
Independent variables:
```

- `(minA, avgA)`

```
NA: Not applicable
```

minA/avgA	15	150	1500	6000
100	NA	423.00	52.67	4.00
1000	NA	NA	80.67	21.00

The table above demonstrates that increasing the both minimum and average interarrival time of the exponential distribution can help to improve performance of the algorithm. I hyphothesized that since I'm conducting the experiment with a short job first algorithm it is logical that longer interarrival times in between CPU bursts helped thee scheduling algorithm to prioritise the CPU bursts with shorter lengths better.

3.3 How long does the CPU burst time (minB, avgB) affect the average waiting time (ms) for a thread?

```
Fixed variables:
```

- `(N = 3, Bcount = 20, minA = 1000, avgA =1500 , ALG = SJF)`

```
Independent variables:
```

- `(minB, avgB)`

minB/avgB	20	200	2000
10	NA	17.00	1709.67
100	NA	80.67	1721.33
1000	NA	NA	4238.33

The table above shows that the longer the CPU burst the longer the average waiting time for a thread which is a trivial conclusion.

3.4 How number of threads (N) affects the average waiting time (ms) of a thread for a fixed total number of CPU bursts (Bcount * N) ?

```
Fixed variables:
   ●  (minA = 1000, avgA =1500 , minB = 100, avgB = 200, ALG=SJF)
Independent variables: (N, Bcount)
```

I conducted the experiment with total of 50 CPU bursts with the following parameters:

	(N = 2, Bcount = 25)	(N = 5, Bcount = 10)	(N = 10, Bcount = 5)
avg thread (ms)	14.00	107.40	207.70
time elapsed (ms)	89.81	38.33	23.54

The table above demonstrates if we fix the total number of tasks for a program `(N * Bcount)` and observe how average waiting time for a thread (ms) and the execution time of a program (ms) changes as we change the number of threads. From the table it can be seen that the average waiting time for a thread increase as we increase the number of threads whereas the program execution time decreases. It is a logical conclusion since now we are benefitting from the advantages of multi-threading to execute same total number of tasks.

4 Conclusion

The project helped me to get more insight into how **`pthreads`** library works for POSIX threads. I learnt a lot about *condition variables*, *mutex locks* and *thread manipulation* techniques. Implementing my own multithreaded scheduling simulator helped me to understand how I can design custom scheduling algorithms and measure the performance of these algorithms. I get more insight into the parameters I can use to perform experiments to measure scheduling algorithms performance.