Assignment 2 - Discussion

Question 1.

a. Explain how "Fair-Share Scheduling " works

The fair-share scheduler aims to ensure that all users or groups (depending on the level of abstraction used) will share equal CPU usage. It works by recursively applying the round robin scheduling strategy at each level of abstraction. For example, if there are 5 users, each executing one process, then each of these users will have, $\frac{100\%}{5} = 20\%$ of the total CPU usage. Let's call these 5 users; user 1, 2, 3, 4 and 5. If for example one of these users, user 2, decides to execute another process then user 2's processes will each use 10% of the total CPU usage, while all the other users are still using the 20% of the initial partitioned CPU usage. Furthermore, another layer of abstraction can be used such that groups are formed, where each group can contain multiple users. The same concept is applied to groups, for example, if there are 3 groups, containing 3, 2, 6 users respectively, then the CPU usage will be distributed as follows:

- Each group uses $\frac{100\%}{3} = 33.33\%$ of the total CPU
  - Each user in group 1 uses $\frac{33.33\%}{3} = 11.11\%$,
  - each user in group 2 uses $\frac{33.33\%}{2} = 16.66\%$, and lastly
  - each user in group 3 uses $\frac{33.33\%}{4} = 8.33\%$ of the total CPU

Furthermore, if each user uses multiple processes then each process will be distributed like the first example, still maintaining equal share in CPU usage for each group and user.

b. Discuss its potential advantages and disadvantages

- **Potential Advantages**
  - Having a fair share in processing power ensures that each group/user will obtain resources at a reasonable time
  - It prevents starvation when large number of programs are awaiting for resources
  - It increases response time for each process
  - It also ensures a consistent user experience especially when multiple users exist
- **Potential Disadvantages**
  - If one process requires much higher demand than all the other processes. For example, if one user is running a process which requires a lot of power such as a program like Quartus Prime, where as another user is running a process like notepad, which doesn't require much CPU power, then it would be desirable for the first process to be assigned most of the CPU usage. This is not the case when Fair Share Scheduling is used.

Question 2.

How do you calculate the value of quantum under different situations?

The value of **quantum** is dependent on different situations, which are as follows:

- If the current process is running and the remaining CPU time is **greater than or equal** to **time_quantum**, then **quantum** is set to **time_quantum**
- If the current process is running and the remaining CPU time is **less than** **time_quantum**, then **quantum** is set to the remaining CPU time
- If there is no current process running, then **quantum** is set to 1
- And, for all other situations, **quantum** is set to 1

Question 3.

a. Run the Round Robin dispatcher program under the following time quanta, giving the average turnaround and waiting times:

i. **time_quantum** = 1,

Average Turnaround Time	34s
Average Wait Time	28.5s

ii. **time_quantum** = 5,

Average Turnaround Time	27s
Average Wait Time	21.5s

iii. **time_quantum** = 10,

Average Turnaround Time	22s
Average Wait Time	16.5s

b. What are the implications by comparing these performance results?

From the results above, it seems that an increase in **time_quantum** results in a decrease in average turnaround time, subsequently a decrease in average wait time as wait time is dependent on turnaround time. It also seems that as **time_quantum** increases the time for each process to receive the resources required shortens, ultimately, **increasing performance for the whole system** (i.e. all the processes will be more time efficient).

To test this hypothesis, the finishing times for each **time_quantum** are compared. The specific input for this question, has 10 jobs which all arrive at time 0 and each job has a service time of 1 greater than the previous job, starting with job 1 having a service time of 1. Notice that wait time is dependent on turnaround time which is dependent on the finishing and arrival time, but all arrival time is 0 for this question, therefore, the arrival time can be ignored and hence, the earlier the finishing time the shorter the wait time.

From the finishing times below, there are clear differences in shorter finishing times in jobs where the **time_quantum** is 5 compared to 1, these differences are in the red circles. Furthermore, the same can be said between a **time_quantum** of 5 and 10, these differences are also shown via blue circles. **Therefore, the larger the time_quantum the shorter the average wait time (however for a time_quantum > 10 it makes no difference, this is further discussed in question 5b).**

Job	Finishing times (**time_quantum** = 1)	Finishing times (**time_quantum** = 5)	Finishing times (**time_quantum** = 25)
1	1	1	1
2	11	3	3
3	20	6	6
4	28	10	10
5	35	15	15
6	41	41	21
7	46	43	28
8	50	46	36
9	53	50	45
10	55	55	55

Question 4.

a. Run the Round Robin dispatcher program under the following time quanta, giving the average turnaround and waiting times:

i. **time_quantum** = 1,

Average Turnaround Time	15s
Average Wait Time	7.5s

ii. **time_quantum** = 5,

Average Turnaround Time	13.9s
Average Wait Time	6.4s

iii. **time_quantum** = 25,

Average Turnaround Time	16s
Average Wait Time	8.5s

b. What are the implications by comparing these performance results?

From question 3b, it was concluded that as **time_quantum** increased the average wait time would decrease. However, the input for this question shows that it is not the case. This is because in the first question, all the arrival times were 0 whereas, the arrival times here are random, furthermore, there is no linearity in service time. Below show the arrival and finishing times of **time_quantum** 1, 5 and 25.

Job	Finishing times (**time_quantum** = 1)	Finishing times (**time_quantum** = 5)	Finishing times (**time_quantum** = 25)
1	39	34	21
2	19	12	26
3	18	20	29
4	38	39	36
5	36	37	39
6	58	55	55
7	89	89	79
8	79	75	84
9	75	77	86
10	83	85	89

From these finishing times, it is evident that as arrival times and services times are randomised, **increasing time_quantum has no effect.** In this question, the optimal **time_quantum** is equal to 5. Therefore, an implication from this result is that the round robin scheduling approach is **ONLY** useful for some, not all, order of processes, furthermore, this should be considered in detail when scheduling processing power and CPU usage.

Question 5.

a. Run the Round Robin dispatcher program under the following time quanta, giving the average turnaround and waiting times:

i. **time_quantum** = 1,

Average Turnaround Time	95.5s
Average Wait Time	85.5s

ii. **time_quantum** = 5,

Average Turnaround Time	77.5s
Average Wait Time	67.5s

iii. **time_quantum** = 10,

Average Turnaround Time	55s
Average Wait Time	45s

b. What are the implications by comparing these performance results?

From questions 3 and 4, it was concluded that a large **time_quantum** is only beneficial for **SOME** order of processes, especially those that contain a pattern of linearity. However, this question provides an input where all the jobs contain the same arrival and service time (0, 10). Below contain the results of this input for different **time_quantum**.

Job	Finishing times (**time_quantum** = 1)	Finishing times (**time_quantum** = 5)	Finishing times (**time_quantum** = 10)
1	91	55	10
2	92	60	20
3	93	65	30
4	94	70	40
5	95	75	50
6	96	80	60
7	97	85	70
8	98	90	80
9	99	95	90
10	100	100	100

For cases where all the jobs have the same arrival and service time, it is similar to the result in question 3 where, as **time_quantum** increases the shorter the average wait time becomes. This pattern continues until **time_quantum** is equal to the largest service time for a job in the input, in this case the largest job is any of the jobs as they are all 10. Whereas for question 3, the largest job is the last job, where the service time is 10.

To conclude, if the **time_quantum** >> largest job's time, then it makes no difference to if **time_quantum** = largest job, furthermore, the response time will be too long as the processes will be waiting for resources hence, the processor will not efficiently used. On the other hand, if **time_quantum** is too small, then it causes inefficient switching between processes, which is also not efficient.

Question 6.

Compare First Come First Served Scheduling, and Round Robin Scheduling. In your comparison, include discussions of their potential advantages and disadvantages, and which scheduling scheme performs better under what job load conditions. (You need to justify your answers.)

First Come First Served Scheduling is the simplest form of scheduling, it is self-explanatory, the first processes that arrive are the first that will be finished regardless of its service time. On the other hand, Round Robin Scheduling is a bit more complicated because a time quantum (time slice) is introduced. This time quantum indicates the maximum time a process can spend executing, before getting pushed back into the end of the queue for continued execution later. There are times when FCFS is more desirable than Round Robin, and there are times when Round Robin is more advantageous. Below contain 4 situations, 2 where FCFS is preferable and 2 where Round Robin are.

- **Situations where FCFS performs better**

    1. When there are **many short processes** that need to be executed, the Round Robin Scheme will be at a disadvantage because the queue will get longer and longer, hence, the average finishing time for the jobs will increase, subsequently the average waiting time will also increase. For example, if someone has opened over 100 tabs of google, then it is desirable to use a FCFS scheme because the tabs (processes) don't require much service time and there are many of them.

    2. If there are processes where the **servicing time for all the jobs are large**, then it is always advantageous to use FCFS because, from question 5 it has become evident that the most efficient scheduling comes from setting the **time_quantum** to equal the longest servicing time job, this is effectively FCFS.

- **Situations where Round Robin performs better**

    3. Given a situation where there **most of the processes have short servicing time, but there are a few that have large servicing time**, if these larger processes are at the beginning of the queue, the shorter processes will suffer greatly in a FCFS scheme. Therefore, this is where a round robin scheme is advantageous. Using the internet browser example again, if someone decides to open 8 tabs of google (which doesn't require much processing power) and 2 tabs of websites which mine cryptocurrency (taking up large processing power). If FCFS scheduling is used and the cryptocurrency mining website is first or second in the queue, the tabs which are trying to open google will suffer and will take much longer to open as opposed to if a Round Robin scheme was used.

    4. Furthermore, there are situations where it is advantageous to have every process using an **equal share of the CPU**. This is exactly what a round robin scheme does. For example, if there is a program which reads and performs calculations on large datasets then having equal processing power will be the most efficient as work load is being shared with all the processors. Compared to FCFS, if a CPU has 4 processors, a round robin scheme will effectively be 4 times faster. Furthermore, another advantage of round robin is that **starvation will never exist**, whereas in FCFS it is very possible.

References

Advantages and Disadvantages of various CPU scheduling algorithms - GeeksforGeeks. (2020). Retrieved 28 October 2020, from https://www.geeksforgeeks.org/advantages-and-disadvantages-of-various-cpu-scheduling-algorithms/

Siahaan, A. (2017). Comparison Analysis of CPU Scheduling FCFS, SJF and Round Robin. doi: 10.31227/osf.io/6dq3p

Sibley, E., & Editor, P. (2020). A Fair Share Scheduler. Retrieved 28 October 2020, from https://proteusmaster.urcf.drexel.edu/urcfwiki/images/KayLauderFairShare.pdf

Stallings, W. (2018). Operating systems (9th ed.). Harlow: Pearson.