OS Project Write-up

Gordon Dauterman, Irakli Gurgenidze

1) Of the four simulated algorithms, which algorithm is the "best" algorithm for CPU-bound processes? Which algorithm is best-suited for I/O-bound processes?

In this context, the "best" algorithm for a given process can be interpreted as referring to the algorithm that yielded the shortest completion time for that process. By this definition, the best algorithm for CPU-bound processes would be the algorithm that resulted in the lowest average turnaround time. The equivalent for IO-bound processes would be the algorithm with the lowest average wait time, since any time spent in the ready queue bottlenecks the rate of IO bursts.

For CPU-bound processes, the standouts were FCFS and SJF, coming in with average turnaround times of 303.726ms and 287.648ms respectively for an input of {16, 2, 0.01, 256, 4, 0.75, 64}. These values are quite close to each other, and to determine a clear frontrunner it is useful to view CPU utilization stats as well. In this category, FCFS yielded a utilization rate of 57.779%, while SJF had a utilization rate of 59.366%, widening its lead over FCFS by a significant margin. These numbers show that SJF both allowed processes to finish their bursts faster, but also minimized CPU idling, allowing for more computations to occur over a given timeframe.

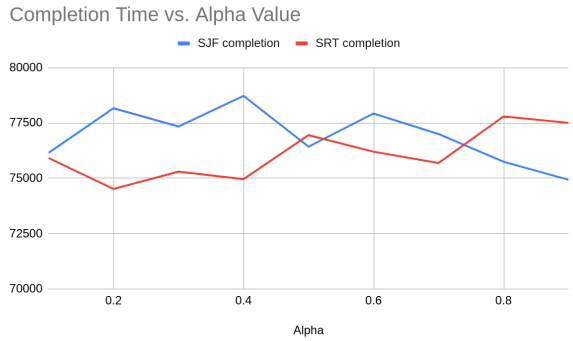
For IO-bound processes, the three highest performers were FCFS, SJF, and SRT, with average wait times of 215.423ms, 199.345ms, and 216.981ms respectively. In this case, SJF has a distinct lead over both FCFS and SJF, clearly demonstrating the dominance of SJF in this

category. By decreasing wait times, processes are able to shift into the IO queue more quickly, and since IO bursts can execute concurrently, this vastly increases the rate of IO burst completion.

2) For the SJF and SRT algorithms, what value of α produced the "best" results?

First we will cover the shortest job first algorithm. Looking at the data shown below (Fig. 1, Fig. 2, Fig. 3), it is a bit confusing as to what alpha value is "best" for this algorithm. On the one hand, as shown by Fig. 1, the completion time of the algorithm goes down a reasonable amount when the alpha value is raised closer to 1. On the other hand, as shown by Fig. 2, the average wait time and average turnaround time are increased slightly when the alpha value is raised closer to 1. Because it would be optimal for all of these values to be low, it seems that a high alpha value makes the algorithm better in some areas, and worse in others. I would say that the optimal alpha value would depend on what use case the user is planning for. However, if there had to be a final answer, it seems that while turnaround time and wait time are very valuable stats, their increase is very minor when compared to the decrease seen in completion time. Therefore, for the SJF algorithm, a higher alpha value, specifically around 0.8, seems to be most efficient.

As for the SRT algorithm, as shown by Fig. 2, a similar trend is observed when the alpha value is increased, with wait times and turnaround times increasing as well. However with this algorithm, the higher alpha value actually leads to a longer runtime. This means that it is a very clear answer that a lower alpha value in the SRT algorithm would be the best option.



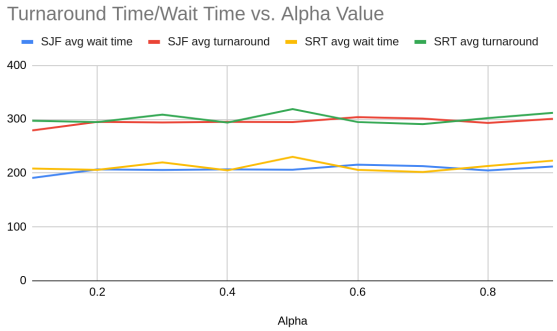


Figure 1 Figure 2

3) For the SJF and SRT algorithms, how does changing from a non-preemptive algorithm to a preemptive algorithm impact your results?

We ran 9 CPU simulations with the input values of {16, 2, 0.01, 256, 4, alpha, 64}, where alpha was a value between 0.1 and 0.9 with increments of 0.1. These changes in alpha were to take into account the difference in performance that each algorithm experiences with different alpha values. The data in Fig 3 is the average values given by simtext.txt averaged together for each alpha value.

Theoretically, when comparing the two algorithms, the preemption will speed up the overall CPU speed, because if the CPU is always running the fastest algorithm to complete, then it likely will always have processes in the ready queue, with little to no downtime between processes (this also increases CPU utilization). However the preemption will in turn increase the average wait time and turnaround time because the processes with a higher average burst time will be left on the backburner for an incredibly long time. This increased starvation is because the preemption leads to those faster bursts having their priority increased more as they run.

When looking at the values in Fig 3, these theoreticals are backed up with some minor yet noticeable differences between the two algorithms. The SRT algorithm is much faster than

the SJF algorithm, along with a minorly increased CPU utilization. However, the average wait time and turnaround time have increased by about 6ms each.

	Avg Completion (ms)	Avg Wait Time (ms)	Avg Turnaround Time (ms)	Avg CPU Utilization (%)
avg SJF	76935.111	206.961	295.265	58.856
avg SRT	76093.556	212.699	301.500	59.506

Figure 3

4) Describe at least three limitations of your simulation, in particular how the project specifications could be expanded to better model a real-world operating system.

Firstly, this simulation in the state that we submitted has no capability to test the effectiveness of algorithms based on whether or not processes are CPU-bound or IO-bound. This could easily be implemented with another user argument that defines scalars for CPU bursts and IO bursts (the default for this project was x10 for IO bursts).

Second, this simulation doesn't allow for testing based on a multitasking CPU, specifically if the CPU is able to run multiple processes at the same time. This would be difficult to implement, due to there being twice as many processes to take care of, however it could be done. The user could perhaps specify the number of processes that could be run simultaneously (the default for this project would be just 1 process).

Finally, we think that it would be very interesting to explore alternative tau recalculations that would put more weight on previously completed bursts. The way that tau currently works in the simulation is with the formula $\tau = \alpha \cdot BD + (1 - \alpha) \cdot \tau_0$. The formula gives more or less weight to the recently finished burst depending on alpha's value. This allows for the CPU to calculate its tau value to be more accurate based on recent bursts, or based on the cumulation of all bursts completed. However, the formula is done in such a way that if a process is made up of many bursts, a few bursts that fall off of the general pattern of the process can mess up the tau value for many bursts to come. An interesting specification might be to use the alternative formula $\tau = \frac{1}{CB} BD + \frac{CB-1}{CB} \tau_0$. This would put an even amount of weight on every burst completed throughout the processes' lifespan (as it is simply a running average of all burst durations.) This new specification would work especially well with another added specification

to allow for control over the average amount of bursts in a process. This would allow for research into different alpha values for the original tau formula, and also perhaps new tau formulas like the one previously described. This would simply be a scalar just like the CPU/IO burst scalars previously described (the default for this project would be a 1x scalar).

5) Describe a priority scheduling algorithm of your own design (i.e., how could you calculate priority?). What are its advantages and disadvantages?

Clearly there is great merit in sorting the ready_queue by tau because of the greatly decreased run time. However, these scheduling algorithms can fall short when they encounter starvation. Starvation is when a process is so long that it never ends up at the front of the ready queue, leading to an incredibly long turnaround time. A possible solution to this problem of starvation would be to modify the SJF algorithm to take into account waiting time. Currently, priority in the queue is calculated by taking the processes' tau and subtracting the total time the process has run. This new algorithm, perhaps named Shortest Job First But Without Starvation (SJFBWS), would take a new input from the user: wait_weight. The priority of processes would be calculated the same as in SJF, but with a new subtraction: the total time the process has spent in the ready queue multiplied by wait_weight, resulting in the new formula: $P = \tau - TR - (WW \bullet WT)$ (P = priority, TR = total runtime, WW=wait_weight, WT = wait time). This would greatly reduce starvation, and the wait_weight value would allow for the user to test for more or less starvation. The higher the value, the less time the process would spend in the queue, and the lower would do the opposite.