Randy Wang <wangr13>
Owen Lockwood <lockwo>
Justin Mai <maij>

CPU Simulacra and Simulation

1. Of the four simulated algorithms, which algorithm is the "best" algorithm? Which algorithm is best-suited for CPU-bound processes? Which algorithm is best-suited for I/O-bound processes? Support your answer by citing specific simulation results.

There is no definitive 'best' algorithm, the best algorithm is dependent on how one defines best and what one's processes look like. For example, if we were to define 'best' as the algorithm with the lowest average wait time, then SRT would take the cake (across the 4 test cases, the average wait times are: FCFS = 966.375, SJF = 904.196, SRT = 887.616, RR = 992.5465), this would be true for turnaround time as well (across the 4 test cases, the average wait times are: FCFS = 1263.134, SJF = 1200.964, SRT = 1185.238, RR = 1290.491). However, if you desired the lowest number of context switches (as your machine was not efficient with those) then a non-preemptive algorithm would be 'best', either FCFS or SJF. Thus, the best algorithm depends on your goals and machinery, but based on our test cases SRT is a solid choice.

For CPU-bound processes, SJF and SRT are the best if you don't include starvation and burst times are moderately varied (Figure 1). They're faster than FCFS when it comes to wait times and turnaround times, as FCFS would have its wait times climb up with more varied process times. The faster one of the two would depend on context switch times and the number of processes. More processes means it's more likely to preempt, causing a larger disparity. SRT has more context switches from preemptions. This adds up if context switch times are large. Even if context switch times weren't large, there could be a difference in preemptions due to a large number of processes. Thus, SRT has more overhead. If you want to remove the chance of starvation and process burst times are similar, FCFS would be the best as it has the least overhead and with similar burst times, the wait times will be low too (Figure 2). Yet, it isn't

"fair." Longer CPU-bound processes may make other processes wait really long. Thus, if you want something fairer, you would use Round Robin. In the case where burst times are small and that overly large processes don't occur (all the processes are around similar burst_times), FCFS dominates. Otherwise, you would need to use Round Robin to address the variation in burst times or for larger average burst times (figure 3).

For I/O-bound processes, the shortest remaining time (SRT) would be the best scheduling algorithm. If a majority of the processes that arrived in the ready queue had a large portion of I/O burst time and relatively short CPU times, then these processes would be able to preempt the CPU and complete their shorter CPU burst time then move to the I/O system and complete their I/O bursts in parallel. Depending on the frequency that new processes arrive, the short CPU burst times of I/O bound processes might still allow CPU bound processes to still be able to utilize the CPU. However, a problem with SRT could be that some processes with high CPU burst times would get starved of using the CPU because the shorter CPU time I/O bound processes would keep preempting any processes with any longer CPU burst times. This trend can be seen in the data from figure 1. Those processes had both longer CPU and I/O burst times yet the SRT scheduling algorithm was able to preempt any longer CPU bursts to get more processes into the I/O system. This ends up being the most efficient use of the CPU and also keeping processes from waiting on the ready queue.

2. For the RR algorithm, how does changing rr_add from END to BEGINNING impact your results? Which approach is "better" here?

In order to test this, we ran 3 tests. Increasing the number of processes from 1 to 26 for each of the sets of parameters (2,3 are the same, then 4 and 5) we then gathered the average wait time. These graphs are included as figures 10, 11 and 12 (for 2/3, 4 and 5 test cases parameters respectively). What this data tells us first and foremost is that more extensive testing needs to be done. Thousands of simulations and variations need to be tried before coming to a conclusion. That being said, based on our

data, the first conclusion of note is that neither is particularly better than the other. They both perform extremely comparably for all sets of parameters. That being said there are some slight differences that might become more apparent with significant increases in testing. First is that the size of the time slice has little impact. Parameters for test cases 2/3 and 4, the only difference was in the time slice (4's being ½ of 3s). However, in both of these results beginning performs just slightly better than end. However, when the CPU burst cap is increased (as it is with test case 5 parameters, in figure 12), the results swap and end becomes more efficient than beginning for large numbers of processes. However, these differences are slight.

3. For the SJF and SRT algorithms, what value of α seemed to produce the best results? Support your answer by citing specific simulation results.

If you refer to figures 8 and 9, those are graphs of the average wait and turnaround times for alpha values ranging from 0.01 - 0.99 with steps of 0.01. This is using the parameters of test case 3, so relatively low CPU and I/O burst times. A noticeable valley in the line graph appears around an α value of 0.15 - 0.20. This would suggest that a lower α value produces the best results. This can further be observed in figures 13 and 14. This data is using the parameters of test case 5, so processes with much higher average CPU and I/O burst times. The data follows a similar trend where in the lower range of α values, the average wait and turnaround times are low, but as the value of α increases, so does the average wait and turnaround times. This could be explained because when the value of α is higher, the prediction of the tau value is more volatile because it will change more frequently, overestimating or underestimating the average time of a CPU burst. However when the value is lower, the tau values would change more slowly, moving asymptotically towards the optimal, mean CPU burst time in the exponential distribution.

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

When the context switching time is really low, it is more efficient for the CPU scheduler to preempt processes as the cost of preemption would be extremely low. SJF might get a couple processes that have comparatively longer CPU burst times which end up blocking I/O bound processes in the ready queue. SRT algorithm would have a better performance than a non-preemptive SJF as it could preempt these processes with longer CPU bursts to finish I/O bound processes' CPU bursts and have them running in parallel in the I/O system. This is reflected in figures 4a and 4b. As the number of processes increase, the average wait and turnaround times stay relatively the same. This is with a context switch time of 4 ms. It is likely if the context switch time were even shorter, then SRT would, on average, perform better than SJF.

However, when the cost of context switching is high, it is more efficient to minimize the number of expensive context switches or avoid preemptions. In figures 6 and 7, the average wait time and turnaround time begin to diverge from each other as the number of processes increase. This occurs because with more processes, it is more likely for the CPU to preempt. In those graphs, the context switch time was set to 40 ms, or ten times the amount of the previous graphs. And if we were to continue past n = 26 for processes, then we would continue to see SJF performing better than SRT simply because it's performing less context switches.

5. 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.

One significant limitation of the simulation is the use of a single CPU core. Contemporary operating systems have access to a multitude of cores, with additional access to multithreading. To expand the simulation and make it more reflective of real world computing, expanding the number of CPUs would be a first step. This would allow processes in the ready queue to access any of the CPUs and have potentially cascading preemptions as a ready queue process could preempt a CPU process that would then preempt a different CPU process. This would enable the simulation to be more accurate.

Another limitation of this simulation is the treatment of all processes as a single class, i.e. all processes are evaluated on the same standard. In real-world operating systems, there can be different types of processes that the CPU will give higher priority to. Some different types might be real-time processes (very high priority), interactive processes (high priority), or batch processes (low priority). Each class of processes would have different sets of workload parameters like a higher average arrival time or different average CPU service time. In order to implement this multiclass system, there would likely need to be a multi-tiered queue implemented. Furthermore, multiple scheduling policies could be considered, one for each group of processes. This would enable the simulation to be more reflective of a real-world operating system.

Finally, the number of processes and duration of the simulation are significantly lower than real world OS. The simulation is capped at 26 processes but real world computers go significantly above that. Currently, my laptop has 392 processes running, which can be significantly larger for different types of machines. Thus expanding the number of processes allowed by our simulation would be critical to implement more realistic ones. Making these simulations longer is also important. Over a brief period of time, only so much insight can be had. These simulations simulate ~50 s at most, when real operating systems run for hours and hours. The effects of the algorithmic choice on wait time and other metrics may be drastically different when a much larger amount of processes is simulated for a much larger amount of time. To increase the accuracy of the simulation, we would need to include multiple CPUs, multiple classes of programs, and raise the cap on the number of programs and increase the length of the programs' run times.

6. Describe a priority scheduling algorithm of your own design (i.e., how do you calculate priority?). What are its advantages and disadvantages? Support your answer by citing specific simulation results.

Randy Wang <wangr13>
Owen Lockwood <lockwo>
Justin Mai <maij>

An optimal priority scheduling algorithm would include priority that could be assigned both internally and externally and also include preemption to finish higher priority jobs. The internal priority would look at metrics such as average burst time, CPU service time, ratio of CPU to I/O activity, arrival time, system resource usage. External priorities could look at what type the type of process is and assign priority based on that. E.g. batch processes might have a lower priority as they could run in the background (non-preemptive), interactive priorities would be high priority as users or servers are waiting for a quick response (preemptive), or real-time processes which would be the highest priority and would immediately preempt the CPU. The priority scheduling algorithm would also keep track of a process's age and would factor this in as part of the internal priority (older age = higher priority). The purpose of aging is to prevent a low priority process from being starved of using the CPU.

The main disadvantage with this approach is the computational overhead. With complex priority calculations and management of queues, there is the potential for a non-trivial algorithmic overhead when you consider the multitude of processes run daily on a CPU. With the changes in priority, if this is a preemptive scheduling algorithm, there is a high likelihood of many context switches, which can be very disadvantageous if the CPU is not well set up to handle them.

The first advantage to this approach is that it prevents CPU starvation. This means that important (potential UI) processes won't get stuck behind long processes. This makes it effective for operating systems which have high levels of human interactivity. The second advantage this can have is to decrease the wait time. As we have seen from question 1, differences in priorities can affect the average wait time, thus this algorithm could serve to reduce the average wait times.

Randy Wang <wangr13>
Owen Lockwood <lockwo>
Justin Mai <maij>

Data and Figures

Figure 1: Simout5

8 processes, all with bursts less than 4096 ms, with 4ms context switches, .5 alpha

```
Algorithm FCFS
-- average CPU burst time: 903.431 ms
-- average wait time: 3185.499 ms
-- average turnaround time: 4092.930 ms
-- total number of context switches: 473
-- total number of preemptions: 0
Algorithm SJF
-- average CPU burst time: 903.431 ms
-- average wait time: 2889.805 ms
-- average turnaround time: 3797.237 ms
-- total number of context switches: 473
-- total number of preemptions: 0
Algorithm SRT
-- average CPU burst time: 903.431 ms
-- average wait time: 2811.782 ms
-- average turnaround time: 3720.338 ms
-- total number of context switches: 606
-- total number of preemptions: 133
Algorithm RR
-- average CPU burst time: 903.431 ms
-- average wait time: 3253.886 ms
-- average turnaround time: 4161.782 ms
-- total number of context switches: 528
-- total number of preemptions: 55
```

Figure 2: Simout4

16 processes, with cpu bursts at 256ms at most, and 4ms context switches

```
Algorithm FCFS
-- average CPU burst time: 84.304 ms
-- average wait time: 622.507 ms
-- average turnaround time: 710.810 ms
-- total number of context switches: 537
-- total number of preemptions: 0
Algorithm SJF
-- average CPU burst time: 84.304 ms
-- average wait time: 669.520 ms
-- average turnaround time: 757.823 ms
-- total number of context switches: 537
-- total number of preemptions: 0
Algorithm SRT
-- average CPU burst time: 84.304 ms
-- average wait time: 695.521 ms
-- average turnaround time: 785.248 ms
-- total number of context switches: 728
-- total number of preemptions: 191
Algorithm RR
-- average CPU burst time: 84.304 ms
-- average wait time: 669.814 ms
-- average turnaround time: 761.492 ms
-- total number of context switches: 990
-- total number of preemptions: 453
```

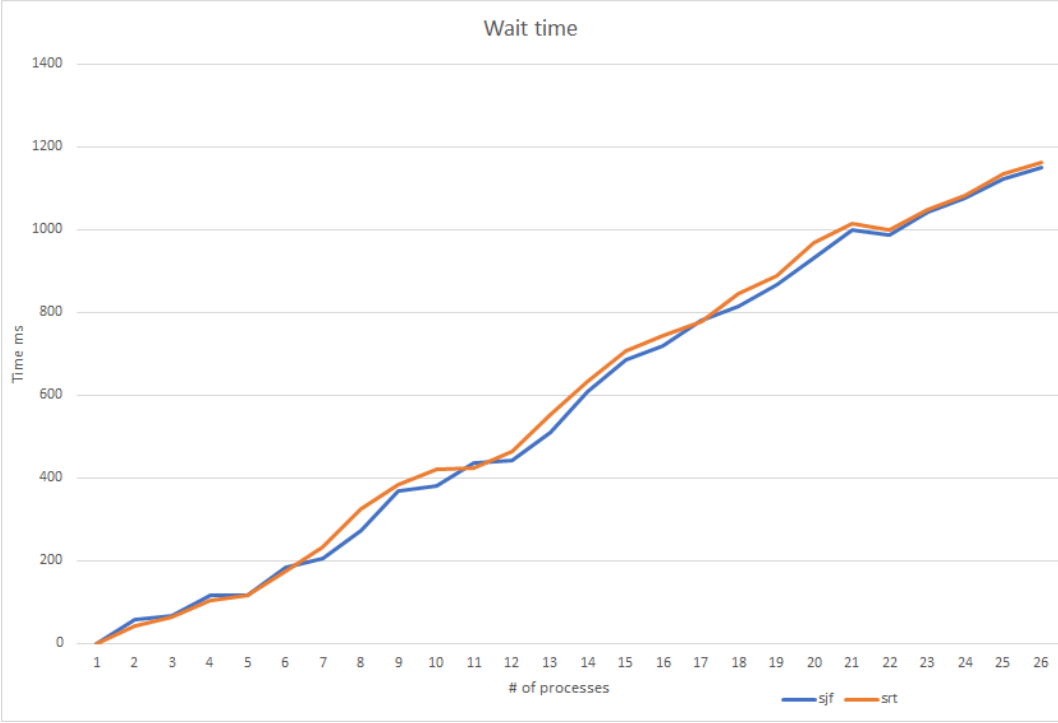
Randy Wang <wangr13>
Owen Lockwood <lockwo>
Justin Mai <maij>

Figure 3: Simout3.txt

For 2 processes with upperbound 256ms and a context switch time of 4ms

```
Algorithm FCFS
-- average CPU burst time: 103.027 ms
-- average wait time: 57.459 ms
-- average turnaround time: 164.486 ms
-- total number of context switches: 37
-- total number of preemptions: 0
Algorithm SJF
-- average CPU burst time: 103.027 ms
-- average wait time: 57.459 ms
-- average turnaround time: 164.486 ms
-- total number of context switches: 37
-- total number of preemptions: 0
Algorithm SRT
-- average CPU burst time: 103.027 ms
-- average wait time: 43.162 ms
-- average turnaround time: 151.054 ms
-- total number of context switches: 45
-- total number of preemptions: 8
Algorithm RR
-- average CPU burst time: 103.027 ms
-- average wait time: 46.486 ms
-- average turnaround time: 154.378 ms
-- total number of context switches: 45
-- total number of preemptions: 8
```

Figure 4a: Wait time graph for sjf and srt, short context switch with test 3 parameters



Randy Wang <wangr13>
Owen Lockwood <lockwo>
Justin Mai <maij>

Figure 4b: Turnaround time graph for sjf and srt, short context switch with test 3 parameters

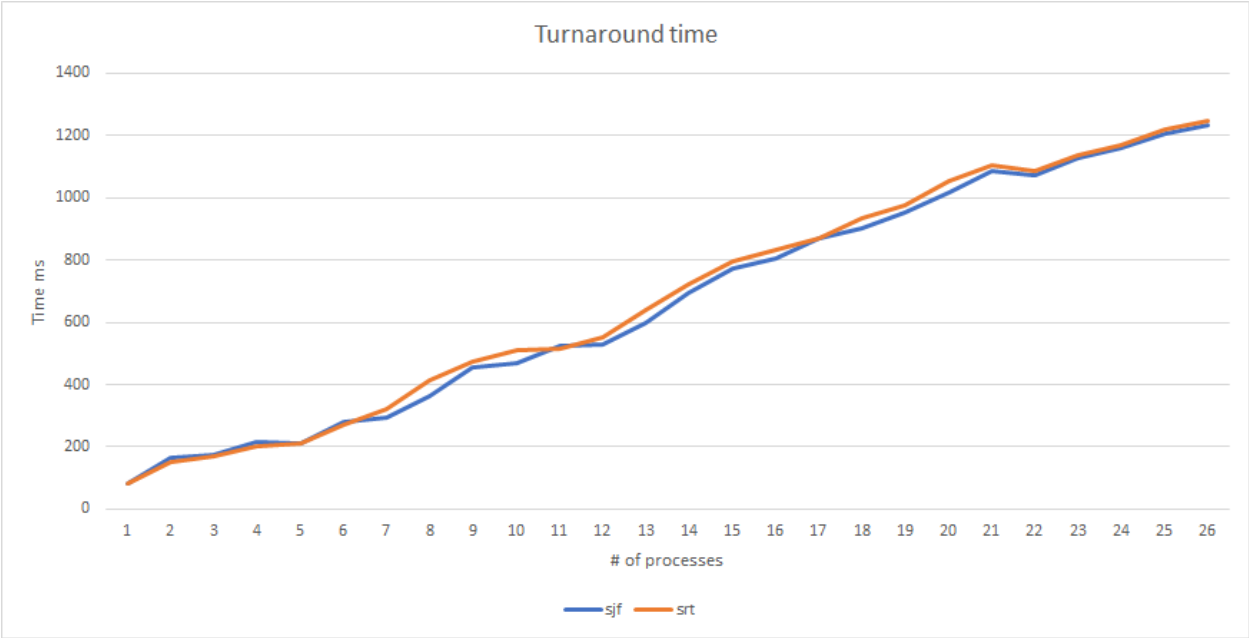
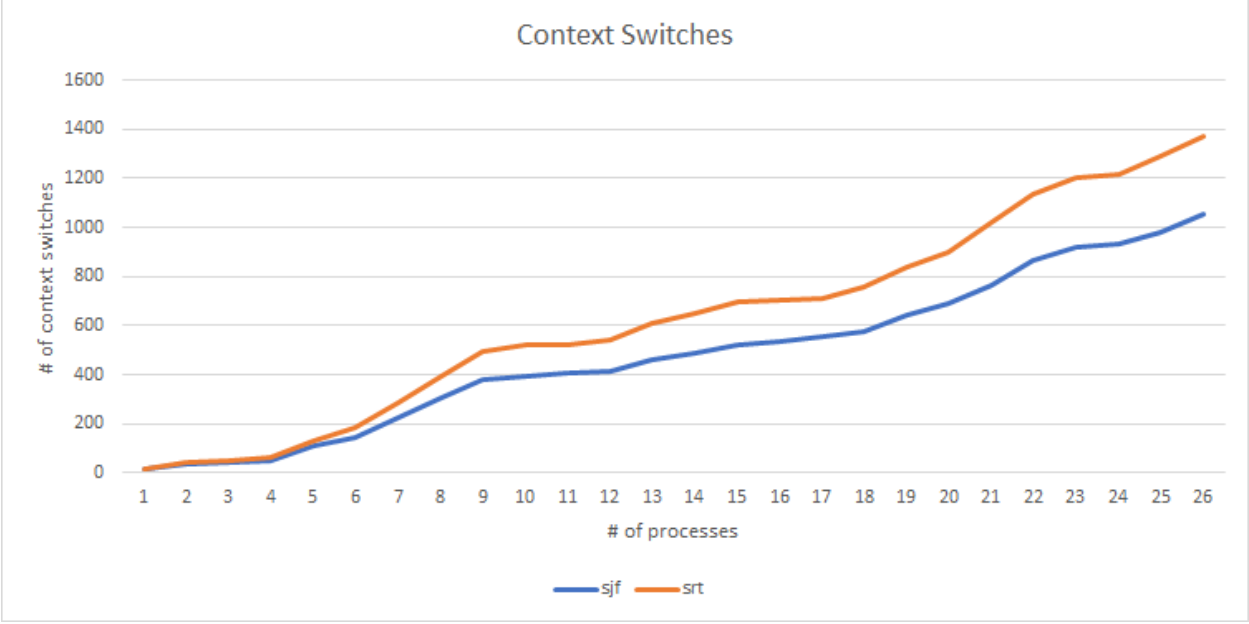


Figure 5: Context switch graph for sjf and srt, with test 3 parameters



Randy Wang <wangr13>
Owen Lockwood <lockwo>
Justin Mai <maij>

Figure 6: average wait times for sjf and srt, long context switch with test 3 parameters

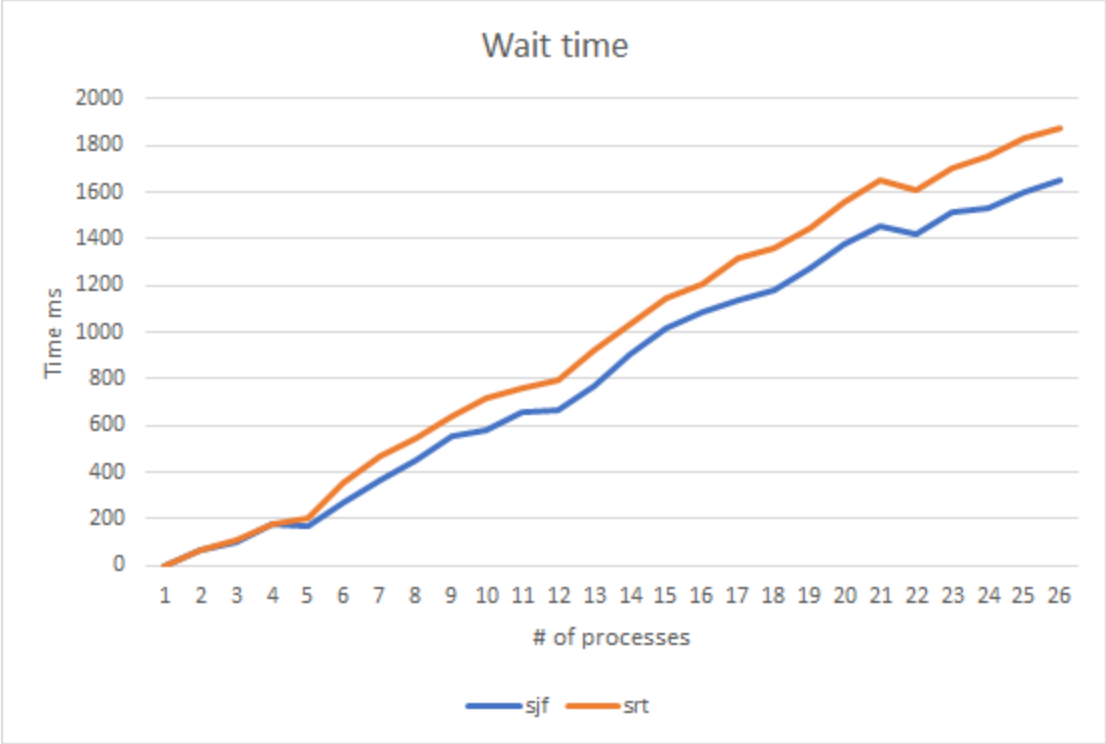


Figure 7: average turnaround times for sjf and srt, long context switch with test 3 parameters

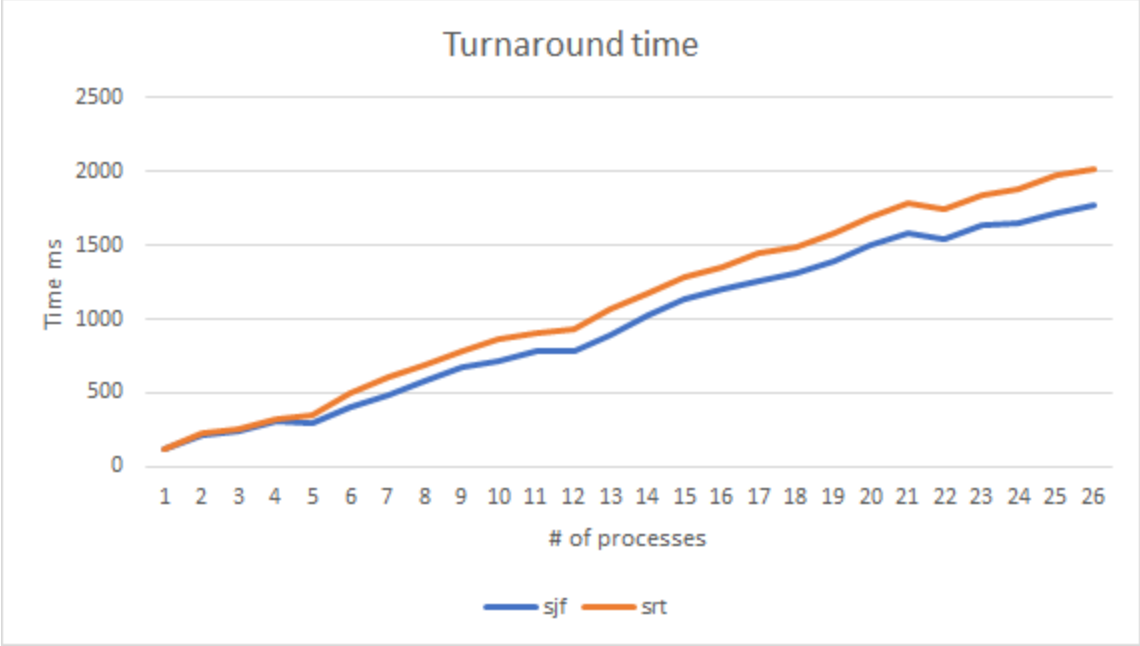


Figure 8: Average wait times for sjf and srt for different alpha values, with test 3 parameters

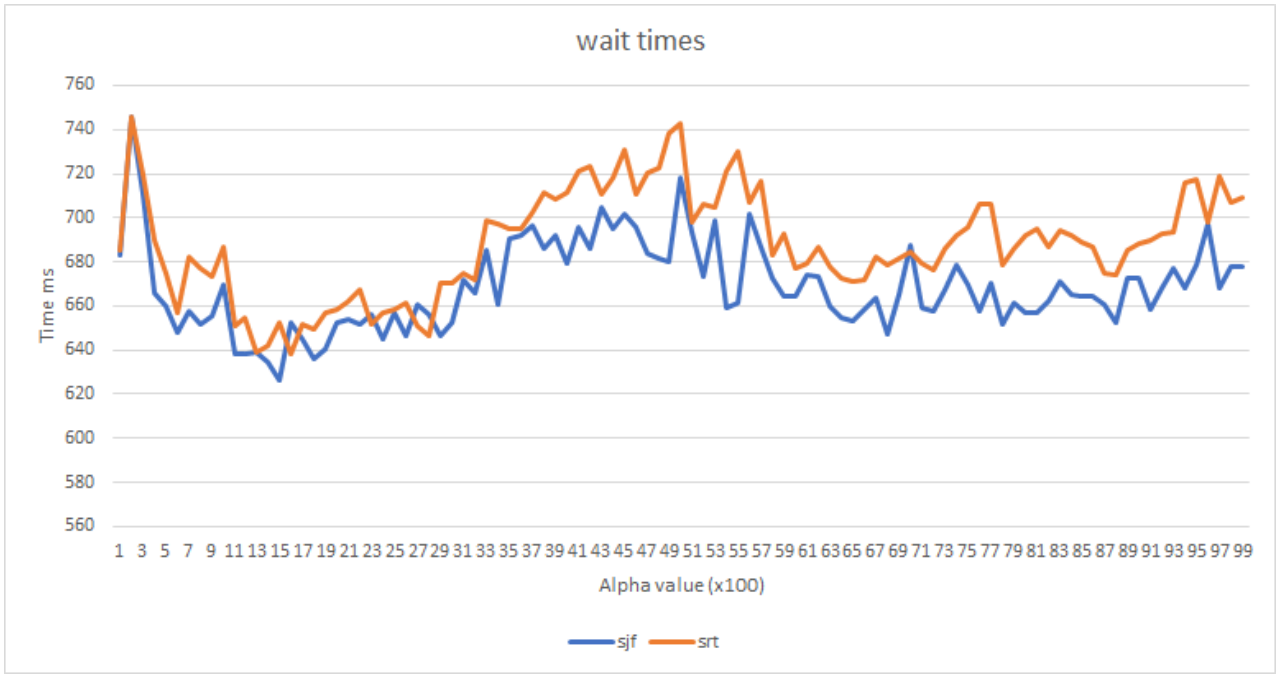
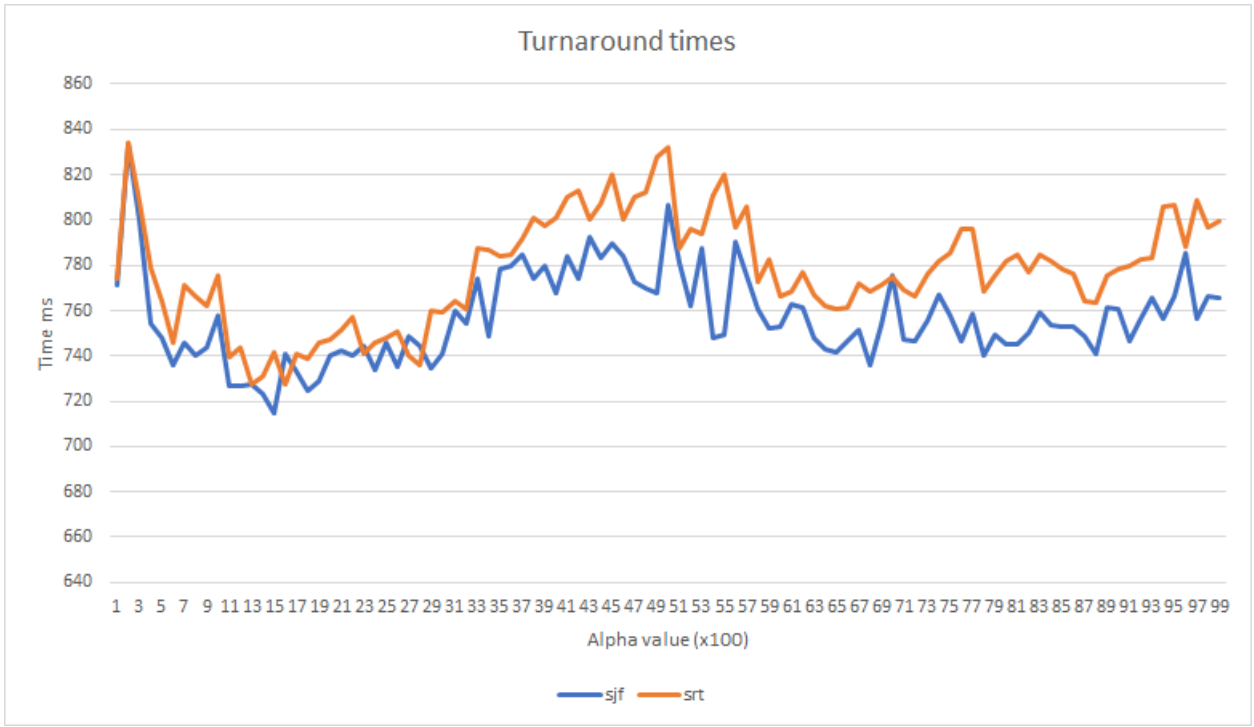


Figure 9: Average turnaround times for sjf and srt for different alpha values, with test 3 parameters



Randy Wang <wangr13>
Owen Lockwood <lockwo>
Justin Mai <maij>

Figure 10: Round Robin ready queue placement with parameter set from test cases 2/3

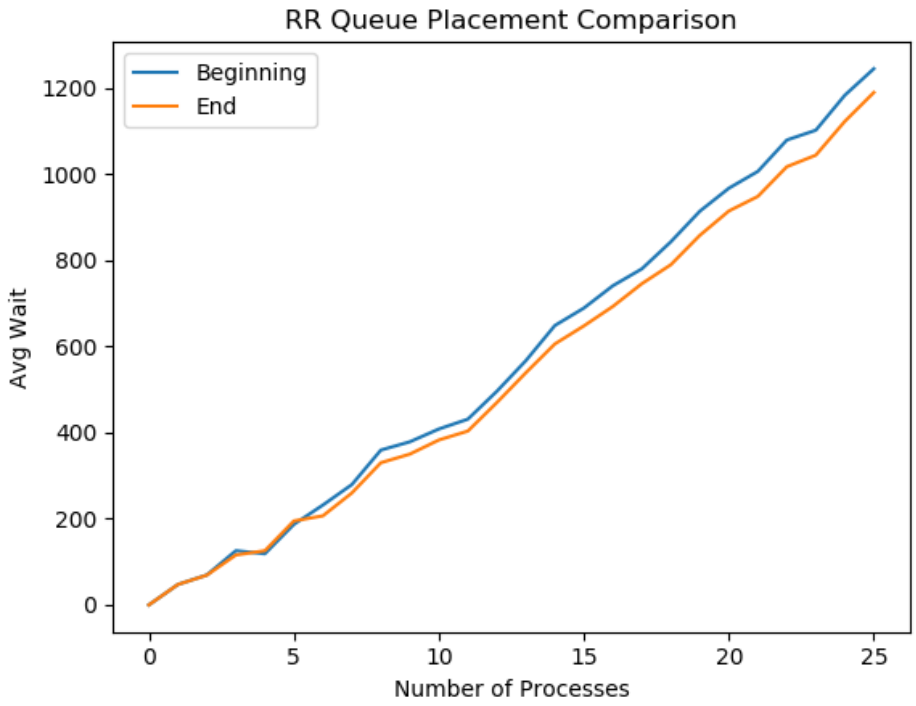
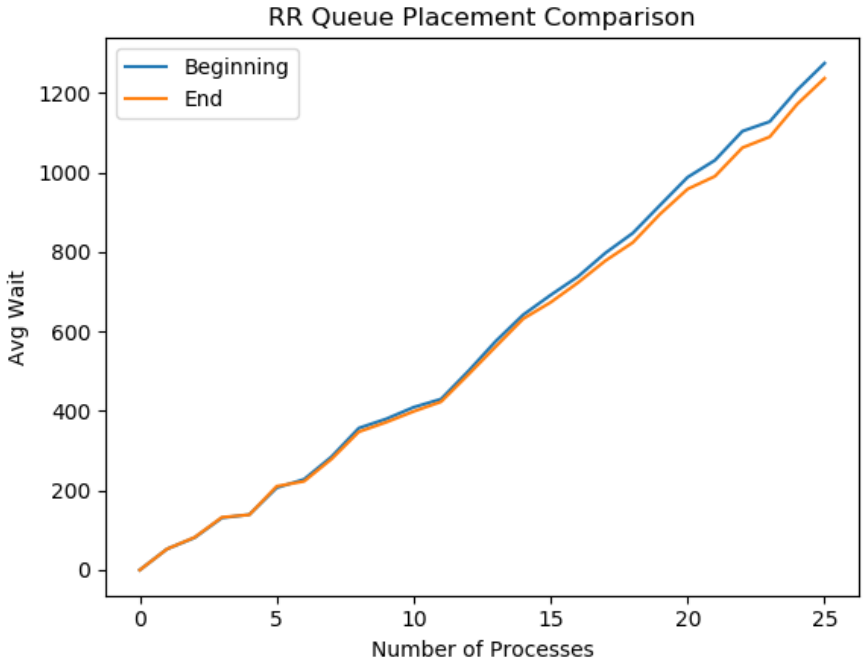


Figure 11: Round Robin ready queue placement with parameter set from test case 4



Randy Wang <wangr13>
Owen Lockwood <lockwo>
Justin Mai <maij>

Figure 12: Round Robin ready queue placement with parameter set from test case 5

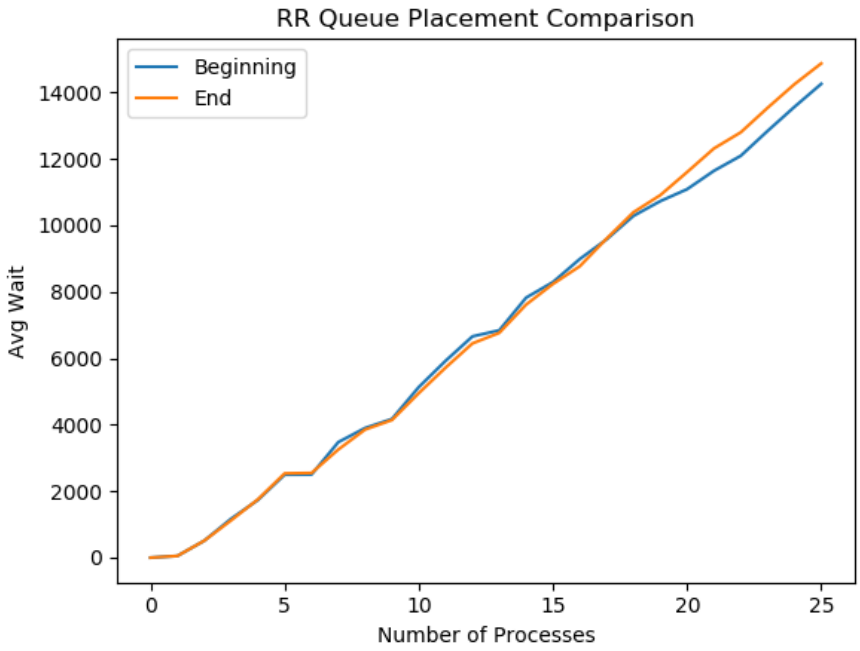
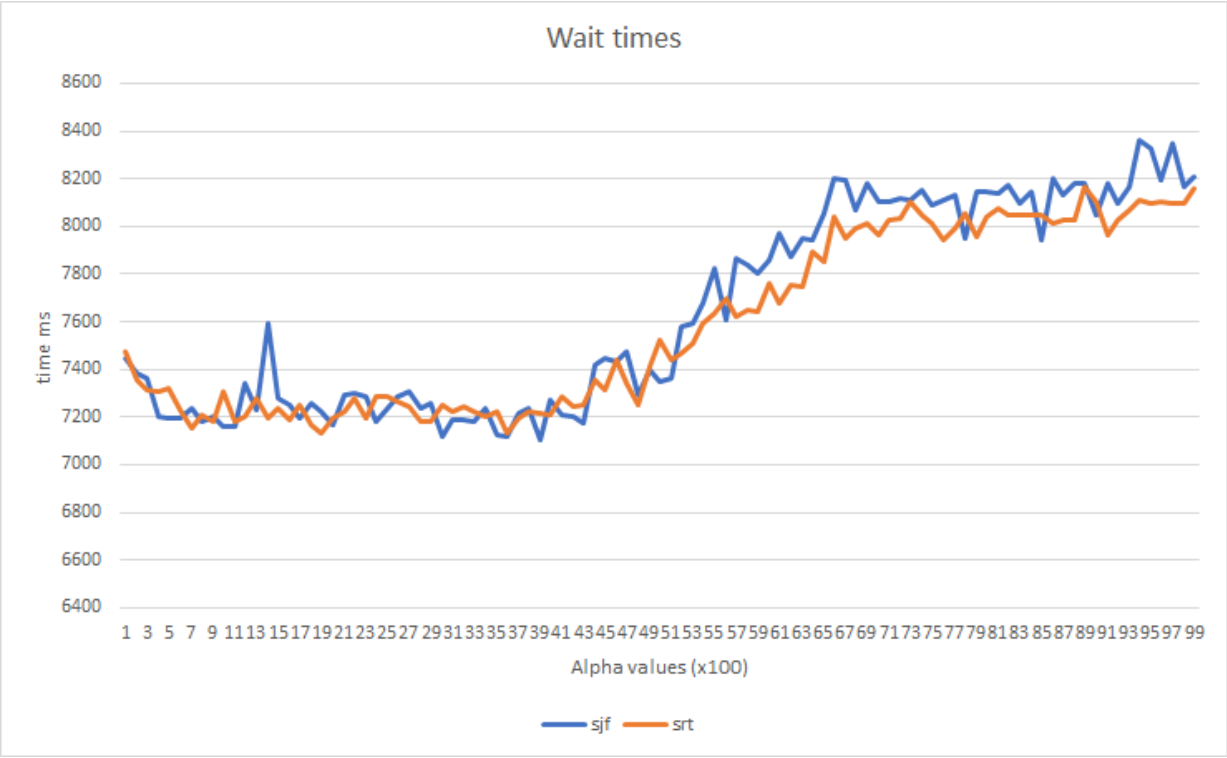


Figure 13: Average wait times for sjf and srt for different alpha values, with test 5 parameters



Randy Wang <wangr13>
Owen Lockwood <lockwo>
Justin Mai <maij>

Figure 14: Average turnaround times for sjf and srt for different alpha values, with test 5 parameters