**Experimentation with Synchronization Methods**

There are a large variety of ways to schedule processes in an operating system. In our experiment, we analyzed the differences between three major process scheduling algorithms: Round Robin (RR), First-In-First-Out (FIFO), and Shortest Job First (SJF). We generated fifty processes with a range of 1,000-11,000 cycles and then proceeded to test the aforementioned three types of scheduling algorithms to determine the average wait time, average completion time, and context switch penalty for the generated processes. To expand on our results, we also included a quad-core simulation to help us determine the benefits of multi core processing and to see what benefits (if any) were gained by more processing cores. Based off of our preliminary knowledge, we hypothesized that the SJF method would process the process the processes the quickest while having the lowest waiting while the RR method would be the slowest in both regards.

To back our hypothesis, we will explain the mathematics behind the different processing methods to back our reasoning. We used three major equations in our analysis: Wait Time of a Process, Average Wait Time, and Average Completion Time.

Wait Time of a Process:

←-----Time(completion)-Time(time to fully execute process)-Time(entrance)

Where the completion time is the time at which the process actually finishes, and entrance time is when the process becomes ready. When we calculate all the Wait Times, we then can calculate the Average Wait Time.   
  
 Average Wait Time of a set of Processes:

←---should divide by n not 50!

Now, with our Average Wait Time established, we need to find our Average Completion Time, which is calculated in a similar fashion. This is probably the most important data because it gives an idea on how good the throughput the processing method provides on a particular set of data.

Average Completion Time of a set of Processes:

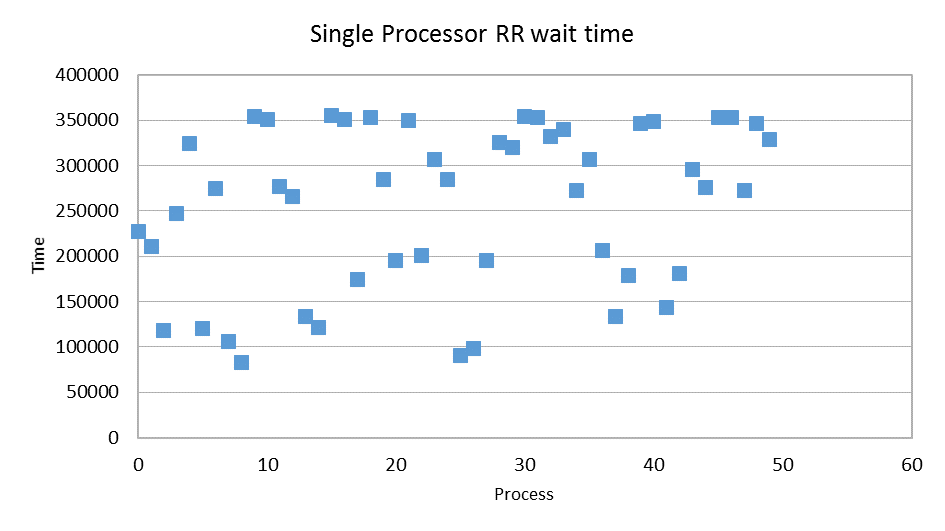
←----same here!

We simply added up the completion times of each process, and divided by the total number of processes. In our experiment, we used fifty processes. Here is the overall statistics on one of the tests we did where used every single processing methods and both single and quad core on the same set of processes:



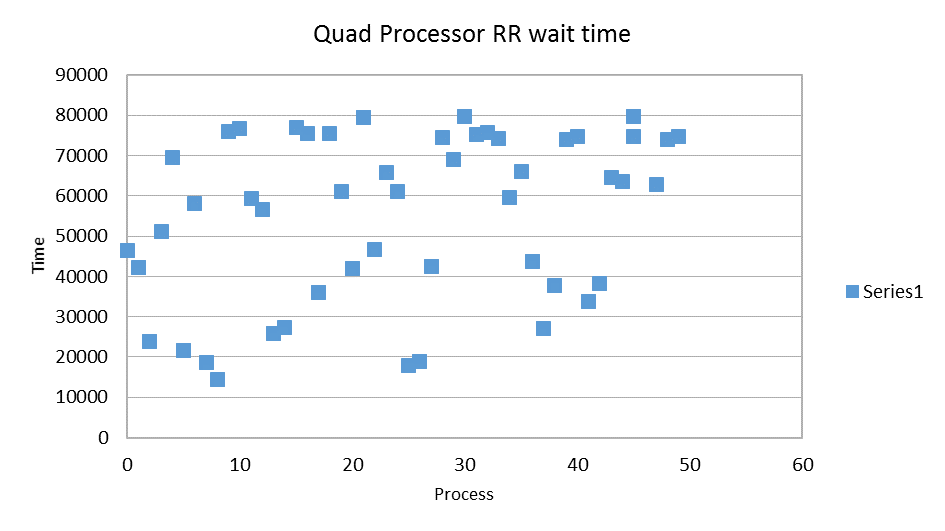
Initially, we began tests with Round Robin, which is a starvation-free, preemptive process scheduling algorithm that uses time slices to determine how long a given program is expected to run. In this program, processes are marked by how much “time” they've spent getting processed. Round Robin utilizes a “quantum”, which is a specific amount of time spent on a process before rotating to the next one in the list whether the process is finished or not. When the process spends time equivalent to the time quantum defined in the header of the program, or the process runs to completion, the program switches out the process for another process in the ready queue that is ready to be processed. With a low quantum value, you are doing very little of each process and constantly rotating between processes. This tends to mean that (unless you have very small processes), it can take a while for processes to finish as they can only do slight amounts before giving up their resources. Also, as the processes are constantly rotating, you also have to deal with high context-switch penalties. The amount of context-switch penalties is inversely proportional to the size of the quantum. With our amount of cycles per process averaging around 6,000, and a quantum of 10 cycles, each process will take a significant amount of time before being able to actually complete its task. A large advantage to Round Robin scheduling is that there is virtually no process starvation. Every process, no matter how large or small, gets an equal chance to use resources (assuming the process is ready to execute of course). This could theoretically help to keep a “balance” between processes in terms of load, but we could find little tangible benefits of using Round Robin compared to other scheduling algorithms, especially given the massive context-switch penalty, and the high wait times.

Our Round Robin data for our single core resulted in the following data:





It should be noted that processes in our program arrived at times of multiples of 50, with the first process arriving at time 0 (i.e, process 0 arrives at time 0, process 1 arrives at time 50, etc.) Based on the data from the scatter plot, it is clear that there is no linear system of time that can be established with Round Robin. With the processes having random times, and the interruption of the time quantum time mechanism, there are massive fluctuations in terms of Process Wait times, with times ranging from roughly 75,000 to 360,000 cycles on a single core system.

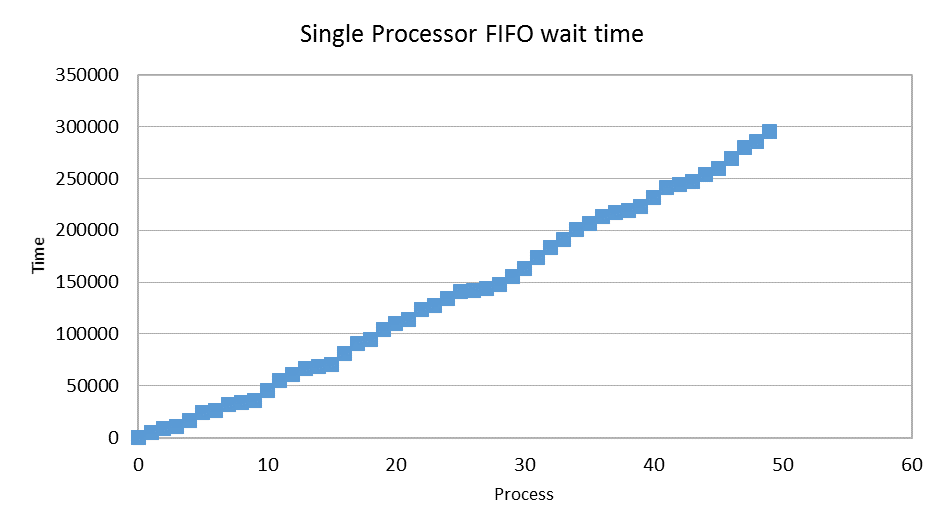




Immediately, one can see how drastically the wait time has dropped for all the processes. With multiple cores serving multiple processes, the range of the wait time for the processes dropped from a range of 75,000-350,000 to around 11,000-82,000. Based on the processes themselves, the wait time would of course change. But it appears that processing will always be faster with a quad core system in these kinds of tests.

Our second type of process scheduling algorithm was the non-preemptive First In First Out method. We intentionally chose a non-preemptive scheduling algorithm to attempt a different perspective from the preemptive processing style of Round Robin. In terms of complexity, the First In First Out style is the simplest form of scheduling, as it simply runs a process as soon as its ready and the current processes has finished executing. There is no organization necessary for the ready processes. As soon as they are ready, they are put into the ready queue. However, one can initially see some advantages as well as some disadvantages of this method. One of the main disadvantages is that short, simple processes can be put at the of the queue and have to wait for the very long processes in front of it to execute first. However, it makes up for this disadvantages with its low context-switch times. As each process runs through in its entirety without stopping, there are only N-1 context-switches for a single core, or N-4 context-switches for a quad core, where N is the number of processes to be executed. In this experiment, the context-switches each had a penalty of ten, and with fifty processes total, our single core resulted in a 490s context-switch penalty, and our quad core resulted in a 460s context-switch penalty. The quad core starts out with three more processes initially, which means there are three fewer context-switches.

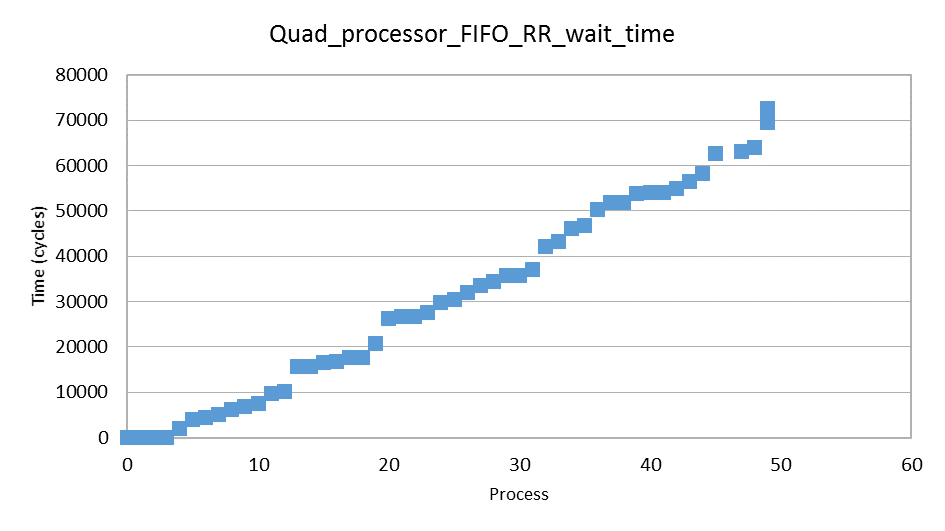
Our single-core simulation resulted in the following data:





A nice, seemingly linear curve is generated for this data because each process that arrives must wait for all the processes before it to finish so that it can begin executing. Thus, processes arriving earlier will have little wait times while processes arriving later will have large times. Process throughput is increased since processes are actually able to be finished much earlier than they are with round robin (however, we also suspect that part of the issue with round robin with these tests is that the time quantum we are meant to use is very small. It would be interesting to try these tests again with a larger time quantum).

Our quad-core tests for FIFO resulted in the following data:

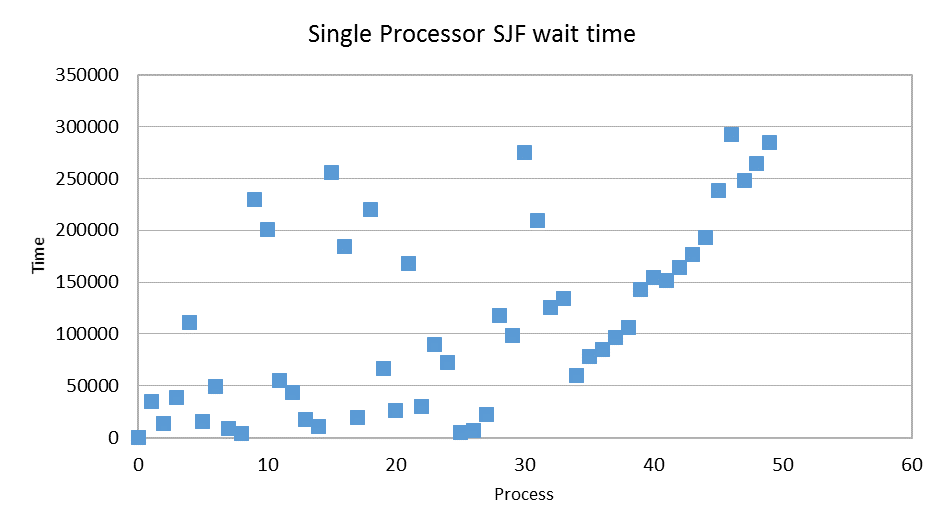




FIFO turned out significantly faster with a quad-core than it did with a single-core as expected. Our data maxed out around 75,000 cycles compared to the 300,000 cycles with a single core which makes sense as the processors can process 4 times many more processes. The low-end of the spectrum was significantly lower than Round Robin’s low-end, but the higher-end matches up similarly. Given that many processes have a lower wait time, FIFO would still seem to be the optimal choice comparatively. However, this depends on our needs. If processes had a much bigger completion time range, round robin might seem like the optimal choice as it ensures that short processes won't have to wait until every long process finishes in order to begin executing.

For our third type of process scheduling algorithm, we opted to use the non-preemptive form of Shortest Job First. Due to its non-preemptive nature, the processes are not constantly shifting between each other, which allows for a very low context switch penalty as with First In First Out. However, the main problem is with process starvation, in which a long process in the ready queue never get processes because many smaller processes always come in front of it and take up processing time. However, this was not really an issue in this analysis because we only had a fixed amount of processes be processed. In terms of overall speed, our non-preemptive Shortest Job First tests for completion and wait time were marginally faster than those for our First In First Out tests and significantly faster than our tests with Round Robin. With our quad-core simulation, our overall time was faster and, due to having four cores at a time, our context-switch penalty was reduced by 30 (with a penalty of ten per context switch) as the extra three cores reduced the initial load.

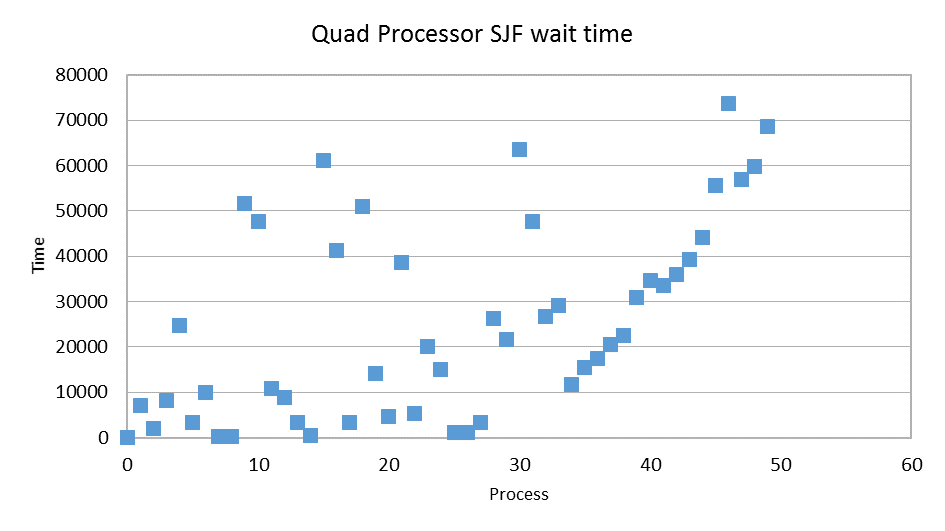
Our single-core simulation for Shortest Job First resulted in the following data:





As seen here, processes with lower cycles times are being processed first rather than processes with higher cycles times. Overall, this proved to have a much better wait time when compared with Round Robin and marginally better wait time when compared with First In First Out. It makes sense that throughput is best with Shortest Job First because shorter processes do not need to wait for longer processes to complete in order to begin executing. Fortunately we were able to not view any process starvation in these tests, but it would be interesting to play with that features with much more data.

Our quad-core simulation for Shortest Job First resulted in the following data:





As with previous tests, quad core systems provides much faster wait times and completion times. Over completion time is 4 times faster than with a single processor and wait time is much lower.

Overall, we have noticed that adding cores always appears to result in performance increases, and that the Round Robin method wastes too much time with context switching to finish processes in a reasonable amount of time (at least with the set of data that we are working with), thereby resulting in significantly longer wait times. Our FIFO and Shortest Job First tests ended with relatively similar wait times, though we suspect this is because we opted for the non-preemptive version of Shortest Job First, as opposed to the preemptive version. It would be interesting to conduct more tests on the preemptive version of Shortest Job First to compare with other processing methods.