**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 tested those processes through 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 gain by multicore programming, and to see what benefits (if any) were gained by more processing cores. Based off of our preliminary knowledge, we believed the SJF method would be the quickest, and the RR method being the slowest.

To explain our conclusions, we need to explain the mathematics behind our reasoning. We had three major equations to deal with calculating: Wait Time of a Process, Average Wait Time, and Average Completion Time.

Wait Time of a Process:

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 as follows.  
  
 Average Wait Time of a set of Processes:

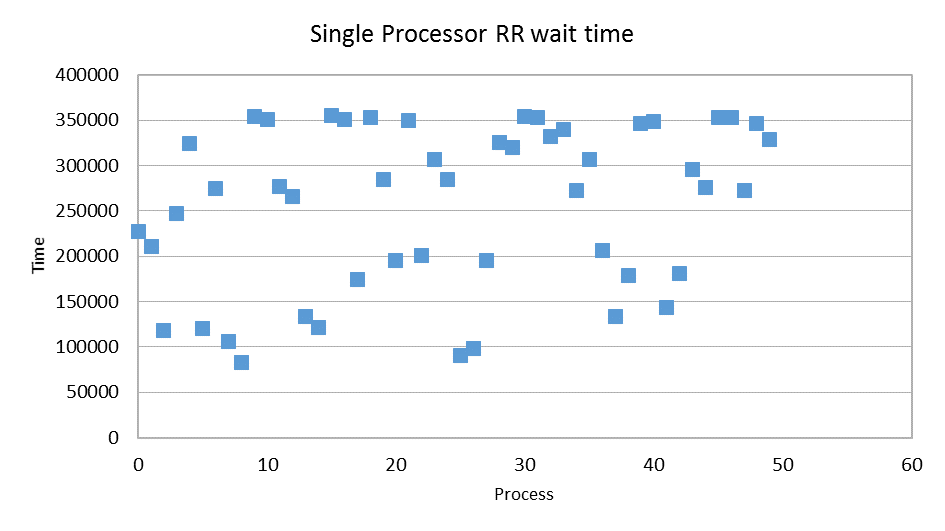
Now, with our Average Wait Time established, we need to find our Average Completion Time, which is concluded in a similar fashion.

Average Completion Time of a set of Processes:

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.

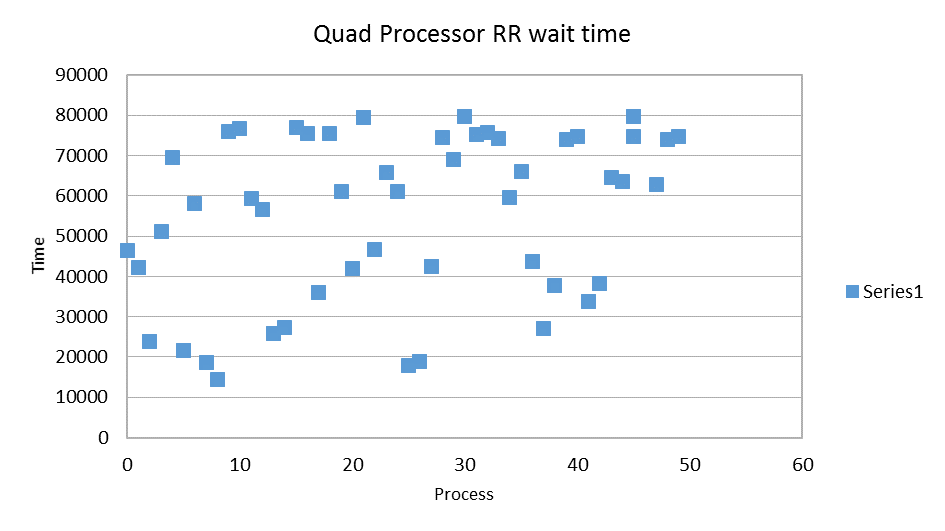
Initially, we began with Round Robin, which is a starvation-free, preemptive process scheduling algorithm that uses time slices to determine how long a given program is expected to run. In this program, processes are marked by how much “time” they've spent getting processed. Round Robin utilizes a “quantum”, 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:



Based on the data on 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 on a single core system.

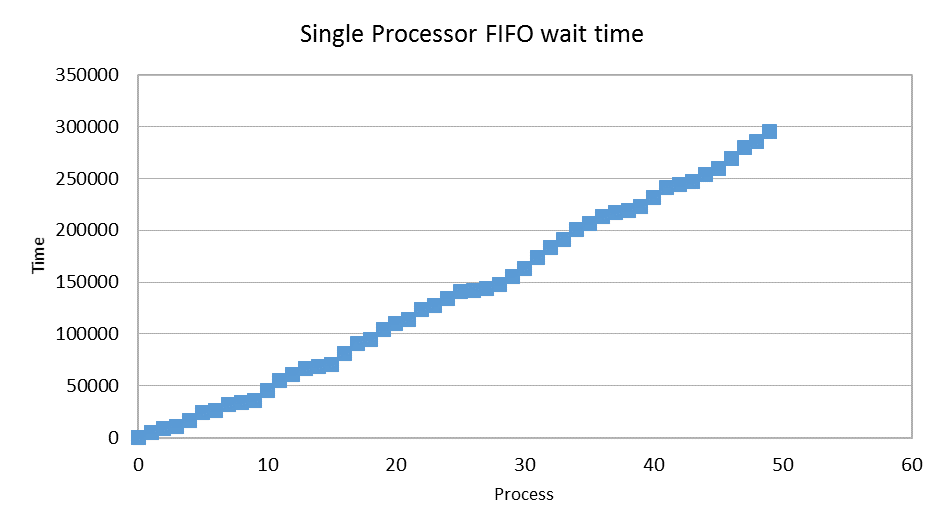
Our quad-core simulation ran significantly faster however:



Immediately, one can see how drastically the wait time has dropped. The range of 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 it will always be faster with a quad core in virtually every test.

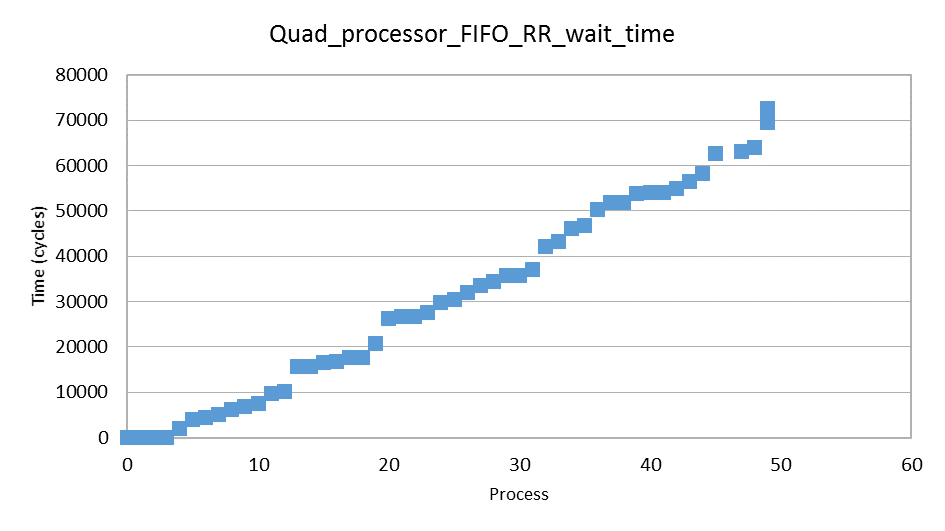
Our second type of process scheduling algorithm was the non-preemptive First In First Out method. We intentionally chose a non-preemptive scheduling algorithm to attempt a different perspective from the preemptive style of Round Robin. In terms of complexity, the First In First Out style is the simplest form of scheduling, as it simply runs a process as soon as its ready. There is no organization necessary of the ready processes. However, one can initially see some minor advantages, as well as some pitfalls of this method. In terms of negative aspects, you could reduce the average wait time if you attempted to run through the smallest processes first, and even Round Robin could theoretically allow for easy completion of small processes, if the quantum is sufficiently large. However, it makes up for this with 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. 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:



Unlike Round Robin, FIFO had a roughly linear increase in time proportional to how many processes were run. On our single-core, the range went from 0 to around 300,000. The Round Robin single core did not have processes that went as low as the FIFO method, but did have many that ended before the 300,000 mark. FIFO turned out to be faster overall given the current data.

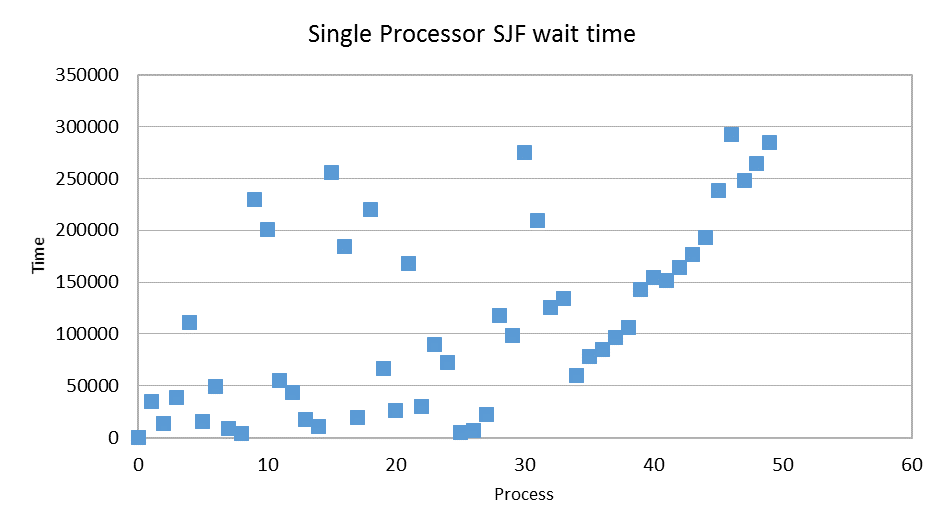
Our quad-core data for FIFO resulted in the following data:



FIFO turned out significantly faster with a quad-core than it did with a single-core. Our data maxed out around 75,000, compared to the 300,000 of a single core. 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 have a lower time, FIFO would still seem to be the optimal choice comparatively.

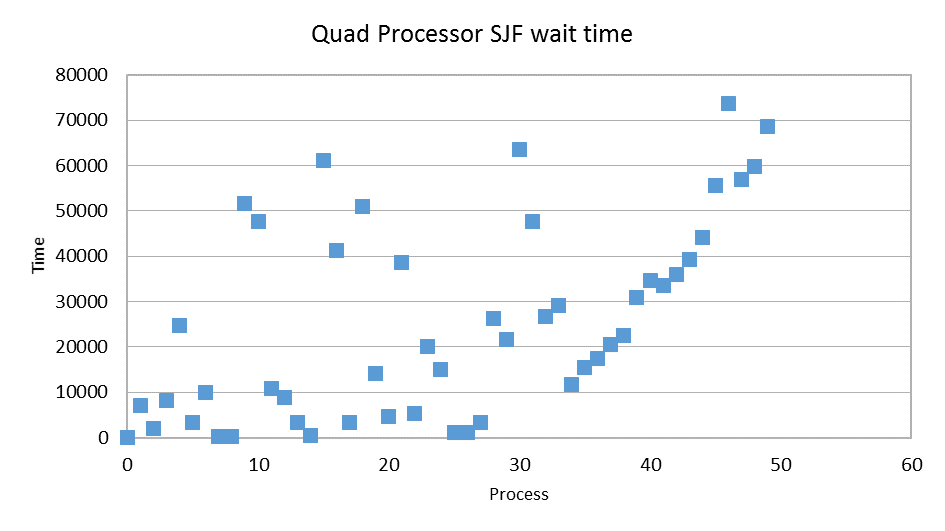
For our third type of process scheduling algorithm, we opted to use the non-preemptive form of Shortest Job First. Due to its non-preemptive nature, the processes are not constantly shifting between each other, which allows for a very low context switch penalty. As each process completes without stopping, the context switch penalty is equivalent to our First In First Out scheduling algorithm. In terms of overall speed, our non-preemptive one was marginally faster than our First In First Out algorithm, and significantly faster than our Round Robin methodology. 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 thirty (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:



Unlike FIFO, our Shortest Job First method was somewhat non-linear, but somewhat more coherent than the scatter plot provided by our Round Robin single core data set. Our data ranges from 0 to 300,000 in terms of the time, which places it right in line with FIFO, but each processor individually is a bit more scattered. It appears to be somewhat faster than the Round Robin methodology.

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



Our quad-core dataset ranges from 0 to 75,000, again in line with FIFO. Although still scattered, our quad-core results mirror our single-core results, but at a lower numeric scale. This displays the inherent increase in performance given by adding three cores to our system.

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, thereby resulting in significantly longer times. Our FIFO and Shortest Job First came out with relatively similar times, though we suspect this is because we opted for the non-preemptive shortest job first, as opposed to the preemptive version.