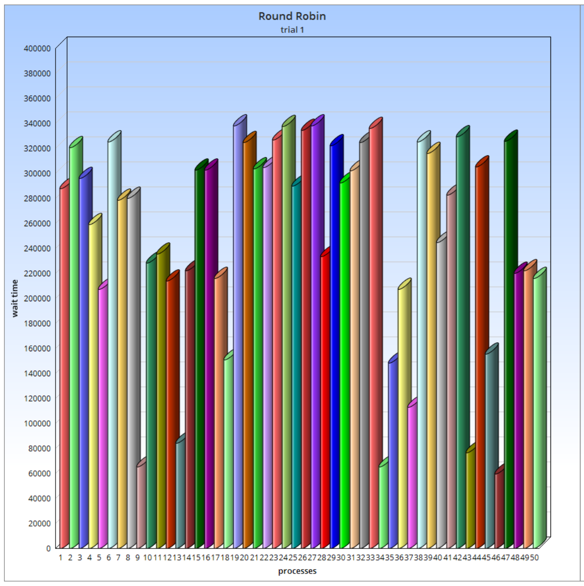
Kristian Forestier
John Alvarez
4600 Project 1
3/20/16

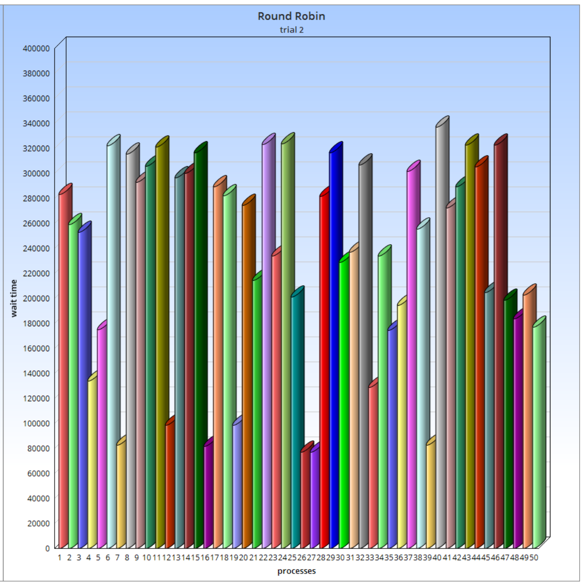
Project 1 Report

For our project we chose 3 different scheduling algorithms to test and compare. We chose Round Robin, First in First Out, and Shortest Job First. Our original program generated a simulation of 50 processes with: a PID from 1 to 50, multiple memory sizes, and multiple cycle times. The cycle times and memory sizes were randomly generated using the rand function but were also within the given scope. We stored all of this sorted data in a linked list and outputted that data in order of increasing PID. We also outputted the average data size and average cycle count for all processes.

We created the different scheduling algorithms in a separate c file and tested them with dummy input. Each scheduling algorithm was created as a separate function. They all printed the data at the same time when the program was compiled and run. Since we only needed the cycle count and the PID for the scheduling functions we moved that data over to a new data structure which also prevented accidentally modifying crucial data and allowed easier manipulation. After we were sure that the sorting functions worked properly with the dummy data we modified them accordingly in order for them to work with our actual data.

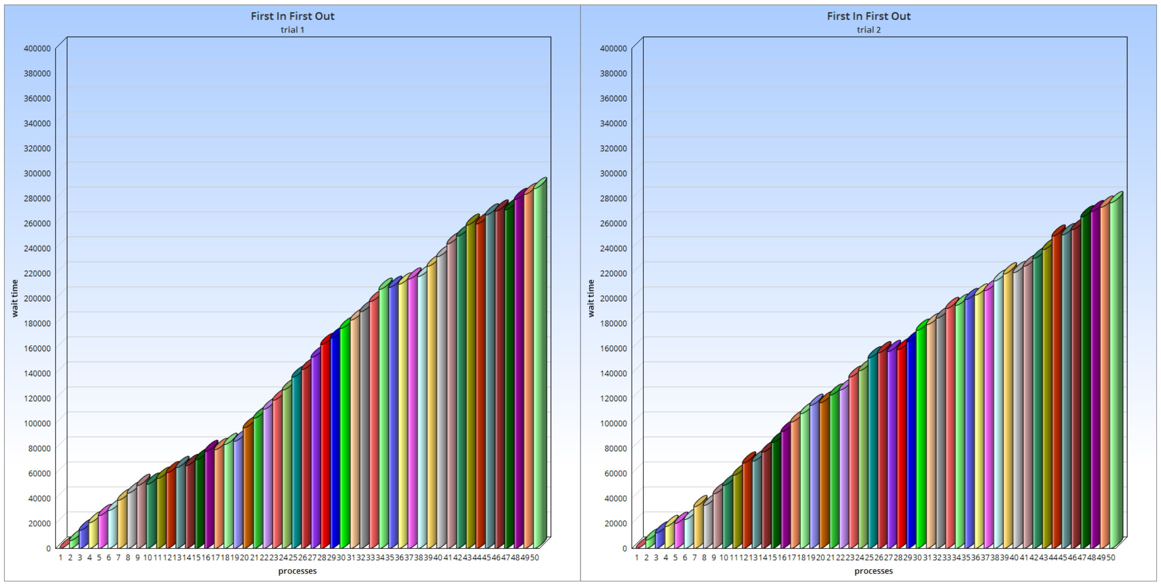
Here is our output for two different runs of Round Robin:





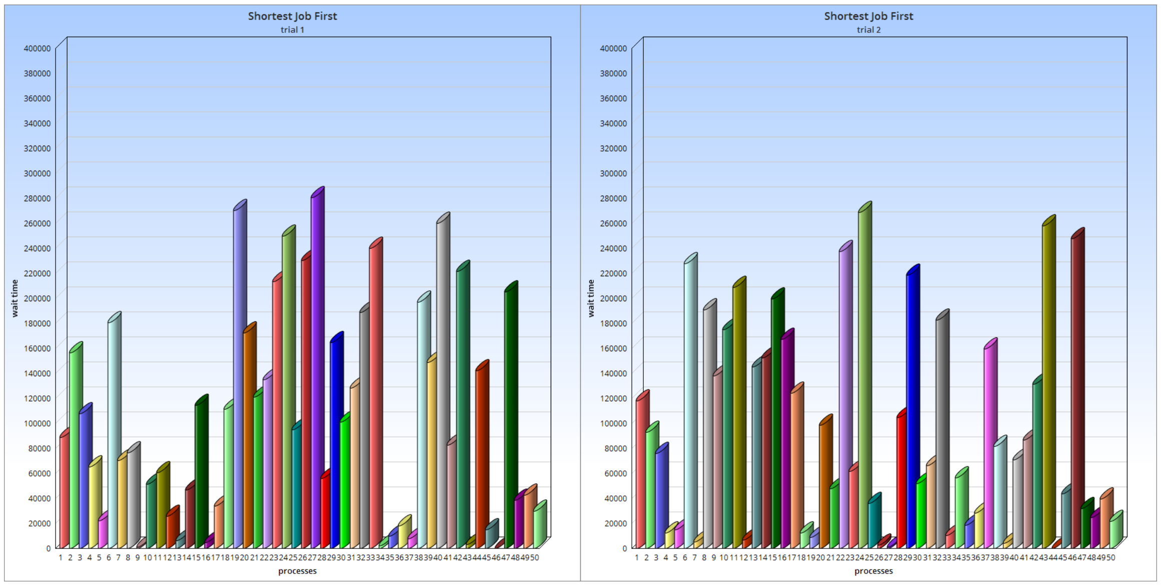
The first sorting algorithm we implemented was Round Robin. For this algorithm we used a for loop and an array in order to set the arrival times in increasing order and in increments of 50 starting from 0. We also stored the remaining cycle times of all the processes at this step in another array in order to keep track of them. We then used another for loop, which checked if the variable “remaining_process” (which was set to 50) was still greater than 0. This variable would be decremented each time a process finished. Our for loop contained many if statements checking the remaining times of each process that passes through it. If the remaining time of that process was less than or equal to the quantum slice and greater than zero, then we would subtract that quantum slice from that process and add that time to the overall time and set a flag saying that a process has finished for another if statement to be true. Else if the overall time was greater than the quantum slice and greater than zero then we would subtract the quantum slice from that time add the quantum slice to the overall time and add 10 to the overall time for the context switch penalty. The last if checked for the flag and if the remaining time of that process was zero. If this was true then it printed out that processes wait time and arrival time information. After the for loop was completed the average wait time was computed and printed out. This sorting algorithm gave us the longest overall waiting time of all of the other sorting algorithms.

Here is our output for two different runs of First In First Out:



The next sorting algorithm we implemented was First in First Out. For this algorithm we also used arrays as buffers. This one was much easier to implement than round robin because of its overall simplicity. For this algorithm we simply used a for loop and used the sum of the cycle times of the previous processes in order to get the wait times of the next processes. We stored all of this in a single array since we didn't need to worry about the PIDs getting mixed around. We then added 10 for the context switch within the first for loop. We then used another for loop in order to print this information out. Adding up all of their individual wait times and dividing by the total number of processes computed the average wait time of the processes. This sorting algorithm gave us the middle waiting time out of all the other sorting algorithms.

Here is our output for two different runs of Shortest Job First:



The last algorithm we implemented was Shortest Job First. For this algorithm we used a for loop to initialize all of the PID’s. We used another for loop to transfer the cycle times from our struct into a temporary array. We then used yet another for loop with “i” as an increment to check that it is less than the number of processes. We then used yet another nested for loop. We stored the times in our temporary array. We then copied these values over to some more temporary variables for the sorting. This all allowed us to move around the id’s alongside the cycle times of the processes. To calculate the overall wait time we just added up the wait times and divided by the total number of processes. We also added 490 to the total time for the context switch penalty. This sorting algorithm gave us the shortest average wait time of all the other sorting algorithms.