# **Project 1 Report**

# **Shortest Job First**

The shortest job first algorithm for the single core processor and the multi-core processor was implemented as a function of the process class. As a function of the class that stored all the available processes and their information, the algorithm could easily use the process information to schedule the processes.

In the single core SJF function, the algorithm initially checks the number of cycles that the first process to arrive requires. Each process arrives in the system every 50 cycles, so all processes will be already generated at 2450 cycles. If that process requires more than 2450 cycles than the first process to arrive may be executed, and all other processes will arrive while it is being ran. If the first process that arrives requires less than 2450 cycles the algorithm compares all the processes that have arrived that could potentially be ready to execute before all processes have arrived. If the first process is less than or equal to 2450, the algorithm checks if the number of cycles required by both the first process and the second process are collectively more than 2450 cycles. If so, the algorithm chooses the second process to arrive as the second process to execute. If not, it will execute the third process to arrive as the third to execute. After that initial check is performed the system can be sure that all processes have arrived and are in the queue. It can then begin sorting the processes in ascending order by their required number of cycles. The SJF algorithm implements the quicksort method of sorting the processes by their runtimes because of its ease of implementation and runtime complexity of O(nlogn). The vector that stores the processes is reorganized to have the shortests jobs first, so the SJF algorithm can increment through the vector and complete each process in the order they are placed in the vector. When a process is being executed the process is marked complete, the wait time is calculated for that process, and the penalty time is stored. The first process that is executed has a wait time of zero, then the wait time for each process is equal to the previous processes number of cycles plus the previous processes total wait time. After each process has been executed the total waits are added up and divided by the number of processes.

In the multi-core implementation of the shortest job first algorithm the first four processes that arrive are each assigned to a different core. The multi-core function distributes the processes among the four cores evenly. The goal of distributing the processes evenly is to ensure that each core is working on a process at all times to minimize the wait times. The algorithm checks the required wait times for the first few processes to arrive. If the initial processes that are assigned to each core finish running before all processes have arrived than it will assign the next suitable process to be executed on that core. After all processes have arrived and been assigned to a core, the algorithm will sort each list of processes in ascending order for each processor core. Then the processes are executed on each core and the wait times are added up in the same fashion as the single core processor.

# **Round Robin**

For the scheduling method of round robin, our code simulation utilized a class data structure to store the 3-tuple process(ID, service time, memory footprint). The class will also hold extra status variables where the scheduling code can record each process’s arrival time, record the total wait time for the process, and a Boolean variable for if the process has completed running. Each process will also have an int variable to keep track of its multi-processor assignment for when the program runs the multiprocessor round robin algorithm. On top of storing each process within a class, we also used a vector of classes to store all of the generated processes into a chronological list. All of these structures and variables are used to determine the round robin schedule for each system of processes that are tested and to calculate the average and total wait times for each process.

The method of simulating round robin that we used for this project was a function that required the number of processes in the system and the list of processes stored in a vector of classes. Several local scope variables are used as counters and trackers to monitor the cycling of the round robin algorithm. We used a completion counter that was used to count the number of completed processes throughout the scheduling. The completion counter was compared to the number of processes that was passed in through the function using a while loop, thus when the two number matched the scheduling algorithm was complete and all processes had finished running within the system. The next loop that scheduler used was a for loop that cycled through every single process(0…number of processes). The logic behind the for loop was to cycle through each process and then use an if statement to check whether or not the process being checked has completed running in the system yet or not. The algorithm will cycle through the whole list over and over until all of the processes are completed in the system.

The next step in our implementation of the round robin algorithm takes the process that has been identified as incomplete and subtracts the round robin time quantum from the number of cycles(or assigned service time) of the process. In our case, the time quantum is 50 cycles. The algorithm will then check if the process has completed running within the system by checking how many cycles it has left until completion(this data is stored in the process’s class). If the process’s number of cycles equates to 0 or is negative, then the process has finished running in the system and can be marked as complete, also adding to the number of process completed counter. If the process records a negative number of cycles, this stat is corrected to zero and the overflow from subtracting the time quantum is added back to the total wait time for all processes. This correction to both the process and the wait times occur due to the fact that the service time during that time quantum was less than the assigned quantum. If the process still has a positive number(above zero) of cycles left to service after the time quantum, then the process is not marked as complete and the scheduling algorithm continues.

The final step in our implementation of the round robin algorithm consists of actually calculating the total wait times for each process and every single context switch, which in our case every context switch equaled 10 cycles. Simply put, we called an outside function into the scheduling algorithm to calculate the wait times for each process in the system regardless of arrival time. It was easier in terms of design and implementation to subtract the arrival times after calculating the total wait times(after all of the processes have completed). This is another area where our simulated code was slightly limited in design. The wait time function is called after every single time quantum until all processes in the system have completed. The wait time calculator cycled through every single process in the system that was marked as incomplete, and added 60 cycles to each process’s wait time variable. 50 cycles represents the round robin time quantum plus 10 cycles for the fixed context switch justifies the 60 cycles being added. Every single process throughout each cycle required a context switch after it was serviced to transfer execution to the next process in the queue. If there was any overflow from a process that did not need the whole time quantum to complete, that value was subtracted from each process after adding the 60 cycles because that amount of overflow time did not actually take place within a real simulation. At the end of service, our simulation cycled through every single process one last time to clean up the total wait times and subtract the projected arrival times from each process’s wait time. The arrival time started at 0 for the first process and then incremented by 50 cycles after each process(0,50,100,150…) as stated in the project assignment. At this point, each process should have their total number of cycles equal to zero and have recorded the correct total wait times after scheduling. The scheduling algorithm also does the job of keeping track of how many context switches occurred after each time quantum in the system and calculated the total context switch penalty time. This value was returned by the scheduling function to main and output to the screen as data for the experiment.

After scheduling, the method of calculating the average wait time for all processes within the system was a simple for loop algorithm that cycled through each process’s wait time and adding them together into one total variable. This total variable was then divided by the total number of processes in the system and output as the average total wait time for this iteration of the experiment.

The multi-scheduling algorithm used to schedule a system of processes for 4 processors was used the exact same method as for the single processor. The only difference was that the algorithm was that the system of processes generated was split up into 4 separate groups to be scheduled separately. This meant that a smaller number of processes(12-13 processes) was being cycled through the scheduler and outputting smaller stats each time compared to the single processor of 50 processes. The scheduling algorithm was called 4 separate times each time with a different set of processes to schedule. But the round robin method was still used the exact same as it was simulated with the single processor.

# **First in First Out**

For the scheduling method of first in first out, our code simulation utilizes a 4-tuple structure containing an identifier (ID), the number of cycles to complete, the memory footprint, and the wait counter for each process. The ID is unique for each process. The overall number of processes is defined as a global variable at the top of the program since the majority of functions use it. Each experiment we conduct sets the number of processes at the beginning of the function. For the different experiments, we use three (3) different sets of process characteristics (as explained below) and schedule them for a single core processor and a quad-core processor. Lastly, each experiment returns the context switch penalty and the average wait time. These numbers are then printed to the screen for the user to read and document.

First off, we utilized a process generator function that once given a process structure and an unique ID would generate a “fake” process with: the cycle count varying from 1,000 to 11,000 with a mean of 6,000, the wait counter initialized to zero (0), and the memory footprint in the range of 1KB to 100KB with a mean of 20KB. To properly calculate the weighted memory footprint, we used a conditional statement to allow 20% of generated numbers under the range from 20KB to 100KB and 80% of generated numbers are under the range from 1KB to 19KB. With the given distribution of pseudo-random numbers above, the final memory footprint now has a weighted mean of 20KB. (Without this “weighted” technique, the average would be left to the random number generator and would be around 50KB) Finally, the wait counter is set to zero (0) and the next process is ready to be generated until all of the “k” number of processes are generated. We are now ready to run our experiments with our new fake processes.

Our FIFO program uses six (6) unique functions that conduct three single core experiments and three quad-core experiments. Each pair of experiments (functions) 1 and 4, 2 and 5, and 3 and 6 takes a list of k processes with the same exact characteristics and schedules them. For experiments 1, 2, and 3 we schedule the processes on a single core and the other experiments are scheduled for a quad-core processor. Each function calculates the wait time (explained in detail below), calculates the context switch penalty (also explained in detail below), and then prints each process to the screen once its cycle count equals zero (i.e. the process is finished running). After scheduling, running, and printing all of “k” processes, we print the total context switch penalty and the wait time for each core individually. Lastly, for the single core experiments, we simply return the average wait time to the main function. For the quad-core experiments, we sum the wait times from each core, divide that sum by four (4), and then return the average wait time to the calling function.

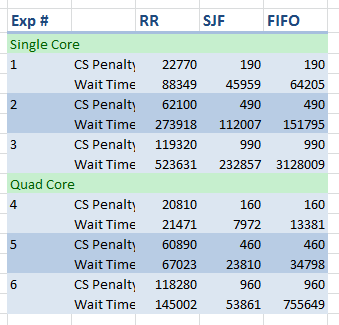
To calculate the average wait time for the single core experiments, we start off with a wait time of zero (0). We know that a new process arrives every 50 cycles. For a new process, the wait time is calculated by summing the number of cycles after the process has entered the ready queue. Our program does this by summing all of the previous’ processes cycles and then subtracting the entrance time (that is, a multiple of 50:0, 50, 10, 150…). Additionally, we have to take 10 cycles every time a context switch occurs between two process and add it to the wait time for that process. For the single core experiments, the wait time per process is complete. We can now sum up all of the k processes’ wait times, divide it by k, and return it to the calling function. Similarly, for the quad-core experiments, the process to calculate the average wait time per core is the exact same as the single core experiments. In this case, the initial four (4) processes will start on each core at cycle 0, 50, 100, and 150 respectively. The average is then calculated for each core individually, summed up, and divided by the number of cores (4). This final average wait time is returned to the calling function to be printed for the user.

For this project, a context switch takes 10 cycles. Given the non-preemptive nature of the FIFO scheduling algorithm, all single core experiments with k processes will have (k - 1) \* 10 cycles of context switch penalty since the next process will always be in the ready queue ( this is true since the arrival time (multiples of 50 cycles) is much lower that the actual run time (thousands of cycles) of the processes, so in this case, it is assumed that there will always be a process in the ready queue). Also, the first process per core does not need a context switch and all of the latter process do. For all quad-core experiments with k processes, (k - 4) \* 10 cycles of context switch penalty will occur since the first process on each core will not need a context switch. After calculating the context switch penalty, each experiment then prints the context switch penalty to the screen.

# **Conclusion - of the overall project**

We simulated three (3) sets of processes on a single core processor using three (3) different scheduling algorithms: Round Robin, Shortest Job First, and FIrst in First Out. Additionally, we simulated those experiments on a quad core processor and recorded the data for a total of eighteen (18) simulations.

After scheduling the processes and running all experiments, we logged the average wait time and the context switch penalty for each experiment. Since we used the random number generator (as explained in the Random Generator section) our experiments used a different set of processes with the same predetermined bounds as defined in said experiments. Therefore, running a single experiment multiple times will yield similar results every single time. Our implementation of the C’s default random number generator provided a certain degree of accuracy towards our experiment. To improve this accuracy, we could seed the random number generator with a specific value. We could go a step further and utilize a non-pseudo-random number generator via seeding to generate more precise numbers that remain the exact same for each algorithm. However, given the time constraints of this project and complexity of a real random number generators, C’s default “rand” function provided us with the workable results below.



*Figure 1: A table of the results of the scheduling algorithms tested.*

**Compare**

According to our final results in figure one (1) above, the SJF and FIFO scheduling algorithms have the same number of context switch penalties across the board. This is true because they both are non-preemptive and both ran the same number of processes across every experiment. SJF and FIFO were very close in average wait times too. However, SJF always had slightly lower average wait times than FIFO. Comparing the single core results to the quad core results for each scheduling algorithm, all quad core experiments were around four (4) times faster. This speedup is understandable since we are utilizing four (4) cores instead of just one (1). Likewise, as we changed the process characteristics and increased the number of processes, the total average times and context penalty times increased. The number of context switches for RR’s single core experiments were nearly identical to its own quad core experiments. RR and FIFO both do not need to know the runtime of the new incoming processes.

**Contrast**

First, let's analyze how Round Robin is different from Shortest Job First and First in First out. RR has a better response time than both SJF and FIFO (each process is advancing a little bit each “round” as opposed to waiting the entire time for a process). RR has substantially more context switches resulting in the longest context switch penalty times as compared to the others.

RR’s wait times for single core processes is double that of SJF, and its wait times for the quad core processes is triple that of SJF. Arguably, RR was the hardest to implement when programming. RR is the only preemptive scheduling algorithm we tested.

Next, we will analyze how SJF is different from RR and FIFO. After a quick review of figure one (1), we can determine that SJF has the lowest wait times of all three (3) scheduling algorithms. SJF has slightly lower wait times than FIFO and substantially lower wait times than RR. Unlike RR, SJF is non-preemptive and it has to know the runtime of the job before execution. On a real system, knowing the exact number of cycles before the process is running may not always be possible. Therefore, the SJF algorithm only applies to systems where the process’s cycle count is known beforehand. As opposed to RR and FIFO, the SJF algorithm can lead to process starvation. If multiple shorter processes continue arriving they can “cut in line” in front of the longer processes and keep them waiting.

Lastly, FIFO will be contrasted with RR and SJF. Most notably, FIFO was is the easiest to implement since the scheduler does not have to calculate which process to run next. FIFO has slightly longer wait times than SJF and much shorter average wait times and fewer context switches that RR. Unlike RR, FIFO is non-preemptive.

**Analysis**

To analyze the results, we took the context switch penalty and average wait times from every simulation and combined them into a table: figure 1. Based on our data in figure 1, we can infer what the SJF scheduling algorithm has the overall smallest average wait times across the board. Likewise, FIFO and SJF tied for the fewest number of context switches. However, in a realtime system, RR should perform better since it will not lead to process starvation (SJF could) and it provides a more responsive system since an equal portion of each active process is ran each time around. For example, let's say a processor is utilizing the SJF algorithm and a number of processes are added to the ready queue and process “73” is the longest process. Then, if an endless number of shorter processes are added to the ready queue, process “73” will be starved and never get to execute. In a RR or FIFO algorithm, this will never happen. Back to figure 1, if we take the Single Core rows containing the Wait Times and divide them by the Quad Core rows containing the Wait Times, we get average speedup multipliers of 3.9, 4.9, and 4.4 for RR, SJF, and FIFO respectively. These numbers average to 4.4 which is acceptable since transitioning from a uni-core to a quad-core should be around four (4) times faster.