Project #1 Report

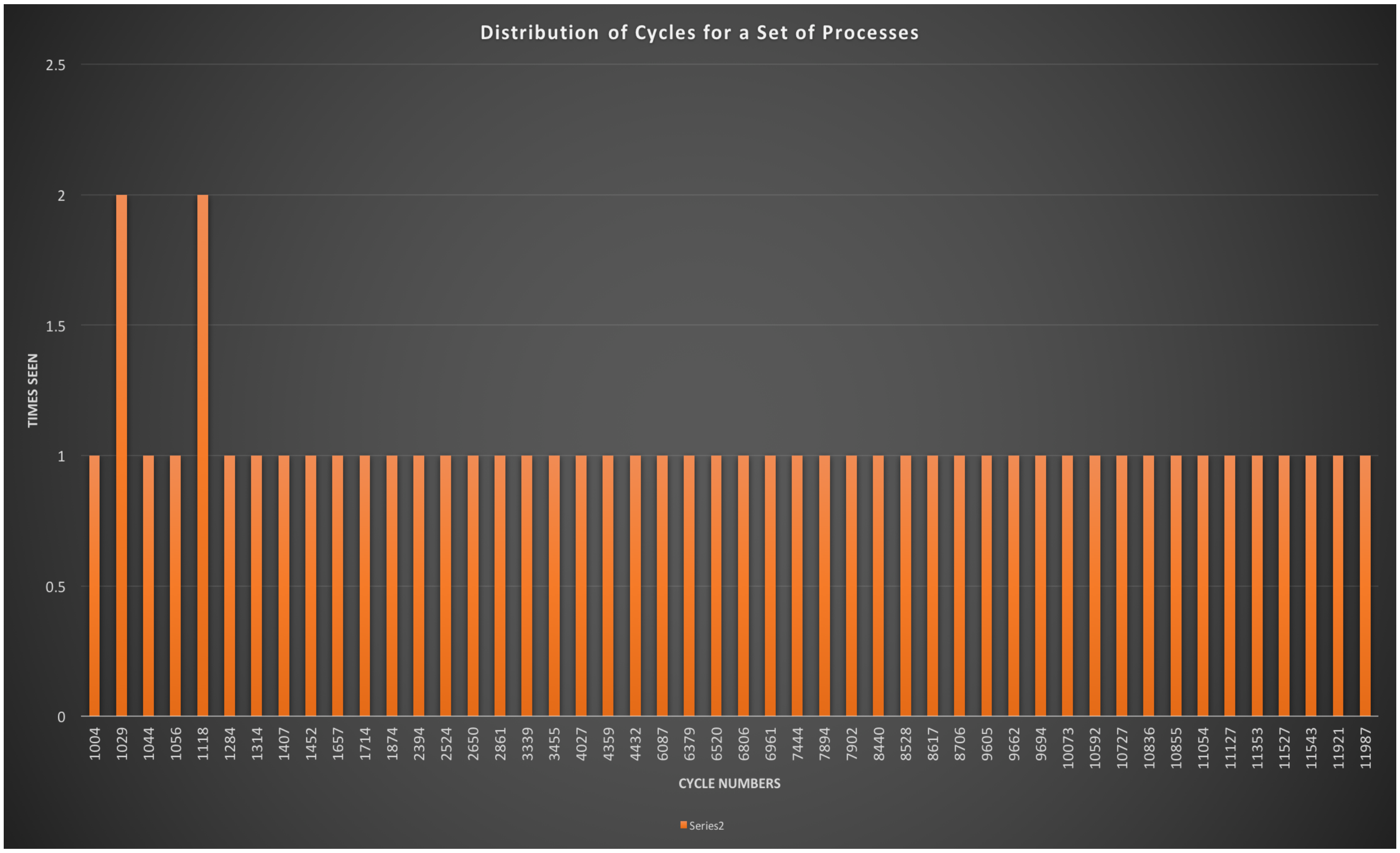
By: Jacob Cole, Scott McKeefer, Mark Doggendorf

Introduction

The purpose of this project is to simulate different scheduling algorithms at work. The algorithms we used were FIFO, SJF, and RR.

We created a process class which held the arrival time, wait time, pid, number of cycles and memory footprint. Then each process object was stored into a “map” C++ container and referenced by its pid.

For all of the algorithms we have represented time as increments of cycles, this assumption was made because requirements defined in the project express arrival time in increments of fifty cycles. We may refer to time in the form of cycles instead of the traditional increments.



Algorithm: FIFO

FIFO is a non-preemptive process scheduling algorithm. This means that processes that arrived first have priority of the processor over those that arrive later. This algorithm is fairly easy to implement, but suffers from high average wait times.

Implementation

This algorithm was implemented by looping from the first process to the last process. Since each process arrives 50 cycles after the previous one, the order of the processes is already correct for FIFO. First, the wait time of each process is determined by the current overall execution time of the system minus its arrival time; waitTime = waitTime - arrivalTime. Then, the overall time is increased by the number of cycles of that process. Next, the total wait time of the system is increased by the current process' wait time. In other words, the total wait time is just a sum of all of the process' wait times. Lastly, the overall time is increased by a context switch if not currently on the last process. Also the process id, number of cycles, and wait time for each process is printed, so you can clearly see the correct timing.

Performance

For a set of 50 processes, the average wait time seems to be consistently be around ~150,000 to ~160,000 cycles. This results in an average wait time about 3000 to 3200 times larger than the number of processes which is 50. The reasoning for this can be given to The total context-switch penalty remains the same at 490 cycles, 49 context switches. This is because the first process in the processor does not have a context-switch and the rest of the 49 do have a context-switch.

Algorithm: FIFO with four processors

This is FIFO, but implemented for a system of four processors. In theory, this will result in a shorter average waiting time.

Implementation

We decided to have each processor only deal with a specific order of processes. So given pid 1 through 50, it will always put them on the same order of processors. This works because each process is 50 cycles after another, and each process has at least 1000 cycles to do. We started by having a loop that continues until the current pid is less than the total num of processes (50 processes total). Each CPU manages its own overallTime (in this case amount of time servicing processes), how many processes have been through it, and its own totalWait time. The waitTime of each process is the same as FIFO for a single processor, except that it resets the waitTime to 0 if it becomes less than 0. If it's less than 0, that means the process hasn't arrived yet so by setting it to 0 it doesn't add to the totalWait time of the processor. Also each line of our output is the iteration of what a processor is doing at the current time. For example, given 5 processes that with pid of 1, 2, 3, 4, 5 respectively, the first line would only have process 1 since it's time 0 and no other process is there yet. The first line would look like "| CPU1: pid= 1, Cycles= 9671, WaitTime=0 | CPU2: pid= 0, Cycles= 0, WaitTime=0 | CPU3: pid= 0, Cycles= 0, WaitTime=0 | CPU4: pid= 0, Cycles= 0, WaitTime=0 |." The next line would have the same format, but CPU1 would have pid 2, CPU2 would have pid 3, CPU3 would have pid 4, CPU4 would have pid 5. Clearly, the wait times and cycles would change as well.

Performance

As hypothesized, this results in a shorter average waiting time for a process in the whole system of processors. For a set of 50 processes, the average wait time of the system is about 30,000 to 37,000. This results in an average wait time about 600 to 740 times larger than the number of processes (50 processes). This is clearly better than the 3000 to 3200 range for the single processor FIFO algorithm. The context switches here will always be the same similar to the single processor FIFO. The only difference is that the total context-switch penalty here versus the single processor FIFO is always 460 cycles, 46 context-swtiches. Otherwise, it behaves the same per processor. This is because there are 3 additional processors that are free at overallTime of 0. This in turn reduces the context switches by 3.

Algorithm: SJF

The shortest job first algorithm schedules the shortest available process and runs it to completion. Since each process runs to completion it is non-preemptive, this means that processes can be subject to starvation or waiting infinitely as shorter processes continuously enter. In our case we run fifty processes so although this may result in large wait times it does not result in deadlock although if we define starvation as "a process that is perpetually denied necessary resources to process its work" (Tanenbaum, Andrew), then we may refer to this as short term starvation.

Implementation:

For shortest job first I implemented a total time variable that allowed me to keep track of the total time in cycles of the SJF implementation as a whole. Since the first process arrived before all others I would always run this to completion regardless of size. Once this process entered and began running I could calculate the runtime in cycles of the process and add this to total time. Total time could then be divided by fifty to find the processes that have entered the program and from there I would calculate the shortest process of those processes to be ran next. Wait time for each process was then easy to calculate with total time at the beginning of the running of the process being subtracted by arrival time of the process.

Performance:

From the values in our program SJF was shown to result in better performance when scheduling by reducing wait time. On the other hand it has a chance of resulting in starvation and because it runs to completion on a process if an incredibly large process enters early on it may result in early starvation instead of rescheduling. In a bar graph presented at the end of this document we can see that SJF seems to have the lowest average wait time calculated by (overall time at the beginning of the process minus arrival time).

Algorithm: SJF with Four Processors

For this part of the project we implemented shortest job first exactly like SJF on a single processor but we simulated three more. This allowed us to share the workload and selectively choose the most available processor to run the next available process.

Implementation:

With four processors the implementation is functionally the same but in this case we must keep four new total times so that we can calculate the first free processor as well as the wait time individually based on processor. Since we output after the running of each process we are able to print which processor is being run on following the running of each process.

Performance:

Performance from the extra processors allowed large gains in effectiveness. Later in the report a bar graph representation shows one of the iterations of each method and how it compared to the non multi processor version. From this graph and from running the program we can see reduction in average wait cut down by 4/5ths. With the simulated quad core CPU running at roughly 20% of the single processors average wait time. Each processor is able to simultaneously work on its own process and since SJF in non-preemptive we do not have to worry about one process being grabbed by a new processor. This also reduces context switching resulting in less of a context switch penalty. Overall the performance is good comparatively to the other scheduling methods we chose. This does not resolve the issue of potential starvation however in our case we can assume it will not be a large problem due to running a set amount of fifty processes.

Algorithm: Round Robin

Round Robin is a process scheduling algorithm that use time slices to work on every process in equal amount every cycle. It is similar to FIFO in the sense that process are worked on in the order that they arrive but based on arrival time the shorter jobs will finish first.

Implementation:

Thanks to the arrival time and the quantum size having the same value of 50 cycles the next process would arrive by the time the previous process had finished. In the event that there was not another process in the round robin the currently process will continue to be worked on in 50 set chunks. Whenever there is less than 50 cycles to process then the program will switch over to the next ready process. Every time the process is being worked on it updates it wait time. The algorithm for the wait time is waitTime =overAllTime –(arrivleTime+(quantumSize * numOfTimesProcess)).

Performance:

The wait time is evenly distributed relative to the cycle time of the process. In others words there is a linear relation between cycle time and wait time as cycle time goes up the wait time goes up. There are also a lot more context switches than the other two schedulers because it switch processes every 50 instead of at the end of the process.

Algorithm: Round Robin with four processors

This is similar to the the one processor but it has to take into account that each processor has a set of values that it processes. For example if there are only 8 process then processor 1 will switch between process 1 and 5, processor 2 will switch between process 2 and 6, processor 3 will switch between process 3 and 7, and processor 4 will switch between process 4 and 8.

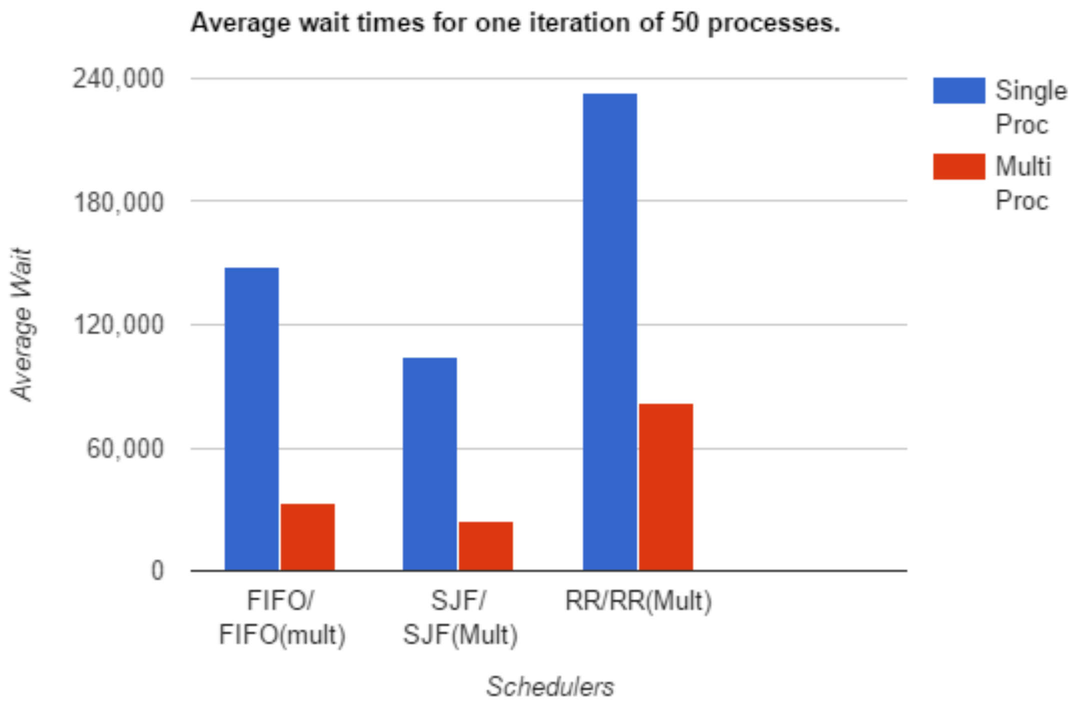
Implementation:

Once again the quantum size and the time between arrival is 50. Each processor will only only be responsible for certain process and have their own total run time based on the process they work on. This could lead to one processor finishing before another but it would affect the overall time or the average wait time. Wait time is still calculated the same way as round robin on one processor but taking into account the current processors over all time.

Performance:

This method was so much faster thanks to have more than one processor. There could be other method to distributing out the process to the other processors to optimize round robin but it is still slower than the other two method because it tries to linearly distributed out the wait time. There are less context switches because I limited the number of process that each processor worked on.

A representation we have made of the average wait times from one iteration of each scheduling method with 50 processes. As mentioned before all number representations are in cycles.



Sources used:

Tanenbaum, Andrew (2001). *Modern Operating Systems*. Prentice Hall. pp. 184–185. ISBN 0-13-092641-8.

Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau(2014). *Operating Systems: Three Easy Pieces*

Arpaci-Dusseau Books