Adrian LaCour Brad Dennis

Process Scheduling

Process scheduling is a process that many people never consider, as it happens seamlessly in the background every time we interact with a computer. To more deeply understand this concept, Brad Dennis and I, Adrian LaCour, developed several programs to emulate process scheduling. Process scheduling is a core concept in operating systems, so understanding how and why the computer does the scheduling will provide us a more holistic view of the operating system concept.

To better understand these concepts, the following tests were performed:

- With 5 identical processors (same speed and memory), minimize the turnaround time to complete 200 processes.
- With 5 processors of different memory, with processors 1 and 2 having 2 GB, 3 and 4 having 4 GB, and 5 having 8 GB, minimize the turnaround time of the 200 processes.
- With 5 processors with the same memory but different speeds, with processors 1 and 2 having 2 GHz, 3 and 4 having 3 GHz, and 5 having 4 GHz, minimize the turnaround time of the 200 processes.
- Develop a scheduling algorithm that deals with the sequential arrival of 200 processes rather than having all the processes known beforehand. Minimize the turnaround time.

To complete these tasks, we use the shortest job first (SJF) algorithm when completing the processes. This algorithm works by checking all the processes, then assigning the process that will require the least processing time to an available processor. This algorithm gives the lowest turnaround time amongst the well-known algorithms. While this algorithm can introduce problems, such as starvation, when a lot of smaller processes are continuously given to it, this is not a problem for our purposes, as we have a set number of processes.

Test 1 - Identical Processors

For this test, all the processors have the same speed and memory. The 200 processes that are being assigned are all generated randomly, so assignments cannot be directly assigned to processors. To satisfy the SJF conditions, the processes are all sorted beforehand, in ascending order of the required cycle counts. To ensure the lowest wait time, the processes were assigned to each processor one after the other. So, process 1 is assigned to processor 1, process 2 is assigned to processor 2, and so on. This, rather than assigning 1/5 of the processes to the first processor, the next 1/5 processes to the second, and so on, ensures that the last processor does not have to process all the larger processes.

The scheduler we created operated at close to the optimal assignment. It was roughly 12% off, but that is because we could omit context switching. If we were to put in the context switching, allowing a processor to pick up another process while it still has available cycles left after completing a process, it would have operated much more efficiently.

Test 2 - Processors with Varying Memory

This test is the same as the first one, but with some memory requirements for the processors. Processors 1 and 2 have 2 GB, processors 3 and 4 have 4 GB, and processor 5 has 8 GB. These were taken into consideration when assigning the processes to the processors. Each of the processes had a memory requirement that was randomly generated, between 0.25 MB and 8 GB. For this task, the 200 randomly generated processes were sorted not only by burst time but also memory requirements. SJF was still the best scheduling algorithm, so we sorted the process by cycle time in ascending order similarly to the first test. However, this time memory requirement was also an issue so we used pythons included lambda sort. Lambda sort took both cycle time and memory of each process and would sort them based on the size difference of both values between each process. We then had to add limits to each processor to only take processes that were equal or below their memory. Additionally, a counter was added so that the processor with the same memory size (1 and 2, 3 and 4) would split the processing load fairly.

Test 3 - Processes with Varying Speeds

This test is the same as the first one, but the processors have different speeds. Processors 1 and 2 run at 2 GHz, processors 3 and 4 run at 3 GHz, and processor 5 runs at 4 GHz. These speeds are faster than any of the previous processor speeds used, for the other tests. The trick that this test has is that the larger processes should be assigned to the fastest processor. This introduces the challenge that the assignment of processes must still follow the SJF algorithm but ensure that the largest processor is given to the fastest processor first. We solved this by sorting by the processes by clock speed in ascending order as we've done in the earlier tests. Then we used changed the limiter from test 2 from memory to clock speed. This is so only clock speeds equaling or below a processors clock speed will be assigned. Also from test 2, we carried over the counter, so that processes would be assigned evenly among all the processors.

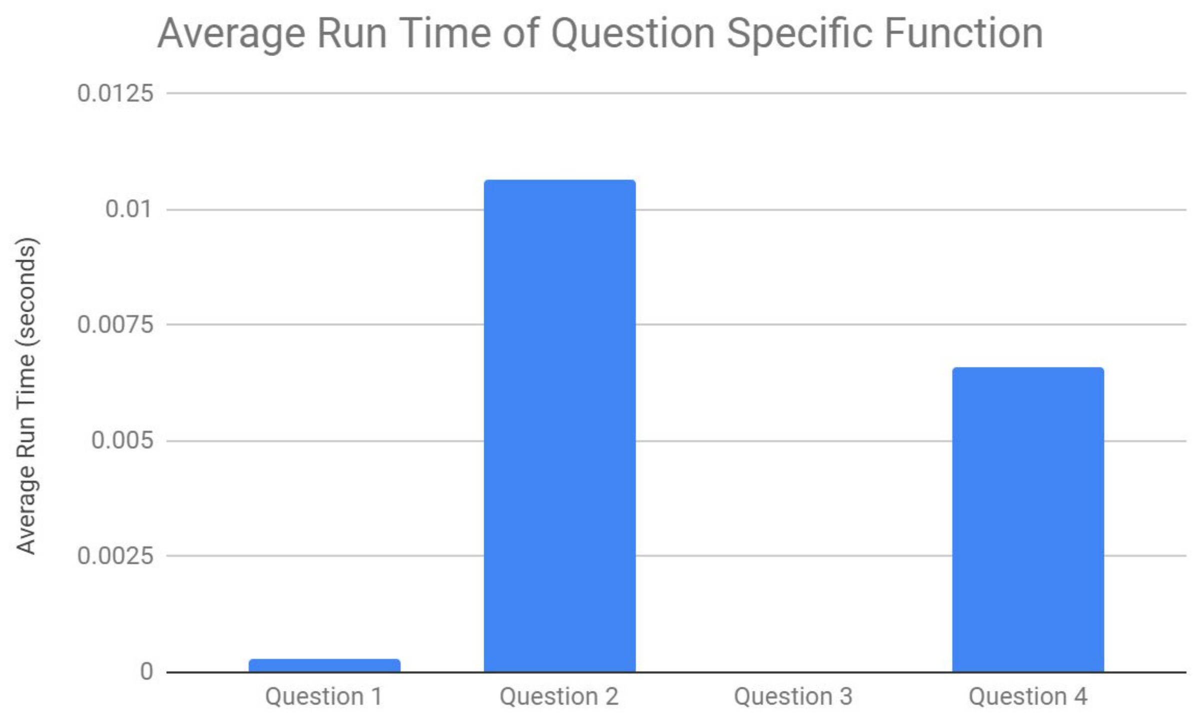
Test 4 - Sequential Arrival

This test is like the first test in that the speed and memory capacity of each processor is the same, however this time the processes are not known beforehand. Thus, the processes cannot be sorted beforehand and must be handled by the processes as they arrive. In order to solve this problem we simply had each processor keep a grand total of all the cycles it had processed. We would assign a new process based on the "laziest" processor, or the one that had done the least amount of cycles. This would assure that even if a processor had to handle a really big process, the other processor would be able to pick up the slack.

Comparison

Our program was written using Python 3. Initially, we used a more precise timekeeping function, but the python version on the UNT CSE machines is less than 3.3, where the improved timekeeping function was introduced. Due to not being able to use the most precise timekeeping function, we were not always able to attain a runtime for the faster processor clock speeds. Because of this, we used a 1 GHz clock speed for all the tests, except for test 3. We ran each test 200 times and calculated the average turnaround time for each process. The results are below:

	Test 1	**Test 2**	**Test 3**	**Test 4**
Average Run Time (seconds)	0.00028208	0.01064	0.00000000532866	0.006611



As you can see in the graph, the runtime for test 3 is the smallest. However, this is due to all the processors being faster than the ones in the other tests. So, from this, we can see that being able to assign the processes freely greatly speeds up the turnaround time. When there are memory requirements that must be satisfied, more time must be taken for specific processes, which may not be optimal for the SJF algorithm. For example, if all 200 processes had an 8 GB memory requirement, all the processes would have to be processed on the 4th processor. This would lead to terrible average turnaround time. Additionally, having to assign the processes as they come in, as seen in test 4, takes much more time, as they are not scheduled optimally.

Conclusion

In conclusion from our tests we discovered that the more limitations added to the assigning process, whether they be memory restrictions or not being able to sort at all, the SJF algorithm suffers greatly in terms of turnaround time. On the other hand, in order to maintain consistency with our tests, we could not increase our processor clock speeds with the exception of test 3. In test 3 the turnaround time was much faster than any other test, and that is just based on the increased clock speed. So, it would be a safe assumption that if the increased clock speed was applied to the other tests, the impact of the other limitations would be much less.