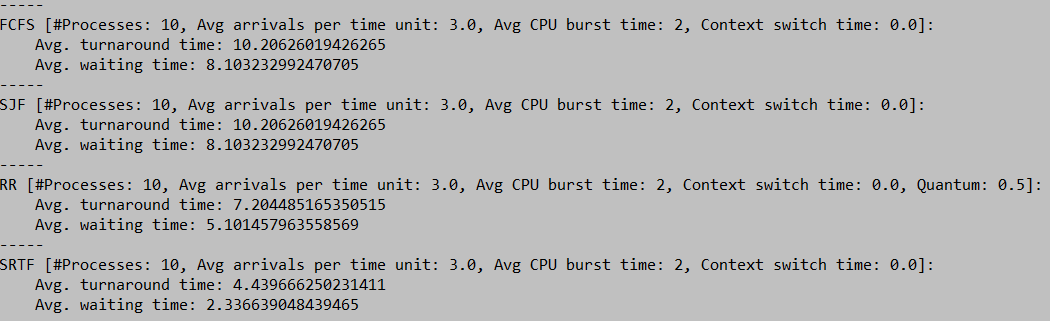
Scheduling Algorithms

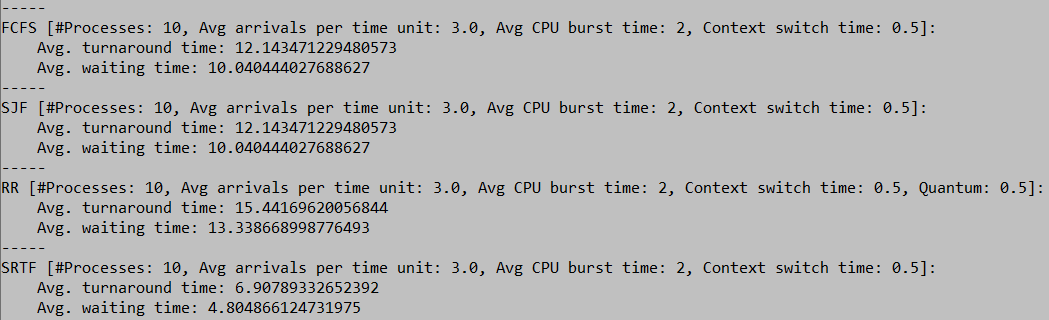
In computing, scheduling is the method by which resources are assigned to work as a means of completing that work. This work covers potentially virtual computation elements such as threads, processes or data flows, which in turn, are scheduled onto hardware resources, namely processors, network links or expansion cards. The manner in which resources are allocated to work, specifically processes with regards to the context of this assignment, is known as a scheduling algorithm. For the purposes of this course, there are four main scheduling algorithms: First-Come-First-Serve/First-In-First-Out (**FCFS/FIFO**), Shortest-Job-First (**SJF**), Round-robin (**RR**), Shortest-Remaining-Time-First (**SRTF**).

1. **FCFS/FIFO**: FCFS/FIFO is a non-pre-emptive scheduler, which prioritizes processes by the order in which they arrive in the system and adopt a ready state. After completing the first to arrive process, the dispatcher terminates the process, and then the scheduler passes the next arrived process to the dispatcher function.
   1. **Scope and limitations**: FCFS is a non-pre-emptive algorithm, meaning its process priority is insignificant. If a process of low priority is being executing, such as a daily routine backup process, which takes significantly more time compared to other processes, and a high priority process arrives, such as an interrupt to avoid a system crash, the high priority process will be forced to wait, thus causing the crash. As a result, the average waiting times aren’t optimal, and the resource utilization is not possible in parallel, leading to a convoy effect, where processes requiring resources for short time slices are blocked by one processes holding the resources for a prolonged period of time.
2. **SJF**: SJF is a non-pre-emptive scheduler, which prioritizes processes by their respective service times. Processes with the shortest service times will be passed by the scheduler to the dispatcher, and, likewise, processes with the longest service times will be passed last.
   1. **Scope and limitations**: Can lead to starvation. This is owed to the fact that a shorter process will have to wait for a significantly longer process to complete before execution can occur. This is due to the fact that this algorithm is not pre-emptive, and will not re-assess the queue of processes should a new, shorter process arrive. This can be solved with concept of aging and making the algorithm pre-emptive.
3. **RR**: Round robin is a pre-emptive scheduler. Processes are, on arrival, added to a queue. The algorithm will, without significance, select the first process in the queue and run it for a time slice, or quantum. If, at the end of this time slice, the process still has time remaining before completion then the dispatcher will pass the process back to the end of our queue. Otherwise, the dispatcher will terminate the process.
   1. **Scope and limitations**: In round robin scheduling, the performance is largely based on the time quantum. If it is significantly larger than most processes in the system, the behaviour exhibited will be that of FCFS. If it is significant shorter than the service times of most processes, then the CPU switches will increase, thus decreasing CPU efficiency.
4. **SRTF**: Shortest-Remaining-Time-First is a pre-emptive scheduler of the SJF algorithm. This algorithm operates just as the SJF algorithm does, but once a change occurs to the system or queue, such as a process turning into the ready state, then the algorithm compares whether the remaining time of the current process is less than that of the newly changed process, while taking into consideration the context switch time! If the result demonstrates that the new system, with consideration of the context switch time, has a lesser service time than the remaining time of the current process, then the algorithm will switch process, otherwise it will continue to work the current process.
   1. **Scope and limitations**: Although in the evaluation below SRTF seems to substantially outperform the remaining 3 algorithms, concerns still exist. The main issue is that processes with a long burst/service time, will have to wait for long periods of time before execution. Additionally, if there are frequent switches to other, newly introduced processes in the system because their service times are less, and the context switch time is large for the system, it could create inefficiencies.

Seed Result (1): 1797410758



***Figure 1.: Using Seed 1797410758 with a context time of 0.***



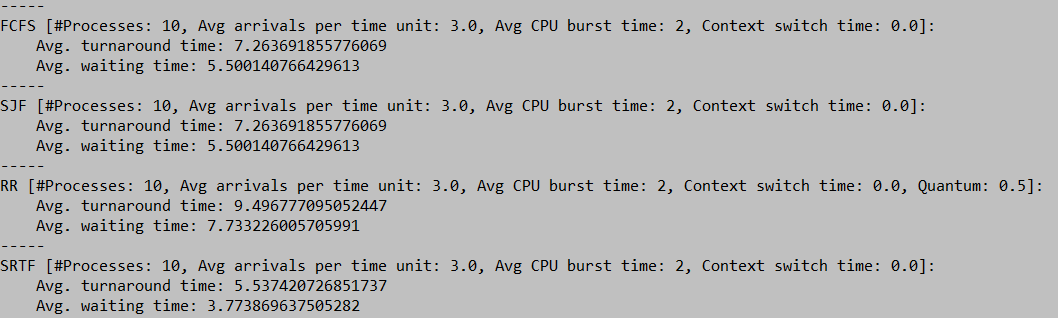
***Figure 2.: Using Seed 1797410758 with a context time of 0.5.***

Using seed 1797410758 and a context switch time of 0 we can see the shortest remaining time algorithm returned the shortest average turnaround and wait time making it the most efficient algorithm, the round robin algorithm came second, then FCFS and SJF coming last both returning similar Average times. This is likely owed to the fact that the SRTF algorithm was allowed to work on quicker processes as they arrived into the ready queue absent of any context switch time which would otherwise compound onto the turnaround and waiting times.

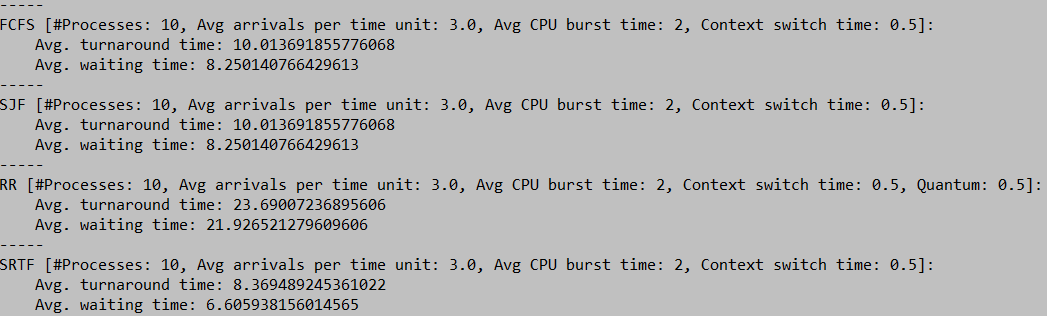
Once the context switch time was increased all of the algorithms are expected to return slower average times with RR and SRTF expecting to slow significantly more than FCFS and SJF as these algorithms usually require more switching. Here we see the SJF algorithm unexpectantly return a smaller average time.

We can only hypothesise the program takes into account the context switch time when evaluating how long processes are going to take and when the context switch is increased it determined that doing less switches would make it faster. When CST was set to zero the algorithm included more switches and caused the average times to increase.

Seed Result (2): 2688744162



***Figure 3.: Using seed 2688744162 with a context time of 0.***



***Figure 4.: Using seed 2688744162 with a context time of 0.5.***

Using seed: 2688744162 we can see SJF and SRTF are the most efficient algorithms returning very similar average turnaround and wait times.

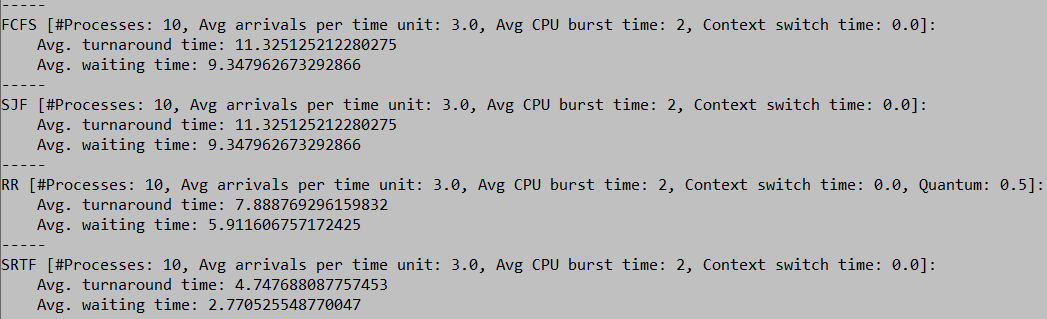
We hypothesise that this means there is minimal switching and the SRTF is able to execute whole programs at a time similar to SJF as a result.

The SRTF algorithm executes a process until another process becomes ready to execute, whereby it switches to the process with the shortest remaining time.

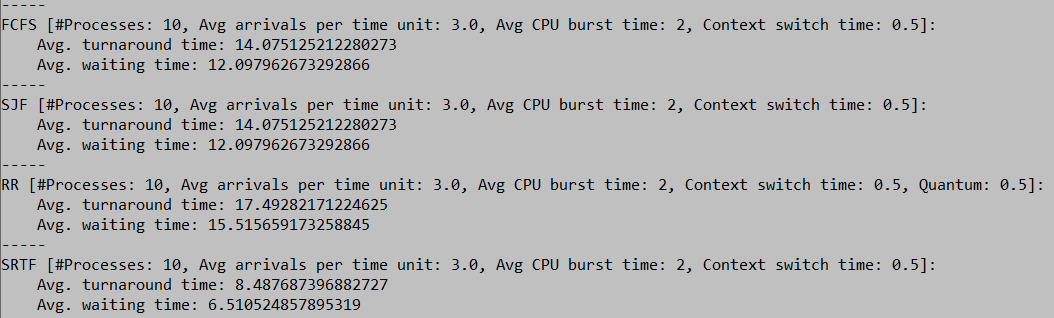
In order for minimal switching to occur we assume that when new processes become ready, the current process being executed has the shortest remaining time therefore the CPU finishes the process to completion like in SJF.

When context switch time is increased to 0.5 we can see SJF and SRTF return identical average turnaround and wait time results indicating the algorithms execute each process in the exact same order. This would mean the current processes in SRTF have the shortest remaining time when the next process becomes ready and the next shortest process is thus in the same order as in SJF.

Seed Result (3): 3399474557



***Figure 5.: Using seed 3399474557 with a context time of 0.***



***Figure 6.: Using seed 3399474557 with a context time of 0.5.***

Here the results appear to be as one would anticipate. The algorithm SRTF returns the fastest result followed by RR followed by SJF followed by RR and finally FCFS. When the context switch time is increased RR average times increase dramatically as it is the algorithm that uses the most switching, the others increasing by a similar amount.

Conclusion

To complete this task, it was imperative to have a detailed understanding of the program skeleton. This is owed to the fact that, otherwise, it becomes increasingly difficult to understand the syntactic construction of the four different algorithms/schedulers. For instance, one would not know the different fields that exist per process, such as the service time, its state, and event types. Additionally, one would not know the different functions that are made available, and which could aid in the construction of the different schedulers. Initially, it was challenging to decipher the program skeleton because there were a plethora of functions and classes that accomplished different tasks. As a result, it was difficult to know where to start and keep track of what would happen in the execution of a particular scheduler/algorithm. I think in the future it may be worth performing a theoretical execution of a particular case, or in this context a specific algorithm, and visualize what the program would have to undertake in relation to the various code skeletons.