Jordan Charest

CSCI 6140 – Operating Systems

Project 1 Analysis

Partner: Jinda Cui

1) **Of the three simulated algorithms, which algorithm is the best algorithm? Which algorithm is the best for CPU-bound processes? Which algorithm is the best for I/O-bound processes? Support your answer by citing specific simulation results.**

The best algorithm cannot be answered because it depends on the circumstances. SRT has the shortest average wait times, but can be susceptible to starvation. Round Robin can have performance that is very close to SRT, without starvation, but that performance is heavily dependent on the chosen time slice. No algorithm will perform the best in all situations.

CPU-bound processes
Using the following input:

```
A|0|123|5|141
B|0|145|1|0
C|0|97|5|113
D|10|1770|3|120          # ← CPU-bound process
E|100|156|3|137
F|120|129|2|135

FCFS:
  Process D terminates at 6793ms, with 3 processes terminating after it
SRT:
  Process D terminates at 7685ms, with 0 processes terminating after it
RR:
  Process D terminates at 7885ms, with 0 processes terminating after it
```

In many cases, FCFS will give the best performance for a CPU-bound process. This is because in both SRT and RR, long CPU bursts are punished. In SRT, a process is preempted if another process with less remaining time shows up, and in RR, long CPU bursts will be preempted because their time slice is up. In FCFS, long CPU bursts don't matter, because each process will be able to finish its burst without getting preempted.

I/O-bound processes
Using the following input:

```
A|0|123|5|141
B|0|145|5|0
C|0|97|5|113
D|10|90|3|1770          # <-- IO-bound process
E|100|156|5|137
```

```
F|120|129|4|135
G|125|143|3|167

FCFS:
  Process D terminates at 4867ms, with 0 processes terminating after it
SRT:
  Process D terminates at 3848ms, with 1 processes terminating after it
RR:
  Process D terminates at 5219ms, with 0 processes terminating after it
```

In many cases, SRT will give the best performance for an I/O-bound process. This is true if the CPU burst is short enough that it will pass most other processes in the ready queue. That way, when the I/O bound process arrives in the ready queue, it can quickly use the CPU and then head back to I/O, where it will be blocked for a while. Not allowing the I/O-bound process to skip ahead, like in FCFS, means it will be blocked for a while on both I/O and the CPU, resulting in very long turnaround times. RR can also have very good performance on I/O-bound processes, but as always, it depends on the choice of time slice. If the time slice is 90ms instead of 80, then process D can finish its CPU burst in single time slice. If that is the case, then process D terminates at 4432ms, much better than FCFS, but still not as fast as SRT.

2) For the RR algorithm, how does changing rr_add from END to BEGINNING change your results?

Changing rr_add to BEGINNING means that process are added to the ready queue at the beginning instead of the end, upon arrival or return from I/O block. The results are dependent on the type of processes that are being scheduled. Typically, the average wait times and turnaround times will increase, but it may be beneficial for specific processes. For an I/O bound process surrounded by CPU bound processes, this change is an improvement. Processes with long I/O bursts can finish their CPU bursts sooner than before and go back to blocking on I/O. For CPU bound processes however, the performance typically worsens. A CPU bound process already blocks for a long time in the CPU, and adding other processes before it in the ready queue will just mean waiting longer. Average performance decreases, but certain processes will see improvements.

3) Can you tune t_slice such that you attain the shortest average turnaround time or shortest average wait time for the given sample test input files? Explain what happens when you use different values of t_slice.

It's not possible to tune t_slice in such a way as to achieve the mathematically optimal shortest average turnaround time or shortest average wait time. That is only the case for the Shortest Remaining Time algorithm. Through various tests, it appears that the optimal value for t_slice is one such that many of the processes can finish their bursts before their time slice is up, but not all. This allows shorter running processes to finish bursts without being preempted, while still maintaining the benefit of reaching the CPU sooner, since long running processes will be preempted. If t_slice is too small, a large number of context switches will occur and performance will rapidly degrade. If t_slice is too large (large enough so all processes can finish a burst without being preempted), the algorithm decays into first come first serve.