**CS492**
Zachary Saegesser & Christopher Lyons
We Pledge Our Honor We Abided by the Stevens Honor System

# Analysis
# Producers & Consumers

## Implementation:

### Initial Construction:

Without skeleton code we had to decide how to construct the project in such a way that wouldn't be overly complicated without reason. We initially wanted to have four files; a main, a producer file, a consumer file and a product file. Ultimately, we decided on combining the producers, consumers and the main file for the sake of not having to pass too many arguments. This was due to POSIX threads requiring a single parameter as a void pointer. So too keep our solution simple we opted to use a few global variables.

### Constructing Producers & First-Come-First-Serve:

In order to have the option to run consumers with Round Robin or First-Come-First-Serve scheduling we created two consumer functions. The structure of First-Come-First-Serve ended up very similar to that of our Producer function. Each Producer was given two binary mutexes, one for preventing other producers from changing the global variables and another specifically for our queue.

### Round Robin:

For deciding how to construct our Round Robin consumer function we used a metaphorical real-world model of the problem. We said the Round Robin algorithm is like a group of people sharing snickers bars. The people being the producers, the snickers being the products but most importantly the bites they take represent the quanta.

## Experimentation:

Zach wrote a python script using numpy and pyplot to test and graph the performance of our implementation.

### How it works:

1. Runs two nested loops, to rotate through Quantums, then to test 5 times for each Quantum
2. Analyzes the stdout of the C++ code using a flag we built into the code to initiate test mode
3. Compares timing values and graphs the analyzed information
4. TO RUN THE PYTHON3 SCRIPT YOU MUST INSTALL NUMPY AND MATPLOTLIB

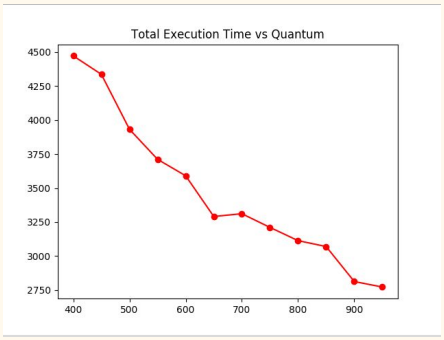
## Expectation:

We expected to find a parabolic curve in the performance of the Round Robin meaning that there would be a “sweet spot” in quantum size. We also expected that quanta outside of this “sweet spot” would yield execution and turnaround times closer to, if not worse than, First-Come-First-Serve.

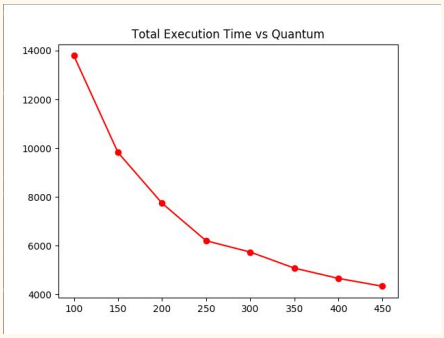
## Results:

To our surprise increasing quanta only makes the Round Robin Scheduling more efficient. We assume this to be the case specifically with our Producers & Consumers problem, being that managing a single queue of objects is much simpler than full CPU scheduling. We hypothesize the main reason for the result varying from our initial expectation to be the fact that the threads we are working with have no I/O wait time embedded in them. In a real world scenario, threads would need to be interacting with the disk or waiting for user input which would cause heavy downfalls in the wait times, turnover times, and total execution time. Below are the graphs we produced to see these results. All graphs are as avg time of Y as Quantum increases.

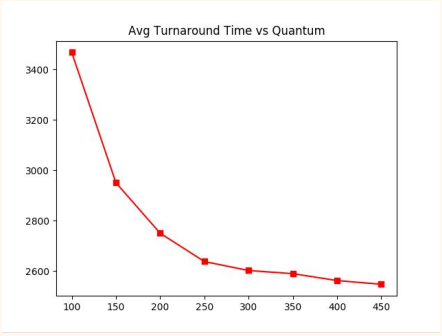
| Input | Producer Throughput | RR Throughput | FCFS ThroughPut |
|---|---|---|---|
| 10 10 10 10 x 10 10 | 10.161 | 540.85 | 10.1744 |
| 10 10 10 10 x 5000 10 | 10.2095 | 10.2105 | Unchanged ^ |
| 10 10 100 10 0 10 10 | 439.711 | 4857.77 | 100.41 |
| 10 10 100 10 0 5000 10 | 10.0377 | 100.451 | unchanged^ |


Total Execution Time vs Quantum
4500
4250
4000
3750
3500
3250
3000
2750
400
500
600
700
800
900

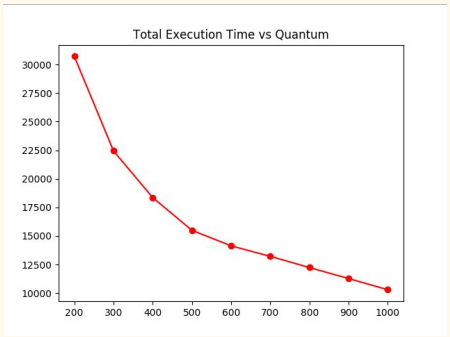
A


Total Execution Time vs Quantum
14000
12000
10000
8000
6000
4000
100
150
200
250
300
350
400
450

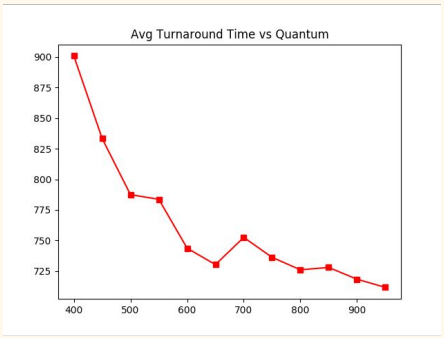
B


Avg Turnaround Time vs Quantum
3400
3200
3000
2800
2600
100
150
200
250
300
350
400
450

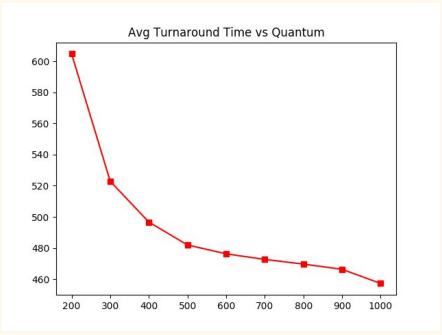
C


Total Execution Time vs Quantum
30000
27500
25000
22500
20000
17500
15000
12500
10000
200
300
400
500
600
700
800
900
1000

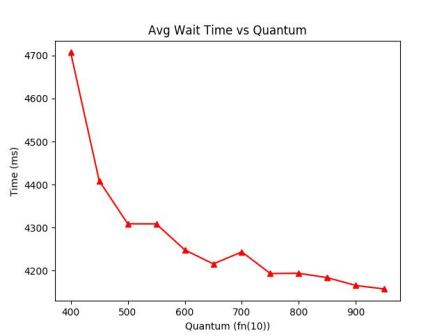
D


Avg Turnaround Time vs Quantum
900
875
850
825
800
775
750
725
400
500
600
700
800
900

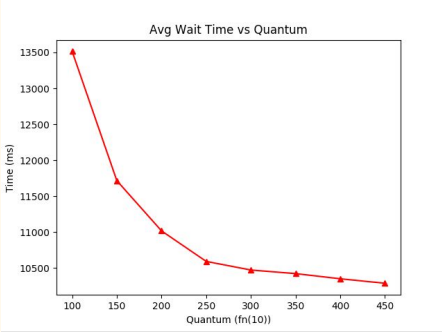
E


Avg Turnaround Time vs Quantum
600
580
560
540
520
500
480
460
200
300
400
500
600
700
800
900
1000

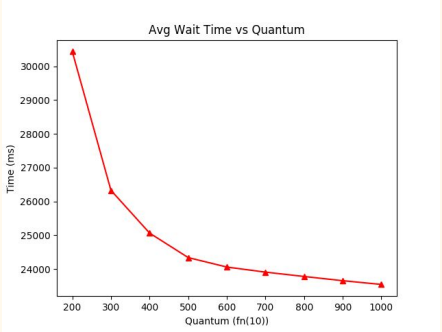
F


Avg Wait Time vs Quantum
Time (ms)
4700
4600
4500
4400
4300
4200
400
500
600
700
800
900
Quantum (fn(10))

G


Avg Wait Time vs Quantum
Time (ms)
13500
13000
12500
12000
11500
11000
10500
100
150
200
250
300
350
400
450
Quantum (fn(10))

H


Avg Wait Time vs Quantum
Time (ms)
30000
29000
28000
27000
26000
25000
24000
200
300
400
500
600
700
800
900
1000
Quantum (fn(10))

I