Task 7: Comparing Scheduling Algorithms

Jana Saleh 900204192
Muhammad Azzazy 900202821
Mariam Dahab 900192441

TABLE OF CONTENTS

TABLE OF CONTENTS

Assumptions

- For all three scheduling algorithms, the arrival times of processes are in increasing order.
- For all three scheduling algorithms, the optimization criteria were assumed to be integers and the result of division was rounded down (integer division).
- For MLFQ, the time quantum of the first two RR scheduling algorithms were fixed to 8 and 16, respectively.
- For MLFQ, there was no aging mechanism implemented. This means that once processes reach FCFS, they never go back to a higher priority queue. In other words, processes which exceed the time quanta of both RR queues exit the algorithm from the tail of the FCFS queue.
- For MLFQ, only a single process can exist in a queue during a certain time interval. This simulates single-core execution for simplicity.

01

Roles

Role of every team member

Name	Role
Jana Saleh	Implemented and tested First Come First Serve(FCFS)
Mariam Dahab	Implemented and tested Round Robin (RR) Created the Graphs
Muhammad Azzazy	Implemented and tested Multilevel Feedback Queue (MLFQ)

02

Pseudocode

Pseudocode of algorithms

FCFS

```
void FCFS_Scheduling (struct process p[], int nproc) {

double total_turnaround_time = 0.0, total_waiting_time =
0.0, avgwaitT = 0.0, totalrespt = 0.0, avgrespt = 0.0, double
sumburst = 0, int totalburst[n], double avgturnT = 0,
p[0].waitT = 0,

p[0].responseT = 0

 for m ← 0 to nproc {
p[m].turnAroundT ←p[m].burst + p[m].waitT }

 for k ← 1 to nproc {

  p[k].waitT ← p[k - 1].burst + p[k - 1].waitT  }

 for i ← 0 to nproc {
 p[i].responseT ← p[i].waitT   }

for v← 0 to nproc {

  total_waiting_time += p[v].waitT;

  total_turnaround_time +=
p[v].turnAroundT;

  totalrespt += p[v].responseT }

 avgwaitT ←total_waiting_time / nproc

 avgturnT ←total_turnaround_time / nproc

 avgrespt ←totalrespt / nproc
```

FCFS

```
int main() {
 int n , x = 0
 struct process p[1000]
 printf("Enter the number of processes: ")
 scanf("%d", &nproc)
 while (x < 10) {
  for l ← 0 to nproc {
    p[l].burst ←0
 }
  for i ← 0 to nproc {
    if (i ==0)
      p[i].arrivalT ← 0
else
      p[i].arrivalT ← p[i - 1].arrivalT + (rand() % n) + 1
    p[i].pid ← i + 1
    p[i].burst = (rand() % nproc) + 1
}
  FCFS_Scheduling(p , nproc)
  x++
 }
```

```c
int main(void) {


  int time_quant = n_process/2;
  int remaining_burst[n_process], previous_arrival = 0;
  int turnaround[n_process], wait[n_process], response[n_process];

  struct PCB pro[n_process];

  // initalize PCB values
  for (int i = 0; i < n_process; i++) {

    if (i == 0) {
      pro[i].arrival = 0;
    } else {
      pro[i].arrival = previous_arrival + (rand() % time_quant);
    }
    // make sure array in FCFS order
    previous_arrival = pro[i].arrival;

    pro[i].execution = (rand() % n_process) + 1;

    remaining_burst[i] = pro[i].execution;
  }

  // total number of processes
  int counter = n_process, n = 0, total_time = 0, total_turnaround = 0,
      total_wait = 0,
      total_response = 0; // i for loop over processes
  bool recently_done = false, time_changed;
```

Round Robin

1- Initialize PCB

```
while (counter != 0) {
  time_changed = false;
  if (remaining_burst[n] > 0) // still need execution
  {
    if (remaining_burst[n] <= time_quant) // last execution
    {
      if (remaining_burst[n] == pro[n].execution) // first execution
      {
        response[n] = total_time - pro[n].arrival;
        total_response += response[n];
      }
      total_time += remaining_burst[n];
      remaining_burst[n] = 0;
      recently_done = true;
      time_changed = true;
    } else {
      if (remaining_burst[n] == pro[n].execution) // first execution
      {
        response[n] = total_time - pro[n].arrival;
        total_response += response[n];
      }
      total_time += time_quant;
      remaining_burst[n] -= time_quant;
      time_changed = true;
    }
  }
  if (remaining_burst[n] == 0 &&
      recently_done) // update recently completed process
  {
    counter--;
```

Round Robin

2- Execute

```
    }
    if (remaining_burst[n] == 0 && recently_done) // update recently completed process
    {
      counter--;

      // update Turnaround
      turnaround[n] = total_time - pro[n].arrival;
      total_turnaround += turnaround[n];

      // update Waiting
      wait[n] = turnaround[n] - pro[n].execution;
      total_wait += wait[n];

      // done updating
      recently_done = false;
    }
    if (n != n_process - 1) {
      if (pro[n + 1].arrival <= total_time) {
        n++;
      } else {
        n = 0;
      }
    } else {
      n = 0;
    }
}
```

Round Robin

3- Update time and select next process

```c
  printf("ID \t Arrival  Burst  Turnaround  Waiting  Response \n");
  for (int i = 0; i < n_process; i++) {
    printf("%d %6d %9d %10d %7d %8d \n", i + 1, pro[i].arrival,
           pro[i].execution, turnaround[i], wait[i], response[i]);
  }

  printf(" \n Average Turnaround : %d \n", total_turnaround/n_process);
  printf("Average Waiting Time : %d \n", total_wait/n_process);
  printf("Average Response Time : %d \n", total_response/n_process);

  return 0;
}
```

Round Robin

4- Print Scheduling Table and Average Time

MLFQ

```
MLFQ(n_proc, Process* processes)
    quanta[0] ← 8
    quanta[1] ← 16
    total_waiting_time ← 0
    total_time ← 0
    total_turnaround_time ← 0
    total_response_time ← 0
    for i ← 1 to n_proc // initialize burst time of procs
        do          remaining_burst[i] ← processes[i].burst_time
    for i ←1 to n          // Q0 RR
        do          If remaining_burst[i] > 0
                    then if remaining_burst[i] == processes[i].burst_time // calculation of response time for proc
                        then    response_time[i] ← total_time - processes[i].arrival_time
                                total_response_time ←  total_response_time + response_time[i]
```

MLFQ

```
If remaining_burst[i] <= quanta[0] // if the process finishes from Q0 RR without preemption
        then total_time ← remaining_burst[i]
                remaining_burst[i] ← 0
                turnaround_time[i] ← total_time - processes[i].arrival_time
                total_turnaround_time ← total_turnaround_time + turnaround_time[i]
                waiting_time[i] ← turnaround_time[i] - processes[i].burst_time
                total_waiting_time ← total_waiting_time + waiting_time[i]
else // if the process exits Q0 RR with preemption
        total_time ← total_time + quanta[0]
        remaining_burst[i] ← remaining_burst[i] - quanta[0]
for i←1 to n_proc
do If remaining_burst[i] > 0
                then        if remaining_burst[i] <= quanta[1] // if the process finishes
from Q1 without preemption
                        then    total_time ← total_time + remaining_burst[i]
                                remaining_burst[i] ← 0
                                turnaround_time[i] ← total_time - processes[i].arrival_time
                                total_turnaround_time ← total_turnaround_time + turnaround_time[i]
                                waiting_time[i] ← turnaround_time[i] - processes[i].burst_time
                                total_waiting_time ← total_waiting_time + waiting_time[i]
```

MLFQ

```
        else // if the process exits from Q1 with preemption
                total_time ← total_time + quanta[1]
                        remaining_burst[i] ← remaining_burst[i] - quanta[1]
for i ← 1 to n
        do              if remaining_burst[i] > 0 // if the process finishes from Q1 FCFS (non-preemptive)
                        then            total_time ← total_time + remaining_burst[i]
                                remaining_burst[i] ← 0
                                turnaround_time[i] ← total_time - processes[i].arrival_time
                                total_turnaround_time ← total_turnaround_time + turnaround_time[i]
                                waiting_time[i] ← turnaround_time[i] - processes[i].burst_time
                                total_waiting_time ← total_waiting_time + waiting_time[i]

average_turnaround_time ←total_turnaround_time/n_proc
average_waiting_time ← total_waiting_time/n_proc
average_response_time ← total_response_time/n_proc
printf("Average turnaround time: %d\n", average_turnaround_time)
printf("Average waiting time: %d\n", average_waiting_time)
printf("Average response time: %d\n", average_response_time)
```

03

Process Attributes

Process attributes chosen

```c
struct PCB {
  int arrival;
  int execution;
};
```

For all three scheduling algorithms , the only attributes needed from a process was its arrival time and burst (execution) time .

These two values were used to determine which process to be executed next , for how long , and to calculate the three optimization criteria for each process

04

Testing

How each algorithm was tested

FCFS

The program asks the user for the number of process then random numbers are generated for the burst time and arrival time for each process and then their response , waiting and turnaround time were calculated

A process table is printed as shown printing all the required times for each process and then their averages is calculated at the end

This is a simple test on 5 processes to illustrate.

```
> make -s
> ./main
Enter the number of processes: 5
PID  BT  WT  RespT  TAT
 1   4    0    0     4
 2   3    4    4     3
 3   4    7    7     4
 4   2    11    11     2
 5   5    13    13     5
Average waiting time = 7.000000
Average turn around time = 3.600000
Average response time = 7.000000
```

RR

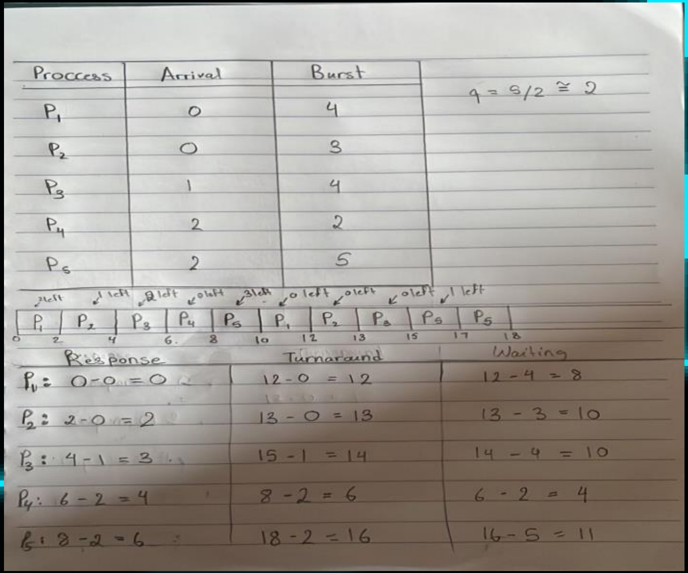
The program automatically generated 5 process each given a random arrival time and burst time .

After going through the scheduling process a scheduling table was printing showing arrival , executing , turnaround , waiting , and response time.

To test , the same arrival and execution time where scheduled manually and the result was compared .

```
> make -s
> ./main
ID    Arrival   Burst   Turnaround   Waiting   Response
1        0        4         12          8         0
2        0        3         13         10         2
3        1        4         14         10         3
4        2        2          6          4         4
5        2        5         16         11         6

 Average Turnaround : 12
Average Waiting Time : 8
Average Response Time : 3
```



Process	Arrival	Burst
P_1	0	4
P_2	0	3
P_3	1	4
P_4	2	2
P_5	2	5

$q = 5/2 \cong 2$

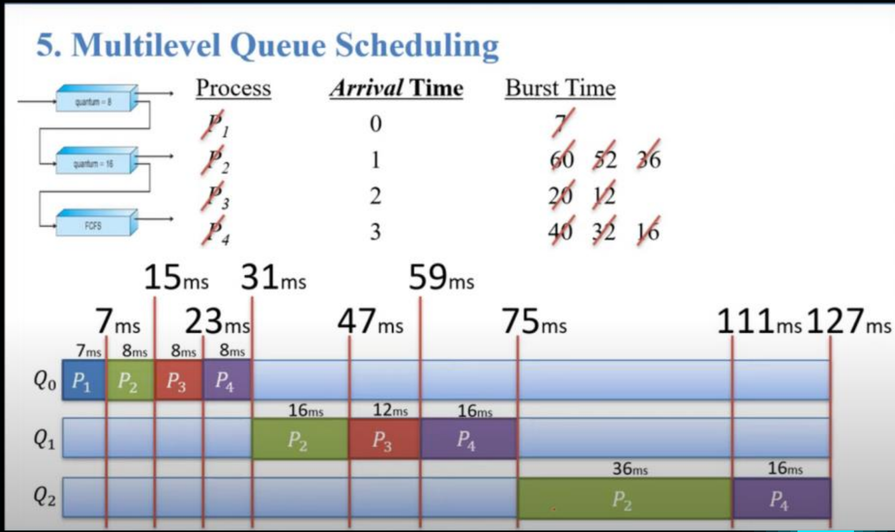
P_1	P_2	P_3	P_4	P_5	P_1	P_2	P_3	P_5	P_5
2left	1 left	2left	0left	3left	0 left	0left	0left	1 left	

0 2 4 6 8 10 12 13 15 17 18

Response	Turnaround	Waiting
P_1: 0-0 = 0	12-0 = 12	12-4 = 8
P_2: 2-0 = 2	13-0 = 13	13-3 = 10
P_3: 4-1 = 3	15-1 = 14	14-4 = 10
P_4: 6-2 = 4	8-2 = 6	6-2 = 4
P_5: 8-2 = 6	18-2 = 16	16-5 = 11

MLFQ

- The test was done on 4 processes for simplicity.
- Echoing the changes to some variables was done to ensure that the total time, turnaround time, waiting time, and response time are accurate.




```
Total time: 7
Total time: 15
Total time: 23
Total time: 31
Total time: 47
Total time: 59
Total time: 75
Total time: 111
Total time: 127
Average turnaround time: 74
Average waiting time: 42
Average response time: 9
```

```
Response time: 0
Response time: 6
Response time: 13
Response time: 20
Average turnaround time: 74
Average waiting time: 42
Average response time: 9
```

```
Turnaround time: 7
Turnaround time: 57
Turnaround time: 110
Turnaround time: 124
Average turnaround time: 74
Average waiting time: 42
Average response time: 9
```

https://www.youtube.com/watch?v=b9EYSLmG8QQ&list=PLxIvc-MGOs6ib0oK1z9C46DeKd9rRcSMY&index=13

05

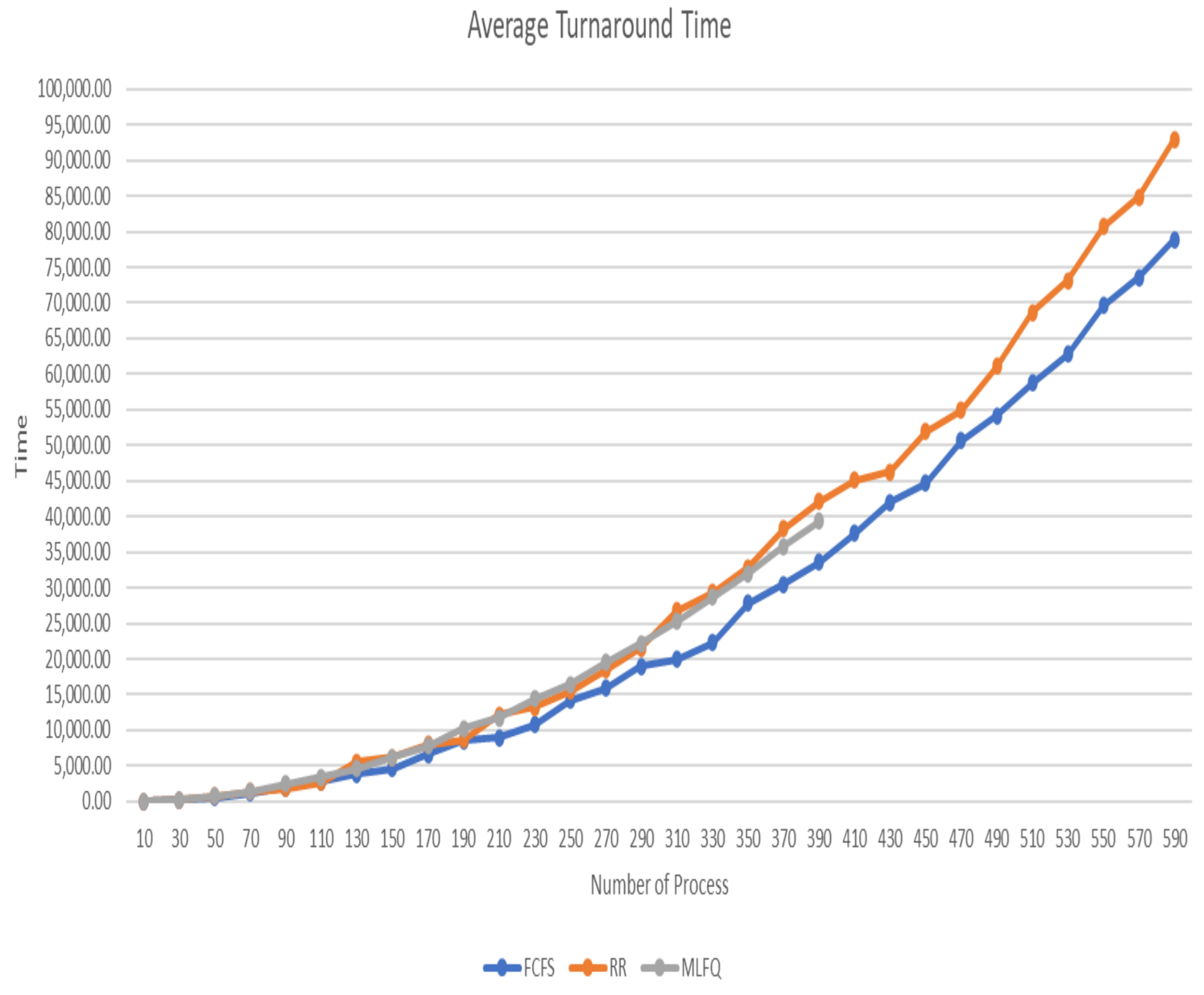
Smoothing

By running each set of processes 10 times and taking the average . We were able to create a smooth curve for each line in the curve.

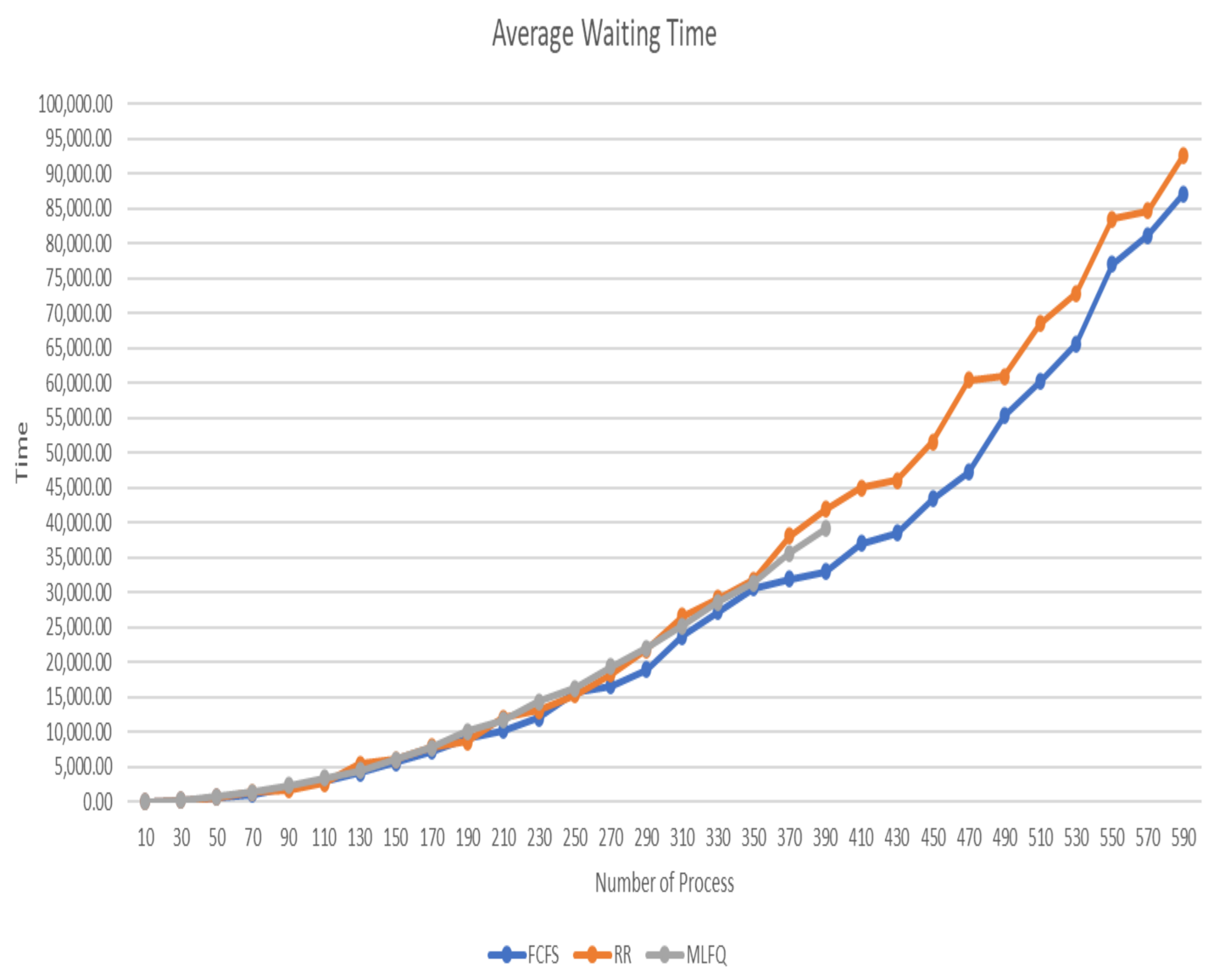
06

Conclusions

Main results and conclusions

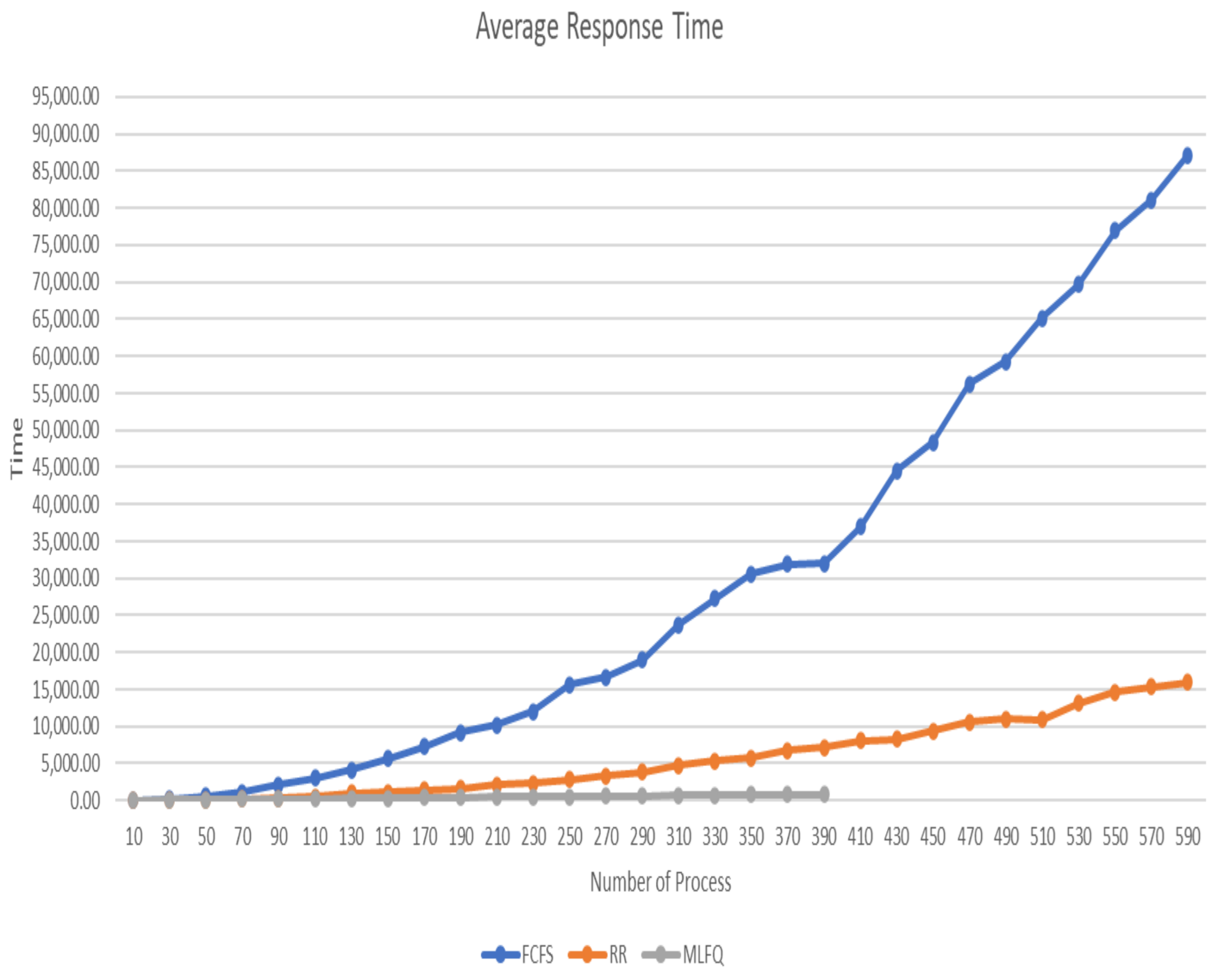


When comparing the scheduling algorithms using the average turnaround values , we could see that FCFS is the optimal choice as the number of process increase



Again , difference in performance can be noticed with large process number .

The winner is clearly the FCFS as it shows a lower average waiting time for large number of processes



In case of response time, MLFQ hard the optimal performance and FCFS should an exponential increase making it the worst performance

07

Bonus

Addressing mistakes and comments

Bonus

A lower value for priority of a process means higher priority.

Submitted a version of xv6 where the default priority is set in the scheduler itself instead of the fork system call.

THANKS!

janasaleh@aucegypt.edu
mariamhdahab@aucegypt.edu
muhammad-azzazy@aucegypt.edu