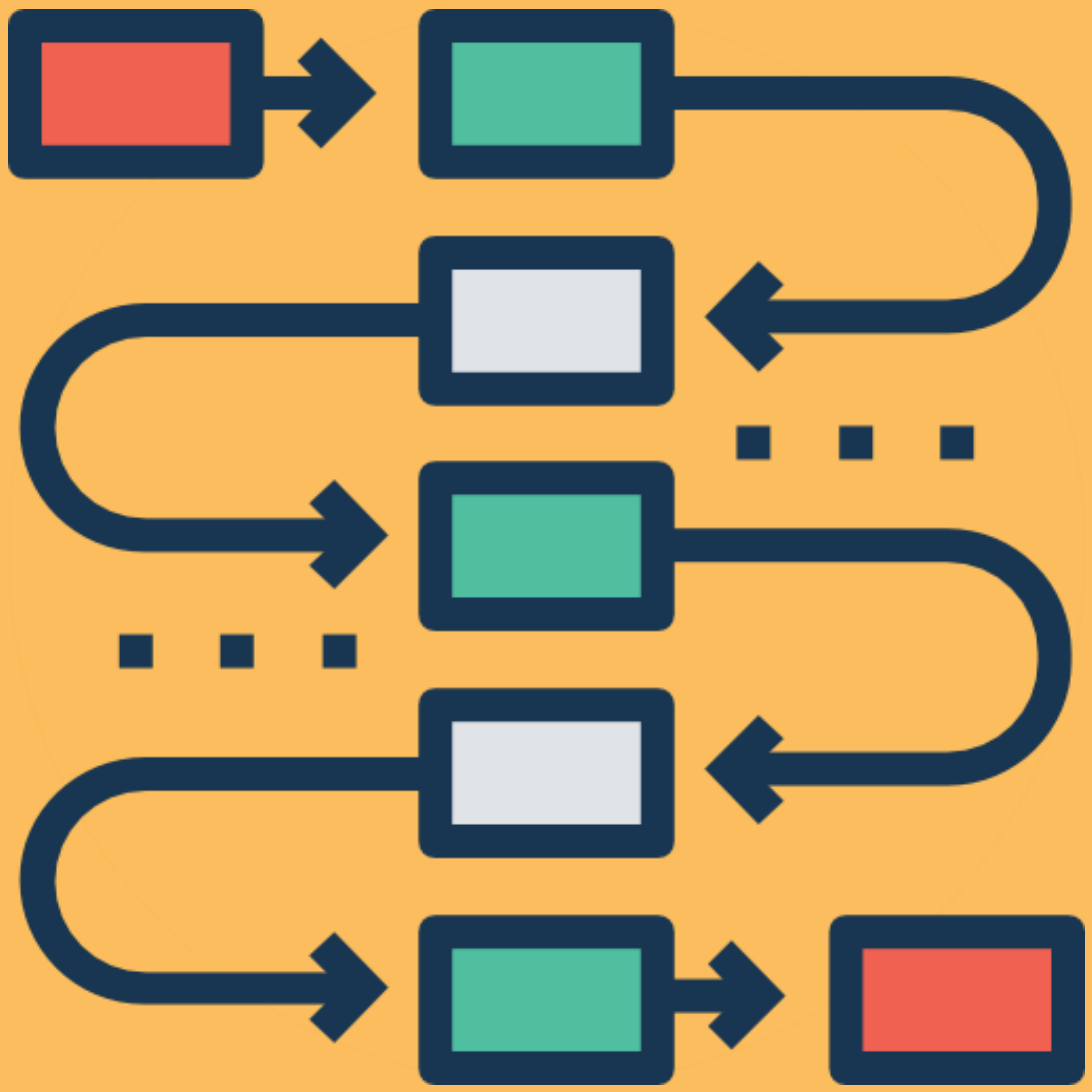


Operating System project

Narges Babaahmadi

610398102

Narges Babaahmadi
610398102

Introduction:

The type of algorithms I chose for my project is scheduling algorithms.
First I give you an intuition about the concept of scheduling algorithms, then I'll mention the specific algorithms I experimented in my project.
After that I'll implement some of the algorithms and compare them to each other.
It's worth mentioning that I've implemented the bonus part of this project in part 4 which is generating random numbers without using randint.

Part 1 and 2:

Scheduling algorithms:

When a computer is **multi-programmed**, it frequently has **multiple processes** or threads competing for the CPU **at the same time**. This situation occurs whenever two or more of them are simultaneously in the ready state. If only one CPU is available, a choice has to be made **which process to run next**. The part of the operating system that **makes the choice** is called the **scheduler**, and the **algorithm** it uses is called the **scheduling algorithm.**
Consider the case where you are using two apps namely a game like Fortnite and a desktop application like Evernote. Both with require the use of a graphics processor and but only one can use it at a time. It is the **CPU scheduling algorithms** which **manages** which process will use a given resource at a time.
The idea is relatively simple. A process is executed until it must wait, typically for the completion of some I/O request. In a simple computer system, the CPU then just sits idle. All this **waiting time** is **wasted**. With **multiprogramming**, we try to **use this time productively**. Several processes are kept in memory at one time. When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues. Every time one process has to wait, another process can take over use of the CPU. On a multicore system, this concept of keeping the CPU busy is extended to all processing cores on the system.
The **scheduling activity** is carried out by a process called **scheduler**. The scheduler is an operating system module that selects the next jobs to be admitted into the system and the next process to run. Operating systems may feature up to **three distinct scheduler** types: a **long-term scheduler** , a **mid-term scheduler**, and a **short-term scheduler**.

Long-term scheduling:

The long-term scheduler, or admission scheduler, decides which jobs or processes are to be admitted to the ready queue (in main memory).

Medium-term scheduling:

The medium-term scheduler temporarily removes processes from main memory and places them in secondary memory (such as a hard disk drive) or vice versa, which is commonly referred to as "swapping out" or "swapping in" .

Narges Babaahmadi
610398102

Short-term scheduling:

The short-term scheduler (also known as the CPU scheduler) decides which of the ready, in-memory processes is to be executed (allocated a CPU) after a clock interrupt, an I/O interrupt, an operating system call or another form of signal. Thus the short-term scheduler makes scheduling decisions much more frequently than the long-term or mid-term schedulers – a scheduling decision will at a minimum have to be made after every time slice, and these are very short.

Preemptive and Non-preemptive Scheduling:

CPU-scheduling decisions may take place under the following four circumstances:

1. When a process switches from the running state to the waiting state.
2. When a process switches from the running state to the ready state.
3. When a process switches from the waiting state to the ready state.
4. When a process terminates.

For situations **1** and **4**, there is **no choice** in terms of scheduling. A new process must be selected for execution. **There is a choice**, however, for situations **2** and **3**. When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is non-preemptive or cooperative. Otherwise, it is preemptive. Under **non-preemptive** scheduling, once the **CPU** has been **allocated** to a process, the process keeps the CPU **until it releases** it either by **terminating** or by **switching** to the **waiting state**. Virtually all modern operating systems including Windows, macOS, Linux, and UNIX use preemptive scheduling algorithms.

The Purpose of a Scheduling algorithm:

- Maximum CPU utilization
- Faire allocation of CPU
- Maximum throughput(number of processes executing per second)
- Minimum turnaround time(time taken to finish execution)
- Minimum waiting time(time for which process waits in ready queue)
- Minimum response time (time when process produces first response)

Key terms to understand different algorithms:

- **Arrival Time :** Time at which any process arrives in ready queue.
- **Burst Time :** Time required by CPU for execution of a process. It is also called as Running Time or Execution Time.
- **Completion Time :** Time at which process completes execution.
- **Turn Around Time :** Time difference between Completion Time and Arrival Time (Completion Time - Arrival Time)
- **Waiting Time :** Time difference between Turn Around Time and Burst Time (Turn Around Time - Burst Time)
- **Response Time :** Time after which any process gets CPU after entering the ready queue.

Narges Babaahmadi
610398102

Different Scheduling Algorithms:

1.**First Come First Served(FCFS):** It schedules according to the **arrival time** of the **process**. It states that process which **request the CPU first** is **allocated the CPU first**. It is implemented by using FIFO (First Come First Serve) queue and is a **Non-Preemptive** scheduling algorithm.

2.**Shortest Job First(SJF):** Process with the **shortest burst time** is scheduled **first**. If two processes have same burst time, then FCFS is used to break the tie. It is a **Non-Preemptive** scheduling algorithm.

3.**Round-Robin Scheduling(RR):** In the Round Robin scheduling algorithm, the OS defines a **time quantum** (slice). **All** of the **processes** will get **executed** in the **cyclic** way. Each of the process will get the CPU for a **small amount of time** (called **time quantum**) and then get back to the ready queue to wait for its next turn. It is a **Preemptive** type of scheduling.

4.**Priority Based scheduling:** In this algorithm, the **priority** will be assigned to each of the processes. The **higher** the **priority**, the **sooner** will the process get the CPU. If the priority of the two processes is **same** then they will be scheduled according to their **arrival time**.

5.**Shortest Remaining Time First(SRTF):** It is one of the CPU scheduling algorithms in which resources are allocated to processes according to the **shortest remaining time**.

There are so many algorithms but I've decided mentioning more would be out of context.

The algorithms I decided to implement for this project are **SJF(SRTF)** and **RR**.

So ,before implementing these two algorithms, I want to explore them more.

Narges Babaahmadi
610398102

Shortest Job First:(I chose the preemptive type of it which is SRTF)

This algorithm associates with each process the **length** of the process's next CPU **burst**. When the CPU is available, it is assigned to the process that has **the smallest next CPU burst**. If the next CPU bursts of two processes are the **same**, **FCFS** scheduling is used to break the tie. Note that a more appropriate term for this scheduling method would be the **shortest-next-CPU-burst algorithm**, because scheduling depends on the length of the next CPU burst of a process, rather than its total length.
Let's have a better understanding of this algorithm using bellow table:

PROCESS	ARRIVAL TIME	BURST TIME	WAITING TIME	TURN AROUND TIME
P1	1	7	0	7
P2	3	3	7	10
P3	6	2	2	4
P4	7	10	14	24
P5	9	8	4	12

Following the algorithm further, process having the least burst time among the available processes will be executed. Till now, we have only one process in the ready queue hence the process will be scheduled no matter what the burst time is.
The process will be executed till 8 units of time. Till then three more processes have arrived in the ready queue therefore, the process with the lowest burst time will be choosen. P3 will be executed next as it has lowest burst time. So that's how the process will go on in shortest job first (SJF) scheduling algorithm.

Advantages of SJF :

- Has minimum waiting time in comparison with other Scheduling Algorithms.
- Maximum throughput

Disadvantages of SJF :

- Very difficult to predict the burst time of processes.
- Long running CPU bound jobs can starve.

Narges Babaahmadi
610398102

Round Robin:

The round-robin (RR) scheduling algorithm is similar to **FCFS** scheduling, but **preemption** is added to enable the system to **switch between processes**. A small unit of time, called a **time quantum** or time slice, is defined. A time quantum is generally **from 10 to 100 milliseconds** in length. The ready queue is treated as a circular queue.
The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum If the process is **still running** at the end of the quantum, the CPU is **preempted** and given to **another** process.
To implement RR scheduling, we again treat the **ready queue** as a **FIFO** queue of processes. New processes are added to the **tail** of the **ready queue**.

Assume quantum time = 5
Now we have:

Processes	**Burst Time**	**Turn Around Time** **Turn Around Time = Completion Time – Arrival Time**	**Waiting Time** **Waiting Time = Turn Around Time – Burst Time**
P1	21	32-0=32	32-21=11
P2	3	8-0=8	8-3=5
P3	6	21-0=21	21-6=15
P4	2	15-0=15	15-2=13

Advantages of RR:

- It doesn't suffer from the problem of starvation or convoy effect.
- All the jobs get a fare allocation of CPU.
- While performing a round-robin scheduling, a particular time quantum is allocated to different jobs

Disadvantages of RR:

- The higher the time quantum, the higher the response time in the system.
- The lower the time quantum, the higher the context switching overhead in the system.
- Deciding a perfect time quantum is really a very difficult task in the system.

Narges Babaahmadi
610398102

Part 3 & 5 & 6 & 7:

I will implement the bonus parts (coding a random generator) later.
For now I just use a library for randomly generating numbers.
Implementation:
As it is mentioned in the projects PDF, I get the following inputs from the user:

- N
- Minimum burst
- Maximum burst
- Number of times to run the algorithm

```
from random import randint
n = int(input('Enter number of processes: '))
min_b = int(input('Enter the minimum burst_time of processes: '))
max_b = int(input('Enter the maximum burst_time of processes: '))
run_num = int(input('Enter the number of times you want to run this programm: '))
avg_waiting_t_srtf = []
avg_tat_srtf = []
avg_waiting_t_rr = []
avg_tat_rr = []
quantum = int(input('Enter the quatum time: '))
```

SRTF:

```
#Shortest Remaining Time First
def SRTF(n, min_b, max_b, bursts):
  burst_time = [0] * (n + 1)
  arrival_time = [0] * (n + 1)
  abt = [0] * (n + 1)

  for i in range(n):
    abt[i] = bursts[i]

    arrival_time[i] = 0
    h = []
    h.append(abt[i])
    h.append(arrival_time[i])
    h.append(i)
    burst_time[i] = h

  #burst_time is a 2d array that includes burst time, arrival time and index of each process
  burst_time.pop(-1)

  sumbt = 0
  i = 0
  #we construct calculator array s.t. it consists of elements as [[a,b,d],e]. e is the completion time of dth process.
  calculator_array = []
  for i in range(0, sum(abt)):
    #appending all proccessesh with arrival time < i(we have them all in l)
    l = []
    for j in burst_time:
      if j[1]<=i:
        l.append(j)
    #sorting that array by burst time
    l.sort(key=lambda x: x[0])
    h_i = burst_time.index(l[0])
    burst_time[h_i][0] -= 1
    for k in burst_time:
      if k[0] == 0:
        t = burst_time.pop(burst_time.index(k))
```

Narges Babaahmadi
610398102

```
      #i+1 is the completion time of k[2]th process
      calculator_array.append([k, i + 1])

completion_time = [0] * (n + 1)
tat = [0] * (n + 1)
waiting_time = [0] * (n + 1)
for i in calculator_array:
  # assigning completion times and saving them in an array for later usage
  completion_time[i[0][2]] = i[1]

for i in range(len(completion_time)):
  #knowing that Turn Around time = Completion Time - Arrival Time, and waiting time = Turn Around Time - Burst Time ,we fill the related arrays with
  #calculated numbers for each process
  tat[i] = completion_time[i] - arrival_time[i]
  waiting_time[i] = tat[i] - abt[i]

#popping the extra zero from all arrays
completion_time.pop(-1)
waiting_time.pop(-1)
tat.pop(-1)
abt.pop(-1)
arrival_time.pop(-1)

avg_waiting_t_srtf.append(sum(waiting_time)/len(waiting_time))

avg_tat_srtf.append(sum(tat)/len(tat))
```

This was the implementation of "**shortest remaining time first**" algorithm.
As you can see from my comments, I first **initialize abt array using the bursts input , then I initialize arrival time** with **zeros** .
Then I make **calculator array** ,which helps in **finding completion times** and so that waiting times and turn around times.
More explanation is available in the comments of my code.

```
for i in range(run_num):
  SRTF(n, min_b, max_b)
print(avg_waiting_t_srtf)
print(avg_tat_srtf)
```

In this cell, I run SRTF algorithm for '**run_num**' times and store the outputs in '**avg_waiting_t_srtf**' and '**avg_tat_srtf**'.

Narges Babaahmadi
610398102

RR:

```
#RR
def RR(n, min_b, max_b, q,bursts):

    #in this variable we find the summation of all burst times so that we can use it as a termination condition
    burst_sum = 0
    time_stamp = 0
    # processes is filled with information of processes in the format of : [arrival_time, burst_time, remaining_time, flag]
    processes = []
    waiting_time = 0
    tat_time = 0
    for i in range(n):
        # filling the process array randomly
        h = []
        #setting all arrival times to zero
        h.append(bursts[i])
        ran =  randint(min_b, max_b)
        h.append(ran)

        arrival = h[0]
        burst = h[1]
        remaining_time = h[1]

        processes.append([arrival, burst, remaining_time, 0])
        burst_sum += burst

    # using the sum of all burst times as termination condition
    while burst_sum != 0:

        for i in range(len(processes)):

            if processes[i][2] <= q and processes[i][2] >= 0:
                time_stamp += processes[i][2]
                burst_sum -= processes[i][2]
                # because the burst  time of this process is less than the quantum time, it will get finished after this pass so we just change the remaining time to zero
                processes[i][2] = 0
```

```
        elif processes[i][2] > 0:
            # In this condition, even after subtracting quantum time, the process won't get finished ,so we just subtract quantum time from remaining time and add it to time_stamp
            processes[i][2] -= q
            burst_sum -= q
            time_stamp += q

        if processes[i][2] == 0 and processes[i][3] != 1:
            #it is the condition where the process in finished and we want to set waiting time and tat, after setting them we change the flag to 1

            waiting_time += time_stamp - processes[i][0] - processes[i][1]
            tat_time += time_stamp - processes[i][0]
            # flag is set to 1 once wait time is calculated
            processes[i][3] = 1

avg_waiting_t_rr.append((waiting_time * 1) / n)
avg_tat_rr.append((tat_time * 1) / n)
```

This is my code for Round Rubin algorithm.
I use the **sum of all burst** time as a **termination variable**, which means I **decrement** it and while it's bigger that zero, I keep running my algorithm.
I also store the information about all processes in processes array and I also have **a time_stamp** which help me **find** the **completion time** of each process.
Other information about the implementation is available in my code as comments.

```
avg_waiting_t_rr = []
avg_tat_rr = []
quantum = int(input('Enter the quatum time: '))
```

```
for i in range(run_num):
  RR(n, min_b, max_b,quantum)
print(avg_waiting_t_rr)
print(avg_tat_rr)
```

I get the **quantum** time **from** the **user** and I **store** the data about each time running the algorithm in '**avg_waiting_t_rr**' and '**avg_tat_tt**'.

Narges Babaahmadi
610398102

Output of running SRTF algorithm 2 times:(I will run this algorithm more, these screenshots are just for showing the near-graphical output)

```
Burst_Time   Arrival_Time   Completion_Time    TAT      Waiting_Time
47             0              166             166             119

43             0              75              75              32

86             0              365             365             279

44             0              119             119             75

97             0              557             557             460

47             0              213             213             166

66             0              279             279             213

95             0              460             460             365

20             0              32              32              12

12             0              12              12              0

Average Waiting Time =  172.1
94             0              431             431             337

53             0              195             195             142

55             0              250             250             195

26             0              65              65              39

41             0              142             142             101

87             0              337             337             250

100            0              531             531             431

15             0              15              15              0

36             0              101             101             65

24             0              39              39              15
```

The information about completion_time, TAT(tourn around time) and waiting time is available at part1 section.

I made that output with the following code:

```
print('Burst_Time   Arrival_Time   Completion_Time    TAT      Waiting_Time')
for i in range(len(completion_time)):
  print("{}\t\t{}\t\t{}\t\t{}\t\t{}\n".format(abt[i], arrival_time[i], completion_time[i], tat[i], waiting_time[i]))
print('Average Waiting Time = ', sum(waiting_time)/len(waiting_time))
```

Narges Babaahmadi
610398102

Main function of my project:

```
import numpy as np
def main(n):
  bursts = []
  # avg_waiting_t_srtf = []
  # avg_tat_srtf = []
  # avg_waiting_t_rr = []
  # avg_tat_rr = []

  for j in range(run_num):
    bursts = []
    for i in range(n):
      bursts.append(randint(min_b, max_b))

    SRTF(n, min_b, max_b, bursts)
    RR(n, min_b, max_b,quantum, bursts)

  std_s_w = np.std(avg_waiting_t_srtf)
  std_s_t = np.std(avg_tat_srtf)
  mean_s_w = np.mean(avg_waiting_t_srtf)
  mean_s_t = np.mean(avg_tat_srtf)


  std_r_w = np.std(avg_waiting_t_rr)
  std_r_t = np.std(avg_tat_rr)
  mean_r_w = np.mean(avg_waiting_t_rr)
  mean_r_t = np.mean(avg_tat_rr)

  return std_s_w, std_s_t, mean_s_w, mean_s_t, std_r_w, std_r_t, mean_r_w, mean_r_t
```

I ,first, make the bursts array randomly and then run srtf and rr algorithms with that random array.
Now I need to **store** some of these **variables** for **later plotting**:

```
std_srtf_waiting = []
std_srtf_tat = []
std_rr_waiting = []
std_rr_tat = []

mean_srtf_waiting = []
mean_srtf_tat = []
mean_rr_waiting = []
mean_rr_tat = []

for i in range(5, 60, 10):

  avg_waiting_t_srtf = []
  avg_tat_srtf = []
  avg_waiting_t_rr = []
  avg_tat_rr = []

  std_s_w, std_s_t, mean_s_w, mean_s_t, std_r_w, std_r_t, mean_r_w, mean_r_t = main(i)
  std_srtf_waiting.append(std_s_w)
  std_srtf_tat.append(std_s_t)
  std_rr_waiting.append(std_r_w)
  std_rr_tat.append(std_r_t)

  mean_srtf_waiting.append(mean_s_w)
  mean_srtf_tat.append(mean_s_t)
  mean_rr_waiting.append(mean_r_w)
  mean_rr_tat.append(mean_r_t)
```

Narges Babaahmadi
610398102

Part 4:Generating random numbers and doing statistical tests(bonus)

To generate random number within a range, I used time module.

```
import time

def random_generator(minimum,maximum):
    now = time.perf_counter()
    t = str(now)
    rand = float(t[::-1][:3:])/1000
    return int(minimum + rand*(maximum-minimum))
```

Instead of using **randint** to initialize bursts, I can use the function I wrote '**random_generator**'.

Part 8 & 9 :

Input:

```
Enter number of processes: 10
Enter the minimum burst_time of processes: 10
Enter the maximum burst_time of processes: 300
Enter the number of times you want to run this programm: 200
Enter the quatum time: 20
```

Output:

```
standard deviation of srtf algorithm waiting times after running for different number of processes each for 200 times :
[79.96468519915526, 174.0649561306864, 221.88479614935315, 255.6698494191687, 280.5996076221925, 330.8833439518906]


standard deviation of srtf algorithm turn around times after running for different number of processes each for 200 times :
[115.05686001277803, 196.57731407232345, 239.276060973178, 269.63495351201885, 292.76634076408965, 342.3864288733124]


standard deviation of rr algorithm waiting times after running for different number of processes each for 200 times :
[167.13827398594256, 330.439367587392, 399.3175874471847, 488.98556256386337, 554.1974772609667, 641.6840030412166]


standard deviation of rr algorithm turn around times after running for different number of processes each for 200 times :
[201.430520207341, 351.64629751293495, 414.75370064996406, 501.96394717077567, 565.6752232934538, 652.6375321229773]


mean of srtf algorithm waiting times after running for different number of processes each for 200 times :
[211.011, 752.3483333333335, 1274.3224, 1811.7215714285715, 2347.985777777778, 2908.349]


mean of srtf algorithm turn around times after running for different number of processes each for 200 times :
[365.15799999999996, 908.459, 1429.3254000000002, 1966.846857142857, 2503.262, 3064.4884545454547]


mean of rr algorithm waiting times after running for different number of processes each for 200 times :
[277.063, 1317.5186666666666, 2406.3644, 3465.364428571429, 4538.999777777778, 5518.093727272728]


standard deviation of rr algorithm turn around times after running for different number of processes each for 200 times :
[435.273, 1471.3323333333333, 2562.2246, 3620.8622857142855, 4694.583111111111, 5672.728545454545]
```

I have already described the output in previous parts.

Narges Babaahmadi
610398102

Part 10: plotting

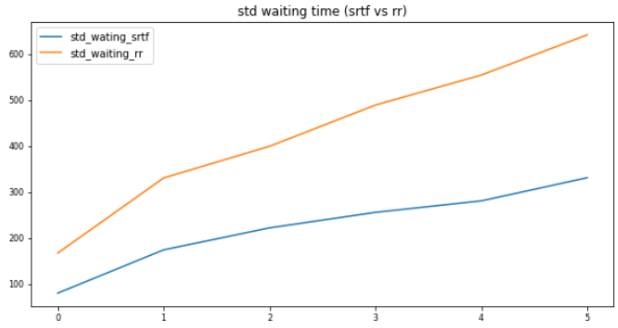
Firstly, it's worth mentioning that x-axis is the number of processes and I change y-axis and draw different plots.

```
import matplotlib.pyplot as plt
x_axis = [5, 15, 25, 35, 45, 55]
```

Standard deviation of waiting times for SRTF algorithm vs RR algorithm:

```
df = pd.DataFrame({'std_wating_srtf':std_srtf_waiting, 'std_waiting_rr':std_rr_waiting})
df.plot(figsize = (10, 5), fontsize = 8)
# plt.legend(loc = 8, prop ={'size':3})
plt.title('std waiting time (srtf vs rr)')
```

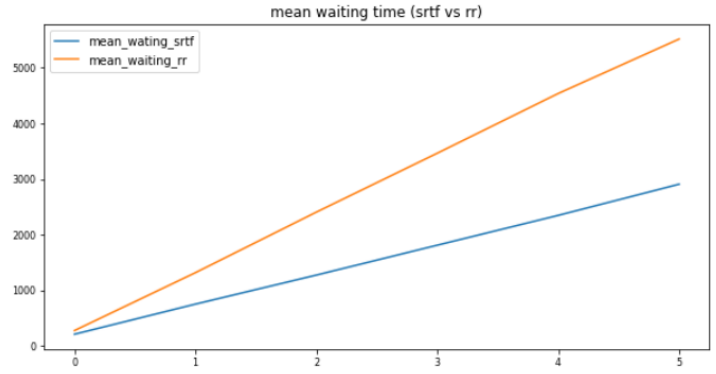
```
Text(0.5, 1.0, 'std waiting time (srtf vs rr)')
```



mean of waiting times for SRTF algorithm vs RR algorithm:

```
df = pd.DataFrame({'mean_wating_srtf':mean_srtf_waiting, 'mean_waiting_rr':mean_rr_waiting})
df.plot(figsize = (10, 5), fontsize = 8)
# plt.legend(loc = 8, prop ={'size':3})
plt.title('mean waiting time (srtf vs rr)')
```

```
Text(0.5, 1.0, 'mean waiting time (srtf vs rr)')
```

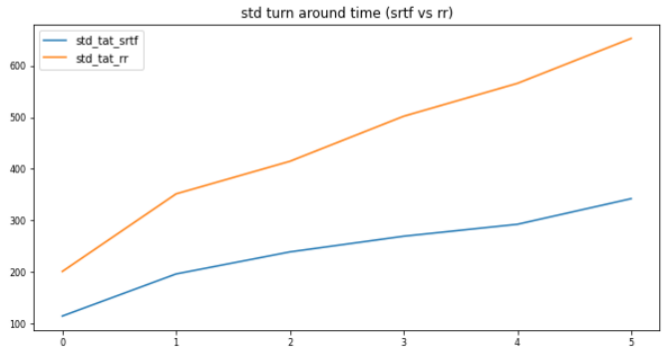


Narges Babaahmadi
610398102

Standard deviation of TAT for SRTF algorithm vs RR algorithm:

```
df = pd.DataFrame({'std_tat_srtf':std_srtf_tat, 'std_tat_rr':std_rr_tat})
df.plot(figsize = (10, 5), fontsize = 8)
# plt.legend(loc = 8, prop ={'size':3})
plt.title('std turn around time (srtf vs rr)')
```

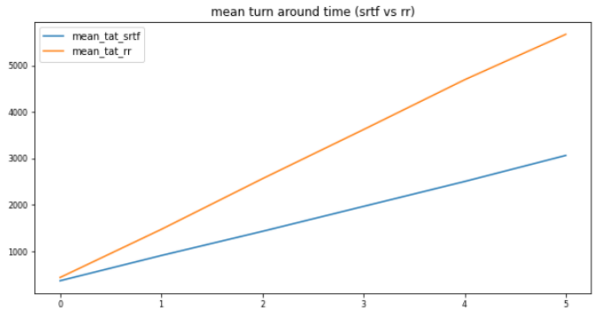
```
Text(0.5, 1.0, 'std turn around time (srtf vs rr)')
```



mean of TAT for SRTF algorithm vs RR algorithm:

```
df = pd.DataFrame({'mean_tat_srtf':mean_srtf_tat, 'mean_tat_rr':mean_rr_tat})
df.plot(figsize = (10, 5), fontsize = 8)
# plt.legend(loc = 8, prop ={'size':3})
plt.title('mean turn around time (srtf vs rr)')
```

```
Text(0.5, 1.0, 'mean turn around time (srtf vs rr)')
```



These plots compare different parameters(standard deviation and mean) of two different algorithms.
What we can get from these plots is that , for both algorithms, when the number of processes increases, both waiting time and turn around time increases.
The other fact that we understand from these plots is that, for the same number of processes, RR has larger waiting time and turn around time.

Narges Babaahmadi
610398102

Resources :

https://iq.opengenus.org/types-of-cpu-scheduling-algorithms/
https://www.javatpoint.com/os-round-robin-scheduling-example
https://www.studytonight.com/operating-system/cpu-scheduling
https://en.wikipedia.org/wiki/Scheduling_(computing)
https://cppsecrets.com/users/1108979711510497121461151049710464115111109971051219746101100117/Python-Shortest-Job-First-Non-Preemptive-Algorithm-with-Different-Arrival-Time.php
https://github.com/adityachavan198/Job-Scheduling-Shortest-job-first-preemptive-python-code/blob/master/Job-scheduling-shortest-remaining-time-first.py