Muhammad Choudhury

Operating Systems

COP 4610

Borko Furht

Programming

Assignment

-

CPU

Scheduler

Table of Contents

Introduction

Operating Systems are one of the most important aspects of modern computers. Windows 10, MacOS, and many other operating systems is essentially software that makes computer hardware, including RAM, SSDs, and CPU accessible to the user and to IO devices. There are many tasks that are assigned to the OS but one task that will be discussed in this report is the feature that allows the OS to act as a process scheduler for the CPU.

At any given time, there are can be many processes that are being ran on the computer. How is it possible that all of these processes are taken care of in an efficient manner? The answer is that the CPU scheduler takes care of which process needs to be sent into CPU for execution. Without taking advantage of schedulers, a computer could never figure out which process will be executed next which will lead to chaos and waste of both time and resources for the computer. Thankfully, the Operating Systems take on the role of being scheduling processes for the CPU to execute which allows computers to run several applications at once, increasing efficiency and productivity for human users to do as they please.

This report contains an in-depth analysis of the following scheduling algorithms: First Come First Serve, Shortest Job First, and Multilevel Feedback Queue. Three programs that run the three scheduling algorithms mentioned above will be shown in this report and it will contain comments that fully explain what is actually happening in the code and the ultimate output was created.

There are three main quantitative values associated with these algorithms schedules and they determine how efficient each algorithm is. The most important value is the waiting time which is the total time that a process spends in the ready queue. The turnaround time is the total amount of time spent by the process from the moment it entered the ready queue until its completion. The response time is the amount of time it took for a process to first get executed starting from the point when it first entered the ready queue.

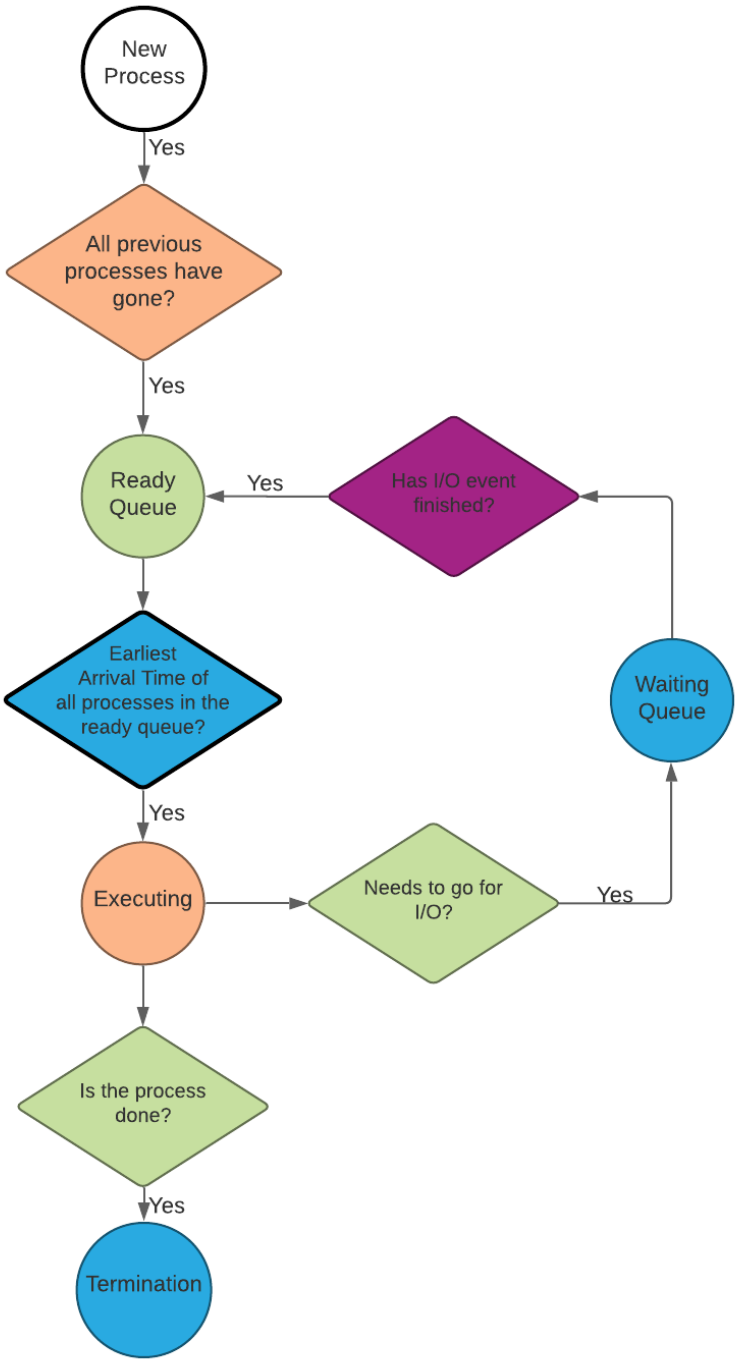
The first scheduling algorithm is called First Come First Serve. A process that goes in first into the queue will always be the first to execute whereas the last program will always be the last to execute regardless of any other factor. This form of algorithm is not optimal because the average response, waiting, and turnaround times tend to be longer than other algorithms.

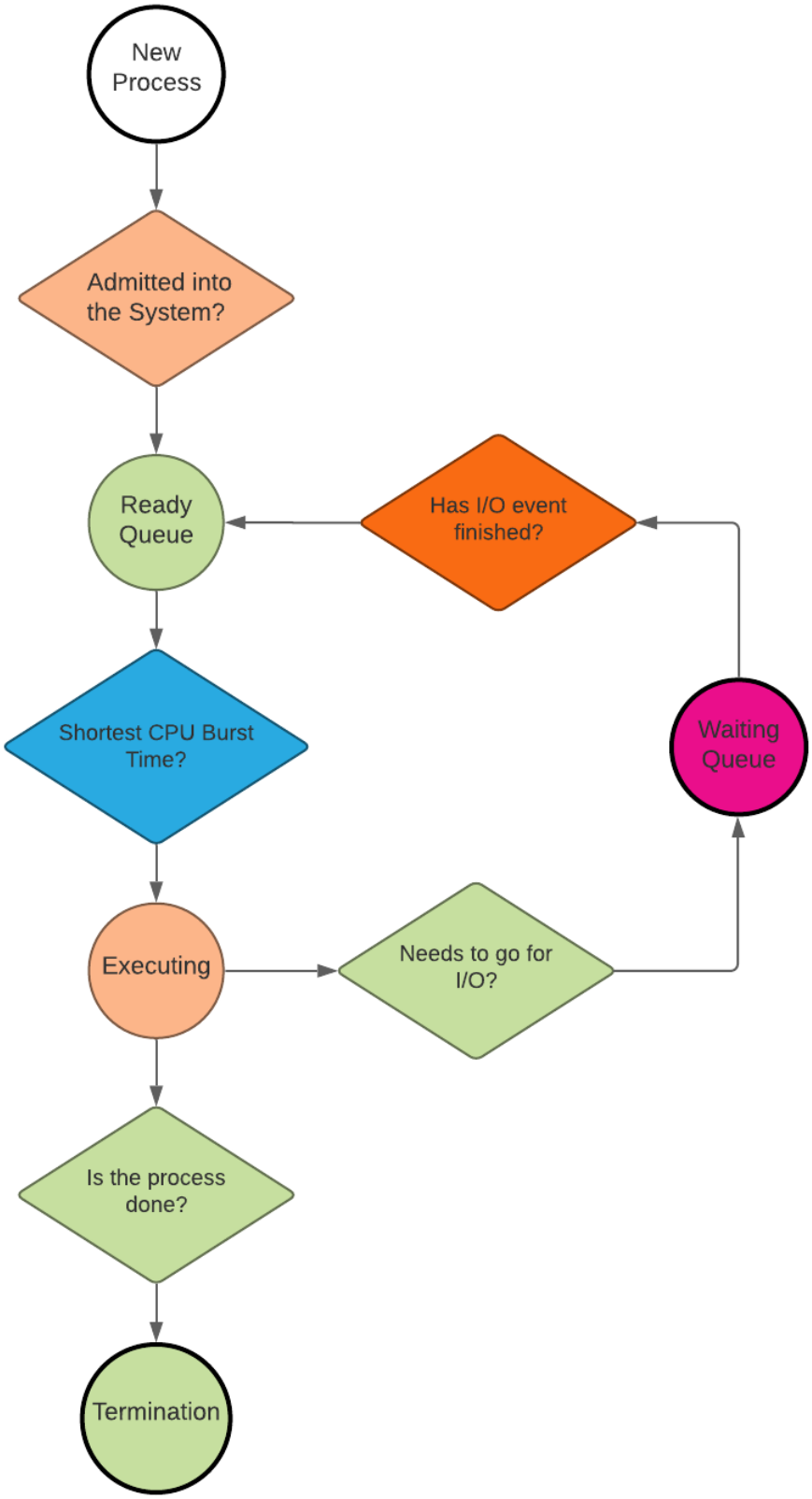
The second algorithm is Shortest Job first. This means that process with the shortest burst time will be executed by the CPU next. Because of this, the average waiting time is far shorter as compared to FCFS even when given the same set of processes. There is a formula that estimates what the length of the next CPU burst likely will be, but there is no guarantee what the next burst time will be.

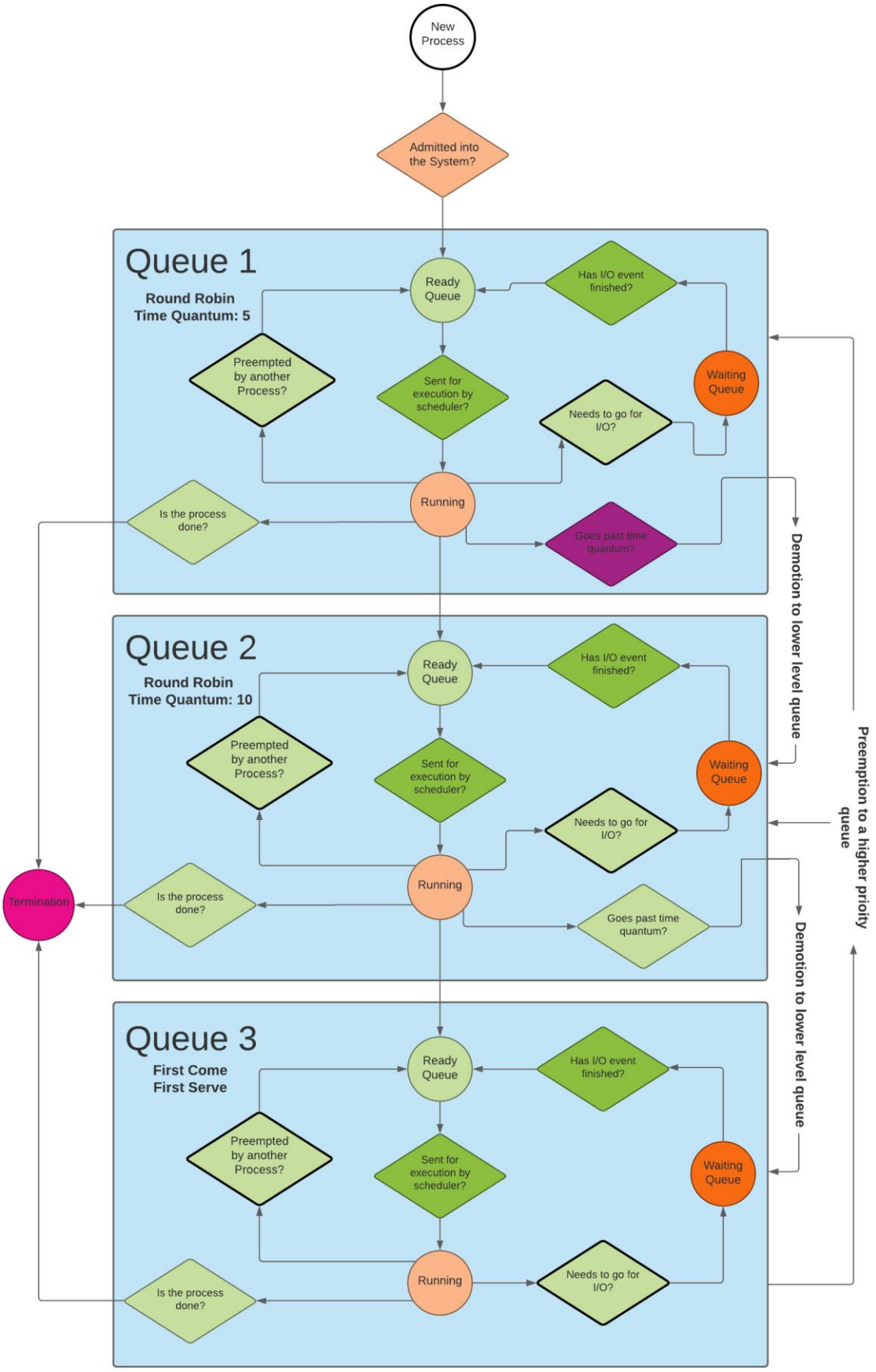
The third algorithm is Multilevel Feedback Queue. Processes can belong to a number of queues that have their own scheduling algorithms and priorities over each other. Unlike the previous scheduling algorithms mentioned, MLFQ is preemptive. This means that if a process in queue 1 arrives it will preempt a process in queue 2 that is being executed since queue 1 has higher priority. Also, processes can move between queues based on predetermined factors such as if it has waited for too long in its queue or if it has been executing for a great amount of time.

The Round Robin algorithm is another scheduling algorithm that includes preemption. Each algorithm has a time quantum that it allocated for every single one of the processes in the

queue. If the time quantum is 10 units of time, each process will run for that amount of time and it will be preempted to the next process if it exceeds that time slice. Two Round Robin queues will be included in the MFLQ algorithm presented in this report.




New Process
Admitted into the System?
Ready Queue
Has I/O event finished?
Shortest CPU Burst Time?
Waiting Queue
Executing
Needs to go for I/O?
Is the process done?
Termination


New Process
Admitted into the System?
Queue 1
Round Robin
Time Quantum: 5
Ready Queue
Has I/O event finished?
Preempted by another Process?
Sent for execution by scheduler?
Waiting Queue
Needs to go for I/O?
Running
Is the process done?
Goes past time quantum?
Demotion to lower level queue
Queue 2
Round Robin
Time Quantum: 10
Ready Queue
Has I/O event finished?
Preempted by another Process?
Sent for execution by scheduler?
Waiting Queue
Needs to go for I/O?
Running
Termination
Is the process done?
Goes past time quantum?
Demotion to lower level queue
Preemption to a higher prioity queue
Queue 3
First Come First Serve
Ready Queue
Has I/O event finished?
Preempted by another Process?
Sent for execution by scheduler?
Waiting Queue
Running
Needs to go for I/O?
Is the process done?

FCFS CPU Utilization: 85.34%			
Process	Tw	Ttr	Tr
P1	170	395	0
P2	164	591	5
P3	165	557	9
P4	17	648	17
P5	20	530	20
P6	36	445	36
P7	47	512	47
P8	61	493	61
Avg	185.25	521.375	24.375

The First Come First Serve algorithm both very large waiting and turnaround times. It also has moderate CPU utilization percentage and response times. The response time for each process is gained from the CPU bursts of all of the processes that have come before it as a result of how FCFS operates. Therefore, the last few processes have to relatively far longer time for it to reach its first execution. And after each process has gone once, the time at which each process will execute depends on how long its I/O burst is. It is very interesting to note that the first three processes have far higher waiting times than the last five. The reason is because the initial processes have longer I/O bursts than the other processes which leads to them arriving back to the ready queue from the I/O queue far later than the other processes. FCFS is a very simple but also inefficient algorithm that doesn't attempt to minimize any waiting, response, or turnaround times.

SJF CPU Utilization: 81.68%			
Process	Tw	Ttr	Tr
P1	34	259	11
P2	118	545	3
P3	285	677	16
P4	39	529	0
P5	240	549	103
P6	150	365	24
P7	134	462	47
P8	112	421	7
Avg	139.0	475.12	26.38

The Shortest Job First algorithm has very lowest waiting and turnaround times. The response times are relatively high because the processes with the longest initial CPU burst go after all the other processes and thus have wait a long time for its first execution. And even then, those processes with long bursts still have to wait on other processes that have already gone because their next CPU bursts are shorter than theirs. But because the processes with shorter I/O bursts executes first, this means that the process will execute quickly and other processes that are in the ready queue can quickly go in afterwards. This reduces the average waiting tremendously and the time between each context switch is relatively small. The average turnaround time is low because several processes have multiple short CPU bursts in succession which means that those processes will eventually terminate relatively quickly. SJF minimizes waiting and turnaround times by placing the processes with the shorter burst first.

MLFQ CPU Utilization: 91.71%			
Process	Tw	Ttr	Tr
P1	55	395	0
P2	13	591	5
P3	282	557	9
P4	20	648	14
P5	327	530	17
P6	314	445	22
P7	211	512	27
P8	79	493	32
Avg	162.53	498.75	15.75

The MLFQ algorithm has relatively low waiting and turnaround times. It also has average CPU utilization and very low response times. This algorithm is very complex as it contains three different queues with different scheduling algorithms in itself. For the response times, each process goes for as long as the time quantum of the first queue, which is Round Robin, will allow them to go for which leads to low initial waiting time or response times here. MLFQ has preemption, which in this case means that a process in a higher priority will take precedence over other process in a lower queue and will even interrupt them when the higher priority process arrives in the ready queue. This algorithm makes processes that stay in the higher priority queues for most of the time will come in and execute for its time quantum then leave. And if it goes the time quantum of its queue then it will be demoted to a lower queue, so it doesn't hold up the other processes. MLFQ uses several queues with different priorities to create optimal waiting and turnaround times.

Comparisons Between Scheduling Algorithms

	SJF	FCFS	MFLQ
CPU Utilization	81.68%	85.34%	91.71%
Avg Waiting Time (Tw)	139.0	185.25	162.63
Avg Turnaround Time (Ttr)	475.13	521.38	498.75
Avg Response Time (Tr)	26.38	24.38	15.75

Between all of these CPU schedulers, it is apparent that certain schedulers are more optimal and more efficient that others. First Come First Serve has both the longest average waiting and turnaround times, has the second longest (or shortest) average response times and second highest CPU utilization. Shortest Job First has the both the shortest average waiting and turnaround times, has the second longest average waiting times and lowest CPU utilization. Multilevel Feedback Queue has both the second shortest average waiting and turnaround times, has the lowest average response times and highest CPU utilization.

By looking at all of the numbers and data has been outputted by the programs that I have created, it is apparent that SJF is the most optimal CPU scheduler whereas FCFS is the least optimal CPU scheduler. The waiting times and turnaround times are the two most important factors when taking into account how useful and efficient an algorithm is. The main goal in designing these algorithms is that you don't want your processes to wait for an extremely long time in general and it is also desirable for processes to terminate and end exit the scheduler as quickly as possible as well. This makes SJF the most ideal schedule as it has both of these qualities.

And while Shortest Job First does appear to be the best on paper, it does have its own flaws. SJF suffers from the concept called starvation, meaning that some processes don't even execute until later on which means that the standard deviation between the waiting time of the process is very high. Also the CPU utilization is lower than even the non-optimal FCFS. Its long average response times are not the best since. But the real problem of the SJF algorithm is that is in not a practical one; it will not always be known what the CPU bursts of the processes will be in the future. Computers nowadays are very powerful, but it is impossible to predict the future.

MLFQ by far the most complex algorithm to understand and create given that it has multiple queues with their own algorithm. The low average response times, as a result of, each initial burst of each process being limited to the time quantum of 5 units, makes MLFQ very efficient. The algorithm is also very smart as it makes sure that a process essentially doesn't take up too much time or else it will be demoted to a lower queue and will be preempted to another process. This performance is much more satisfactory than FCFS which has no power to do a context switch in the middle of a CPU burst.

And while MFLQ does not have the extremely low waiting and turnaround times as SJF, it is a far more practical to use in real life because you do not need to know any future CPU curst times to do any context switches. The multiple queued algorithm doesn't exactly prioritize shorter CPU bursts like SJF since that's not feasible; it just limits how far a process can execute for based on the set time quantum. The second Round Robin queue has a larger time quantum of ten instead of five and the last queue is FCFS so that processes with long bursts can execute really only when most other processes are in the waiting queue.

The First Come First Serve algorithm is the least efficient scheduler. The scheduler doesn't do anything unique like prioritizing processes with lower CPU or restricting the processes that

have executed for too long might fix. It simply allows for one process to go then the next one. This is bad for the response time because some of the initial processes have fairly long CPU bursts right off the bat. It is still better than SJF when it comes to response times since there's no starvation occurring. It doesn't have lower response times than MLFQ because it has to wait for the entire CPU burst to occur instead of cutting it short when the remaining burst exceeds the time quantum.

But one problem that FCFS has that makes it categorially the worst scheduler is that is suffers from the convoy effect. This means that several processes have to wait in the ready queue if a process with a long burst is executing at that moment. This leads to annoyingly high waiting and turnaround times. In SJF this is avoided since all of the processes with long bursts only start going when most other process which shorter bursts are terminated. It is also avoided in MLFQ since all the processes with long burst time are interrupted demoted to the lower queue allowing for other processes to go. FCFC is ultimately more inefficient in every regard in comparison to the other two schedulers that were discussed.

Conclusion

In conclusion, there are three scheduling algorithms that were discussed in this report. First, it was explained how they function and the method in which they choose the next process to execute. Then, all relevant data like waiting and turnaround times were shown, and that data was discussed for every algorithm. Finally, all three CPU scheduling algorithms were compared and contrasted to discover which scheduler is more efficient that the other two.

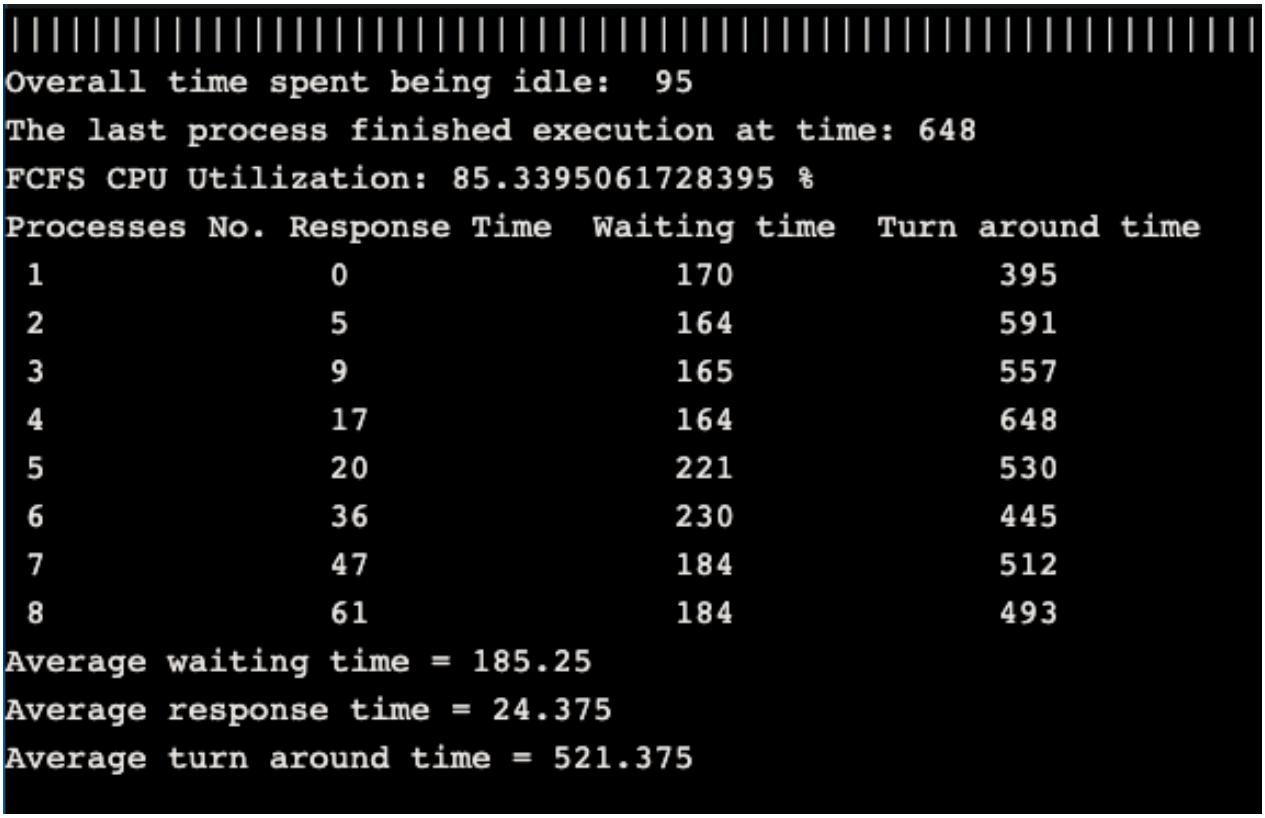
Shortest Job First is the most ideal scheduling algorithm mainly because it has the lowest waiting and turnaround times. But it is not an ideal algorithm because you would have to know the future CPU bursts since that's how the algorithm functions. First Come First Serve is a very inefficient scheduling algorithm as it does not attempt to minimize anything like waiting or turnaround times. But is a good algorithm for people who need to understand how CPU schedulers work but it is overall not practical because of relatively poor performance. The Multilevel Feedback Queue is clearly the best algorithm in practice because of its relatively good performance and because its also very feasible to implement, even if it may be a bit complex to understand at first. Transferring processes to lower priority queues is very efficient and it a very smart system as it avoids the convoy effect of FCFS and starvation of SJF.

Overall, the type of CPU scheduler than you want to implement depends on what you need. FCFS is fine for systems in which there are not too many processes and when minimizing turnaround time isn't too important. SJF is optimal in all cases in which it is absolutely known what the future CPU bursts will be. And MLFQ is made for systems that needs to minimize

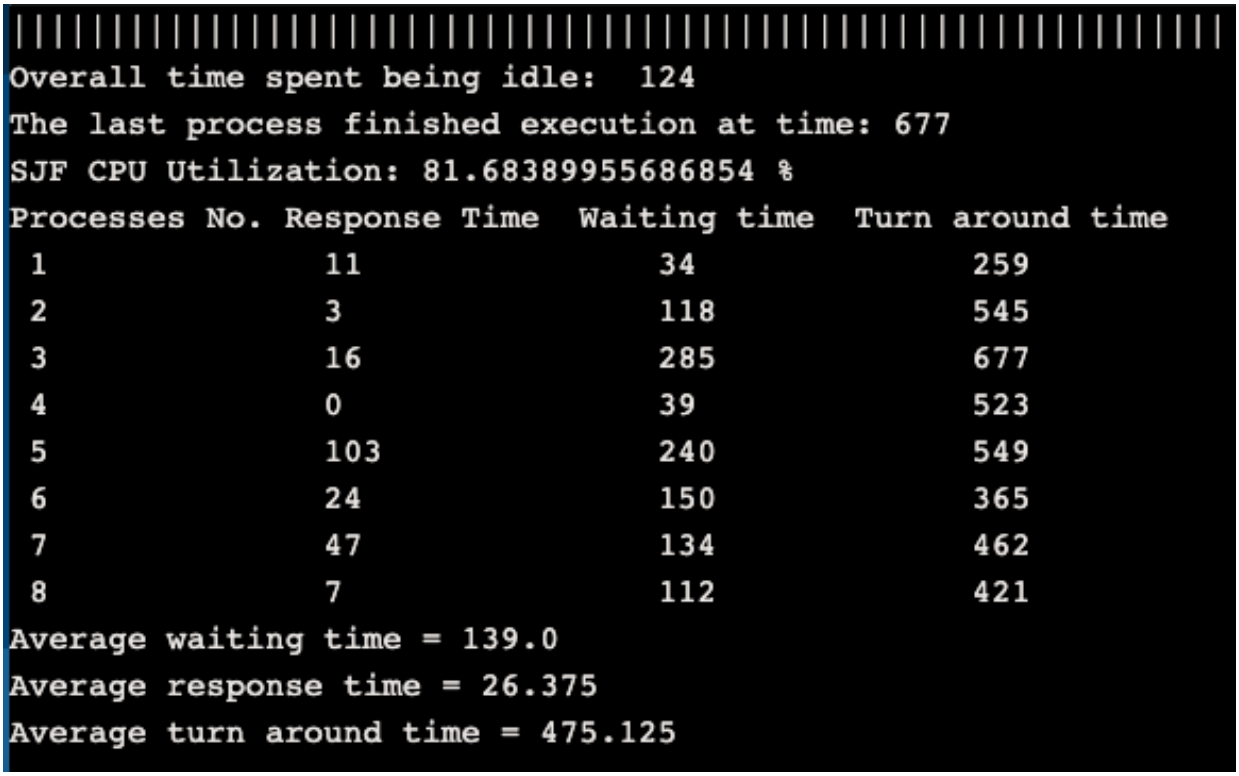
response, waiting, and turnaround times in an otherwise unpredictable system where future bursts aren't known.

Screenshots of Results

FCFS



SJF



```
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Overall time spent being idle:  124
The last process finished execution at time: 677
SJF CPU Utilization: 81.68389955686854 %
Processes No. Response Time  Waiting time  Turn around time
 1               11                34               259
 2               3                 118              545
 3               16                285              677
 4               0                 39               523
 5               103               240              549
 6               24                150              365
 7               47                134              462
 8               7                 112              421
Average waiting time = 139.0
Average response time = 26.375
Average turn around time = 475.125
```

MLFQ

```
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Overall time spent being idle:  50
The last process finished execution at time: 603
MLFQ CPU Utilization: 91.70812603648424 %
Processes No. Response Time  Waiting time  Turn around time
 1                0                55                280
 2                5                13                440
 3                9                282               674
 4                14               20                504
 5                17               327               636
 6                22               314               529
 7                27               211               539
 8                32               79                388
Average waiting time = 162.625
Average response time = 15.75
Average turn around time = 498.75
```

Code for FCFS

```
'''

This program uses a two-dimmensional array to store date about the processes.

The first dimmension are the 8  processess indexed from 0-7


The second dimnension have data relevant to that specific process

Here is what the indexes for the second dimmension means (this is for EACH process)

0: Time in which the process will end up back in ready queue from waiting queue

1: Waiting Time

2: Process Number

3: Index of the where the current CPU burst is

4: Current CPU Burst

5: Inndicator for when process is finished

6: First CPU Burst

7: First I/O Burst

8: Second CPU Burst....

'''


print("Hello World. FCFS scheduler created in Pycharm")
```

```
# Function finds Waiting, Turnaround, and Response Times
# of both individual process and averages. Also finds
# CPU utilization
def findTimes(processes, n, m, lot):
    io = [0] * n
    bt = [0] * n
    rt = [[0] * 2 for i in range(8)]
    wt = [0] * n
    tat = [[0] * 2 for i in range(8)]
    total_wt = 0
    total_rt = 0
    total_tat = 0


    t = 0
    r = 0
    f = 1
    b = 3
    d = 5
    m = 6
    z = 0
    h = 0
```

```
v = 8
tm = [0] * n
rm = [0] * n
rd = [0] * n
io = [0] * n
c = [0] * n
response = False
progress = True
cpu = 0


for i in range(n):
    rt[i][0] = i+1


# Adds all CPU and I/O burst times to array
# and sets up all other fields like position
# of CPU burst time and if process is terminated
for j in range(20):
    for i in range(n):
        lot[i][6 + j] = processes[i][j]
        if j == 0:
            lot[i][2] = i + 1
            lot[i][3] = 6
            lot[i][4] = lot[i][6]
```

```
        lot[i][5] = 1


while(progress):
    # Will continue until every process is
    # finished with its burst times
    s = 9999
    g = 10
    e = 0


    for i in range(n):
        #Determines which process will execute next depending on which
        #process came in the ready queue the earliest
        if lot[i][5] == 1 and lot[i][0] < s and lot[i][0] <= t:
            if lot[i][0] < s:
                s = lot[i][0]
                g = i
        if lot[i][5] == 1 and lot[i][0] <= t:
            # Places the processes who have arrived in the Ready Queue
            rd[i] = 1
            io[i] = 0
        if lot[i][5] == 1 and lot[i][0] > t:
            # Places the processes who havent arrived yet in the I/O Queue
            io[i] = 1
```

```
            rd[i] = 0
        if lot[i][5] == 0:
            # Puts terminated process in terminated queue
            tm[i] = 1

print("|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||")
if g == 10:
    #If no process is executing, will continue to next iteration in loop
    #Increases idle and time counters. Prints processes in  I/O Queue
    h += 1
    t += 1
    for i in range(n):
        if lot[i][0] > t and lot[i][5] == 1:
            p = lot[i][3]
            print("Process", lot[i][2], "is in the I/O queue with I/O burst of", lot[i][p-1], "units")
    print("Idle. Total idle time (so far):", h, )
    print("Current Execution Time:", t)
    continue

for i in range(n): #Prints out every process thats in the Ready Queue
    if lot[i][0] <= t and lot[i][5] == 1:
        print("Process", lot[i][2], " is in the Ready queue with CPU burst of ", lot[i][4], "units")
```

```
for i in range(n): #Prints out every process thats in the I/O Queue
    if lot[i][0] > t and lot[i][5] == 1:
        p = lot[i][3]
        print("Process", lot[i][2], "is in the I/O queue with I/O burst of", lot[i][p-1], "units")


for i in range(n):
    if lot[i][5] == 0:
        print("Process", lot[i][2],"has already been terminated")


print("Process", g+1, "is currently executing")


if lot[g][1] == 0 and c[g] == 0:
    #Records if a process has executed for the first time
    c[g] = 1
    response = True


#Adds waiting time to the process that will be executed
r = t - lot[g][0]
lot[g][1] += r


#Process executes. CPU burst time gets added to the
#updated arrival time and also the time counter
lot[g][0] = t
```

```
lot[g][0] += lot[g][4]

t += lot[g][4]


#Finds CPU burst position and also checks to see

#if this is the last CPU or I/O burst or not

p = lot[g][3]

if lot[g][p+1] == 0:

    #Process has no more CPU or I/O bursts. Terminates process

    lot[g][5] = 0

    rd[g] = 0

    io[g] = 0

    print("Process", lot[g][2], "has finished total execution.")

    print("Total Turnaround Time: ",lot[g][0])


else:

    #Adds the I/O burst time so it can go to I/O Also adjusts array

    #so that the next CPU burst is available when it next executes

    lot[g][0] += lot[g][p + 1]

    lot[g][3] += 2

    lot[g][4] = lot[g][p + 2]


if response == True:

    #Records and prints the response time of the process
```

```
        rt[g][1] = lot[g][1]
        print("Process",g+1, "has gone first execution. Response Time: ",rt[g][1])
    response = False

    print("Current Execution Time:", t) #Prints out current execution time
    for i in range(n): #Checks to see if all processes are done executing
        e += lot[i][5]

    if e == 0: # If all are done executing, loops will end
        progress = False

print("|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||")
# Formula to find the CPU utilization percentage
cpu = 100 - (h / t * 100)
print("Overall time spent being idle: ", h)
print("The last process finished execution at time:", t)
print("FCFS CPU Utilization:", cpu,"%")

# Finds turnaround times. Since the arrival time for all
# processes are 0, the time in which the process ends
# is in fact the turnaround time
for i in range(n):
    tat[i][0] = lot[i][2]
```

```
        tat[i][1] = lot[i][0]
    tat.sort(key=lambda tat: tat[0])
    rt.sort(key=lambda rt: rt[1])

    print("Processes No. " + "Response Time " +
          " Waiting time " + " Turn around time")

    for i in range(n):
        total_wt += lot[i][1]
        total_rt += rt[i][1]
        total_tat += tat[i][1]
        print(" " + str(i + 1) + "\t\t" +
              str(rt[i][1]) + "\t\t " +
              str(lot[i][1]) + "\t\t " +
              str(tat[i][1]))

    print("Average waiting time = " + str(total_wt / n))
    print("Average response time = " + str(total_rt / n))
    print("Average turn around time = " + str(total_tat / n))


if __name__ == "__main__":
    processes = [[5, 27, 3, 31, 5, 43, 4, 18, 6, 22, 4, 26, 3, 24, 4, 0, 0, 0, 0, 0],
```

```
        [4, 48, 5, 44, 7, 42, 12, 37, 9, 76, 4, 41, 9, 31, 7, 43, 8, 0, 0, 0],
        [8, 33, 12, 41, 18, 65, 14, 21, 4, 61, 15, 18, 14, 26, 5, 31, 6, 0, 0, 0],
        [3, 35, 4, 41, 5, 45, 3, 51, 4, 61, 5, 54, 6, 82, 5, 77, 3, 0, 0, 0],
        [16, 24, 17, 21, 5, 36, 16, 26, 7, 31, 13, 28, 11, 21, 6, 13, 3, 11, 4, 0],
        [11, 22, 4, 8, 5, 10, 6, 12, 7, 14, 9, 18, 12, 24, 15, 30, 8, 0, 0, 0],
        [14, 46, 17, 41, 11, 42, 15, 21, 4, 32, 7, 19, 16, 33, 10, 0, 0, 0, 0, 0],
        [4, 14, 5, 33, 6, 51, 14, 73, 16, 87, 6, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
m = 24
n = len(processes)

lot = [[0] * (m + 2) for i in range(n)]
print("1 lot =", lot)

findTimes(processes, n, m, lot)
```

Code for SJF

```
'''

This program uses a two-dimmensional array to store date about the processes.

The first dimmension are the 8  processess indexed from 0-7

The second dimmnension have data relevant to that specific process

Here is what the indexes for the second dimmension means (this is for EACH process)

0: Time in which the process will end up back in ready queue from waiting queue

1: Waiting Time

2: Process Number

3: Index of the where the current CPU burst is

4: Current CPU Burst

5: Indicator for when process is finished

6: First CPU Burst

7: First I/O Burst

8: Second CPU Burst....

'''

print("Hello World. SJF scheduler created in Pycharm")
```

```
# Function finds Waiting, Turnaround, and Response Times
# of both individual process and averages. Also finds
# CPU utilization
def findTimes(processes, n, m, lot):
    io = [0] * n
    bt = [0] * n
    rt = [0] * n
    wt = [0] * n
    tat = [[0] * 2 for i in range(8)]
    total_wt = 0
    total_rt = 0
    total_tat = 0

    t = 0
    r = 0
    f = 1
    b = 3
    d = 5
    m = 6
    tm = [0] * n
    rd = [0] * n
    io = [0] * n
```

```
c = [0] * n

z = 0

h = 0

v = 8

response = False

progress = True

cpu = 0


# Adds all CPU and I/O burst times to array

# and sets up all other fields like position

# of CPU burst time and if process is terminated

for j in range(20):

    for i in range(n):

        lot[i][6 + j] = processes[i][j]

        if j == 0:

            lot[i][2] = i + 1

            lot[i][3] = 6

            lot[i][4] = lot[i][6]

            lot[i][5] = 1


while (progress):

    # Will continue until every process is

    # finished with its burst times
```

```
s = 9999
g = 10
e = 0

for i in range(n):
    # Determines which process will execute next depending on which
    # process has the shortest CPU burst time
    if lot[i][5] == 1 and lot[i][4] < s and lot[i][0] <= t:
        s = lot[i][4]
        g = i
        if lot[i][5] == 1 and lot[i][0] <= t:
            # Places the processes who have arrived in the Ready Queue
            rd[i] = 1
            io[i] = 0
        if lot[i][5] == 1 and lot[i][0] > t:
            # Places the processes who havent arrived yet in the I/O Queue
            io[i] = 1
            rd[i] = 0
        if lot[i][5] == 0:
            # Puts terminated process in terminated queue
            tm[i] = 1

print("||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||")
```

```
if g == 10:
    # If no process is executing, will continue to next iteration in loop
    # Increases idle and time counters. Prints processes in  I/O Queue
    h += 1
    t += 1
    for i in range(n):
        if lot[i][0] > t and lot[i][5] == 1:
            p = lot[i][3]
            print("Process", lot[i][2], "is in the I/O queue with I/O burst of", lot[i][p - 1], "units")
    print("Idle. Total idle time (so far):", h, )
    print("Current Execution Time:", t)
    continue


for i in range(n):
    # Prints out every process thats in the Ready Queue
    if lot[i][0] <= t and lot[i][5] == 1:
        print("Process", lot[i][2], " is in the Ready queue with CPU burst of ", lot[i][4], "units")


for i in range(n):
    # Prints out every process thats in the I/O Queue
    if lot[i][0] > t and lot[i][5] == 1:
        p = lot[i][3]
```

```
        print("Process", lot[i][2], "is in the I/O queue with I/O burst of", lot[i][p - 1], "units")


for i in range(n):
    # Prints out evey process that has been terminated so far
    if lot[i][5] == 0:
        print("Process", lot[i][2], "has already been terminated")


if lot[g][1] == 0 and c[g] == 0:
    # Records if a process has executed for the first time
    c[g] = 1
    response = True


print("Process", g + 1, "is currently executing")


# Adds waiting time to the process that will be executed
r = t - lot[g][0]
lot[g][1] += r


# Process executes. CPU burst time gets added to the
# updated arrival time and also the time counter
lot[g][0] = t
lot[g][0] += lot[g][4]
t += lot[g][4]
```

```
# Finds CPU burst position and also checks to see
# if this is the last CPU or I/O burst or not
p = lot[g][3]
if lot[g][p + 1] == 0:
    # Process has no more CPU or I/O bursts. Terminates process
    lot[g][5] = 0
    rd[g] = 0
    io[g] = 0
    print("Process", lot[g][2], "has finished total execution.")
    print("Total Turnaround Time: ", lot[g][0])
else:
    # Adds the I/O burst time so it can go to I/O Also adjusts array
    # so that the next CPU burst is available when it next executes
    lot[g][0] += lot[g][p + 1]
    lot[g][3] += 2
    lot[g][4] = lot[g][p + 2]

if response == True:
    # Records and prints the response time of the process
    rt[g] = lot[g][1]
    print("Process", g + 1, "has gone first execution. Response Time: ", rt[g])
response = False
```

```
    print("Current Execution Time:", t)  # Prints out current execution time


    # Checks to see if all processes are done executing
    for i in range(n):
        e += lot[i][5]
    # If all are done executing, loops will end
    if e == 0:
        progress = False


print("|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||")
cpu = 100 - (h / t * 100)
print("Overall time spent being idle: ", h)
print("The last process finished execution at time:", t)
print("SJF CPU Utilization:", cpu, "%")


# Finds turnaround times. Since the arrival time for all
# processes are 0, the time in which the process ends
# is in fact the turnaround time
for i in range(n):
    tat[i][0] = lot[i][2]
    tat[i][1] = lot[i][0]
tat.sort(key=lambda tat: tat[0])
```

```
    print("Processes No. " +  "Response Time " +
          " Waiting time " + " Turn around time")


    for i in range(n):
        total_wt += lot[i][1]
        total_rt += rt[i]
        total_tat += tat[i][1]
        print(" " + str(i + 1) + "\t\t" +
              str(rt[i]) + "\t\t " +
              str(lot[i][1]) + "\t\t " +
              str(tat[i][1]))


    print("Average waiting time = " + str(total_wt / n))
    print("Average response time = " + str(total_rt / n))
    print("Average turn around time = " + str(total_tat / n))


# Driver code
if __name__ == "__main__":
```

```
# process id's
processes = [[5, 27, 3, 31, 5, 43, 4, 18, 6, 22, 4, 26, 3, 24, 4, 0, 0, 0, 0, 0],
        [4, 48, 5, 44, 7, 42, 12, 37, 9, 76, 4, 41, 9, 31, 7, 43, 8, 0, 0, 0],
        [8, 33, 12, 41, 18, 65, 14, 21, 4, 61, 15, 18, 14, 26, 5, 31, 6, 0, 0, 0],
        [3, 35, 4, 41, 5, 45, 3, 51, 4, 61, 5, 54, 6, 82, 5, 77, 3, 0, 0, 0],
        [16, 24, 17, 21, 5, 36, 16, 26, 7, 31, 13, 28, 11, 21, 6, 13, 3, 11, 4, 0],
        [11, 22, 4, 8, 5, 10, 6, 12, 7, 14, 9, 18, 12, 24, 15, 30, 8, 0, 0, 0],
        [14, 46, 17, 41, 11, 42, 15, 21, 4, 32, 7, 19, 16, 33, 10, 0, 0, 0, 0, 0],
        [4, 14, 5, 33, 6, 51, 14, 73, 16, 87, 6, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
m = 24
n = len(processes)

# Burst time of all processes
lot = [[0] * (m + 2) for i in range(n)]
print("1 lot =", lot)

findTimes(processes, n, m, lot)
```

Code for MLFQ

'''

This program uses a two-dimmensional array to store date about the processes.

The first dimmension are the 8 processess indexed from 0-7

The second dimmnension have data relevant to that specific process

Here is what the indexes for the second dimmension means (this is for EACH process)

0: Time in which the process will end up back in ready queue from waiting queue

1: Waiting Time

2: Process Number

3: Which queue the process is in

4: Remaining Burst Time of current process

5: Index of the where the current CPU burst is

6: First CPU Burst

7: First I/O Burst

8: Second CPU Burst....

'''

```
print("Hello World. MultiLevel Feedback Queue scheduler created in Pycharm")


# Function finds Waiting, Turnaround, and Response Times
# of both individual process and averages. Also finds
# CPU utilization
def findTimes(processes, n, m, lot):
    io = [0] * n
    bt = [0] * n
    rt = [0] * n
    wt = [0] * n
    tat = [[0] * 2 for i in range(n)]
    totalWT = 0
    totalRT = 0
    totalTAT = 0


    t = 0
    r = 0
    f = 1
    b = 3
    d = 5
    m = 6
    l = 10
    z = 0
```

```
h = 0

tm = [0] * n

rd = [0] * n

io = [0] * n

u = [0] * n

last = 1

tres = 0

dos = 0


c = [0] * 2

c[0] = 10

c[1] = 0

tq1 = 5

tq2 = 10

p = 0

cpu = 0

progress = True

response = False

t = 0


# Adds all CPU and I/O burst times to array

# and sets up all other fields like position

# of CPU burst time and if process is terminated
```

```
for j in range(19):
    for i in range(n):
        lot[i][6 + j] = processes[i][j]
        if j == 0:
            lot[i][2] = i + 1
            lot[i][3] = 1
            lot[i][4] = lot[i][6]
            lot[i][5] = 6


while (progress):
    # Will continue until every process is
    # finished with its burst times
    s = 9999
    g = 10
    x = 3


    print("||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||| ||||||||||||||||||||||||||||||||||||||||")


    for j in range(3):
        for i in range(n):
            # Determines which process should go next based on which queue
            # it is in and also and if has arrived yet
            if lot[i][3] == x and lot[i][0] < s and lot[i][0] <= t:
```

```
            s = lot[i][0]
            g = i
            if lot[i][3] != 0 and lot[i][0] <= t:
                # Places the processes who have arrived in the Ready Queue
                rd[i] = 1
                io[i] = 0
            if lot[i][3] != 0 and lot[i][0] > t:
                # Places the processes who havent arrived yet in the I/O Queue
                io[i] = 1
                rd[0] = 0
            if lot[i][5] == 0:
                # Puts terminated process in terminated queue
                tm[i] = 1
        x -= 1  # Checks next, higher queue
        s = 9999


    if g == 10:
        # If no process is executing, will continue to next iteration in loop
        # Increases idle and time counters. Prints processes in  I/O Queue
        h += 1
        t += 1
        for i in range(n):
            if lot[i][0] > t and lot[i][3] != 0:
```

```
            p = lot[i][5]
            print("Process", lot[i][2], "is in the I/O queue with I/O burst of", lot[i][p - 1], "units")
    print("Idle. Total idle time (so far):", h, )
    print("Current Execution Time:", t)
    continue

for i in range(n):
    # Prints out every process thats in the Ready Queue
    if lot[i][0] <= t and lot[i][3] != 0:
        print("Process", lot[i][2], " is in the Ready queue with CPU burst of ", lot[i][4], "units")

for i in range(n):
    # Prints out every process thats in the I/O Queue
    if lot[i][0] > t and lot[i][3] != 0:
        p = lot[i][5]
        print("Process", lot[i][2], "is in the I/O queue with I/O burst of", lot[i][p - 1], "units")

for i in range(n):
    # Prints out evey process that has been terminated so far
    if lot[i][3] == 0:
        print("Process", lot[i][2], "has already been terminated")

if lot[g][1] == 0 and u[g] == 0:
```

```
    u[g] = 1
    response = True

# If last process executed was in queue 3 and current process
# is in queue 3, the last process will be executing now
if last == 3 and lot[g][3] == 3:
    g = l

# If last process executed was in queue 2 and current process
# is in queue 2, the last process will be executing now
if last == 2 and lot[g][3] == 2 and c[0] != 10:
    g = c[0]

# If current process is in queue 3, the following will execute
if lot[g][3] == 3:
    t += 1
    l = g
    c[0] = 10
    c[1] = 0

    # Adds waiting time to the process that will be executed
    r = t - lot[g][0]
    lot[g][1] += r
```

```
# Updates arrival time and decreases remaining CPU burst time
p = lot[g][5]
lot[g][0] = t
lot[g][4] -= 1

if lot[g][4] == 0 and lot[g][p + 1] != 0:
    # Adds I/O burst time to the process that has executed
    # Positions process for next CPU burst time
    c[0] = 10
    c[1] = 0
    lot[g][0] += lot[g][p + 1]
    lot[g][5] += 2
    p += 2
    lot[g][4] = lot[g][p]

if lot[g][4] == 0 and lot[g][p + 1] == 0:
    # Process will be terminated if no more bursts to run
    print("process", lot[g][2], " has ended")
    lot[g][3] = 0
    lot[g][4] = 0
last = 3
print("Process", g + 1, "is executing and is in Queue 3")
```

```
# print("green")


# If current process is in queue 2, the following will execute
elif lot[g][3] == 2:
    # Counter for the amount of time units that
    # a process in queue 2 has been executing
    t += 1
    if c[0] == g:
        c[1] += 1
    else:
        c[0] = g
        c[1] = 1

    # Adds waiting time to the process that will be executed
    # Decreases remaining CPU burst time
    p = lot[g][5]
    if c[1] == 1:
        r = t - lot[g][0]
        lot[g][1] += r
    lot[g][4] -= 1

    if c[0] == g and c[1] == tq2 and lot[g][4] > 0:
        # If process still has CPU burst time after it reaches
```

```
    # time quantum of 10 units, process moved to Queue 3
    print("Process", g + 1, "will be moved to queue 3")
    lot[g][3] = 3
    c[0] = 10
    c[1] = 0

lot[g][0] = t
if lot[g][4] == 0 and lot[g][p + 1] != 0:
    # Adds I/O burst time to the process that has executed
    # Positions the process for next CPU burst time
    c[0] = 10
    c[1] = 0
    lot[g][0] += lot[g][p + 1]
    lot[g][5] += 2
    p += 2
    lot[g][4] = lot[g][p]

if lot[g][4] == 0 and lot[g][p + 1] == 0:
    # If the process has no more burst times left, will terminate
    print("process has ended")
    lot[g][3] = 0
    lot[g][4] = 0
l = 10
```

```
    last = 2
    print("Process", g + 1, "is executing and is in Queue 2")

# If current process is in queue 1, the following will execute
else:  # lot[g][3] == 1
    # Stops the counting for for a process was executing in queue 2
    l = 10
    last = 1
    c[0] = 10
    c[1] = 0

    # Adds waiting time to the process that will be executed
    # Also finds CPU burst time for current process
    r = t - lot[g][0]
    lot[g][1] += r
    p = lot[g][5]
    lot[g][0] = t

    if lot[g][4] > tq1:
        # If current burst is larger that time quantum (5 time units)
        # will be moved to Queue 2. Time updated
        print("Process",g+1,"will be moved to queue 2")
        lot[g][3] = 2
```

```
    lot[g][0] += tq1
    lot[g][4] -= tq1
    t += tq1

else:
    # If current burst is not larger than time quantum, it
    # will execute fully the remaining current burst
    lot[g][0] += lot[g][4]
    t += lot[g][4]
    if lot[g][p + 1] == 0:
        # Process will end if no more burst times remaining
        print("Process",g+1," has ended")
        lot[g][3] = 0
        lot[g][4] = 0

    else:
        # Adds I/O to the process so that it will come back
        # after that I/O time has been fullfilled
        lot[g][0] += lot[g][p + 1]
        lot[g][5] += 2
        p += 2
        lot[g][4] = lot[g][p]
print("Process", g + 1, "is executing and is in Queue 1")
```

```
    if response == True:
        rt[g] = lot[g][1]
        print("Process", g + 1, "has gone for the first time")
        print("Response time: ", rt[g])
    response = False

    print("Current Execution Time: ", t)

    # Checks to see if all processes are done executing
    e = 0
    for i in range(n):
        e += lot[i][4]
    # If all are done executing, loops will end
    if e == 0:
        progress = False

lot.sort(key=lambda lot: lot[2])
print("||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||")
cpu = 100 - (h / t * 100)
print("Overall time spent being idle: ", h)
print("The last process finished execution at time:", t)
print("MLFQ CPU Utilization:", cpu, "%")
```

```
# Finds turnaround times. Since the arrival time for all
# processes are 0, the time in which the process ends
# is in fact the turnaround time
for j in range(n):
    tat[j][0] = lot[j][2]
    for i in range(6, 25):
        tat[j][1] += lot[j][i]
tat.sort(key=lambda tat: tat[0])
for i in range(n):
    tat[i][1] += lot[i][1]

# Print process names
print("Processes No. " +  "Response Time " +
      " Waiting time " + " Turn around time")

for i in range(n):
    totalWT += lot[i][1]
    totalRT += rt[i]
    totalTAT += tat[i][1]
    print(" " + str(i + 1) + "\t\t" +
          str(rt[i]) + "\t\t " +
          str(lot[i][1]) + "\t\t " +
```

```
            str(tat[i][1]))

    print("Average waiting time = " + str(totalWT / n))
    print("Average response time = " + str(totalRT / n))
    print("Average turn around time = " + str(totalTAT / n))

# Driver code
if __name__ == "__main__":
    # process id's
    processes = [[5, 27, 3, 31, 5, 43, 4, 18, 6, 22, 4, 26, 3, 24, 4, 0, 0, 0, 0, 0],
                 [4, 48, 5, 44, 7, 42, 12, 37, 9, 76, 4, 41, 9, 31, 7, 43, 8, 0, 0, 0],
                 [8, 33, 12, 41, 18, 65, 14, 21, 4, 61, 15, 18, 14, 26, 5, 31, 6, 0, 0, 0],
                 [3, 35, 4, 41, 5, 45, 3, 51, 4, 61, 5, 54, 6, 82, 5, 77, 3, 0, 0, 0],
                 [16, 24, 17, 21, 5, 36, 16, 26, 7, 31, 13, 28, 11, 21, 6, 13, 3, 11, 4, 0],
                 [11, 22, 4, 8, 5, 10, 6, 12, 7, 14, 9, 18, 12, 24, 15, 30, 8, 0, 0, 0],
                 [14, 46, 17, 41, 11, 42, 15, 21, 4, 32, 7, 19, 16, 33, 10, 0, 0, 0, 0, 0],
                 [4, 14, 5, 33, 6, 51, 14, 73, 16, 87, 6, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
    m = 24
    n = len(processes)

    lot = [[0] * (m + 2) for i in range(n)]

    findTimes(processes, n, m, lot)
```