Programming Assignment

10.22.2023

—

Chase Moffat

Programming Assignment

Borivoje Furht

COP 4610-001

Z23550757

October 22 2023

Table of Contents

Introduction

The CPU scheduling programming assignment gave us the task to implement three different CPU scheduling algorithms. I chose to use Python as my programming language for this project. I coded my algorithms in Google Colab to give me the ability to run different code cells which I felt would be very useful for this project.

The first algorithm I implemented was the FCFS Non-preemptive algorithm. This algorithm starts by initializing a ready queue that holds all the processes. The order that these processes run their CPU bursts depends on the time they arrive in the queue. But in this project we are assuming each process arrives at 0. Which means each process will just initially run in order from P1 to P8.

The second algorithm I implemented was the SJF Non-preemptive algorithm. This algorithm is similar to FCFS except for the order the processes run. The order that each process runs their CPU burst is determined by which burst is the shortest. Each time a process is added to the ready queue, the queue is then re sorted so that the process with the shortest cpu burst gets priority to run.

The third and final algorithm that I implemented was the MLFQ. The MLFQ has three different queues. The first two are Round Robin queues and the third queue is a FCFS. All processes start in the first Round Robin queue with a tq time of 5. If the processes CPU burst exceeds this tq limit it is then downgraded to the second queue which has a tq time of 10. If a

processes CPU burst were to also exceed the second queue tq time of 10, it would then be downgraded to the FCFS queue.

Throughout the remainder of my report I will display my code for each algorithm as well as the logic behind them. I will also include the results from each algorithm, as well as tables to compare the results. Finally I will go into discussion comparing the difference between the algorithms as well as some general thoughts that I gathered while completing this assignment.

The Process Class

```
"""
The process class stores each processes CPU and I/O bursts, process id, total burst time,
wait time, turnaround time, and response time.
"""
class Process:

    def __init__(self, pid, bursts):
        self.pid = pid
        self.bursts = bursts
        self.total_burst_time = sum(bursts) #Total burst time to calculate wait time
        self.wait_time = 0
        self.turnaround_time = 0
        self.response_time = 1 #Set to 1 initially
```

```
P1 = Process(1, [5, 27, 3, 31, 5, 43, 4, 18, 6, 22, 4, 26, 3, 24, 4])
P2 = Process(2, [4, 48, 5, 44, 7, 42, 12, 37, 9, 76, 4, 41, 9, 31, 7, 43, 8])
P3 = Process(3, [8, 33, 12, 41, 18, 65, 14, 21, 4, 61, 15, 18, 14, 26, 5, 31, 6])
P4 = Process(4, [3, 35, 4, 41, 5, 45, 3, 51, 4, 61, 5, 54, 6, 82, 5, 77, 3])
P5 = Process(5, [16, 24, 17, 21, 5, 36, 16, 26, 7, 31, 13, 28, 11, 21, 6, 13, 3, 11, 4])
P6 = Process(6, [11, 22, 4, 8, 5, 10, 6, 12, 7, 14, 9, 18, 12, 24, 15, 30, 8])
P7 = Process(7, [14, 46, 17, 41, 11, 42, 15, 21, 4, 32, 7, 19, 16, 33, 10])
P8 = Process(8, [4, 14, 5, 33, 6, 51, 14, 73, 16, 87, 6])

processes = [P1, P2, P3, P4, P5, P6, P7, P8]

#Total for cpu bursts to later calculate cpu utilization
burst_time = 553
```

Above is the code for the Process class. This was a design choice I decided to implement as an easy way to keep track of each process's individual stats such as wait time, response time, and turnaround time. The class has eight instances for each process. Each instance has its own unique id, bursts, total_burst_time, wait_time, turnaround_time, and response_time. The total_burst_time is referenced in each algorithm as a way to calculate wait time by subtracting turnaround time with burst time. The response time is also set to 1 initially. In all of my scheduling functions there is an if statement to check if the response time is equal to 1, if it is then it sets the response time to the clock value. This ensures the response time is set for each process's first time running a CPU burst.

FCFS Non-preemptive

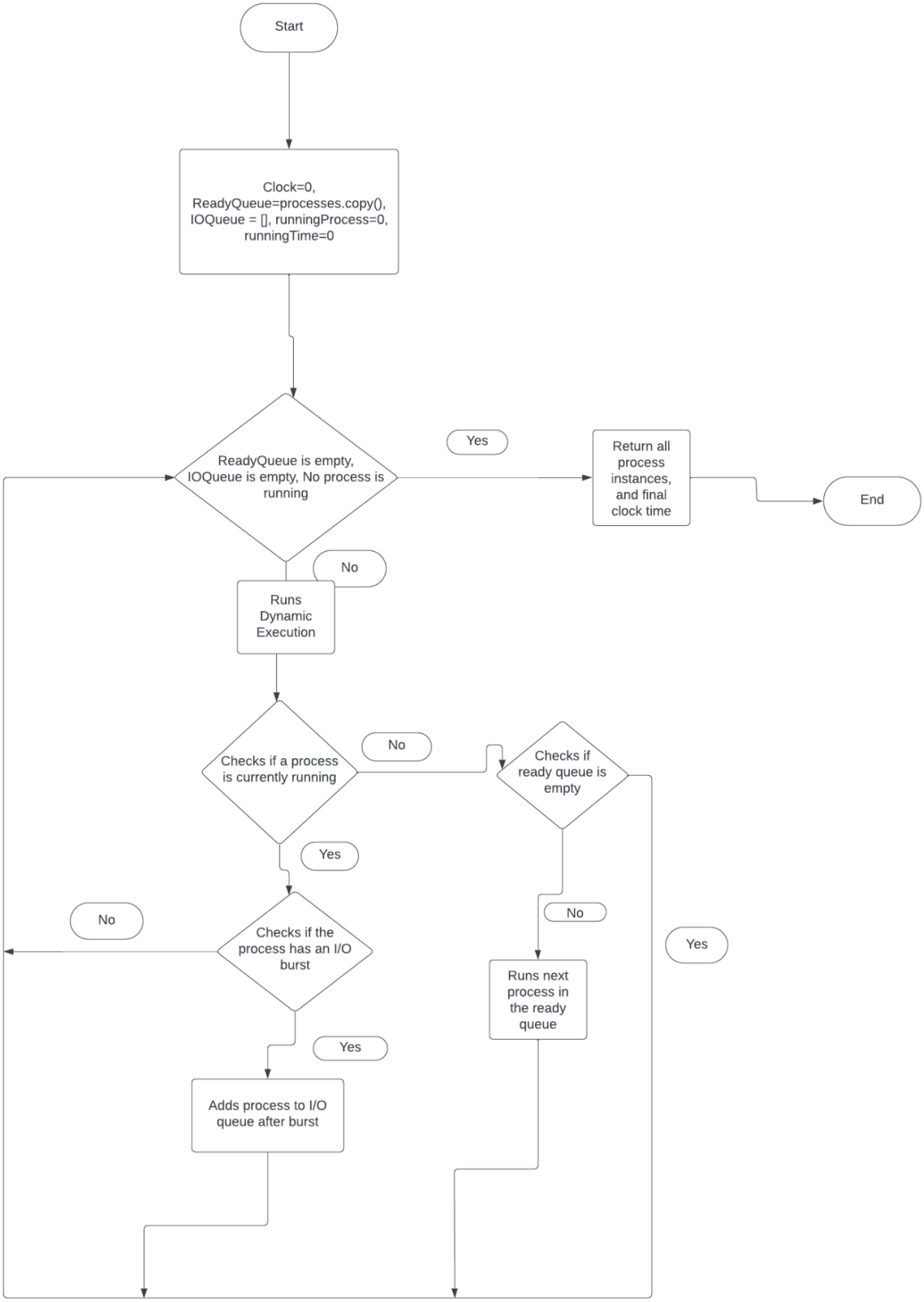
```
def fcfs(processes):
    clock = 0
    ready_queue = processes.copy()  #Initializes a copy of all processes that start in the ready queue
    io_queue = []
    running_process = 0  #0 means no process is running
    running_time = 0

    def printExecution():
      print(f"Clock: {clock}")

      if running_process == 0:
          print("Running Process: None")
      else:
          print(f"Running Process: P{running_process.pid} Burst Time: {running_process.bursts[0]}")

      print("Ready Queue:", [f"P{p.pid}({p.bursts})" for p in ready_queue])
      print("I/O Queue:", [f"P{p[0].pid}({p[1]})" for p in io_queue])
      print("\n")

    #checks if there are processes in ready queue, io queue, or if a process is currently running
    while len(ready_queue) > 0 or len(io_queue) > 0 or running_process != 0:
```

```
  printExecution()

#If a process is running, running time gets incremented
  if running_process != 0:
     running_time += 1
     #If the run time equals the burst value it removes the process
     if running_time == running_process.bursts[0]:
        running_process.bursts.pop(0)
        #if a process finishes its cpu burst and still has values, it gets sent to the io queue
        if running_process.bursts:
           io_time = running_process.bursts.pop(0)
           #adds the process and their io burst value to the io queue
           io_queue.append((running_process, io_time))
        #else the process has finished
        else:
           running_process.turnaround_time = clock
           print(f"Process P{running_process.pid} has completed its total execution.")
        #resets the run time and running process
        running_process = 0
        running_time = 0

  #Checks if there are not processes currently running
  #Also checks if there are processes in the ready queue
  if running_process == 0 and len(ready_queue) > 0:
    #Sets the next process in the queue to run its cpu burst
     running_process = ready_queue.pop(0)
     running_time = 0
```

```
        if running_process.response_time == 1:
          #All processes are initialized with a response time of 1
          #This then sets the response time to the clock value if its their first time running
            running_process.response_time = clock

    #creates a new io queue
    new_io_queue = []
    #checks if a process still has time left for its io burst
    for p, time_left in io_queue:
        time_left -= 1
        #once a process has finished it is sent back to ready queue
        if time_left == 0:
            ready_queue.append(p)
        #else the remaining processes are added to a new queue to finish their io bursts
        else:
            new_io_queue.append((p, time_left))
    #remakes the io queue and increments the clock by 1
    io_queue = new_io_queue
    clock += 1

#Calculates wait time for all processes
for p in processes:
    p.wait_time = p.turnaround_time - p.total_burst_time
#returns each process instance and the final run time
return processes, clock
```

```
#Generates all the final results and prints them to the terminal
def generateStats(processData, clock):
  #prints all the final stats
    avg_wait_time = sum(p.wait_time for p in processes) / 8
    avg_turnaround_time = sum(p.turnaround_time for p in processes) / 8
    avg_response_time = sum(p.response_time for p in processes) / 8
    #burst_time is the sum of all the cpu bursts
    cpu_utilization = ( burst_time / clock)

    print("Process\tWaiting Time\tTurnaround Time\tResponse Time")
    for p in processData:
       print(f"P{p.pid}\t{p.wait_time}\t\t{p.turnaround_time}\t\t{p.response_time}")
    print(f"\nAverage Waiting Time: {avg_wait_time}")
    print(f"Average Turnaround Time: {avg_turnaround_time}")
    print(f"Average Response Time: {avg_response_time}")
    print(f"CPU Utilization: {cpu_utilization}%")
    print(f"Total Run Time: {clock}")

if __name__ == "__main__":
    processData, clock = fcfs(processes)
    generateStats(processData, clock)
```

Sequence Diagram:


Start
Clock=0, ReadyQueue=processes.copy(), IOQueue = [], runningProcess=0, runningTime=0
ReadyQueue is empty, IOQueue is empty, No process is running
Yes
Return all process instances, and final clock time
End
No
Runs Dynamic Execution
Checks if a process is currently running
No
Checks if ready queue is empty
Yes
No
Checks if the process has an I/O burst
No
Yes
Runs next process in the ready queue
Yes
Adds process to I/O queue after burst

Dynamic Execution & Results:

```
Clock: 209
Running Process: P6 Burst Time: 7
Ready Queue: ['P5([16, 26, 7, 31, 13, 28, 11, 21, 6, 13, 3, 11, 4])']
I/O Queue: ['P3(16)', 'P4(1)', 'P2(10)', 'P8(25)', 'P1(2)', 'P7(37)']


Clock: 210
Running Process: P6 Burst Time: 7
Ready Queue: ['P5([16, 26, 7, 31, 13, 28, 11, 21, 6, 13, 3, 11, 4])', 'P4([3, 51, 4, 61, 5, 54, 6, 82, 5, 77, 3])']
I/O Queue: ['P3(15)', 'P2(9)', 'P8(24)', 'P1(1)', 'P7(36)']


Clock: 211
Running Process: P6 Burst Time: 7
Ready Queue: ['P5([16, 26, 7, 31, 13, 28, 11, 21, 6, 13, 3, 11, 4])', 'P4([3, 51, 4, 61, 5, 54, 6, 82, 5, 77, 3])', 'P1([6, 22, 4, 26, 3, 24, 4])']
I/O Queue: ['P3(14)', 'P2(8)', 'P8(23)', 'P7(35)']


Clock: 212
Running Process: P5 Burst Time: 16
Ready Queue: ['P4([3, 51, 4, 61, 5, 54, 6, 82, 5, 77, 3])', 'P1([6, 22, 4, 26, 3, 24, 4])']
I/O Queue: ['P3(13)', 'P2(7)', 'P8(22)', 'P7(34)', 'P6(13)']


Clock: 213
Running Process: P5 Burst Time: 16
Ready Queue: ['P4([3, 51, 4, 61, 5, 54, 6, 82, 5, 77, 3])', 'P1([6, 22, 4, 26, 3, 24, 4])']
I/O Queue: ['P3(12)', 'P2(6)', 'P8(21)', 'P7(33)', 'P6(12)']
```

```
Process Waiting Time    Turnaround Time Response Time
P1      170             395             0
P2      164             591             5
P3      165             557             9
P4      164             648             17
P5      221             530             20
P6      230             445             36
P7      184             512             47
P8      184             493             61

Average Waiting Time: 185.25
Average Turnaround Time: 521.375
Average Response Time: 24.375
CPU Utilization: 0.8520801232665639%
Total Run Time: 649
```

SJF Non-preemptive

#Funtion to sort processes based on shortest burst times

def insert_sorted(readyQueue, process):

    i = 0

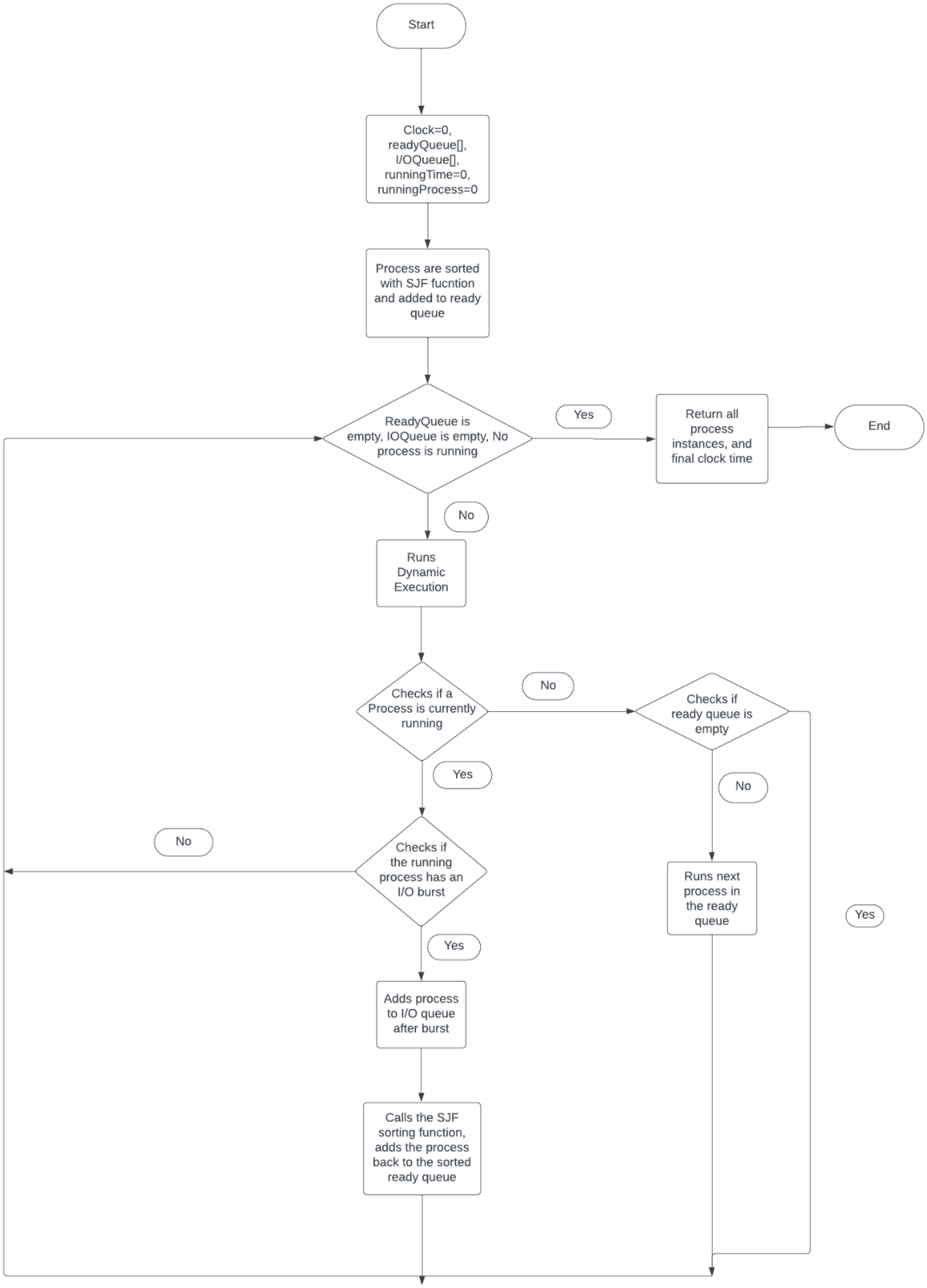
```
    while i < len(readyQueue) and readyQueue[i].bursts[0] <= process.bursts[0]:
        i += 1
    #Adds the sorted process to the ready queu
    readyQueue.insert(i, process)

def sjf(processes):
    clock = 0
    ready_queue = []
    io_queue = []
    running_process = 0 #0 means no process is running
    running_time = 0

    def printExecution():
      #Prints the current run time, the current running process, and the ready / IO queues
      print(f"Clock: {clock}")

      if running_process == 0:
          print("Running Process: None")
      else:
          print(f"Running Process: P{running_process.pid} Burst Time:
{running_process.bursts[0]}")

      print("Ready Queue:", [f"P{p.pid}({p.bursts})" for p in ready_queue])
      print("I/O Queue:", [f"P{p[0].pid}({p[1]})" for p in io_queue])
      print("\n")

    #Sorts the processes and adds them to  the ready queue
```

```
for p in processes:
    insert_sorted(ready_queue, p)

#checks if there are processes in ready queue, io queue, or if a process is currently running
while len(ready_queue) > 0 or len(io_queue) > 0 or running_process != 0:
    printExecution()
    #If a process is running, running time gets incremented
    if running_process:
        running_time += 1
        #If the run time equals the burst value it removes the process
        if running_time == running_process.bursts[0]:
            running_process.bursts.pop(0)
            #if a process finishes its cpu burst and still has bursts to run, it gets sent to the io queue
            if running_process.bursts:
                #adds the io burst time
                io_time = running_process.bursts.pop(0)
                #adds the process to the io queue
                io_queue.append((running_process, io_time))
            #else the process has finished
            else:
                running_process.turnaround_time = clock
                print(f"Process P{running_process.pid} has completed its total execution.")

            #resets the run time and running process
            running_process = 0
            running_time = 0
```

```
    #Checks if there are not processes currently running
    #Also checks if there are processes in the ready queue
    if running_process == 0 and len(ready_queue) > 0:
      #Sets the next process in the queue to run its cpu burst
      running_process = ready_queue.pop(0)

      if running_process.response_time == 1:
        #All processes are initialized with a response time of 1
        #This then sets the response time to the clock value if its their first time running
          running_process.response_time = clock

    #creates a new io queue
    new_io_queue = []
    #checks if a process still has time left for its io burst
    for p, time_left in io_queue:
        time_left -= 1
      #once a process has finished it is sorted by sjf and added to ready queue
        if time_left == 0:
            insert_sorted(ready_queue, p)
      #else the remaining processes are added to a new queue to finish their io bursts
        else:
            new_io_queue.append((p, time_left))
    #remakes the io queue and increments the clock by 1
    io_queue = new_io_queue
    clock += 1
#Calculates wait time for all processes
for p in processes:
```

```
        p.wait_time = p.turnaround_time - p.total_burst_time
    #returns each process instance and the final run time
    return processes, clock


def generateStats(processData, clock):
  #Prints all the final stats
    avg_wait_time = sum(p.wait_time for p in processes) / 8
    avg_turnaround_time = sum(p.turnaround_time for p in processes) / 8
    avg_response_time = sum(p.response_time for p in processes) / 8
    cpu_utilization = (burst_time / clock)

    print("Process\tWaiting Time\tTurnaround Time\tResponse Time")
    for p in processData:
        print(f"P{p.pid}\t{p.wait_time}\t\t{p.turnaround_time}\t\t{p.response_time}")
    print(f"\nAverage Waiting Time: {avg_wait_time}")
    print(f"Average Turnaround Time: {avg_turnaround_time}")
    print(f"Average Response Time: {avg_response_time}")
    print(f"CPU Utilization: {cpu_utilization}%")
    print(f"Total Run Time: {clock}")

if __name__ == "__main__":
    processData, clock = sjf(processes)
    generateStats(processData, clock)
```

Sequence Diagram:


Start
Clock=0, readyQueue[], I/OQueue[], runningTime=0, runningProcess=0
Process are sorted with SJF fucntion and added to ready queue
ReadyQueue is empty, IOQueue is empty, No process is running
Yes
Return all process instances, and final clock time
End
No
Runs Dynamic Execution
Checks if a Process is currently running
No
Checks if ready queue is empty
Yes
No
Yes
No
Checks if the running process has an I/O burst
Runs next process in the ready queue
Yes
Yes
Adds process to I/O queue after burst
Calls the SJF sorting function, adds the process back to the sorted ready queue

Dynamic Execution & Results:

```
Clock: 108
Running Process: P6 Burst Time: 6
Ready Queue: ['P5([16, 24, 17, 21, 5, 36, 16, 26, 7, 31, 13, 28, 11, 21, 6, 13, 3, 11, 4])', 'P7([17, 41, 11, 42, 15, 21, 4, 32, 7, 19, 16, 33, 10])']
I/O Queue: ['P2(6)', 'P3(15)', 'P1(27)', 'P4(34)', 'P8(46)']


Clock: 109
Running Process: P6 Burst Time: 6
Ready Queue: ['P5([16, 24, 17, 21, 5, 36, 16, 26, 7, 31, 13, 28, 11, 21, 6, 13, 3, 11, 4])', 'P7([17, 41, 11, 42, 15, 21, 4, 32, 7, 19, 16, 33, 10])']
I/O Queue: ['P2(5)', 'P3(14)', 'P1(26)', 'P4(33)', 'P8(45)']


Clock: 110
Running Process: P5 Burst Time: 16
Ready Queue: ['P7([17, 41, 11, 42, 15, 21, 4, 32, 7, 19, 16, 33, 10])']
I/O Queue: ['P2(4)', 'P3(13)', 'P1(25)', 'P4(32)', 'P8(44)', 'P6(11)']


Clock: 111
Running Process: P5 Burst Time: 16
Ready Queue: ['P7([17, 41, 11, 42, 15, 21, 4, 32, 7, 19, 16, 33, 10])']
I/O Queue: ['P2(3)', 'P3(12)', 'P1(24)', 'P4(31)', 'P8(43)', 'P6(10)']


Clock: 112
Running Process: P5 Burst Time: 16
Ready Queue: ['P7([17, 41, 11, 42, 15, 21, 4, 32, 7, 19, 16, 33, 10])']
I/O Queue: ['P2(2)', 'P3(11)', 'P1(23)', 'P4(30)', 'P8(42)', 'P6(9)']
```

```
Process Waiting Time    Turnaround Time Response Time
P1      43              268             11
P2      73              500             3
P3      276             668             16
P4      50              534             0
P5      237             546             109
P6      121             336             24
P7      149             477             47
P8      119             428             7

Average Waiting Time: 133.5
Average Turnaround Time: 469.625
Average Response Time: 27.125
CPU Utilization: 0.8266068759342302%
Total Run Time: 669
```

MLFQ

```
def mlfq(processes):
    clock = 0

    queue1 = processes.copy() #Initialzes a copy of all processes that start in RR queue 1
    queue2 = []
```

```
    queue3 = []
    io_queue = []

    running_process = 0 #0 means no process is running
    running_time = 0
    tq1 = 5
    tq2 = 10
    tq_count = 0 #used to keep track of tq time

    def printExecution():
      #Prints the current run time, the current running process, and the ready / IO queues
      print(f"Clock: {clock}")

      if running_process == 0:
         print("Running Process: None")
      else:
         print(f"Running Process: P{running_process.pid} Burst Time:
{running_process.bursts[0]}")

      print("Queue 1:", [f"P{p.pid}({p.bursts})" for p in queue2])
      print("Queue 2:", [f"P{p.pid}({p.bursts})" for p in queue2])
      print("Queue 3:", [f"P{p.pid}({p.bursts})" for p in queue3])
      print("I/O Queue:", [f"P{p[0].pid}({p[1]})" for p in io_queue])
      print("\n")

    #checks if there are processes in ready queue, io queue, or if a process is currently running
    while len(queue1) > 0 or len(queue2) > 0 or len(queue3) > 0 or len(io_queue) > 0  or
running_process !=  0:
```

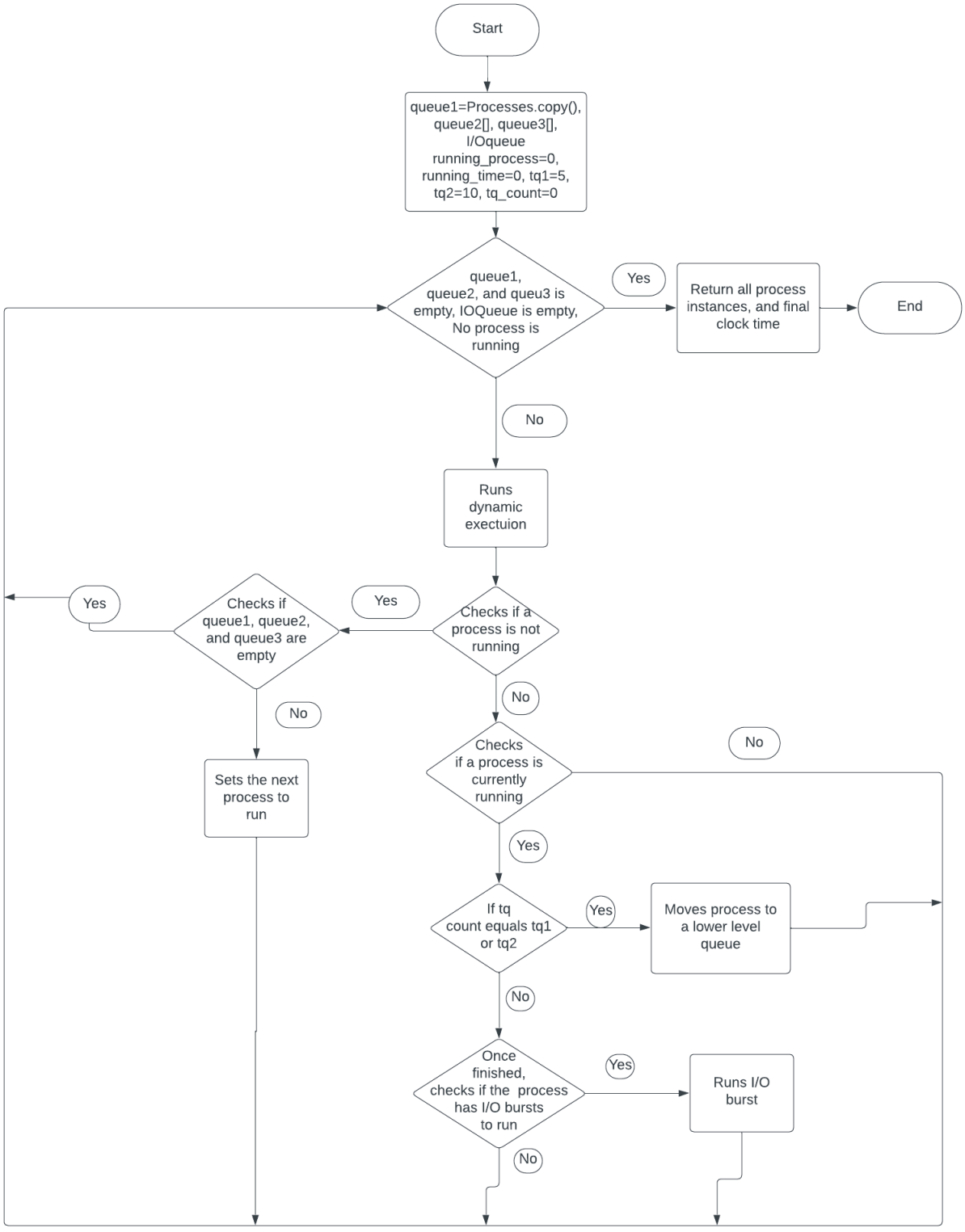
```
printExecution()

#If no process is running it checks all the queues
if running_process == 0:
    if queue1:
        running_process = queue1.pop(0)
        running_time = 0
        tq_count = tq1

    elif queue2:
        running_process = queue2.pop(0)
        running_time = 0
        tq_count = tq2

    elif queue3:
        running_process = queue3.pop(0)
        running_time = 0

#Checsk if a process is running
if running_process != 0:

    if running_process.response_time == 1:
      #All processes are initialized with a response time of 1
      #This then sets the response time to the clock value if its their first time running
        running_process.response_time = clock
```

```
#Subtracts from the current processes cpu burst and increments run time
running_process.bursts[0] -= 1
running_time += 1
#Checks if a burst is finished and removes  it
if running_process.bursts[0] == 0:
    running_process.bursts.pop(0)
    #If a process has more burst after the cpu, it is added to the I/O queue
    if running_process.bursts:
        io_time = running_process.bursts.pop(0)
        io_queue.append((running_process, io_time))


    else:
        #Else a process is finished, assigns turnaround time
        running_process.turnaround_time = clock
        print(f"Process P{running_process.pid} has completed its total execution.")
    #Resets running process and time
    running_process = 0
    running_time = 0
#Checks if a run time reaches a tq count
elif running_time == tq_count:
    #If a process in queue 1 exceeds tq(5), the process is added to RR queue 2
    if tq_count == tq1:
        queue2.append(running_process)
    #If a process in queue 2 exceeds tq(10), the process is added to FCFS queue 3
    elif tq_count == tq2:
        queue3.append(running_process)
    #Resets running process and time
```

```
            running_process = 0
            running_time = 0



    #creates a new io queue
    new_io_queue = []
    #checks if a process still has time left for its io burst
    for p, time_left in io_queue:
      #decreases the cpu bursts by 1
        time_left -= 1
        #Checks if a I/O burst is done
        if time_left == 0:
            #Removes the process and adds it back to queue 1
            queue1.append(p)
      #else the remaining processes are added to a new queue to finish their io bursts
        else:
             new_io_queue.append((p, time_left))
    #remakes the io queue and increments the clock by 1
    io_queue = new_io_queue
    clock += 1
#Calculates wait time for all processes
for p in processes:
        p.wait_time = p.turnaround_time - p.total_burst_time

#returns each process instance and the final run time
return processes, clock
```

```
def generateStats(processData, clock):
    # Prints all the final stats
    avg_wait_time = sum(p.wait_time for p in processData) / 8
    avg_turnaround_time = sum(p.turnaround_time for p in processData) / 8
    avg_response_time = sum(p.response_time for p in processData) / 8
    cpu_utilization = (burst_time / clock)

    print("Process\tWaiting Time\tTurnaround Time\tResponse Time")
    for p in processData:
        print(f"P{p.pid}\t{p.wait_time}\t\t{p.turnaround_time}\t\t{p.response_time}")
    print(f"\nAverage Waiting Time: {avg_wait_time}")
    print(f"Average Turnaround Time: {avg_turnaround_time}")
    print(f"Average Response Time: {avg_response_time}")
    print(f"CPU Utilization: {cpu_utilization}%")
    print(f"Total Run Time: {clock}")

if __name__ == "__main__":

  processData, clock = mlfq(processes)
  generateStats(processData, clock)
```

Sequence Diagram:



Dynamic Execution & Results:

```
Clock: 526
Running Process: P5 Burst Time: 7
Queue 1: ['P2([3])']
Queue 2: ['P2([3])']
Queue 3: ['P7([1, 33, 10])']
I/O Queue: ['P4(37)', 'P3(5)']


Clock: 527
Running Process: None
Queue 1: ['P2([3])', 'P5([6, 21, 6, 13, 3, 11, 4])']
Queue 2: ['P2([3])', 'P5([6, 21, 6, 13, 3, 11, 4])']
Queue 3: ['P7([1, 33, 10])']
I/O Queue: ['P4(36)', 'P3(4)']


Clock: 528
Running Process: P2 Burst Time: 2
Queue 1: ['P5([6, 21, 6, 13, 3, 11, 4])']
Queue 2: ['P5([6, 21, 6, 13, 3, 11, 4])']
Queue 3: ['P7([1, 33, 10])']
I/O Queue: ['P4(35)', 'P3(3)']


Clock: 529
Running Process: P2 Burst Time: 1
Queue 1: ['P5([6, 21, 6, 13, 3, 11, 4])']
Queue 2: ['P5([6, 21, 6, 13, 3, 11, 4])']
Queue 3: ['P7([1, 33, 10])']
I/O Queue: ['P4(34)', 'P3(2)']


Process P2 has completed its total execution.
Clock: 530
Running Process: None
Queue 1: ['P5([6, 21, 6, 13, 3, 11, 4])']
Queue 2: ['P5([6, 21, 6, 13, 3, 11, 4])']
Queue 3: ['P7([1, 33, 10])']
I/O Queue: ['P4(33)', 'P3(1)']
```

```
Process P5 has completed its total execution.
Process Waiting Time    Turnaround Time Response Time
P1      47              272             0
P2      102             529             5
P3      192             584             9
P4      81              565             14
P5      288             597             17
P6      197             412             22
P7      261             589             27
P8      197             506             32

Average Waiting Time: 170.625
Average Turnaround Time: 506.75
Average Response Time: 15.75
CPU Utilization: 0.9247491638795987%
Total Run Time: 598
```

Final Results

	SJF	FCFS	MLFQ
CPU Utilization	82.66%	85.21%	92.47%
Avg Waiting Time(Tw)	133.5	185.25	170.63
Avg Turnaround Time(Ttr)	469.63	521.38	506.75
Avg Response Time(Tr)	27.13	24.38	15.75

SJF	FCFS	MLFQ
CPU Utilization: 82.66%	CPU Utilization: 85.21%	CPU Utilization: 92.47%

	Tw	Ttr	Tr	Tw	Ttr	Tr	Tw	Ttr	Tr
P1	43	268	11	170	395	0	47	272	0
P2	73	500	3	164	591	5	102	529	5
P3	276	668	16	165	557	9	192	584	9
P4	50	534	0	164	648	17	81	565	14
P5	237	546	109	221	530	20	288	597	17
P6	121	336	24	230	445	36	197	412	22
P7	149	477	47	184	512	47	261	589	27
P8	119	428	7	184	493	61	197	506	32
AVG	113.5	469.63	27.13	185.25	521.38	24.38	170.63	506.75	15.75

Discussion & Conclusion

Observations I made while completing this assignment were how different some of the algorithms were. The main difference I noticed was the difference of response times. For these processes the SJF algorithm had very high response times, for example in my program P5 did not begin its first process until the 109. This is a huge jump when compared to the otter algorithms. Another thing I noticed was how the MLFQ had a shorter run time than both SJF and FCFS. The CPU utilization was also different for each process. SJF had the lowest CPU utilization while MLFQ had the highest. I believe that reason for this is because it is a much more complex algorithm when compared to the other two and it has more queues.

In conclusion, this assignment was a great learning experience. It really allowed me to experiment with different code and understand the logic behind these algorithms. Having the

results for the FCFS algorithm was also extremely helpful. This allowed me to gain confidence in my ability to complete the other algorithm. This project was a great experience for me as a programmer and to further my understanding of the algorithms in operating systems.