MLFQ Scheduler in Python

DELA VEGA, Audrey Kirsten R; GALANG, Elijah Israel A.

Overview

A multilevel feedback queue is a scheduling algorithm that schedules processes by using multiple queues with different priority levels and scheduling policies. This project aims to simulate one using Python, where it will output the state of the scheduler—the contents of its queues, the current running process in the CPU, and which processes are in input-output (I/O)—given a list of set properties for the scheduler and multiple process inputs. **Unfortunately, this implementation is only correct and accurate if context-switching is negligible, but it outputs slightly inaccurate wait times per process.**

Implementation

This implementation utilizes python to represent the queues, the individual processes, and to aid in how the main logic for the MLFQ was facilitated. This section details how they work.

Processes

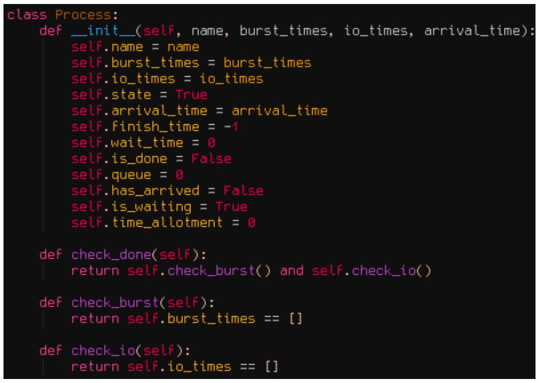
Processes are represented by class Process, illustrated in the figure below, and has the following fields:

1. name - keeps the process name (single capital letter, e.g. "A")
2. burst_times - a list of the times the process will be occupying the CPU in chronological order
3. io_times - a list of the times the process will be in I/O
4. state -
5. arrival_time - a single integer showing the process's arrival time in ms
6. finish_time - a single integer showing how long it took for the process to complete from arrival
7. wait_time - a single integer showing the process's total wait time, which is incremented at each ms whenever the is_waiting field is True
8. is_done - a boolean value set as False initially,; a flag that indicates whether a processes has run to completion or not
9. queue - a single integer showing the process's current queue
10. has_arrived - a boolean value set as False initially, a flag that indicates whether a process has arrived for scheduling. It is set to True when the current time is equal to the process's arrival_time
11. is_waiting - a boolean value set as True initially, a flag that indicates whether a process is waiting, and is set to True when it is neither in the ready queue and neither using the CPU not in I/O

12. time_allotment - a single integer showing the total time allotment a process has in the queue before it is demoted to the next lower queue

classs Process has the following methods:

1. check_done - is True when the process has run to completion
2. check_burst - is True when all burst times are completed
3. check_io - is True when all IO times are completed



```python
class Process:
    def __init__(self, name, burst_times, io_times, arrival_time):
        self.name = name
        self.burst_times = burst_times
        self.io_times = io_times
        self.state = True
        self.arrival_time = arrival_time
        self.finish_time = -1
        self.wait_time = 0
        self.is_done = False
        self.queue = 0
        self.has_arrived = False
        self.is_waiting = True
        self.time_allotment = 0

    def check_done(self):
        return self.check_burst() and self.check_io()

    def check_burst(self):
        return self.burst_times == []

    def check_io(self):
        return self.io_times == []
```

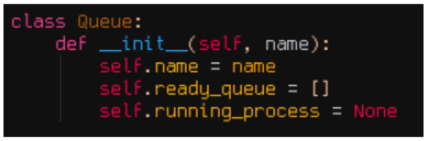
Queues

The queues are represented by class Queue, which is illustrated below. The three different queues are made by extending this class, essentially making subclasses of Queue named Queue1, Queue2, and Queue3. The main class Queue contains fields and methods that are used across all queues, and the logic specific to each queue pertaining to how it handles incoming processes are within the specific queue subclasses.

It is noted that the explanations below will be mentioning the modification of a class named MLFQ during certain conditions. This will be detailed in the next section. For now, know that it is the main driver of the scheduler which contains general information such as the current time (in ms), and several flags that keep track of important information. It is also responsible for calling the active queue, tracking waiting time per process, and the I/O queue among others.

The main queue class is shown below, and has three fields:

1. name - the queue identifier, which is a single integer.
2. ready_queue - a list containing all processes waiting to use the CPU and are not in I/O
3. running_process - contains the current Process using the CPU

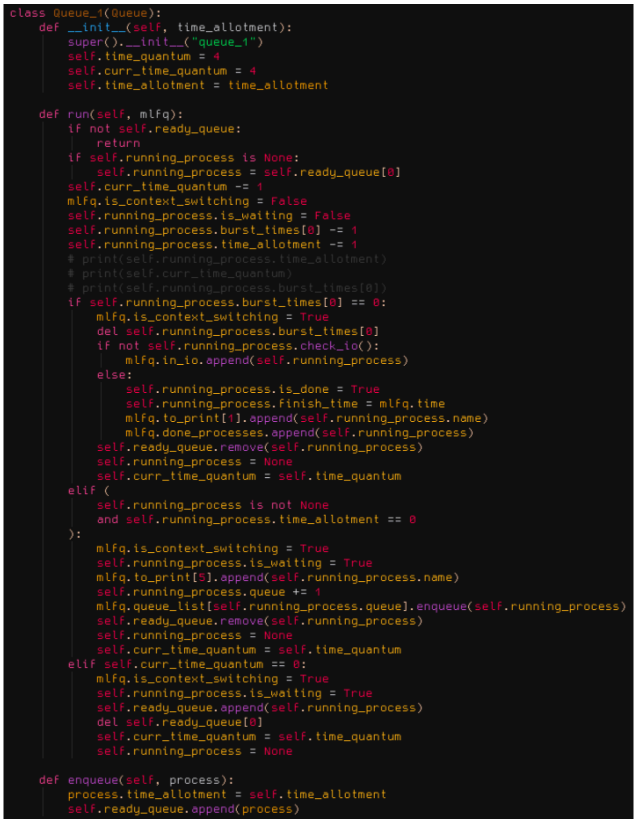


```python
class Queue:
    def __init__(self, name):
        self.name = name
        self.ready_queue = []
        self.running_process = None
```

Queue 1

Below is the extension of the queue class named Queue_1, which is the topmost queue. The topmost queue adopts a round-robin algorithm with a quantum of 4 ms. Thus, in addition to the fields under the general queue class, it has the following:

1. time_quantum - a singular integer hardcoded to 4 ms, indicates the time-slice a process will get before they yield the CPU for the next process in the round robin
2. curr_time_quanum - a singular integer that tracks how much of their quantum a process has consumed, so it is also initially set to 4. It is decremented per ms. and make the process yield their control of the CPU when it reaches 0
3. time_allotment - a singular integer that shows how much time a process has in the queue before it is demoted to Queue 2. This value is initially set to the time allotment inputted by the user for Queue 1, and decremented per ms. The process is demoted when it reaches 0. If the process is returning to the CPU from I/O, this value is reset back to the initial value for time allotment.



```python
class Queue_1(Queue):
    def __init__(self, time_allotment):
        super().__init__("queue_1")
        self.time_quantum = 4
        self.curr_time_quantum = 4
        self.time_allotment = time_allotment

    def run(self, mlfq):
        if not self.ready_queue:
            return
        if self.running_process is None:
            self.running_process = self.ready_queue[0]
        self.curr_time_quantum -= 1
        mlfq.is_context_switching = False
        self.running_process.is_waiting = False
        self.running_process.burst_times[0] -= 1
        self.running_process.time_allotment -= 1
        # print(self.running_process.time_allotment)
        # print(self.curr_time_quantum)
        # print(self.running_process.burst_times[0])
        if self.running_process.burst_times[0] == 0:
            mlfq.is_context_switching = True
            del self.running_process.burst_times[0]
            if not self.running_process.check_io():
                mlfq.in_io.append(self.running_process)
            else:
                self.running_process.is_done = True
                self.running_process.finish_time = mlfq.time
                mlfq.to_print[1].append(self.running_process.name)
                mlfq.done_processes.append(self.running_process)
            self.ready_queue.remove(self.running_process)
            self.running_process = None
            self.curr_time_quantum = self.time_quantum
        elif (
            self.running_process is not None
            and self.running_process.time_allotment == 0
        ):
            mlfq.is_context_switching = True
            self.running_process.is_waiting = True
            mlfq.to_print[5].append(self.running_process.name)
            self.running_process.queue += 1
            mlfq.queue_list[self.running_process.queue].enqueue(self.running_process)
            self.ready_queue.remove(self.running_process)
            self.running_process = None
            self.curr_time_quantum = self.time_quantum
        elif self.curr_time_quantum == 0:
            mlfq.is_context_switching = True
            self.running_process.is_waiting = True
            self.ready_queue.append(self.running_process)
            del self.ready_queue[0]
            self.curr_time_quantum = self.time_quantum
            self.running_process = None

    def enqueue(self, process):
        process.time_allotment = self.time_allotment
        self.ready_queue.append(process)
```

The run method details the logic that contains process state handling. It is first checked if there are processes in the ready queue—if none, it simply returns. Otherwise, it gets the current process by reading the ready queue. Upon reading the first process that arrives (or the one with highest priority if multiple processes arrive at the same time, which in this case, is the one that is the most inferior lexicographically), it changes the following fields:

1. (MLFQ) is_context_switching = False
2. (Process) is_waiting = False

When a process completes a burst time before consuming its timeslice (if self.running_process.burst_times[0] == 0), it checks if the process has a pending I/O time, if yes, then the process is put into the I/O queue, and the leftmost element of the process's burst_times list is also popped from the list. Otherwise, the process is considered done and the following are modified:

1. (Process) is_done = True
2. (Process) finish_time = set to current time

The process is then appended to the list of done processes and removed from the ready queue.

1. (Queue) running_process = None
2. (Queue) curr_time_quantum = reset to time quantum (4)

When a process has consumed its time allotment before completion, a context-switch is initiated within the queue. The following fields are changed:

1. (Process) is_waiting = True (because the process will be placed into the ready queue and will yield control of the CPU)
2. (MLFQ) is_context_switching = True
3. (Process) queue += 1 (the process is demoted to the next queue)

The process is then appended to the ready queue of Queue 2 and removed from the ready_queue of Queue 1.

1. (Queue) running_process = None
2. (Queue) curr_time_quantum = reset to time quantum (4)

If the process has consumed its current quantum and has to yield the CPU to the next process in the round robin, the following are modified:

1. (MLFQ) is_context_switching = True
2. (Process) is_waiting = True (since it will move into the ready queue)

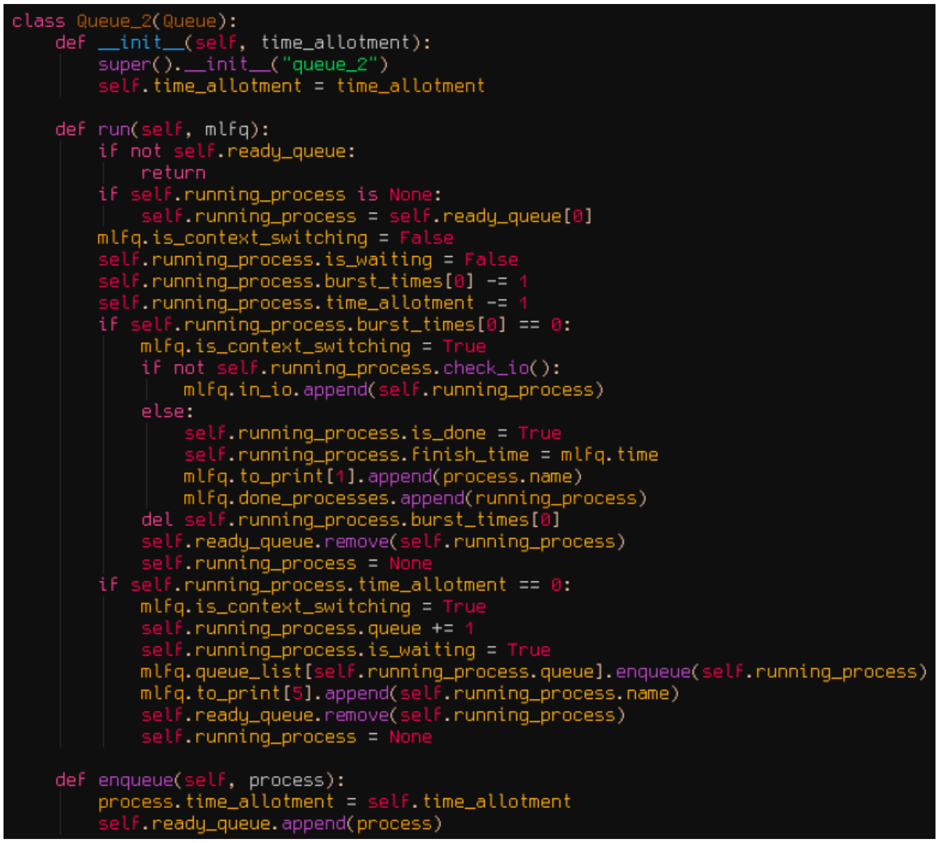
The process is then popped from the ready queue and then appended to the end ready queue.

1. (Queue) curr_time_quantim = reset to time_quantum (4)
2. (Queue) running_process = None

The enqueue method handles appending a process into the ready queue and setting its time allotment.

Queue 2

Below is the extension of the queue class named Queue_2, which uses a first-come-first-serve scheduling logic with time allotment, so it also adds an additional field time_allotment on top of the fields already under the general Queue class.



```python
class Queue_2(Queue):
    def __init__(self, time_allotment):
        super().__init__("queue_2")
        self.time_allotment = time_allotment

    def run(self, mlfq):
        if not self.ready_queue:
            return
        if self.running_process is None:
            self.running_process = self.ready_queue[0]
        mlfq.is_context_switching = False
        self.running_process.is_waiting = False
        self.running_process.burst_times[0] -= 1
        self.running_process.time_allotment -= 1
        if self.running_process.burst_times[0] == 0:
            mlfq.is_context_switching = True
            if not self.running_process.check_io():
                mlfq.in_io.append(self.running_process)
            else:
                self.running_process.is_done = True
                self.running_process.finish_time = mlfq.time
                mlfq.to_print[1].append(process.name)
                mlfq.done_processes.append(running_process)
            del self.running_process.burst_times[0]
            self.ready_queue.remove(self.running_process)
            self.running_process = None
        if self.running_process.time_allotment == 0:
            mlfq.is_context_switching = True
            self.running_process.queue += 1
            self.running_process.is_waiting = True
            mlfq.queue_list[self.running_process.queue].enqueue(self.running_process)
            mlfq.to_print[5].append(self.running_process.name)
            self.ready_queue.remove(self.running_process)
            self.running_process = None

    def enqueue(self, process):
        process.time_allotment = self.time_allotment
        self.ready_queue.append(process)
```

The run method details the logic that contains process state handling. It is first checked if there are processes in the ready queue—if none, it simply returns. Otherwise, it gets the current process by reading the ready queue. Upon giving the CPU to the process with highest priority, the following are changed:

1. (MLFQ) is_context_switching = False
2. (Process) is_waiting = True

When a process finishes a burst time before consuming the quantum, a context-switch is initiated, and the scheduler checks if the process has pending I/O times. If yes, then the process is put into the I/O queue; otherwise, the process is considered done and the following are changed:

1. (Process) is_done = True
2. (Process) finish_time = current time

The process is then appended to the list of done processes and removed from the ready queue of the current queue.

1. (Queue) running_process = None

When the process exceeds the set time allotment for the queue, a context-switch is initiated:

1. (MLFQ) is_context_switching = True
2. (Process) queue += 1 (the process is demoted to the next queue)
3. (Process) is_waiting = True (the process is neither using the CPU or in I/O and is in the ready queue)

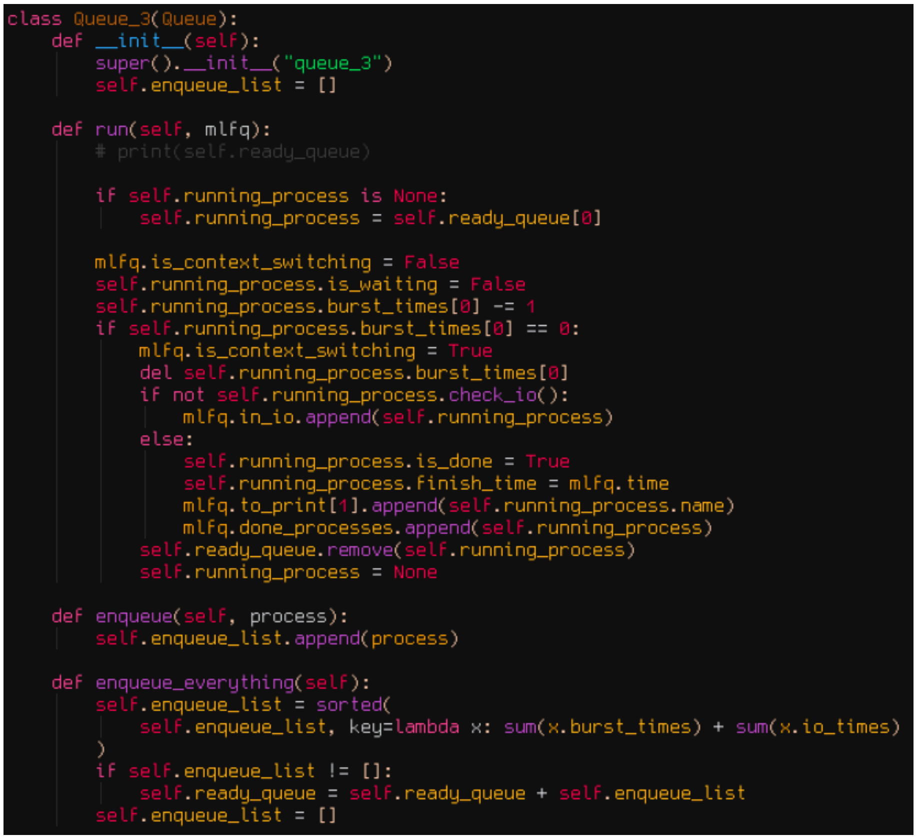
The process is then appended to the ready queue of Queue 3 and removed from the ready queue for Queue 2.

1. (Queue) running_process = None

The enqueue method for Queue 2 is the same as Queue 1's.

Queue 3

Below is the extension of the queue class named Queue_3, which uses shortest-job-first scheduling logic. In addition to the fields under Queue, it adds another field enqueue_list, initialized as an empty list [].



```
class Queue_3(Queue):
    def __init__(self):
        super().__init__("queue_3")
        self.enqueue_list = []

    def run(self, mlfq):
        # print(self.ready_queue)

        if self.running_process is None:
            self.running_process = self.ready_queue[0]

        mlfq.is_context_switching = False
        self.running_process.is_waiting = False
        self.running_process.burst_times[0] -= 1
        if self.running_process.burst_times[0] == 0:
            mlfq.is_context_switching = True
            del self.running_process.burst_times[0]
            if not self.running_process.check_io():
                mlfq.in_io.append(self.running_process)
            else:
                self.running_process.is_done = True
                self.running_process.finish_time = mlfq.time
                mlfq.to_print[1].append(self.running_process.name)
                mlfq.done_processes.append(self.running_process)
            self.ready_queue.remove(self.running_process)
            self.running_process = None

    def enqueue(self, process):
        self.enqueue_list.append(process)

    def enqueue_everything(self):
        self.enqueue_list = sorted(
            self.enqueue_list, key=lambda x: sum(x.burst_times) + sum(x.io_times)
        )
        if self.enqueue_list != []:
            self.ready_queue = self.ready_queue + self.enqueue_list
        self.enqueue_list = []
```

The run method details the logic that contains process state handling. It is first checked if there are processes in the ready queue—if none, it simply returns. Otherwise, it gets the current process by reading the ready queue. Upon giving the CPU to the process with highest priority, the following are changed:

1. (MLFQ) is_context_switching = False
2. (Process) is_waiting) = False

When a process finishes a burst time before consuming the quantum, a context-switch is initiated, and the scheduler checks if the process has pending I/O times. If yes, then the process is put into the I/O queue; otherwise, the process is considered done and the following are changed:

3. (Process) is_done = True
4. (Process) finish_time = current time

The process is then appended to the list of done processes and removed from the ready queue of the current queue.

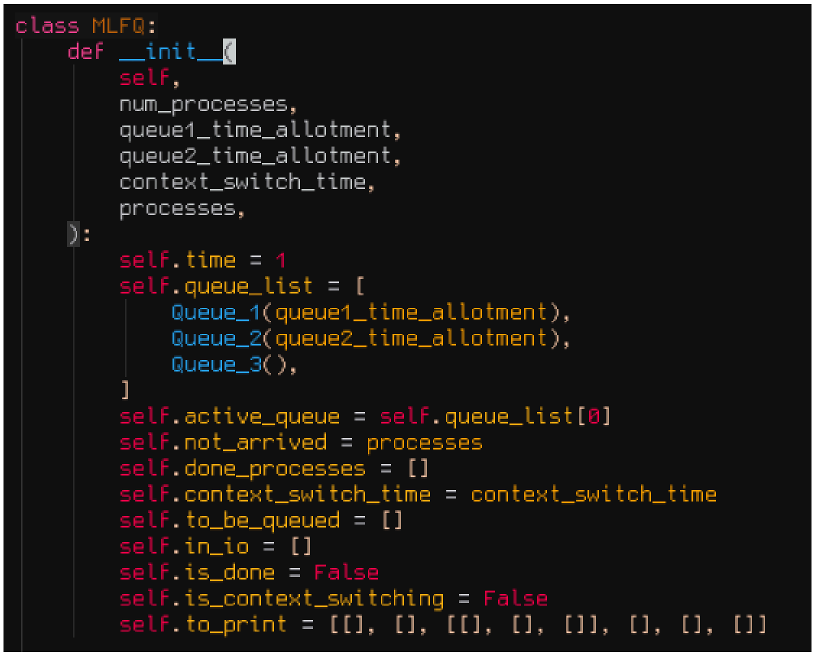
2. (Queue) running_process = None

The enqueue method for Queue 3 is different from the enqueue method for the previous queues since it appends to the field enqueue_list instead of the queue's ready queue. There is another method called enqueue_everything, which is called by the driver class. This ensures that the queue will always schedule the shortest job first, and assesses which process is shortest at the time when Queue 3 is first set as the active queue. This way, the queue is non-preemptive. The contents of enqueue_list are sorted using the processes' total burst and io times, and then this sorted list is appended to Queue 3's ready queue, ensuring that processes are arranged from shortest to longest. The enqueue_list is then reset to an empty list.

MLFQ

The MLFQ class is the main driver class which stores the current active queue, the I/O queue, and the output printing. It is initialized with the scheduler details inputted by the user and contains the field time which increments until all processes are completed. All of its fields are listed below:

1. time = integer initialized to 1
2. queue_list = list containing the three queues
3. active_queue = current Queue that is using the CPU, which is the head of the queue_list
4. not_arrived = initially contains all processes, and are popped when the time is equal to its set arrival time
5. done_processes = list Process containing all processes that are complete
6. context_switch_time = integer set to whatever the set time for context switching was
7. in_io = list of processes in I/O, the I/O queue
8. is_context_switching = initialized as False, is True when a queue initiates a context switch
9. to_print = stores values for printing the output

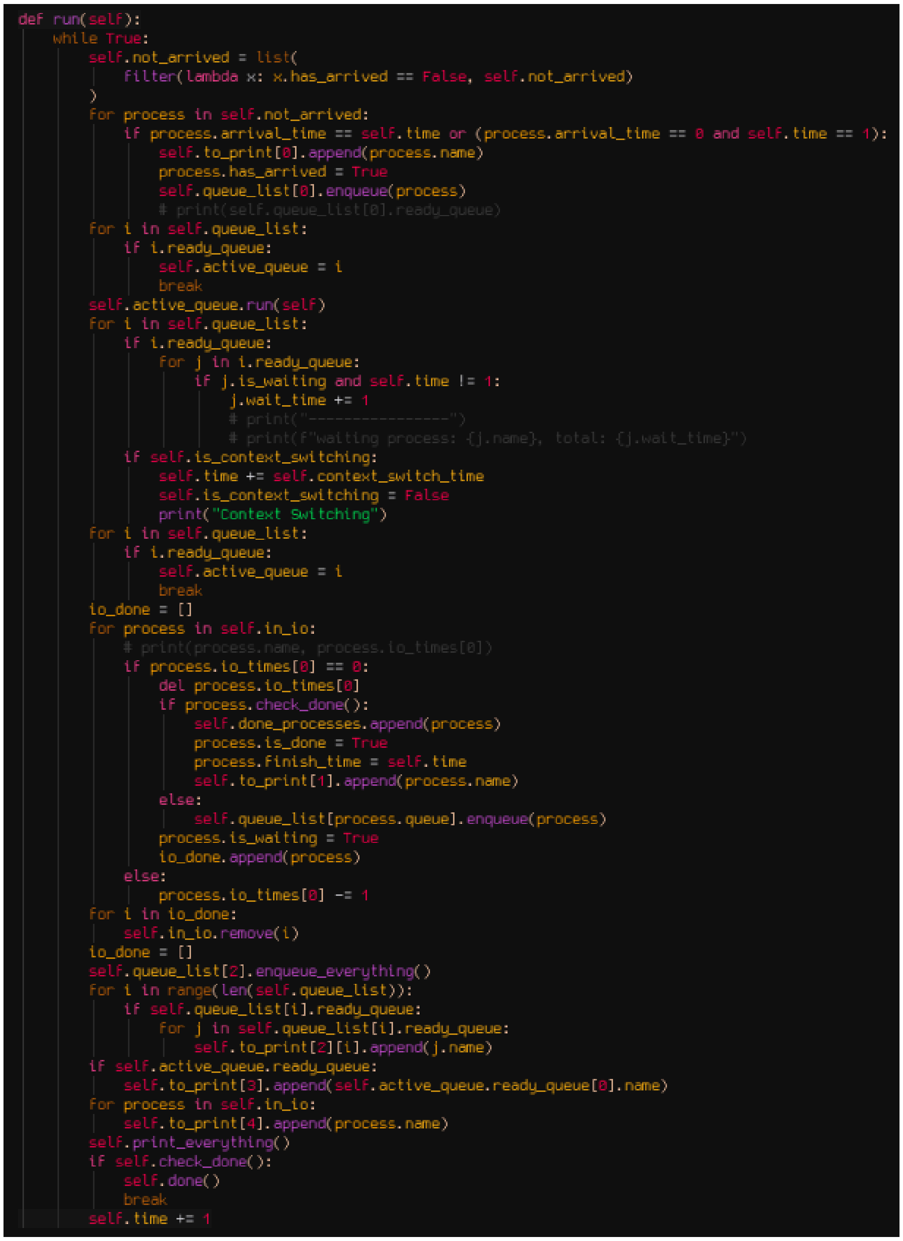


```python
class MLFQ:
    def __init__(
        self,
        num_processes,
        queue1_time_allotment,
        queue2_time_allotment,
        context_switch_time,
        processes,
    ):
        self.time = 1
        self.queue_list = [
            Queue_1(queue1_time_allotment),
            Queue_2(queue2_time_allotment),
            Queue_3(),
        ]
        self.active_queue = self.queue_list[0]
        self.not_arrived = processes
        self.done_processes = []
        self.context_switch_time = context_switch_time
        self.to_be_queued = []
        self.in_io = []
        self.is_done = False
        self.is_context_switching = False
        self.to_print = [[], [], [[], [], []], [], [], []]
```

The run method contains the driving code for the scheduler. For every ms (whenever MLFQ.time is incremented) this loop is executed:

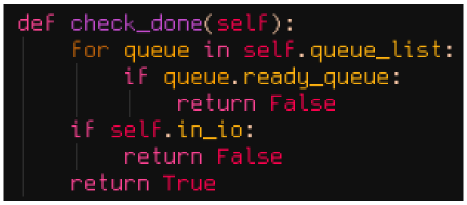
1. The not_arrived list of processes is filtered down to processes with has_arrived set to False
2. Each process in not_arrived is assessed if the current time is equal to their arrival_time; if yes, they are appended to the arriving list and are queued into the topmost queue.
3. The active queue is set and run.
4. Each process in the ready queues of all the queues are checked if they are neither using the CPU nor in the I/O queue—if yes, their wait_time is incremented by 1.
5. Checks if the is_context_switching flag is True and increases the time field by the amount set by context_switch_time, and then switches the flag back to False.
6. Initializes a list called io_done.
7. Checks each process in in_io and checks if the head of their io_times has reached 0.
   If yes, it pops the head of their io_times list. It then checks if their io_times list is now empty—if yes, the process is considered done and their finish_time is set to current time. Otherwise, the process is put back into the ready queue of the current active queue. The is_waiting field of the process is set to True and the process is appended to io_done.
   Otherwise, the head of their io_times is decremented.
8. Processes in io_done are removed from in_io as they are no longer in I/O. io_done is reinitialized to an empty list.
9. The enqueue_everything() method under queue 3 is called (sorts processes).
10. It is checked if all processes have run to completion and if the simulation can proceed to printing the output (via check_done() method).
11. time is incremented.

12. Collects all relevant data for the print output (print_everything()).



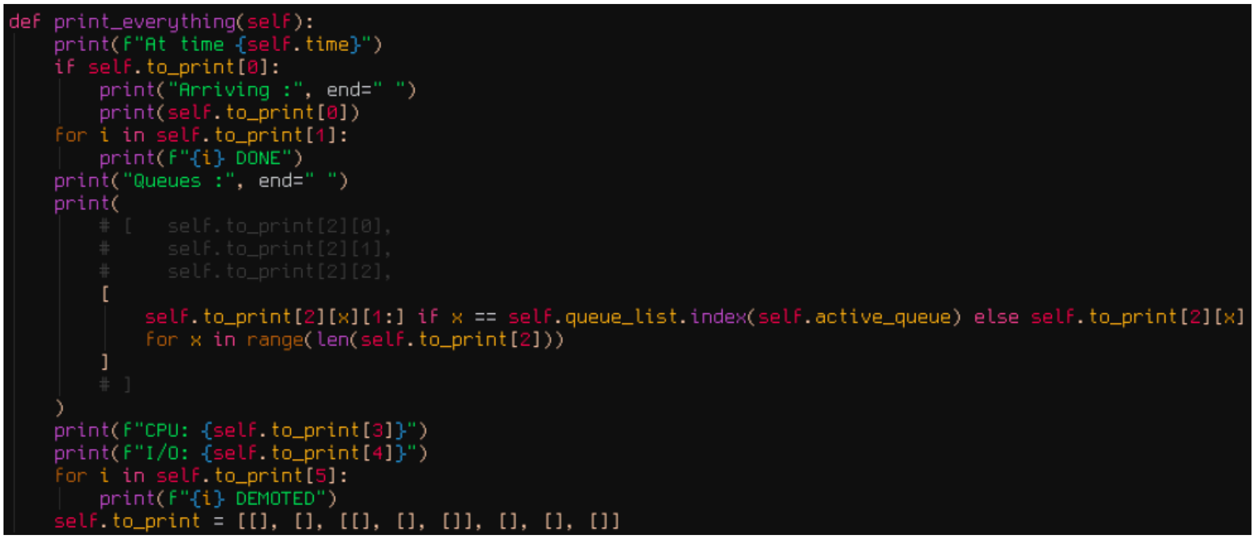
```python
def run(self):
    while True:
        self.not_arrived = list(
            filter(lambda x: x.has_arrived == False, self.not_arrived)
        )
        for process in self.not_arrived:
            if process.arrival_time == self.time or (process.arrival_time == 0 and self.time == 1):
                self.to_print[0].append(process.name)
                process.has_arrived = True
                self.queue_list[0].enqueue(process)
                # print(self.queue_list[0].ready_queue)
        for i in self.queue_list:
            if i.ready_queue:
                self.active_queue = i
                break
        self.active_queue.run(self)
        for i in self.queue_list:
            if i.ready_queue:
                for j in i.ready_queue:
                    if j.is_waiting and self.time != 1:
                        j.wait_time += 1
                        # print("----------------")
                        # print(f"waiting process: {j.name}, total: {j.wait_time}")
            if self.is_context_switching:
                self.time += self.context_switch_time
                self.is_context_switching = False
                print("Context Switching")
        for i in self.queue_list:
            if i.ready_queue:
                self.active_queue = i
                break
        io_done = []
        for process in self.in_io:
            # print(process.name, process.io_times[0])
            if process.io_times[0] == 0:
                del process.io_times[0]
                if process.check_done():
                    self.done_processes.append(process)
                    process.is_done = True
                    process.finish_time = self.time
                    self.to_print[1].append(process.name)
                else:
                    self.queue_list[process.queue].enqueue(process)
                process.is_waiting = True
                io_done.append(process)
            else:
                process.io_times[0] -= 1
        for i in io_done:
            self.in_io.remove(i)
        io_done = []
        self.queue_list[2].enqueue_everything()
        for i in range(len(self.queue_list)):
            if self.queue_list[i].ready_queue:
                for j in self.queue_list[i].ready_queue:
                    self.to_print[2][i].append(j.name)
        if self.active_queue.ready_queue:
            self.to_print[3].append(self.active_queue.ready_queue[0].name)
        for process in self.in_io:
            self.to_print[4].append(process.name)
        self.print_everything()
        if self.check_done():
            self.done()
            break
        self.time += 1
```

The check_done method checks if all queues have empty ready queues and if there are no processes inside in_io. If both are true, then the simulation can end since all processes have run to completion. This method is illustrated below.



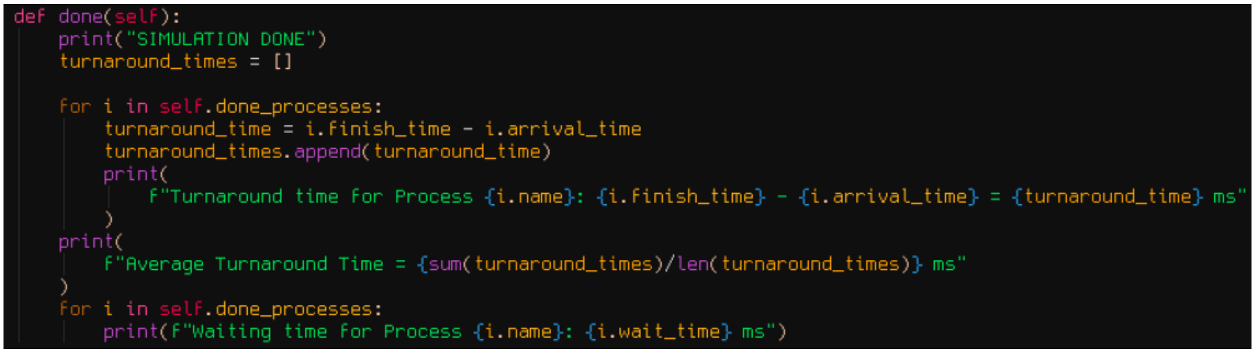
```python
def check_done(self):
    for queue in self.queue_list:
        if queue.ready_queue:
            return False
    if self.in_io:
        return False
    return True
```

The print_everything method organizes all data that is required for the output, illustrated below.



```
def print_everything(self):
    print(f"At time {self.time}")
    if self.to_print[0]:
        print("Arriving :", end=" ")
        print(self.to_print[0])
    for i in self.to_print[1]:
        print(f"{i} DONE")
    print("Queues :", end=" ")
    print(
        # [   self.to_print[2][0],
        #     self.to_print[2][1],
        #     self.to_print[2][2],
        [
            self.to_print[2][x][1:] if x == self.queue_list.index(self.active_queue) else self.to_print[2][x]
            for x in range(len(self.to_print[2]))
        ]
        # ]
    )
    print(f"CPU: {self.to_print[3]}")
    print(f"I/O: {self.to_print[4]}")
    for i in self.to_print[5]:
        print(f"{i} DEMOTED")
    self.to_print = [[], [], [[], [], []], [], [], []]
```

The done method prints all the information obtained at the end of the simulation.



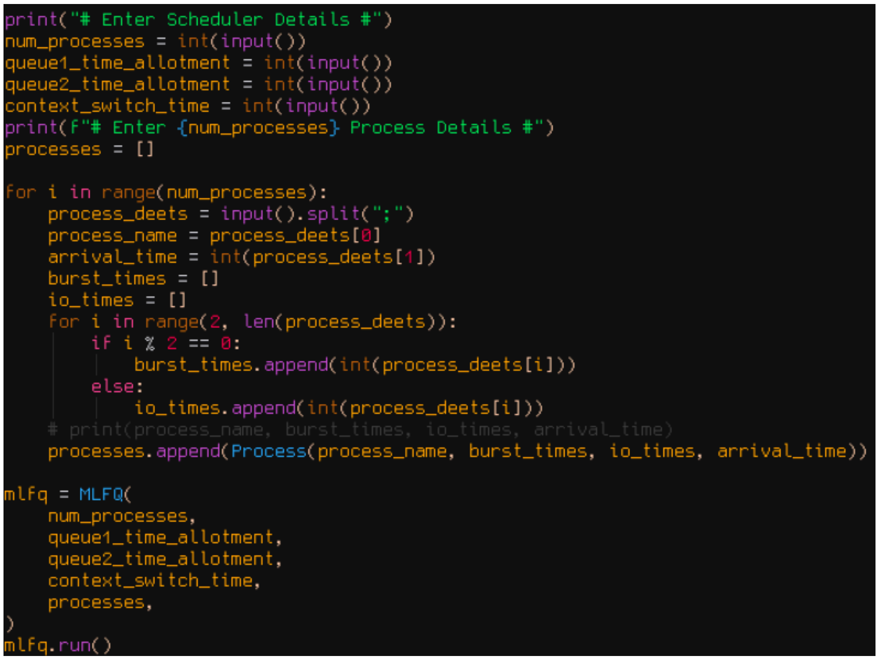
```
def done(self):
    print("SIMULATION DONE")
    turnaround_times = []

    for i in self.done_processes:
        turnaround_time = i.finish_time - i.arrival_time
        turnaround_times.append(turnaround_time)
        print(
            f"Turnaround time for Process {i.name}: {i.finish_time} - {i.arrival_time} = {turnaround_time} ms"
        )
    print(
        f"Average Turnaround Time = {sum(turnaround_times)/len(turnaround_times)} ms"
    )
    for i in self.done_processes:
        print(f"Waiting time for Process {i.name}: {i.wait_time} ms")
```

Input parsing

The figure below shows how the inputs are collected. They are organized into the following variables:

1. num_processes - stores number of processes (single int)
2. queue1_time_allotment - stores time allotment for Queue 1 (single int)
3. queue2_time_allotment - stores time allotment for Queue 2 (single int)
4. context_switch_time - stores time it takes for a context switch (single int)
5. processes - list of all processes and their details (list Process)

An MLFQ class is then initialized with the user input, and called via mlfq.run().



```python
print("# Enter Scheduler Details #")
num_processes = int(input())
queue1_time_allotment = int(input())
queue2_time_allotment = int(input())
context_switch_time = int(input())
print(f"# Enter {num_processes} Process Details #")
processes = []

for i in range(num_processes):
    process_deets = input().split(";")
    process_name = process_deets[0]
    arrival_time = int(process_deets[1])
    burst_times = []
    io_times = []
    for i in range(2, len(process_deets)):
        if i % 2 == 0:
            burst_times.append(int(process_deets[i]))
        else:
            io_times.append(int(process_deets[i]))
    # print(process_name, burst_times, io_times, arrival_time)
    processes.append(Process(process_name, burst_times, io_times, arrival_time))

mlfq = MLFQ(
    num_processes,
    queue1_time_allotment,
    queue2_time_allotment,
    context_switch_time,
    processes,
)
mlfq.run()
```

Test Cases

The following test cases were used to test the implementation.

Given

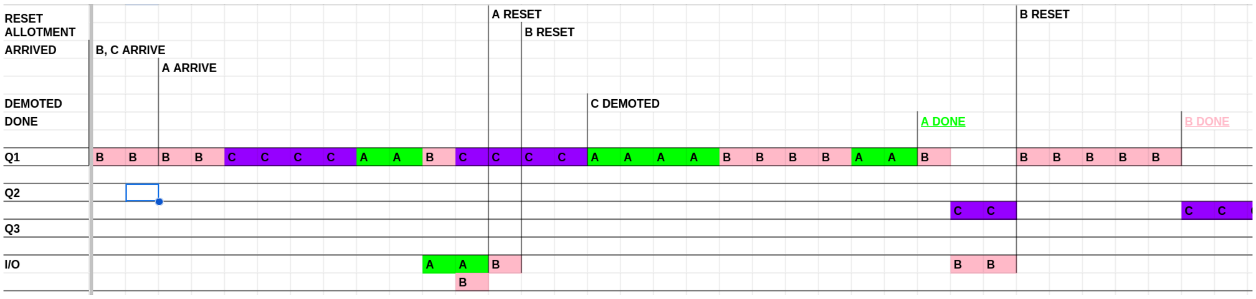
The scheduler details are as follows:

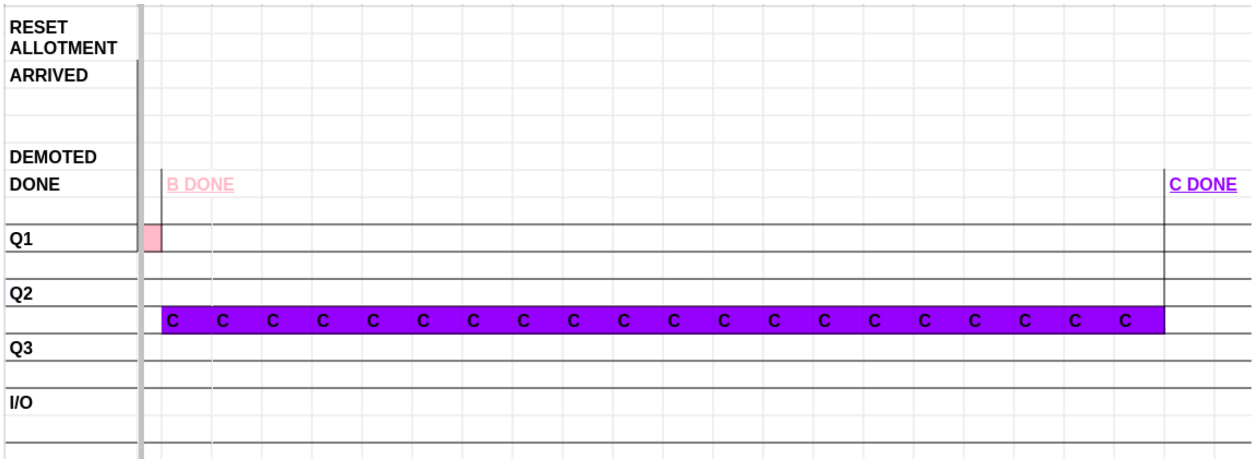
- Three (3) processes
- Queue 1 time allotment: 8 ms
- Queue 2 time allotment: 8 ms
- Context-switching: 0 ms

The following table shows the process information:

Process	**Arrival**	**Burst Time**
Process A	t = 2	2 ms CPU; 2 ms I/O; 6 ms CPU
Process B	t = 0	5 ms CPU; 2 ms I/O; 5 ms CPU; 2 ms I/O; 5 ms CPU
Process C	t = 0	30 ms CPU

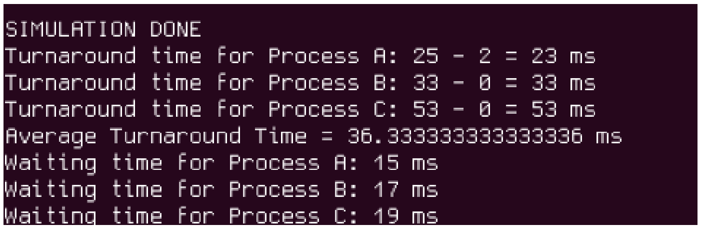
The expected solution to this is illustrated below. Note that each column is 1ms.





Process	Turn-around time	Waiting time
A	23 ms	13 ms
B	33 ms	14 ms
C	53 ms	23 ms
Average	36.33333333334 ms	16.66666666667 ms

The code outputs the following:



```
SIMULATION DONE
Turnaround time for Process A: 25 - 2 = 23 ms
Turnaround time for Process B: 33 - 0 = 33 ms
Turnaround time for Process C: 53 - 0 = 53 ms
Average Turnaround Time = 36.333333333333336 ms
Waiting time for Process A: 15 ms
Waiting time for Process B: 17 ms
Waiting time for Process C: 19 ms
```

Sample #1 (set1.txt)

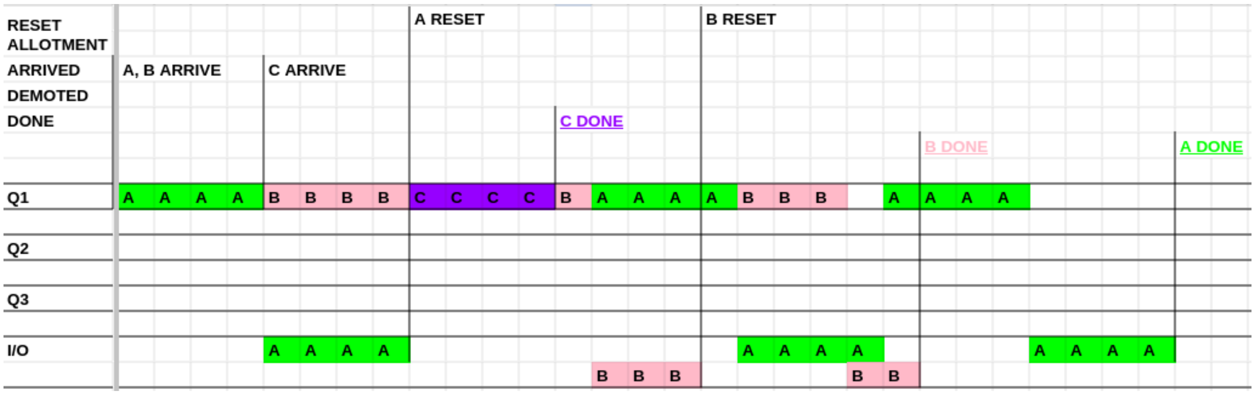
The scheduler details are as follows:

- Three (3) processes
- Queue 1 time allotment: 8 ms
- Queue 2 time allotment: 8 ms
- Context-switching: 0 ms

The following table shows the process information:

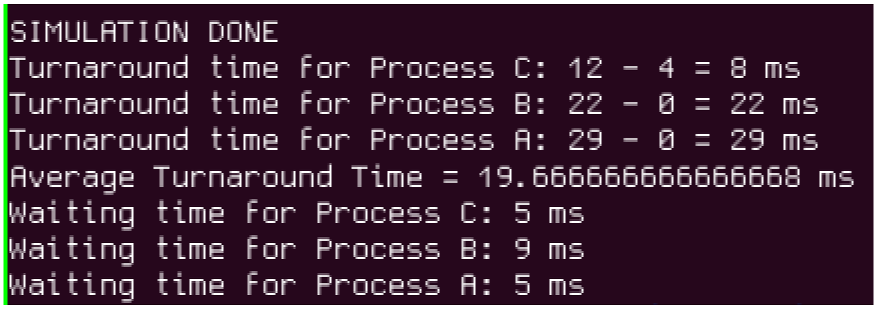
Process	Arrival	Burst Time
Process A	t = 0	4 ms CPU; 4 ms I/O; 4 ms CPU; 4 ms I/O; 4 ms CPU; 4 ms I/O;
Process B	t = 0	5 ms CPU; 3 ms I/O; 3 ms CPU; 2 ms I/O
Process C	t = 4	4 ms CPU

The expected solution to this is illustrated below. Note that each column is 1ms.



Process	Turn-around time	Waiting time
A	29 ms	5 ms
B	22 ms	9 ms
C	8 ms	4 ms
Average	19.6666666667 ms	6 ms

The code outputs the following:



```
SIMULATION DONE
Turnaround time for Process C: 12 - 4 = 8 ms
Turnaround time for Process B: 22 - 0 = 22 ms
Turnaround time for Process A: 29 - 0 = 29 ms
Average Turnaround Time = 19.666666666666668 ms
Waiting time for Process C: 5 ms
Waiting time for Process B: 9 ms
Waiting time for Process A: 5 ms
```

Sample #2 (set2.txt)

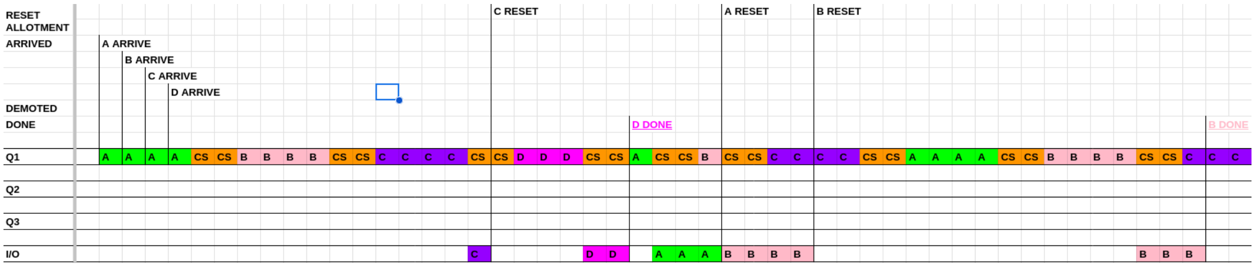
The scheduler details are as follows:

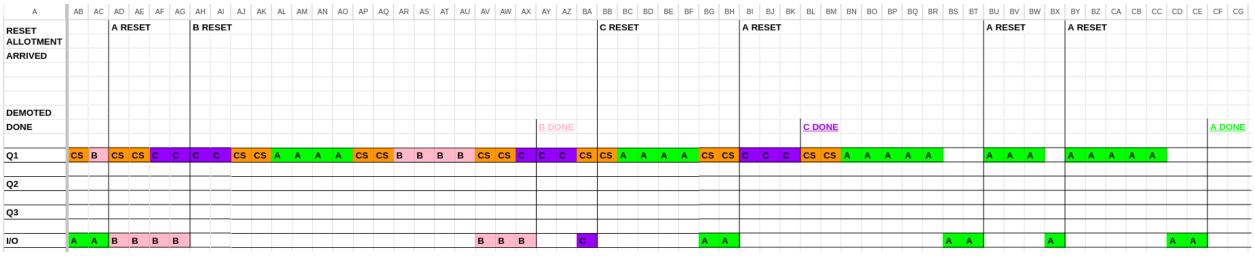
- Four (4) processes
- Queue 1 time allotment: 10 ms
- Queue 2 time allotment: 8 ms
- Context-switching: 2 ms

The following table shows the process information:

Process	**Arrival**	**Burst Time**
Process A	t = 1	5 ms CPU; 3 ms I/O; 8 ms CPU; 2 ms I/O; 3 ms CPU; 2 ms I/O; 3 ms CPU; 1 ms I/O; 5 ms CPU; 2 ms I/O
Process B	t = 2	5 ms CPU; 4 ms I/O; 4 ms CPU; 3 ms I/O
Process C	t = 3	4 ms CPU; 1 ms I/O; 7 ms CPU; 1 ms I/O; 3 ms CPU
Process D	t = 4	3 ms CPU; 2 ms I/O

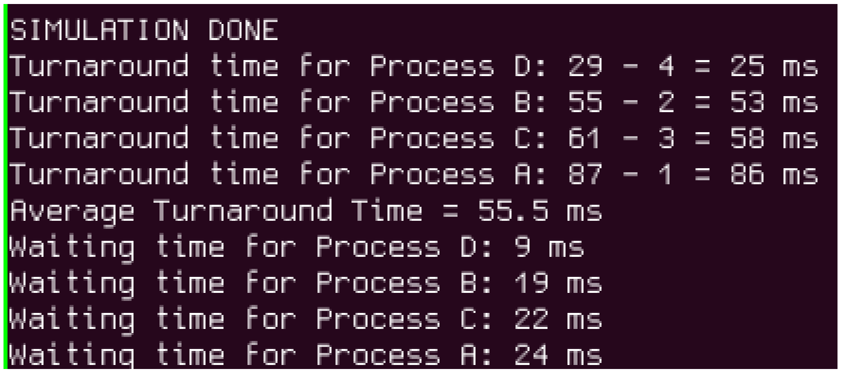
The expected solution to this is illustrated below. Note that each column is 1ms, and that context switches are denoted by cells shaded orange.


RESET
ALLOTMENT
ARRIVED
DEMOTED
DONE
Q1
Q2
Q3
I/O
C RESET
A RESET
B RESET
A ARRIVE
B ARRIVE
C ARRIVE
D ARRIVE
D DONE
B DONE
A A A A CS CS B B B B CS CS C C C C CS CS D D D CS CS A CS CS B CS CS C C C C CS CS A A A A CS CS B B B B CS CS C C C
C D D A A A B B B B B B B



Process	Turn-around time	Waiting time
A	81 ms	45 ms
B	47 ms	31 ms
C	59 ms	43 ms
D	20 ms	15 ms
Average	51.75 ms	33.5 ms

The code outputs the following:



```
SIMULATION DONE
Turnaround time for Process D: 29 - 4 = 25 ms
Turnaround time for Process B: 55 - 2 = 53 ms
Turnaround time for Process C: 61 - 3 = 58 ms
Turnaround time for Process A: 87 - 1 = 86 ms
Average Turnaround Time = 55.5 ms
Waiting time for Process D: 9 ms
Waiting time for Process B: 19 ms
Waiting time for Process C: 22 ms
Waiting time for Process A: 24 ms
```

Comments

The most challenging part was implementing the different type classes and organizing all the important information needed for the scheduler to work. Debugging how the waiting_time was tallied and how the context switches affected the completion times were also difficult to track.