Assignment 8-Multilevel Queue Scheduling

22000021

March 8, 2024

1 Introduction

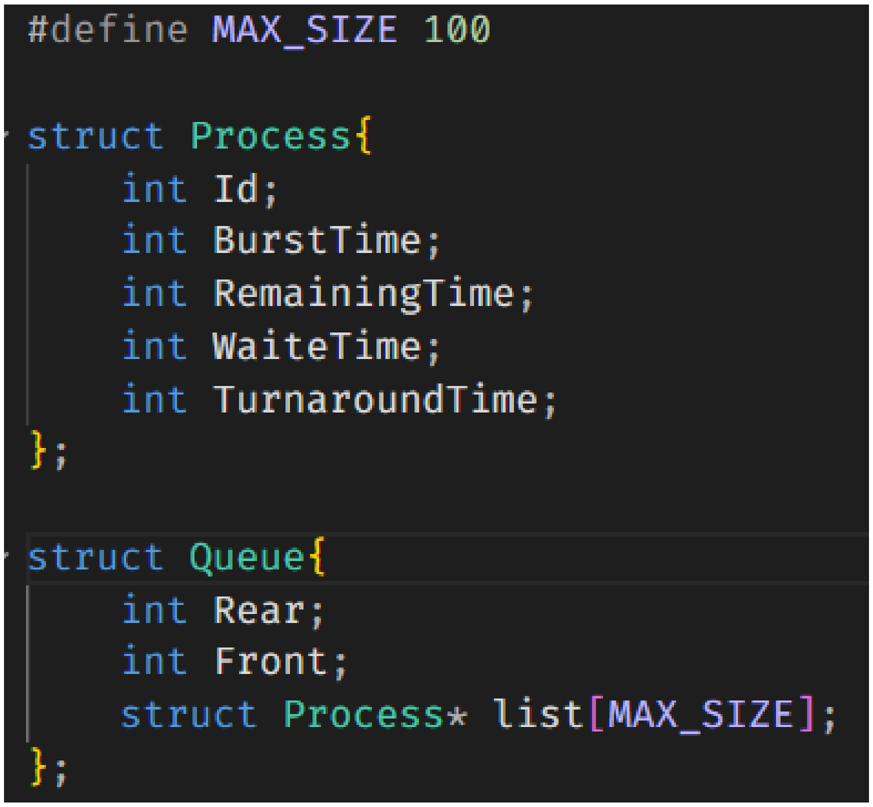
This program aims to simulate a multilevel queue scheduling algorithm, a crucial aspect of operating system design, where processes are organized into priority-based queues. With four distinct queues assigned priorities from q0 to q3, the algorithm employs different scheduling strategies tailored to each queue's priority level. Specifically, Round Robin (RR) scheduling is used for q0, while Shortest Job First (SJF) scheduling is employed for q1 and q2, and First-In-First-Out (FIFO) scheduling for q3. Each queue is allocated a time quantum of 20 seconds, after which the CPU switches between queues. The program prompts users to input the number of processes and their respective priorities for each queue, facilitating dynamic simulation scenarios. Through this simulation, we aim to analyze the efficiency and performance of various scheduling algorithms under differing workload conditions, providing valuable insights for optimizing system scheduling strategies.

2 Methodology

In this section, we will examine the implementation details of the algorithm.

2.1 Process Implementation

The multilevel queue scheduling algorithm is implemented using **C++**'s class-based approach. The 'Process' structure represents a process with properties such as process ID, burst time, remaining time, wait time Turnaround time. The 'Queue' structure represents a queue with properties such as front, rear, and an array to hold the processes For the implementation purpose, I have used the maximum number of processes in the queue as 100

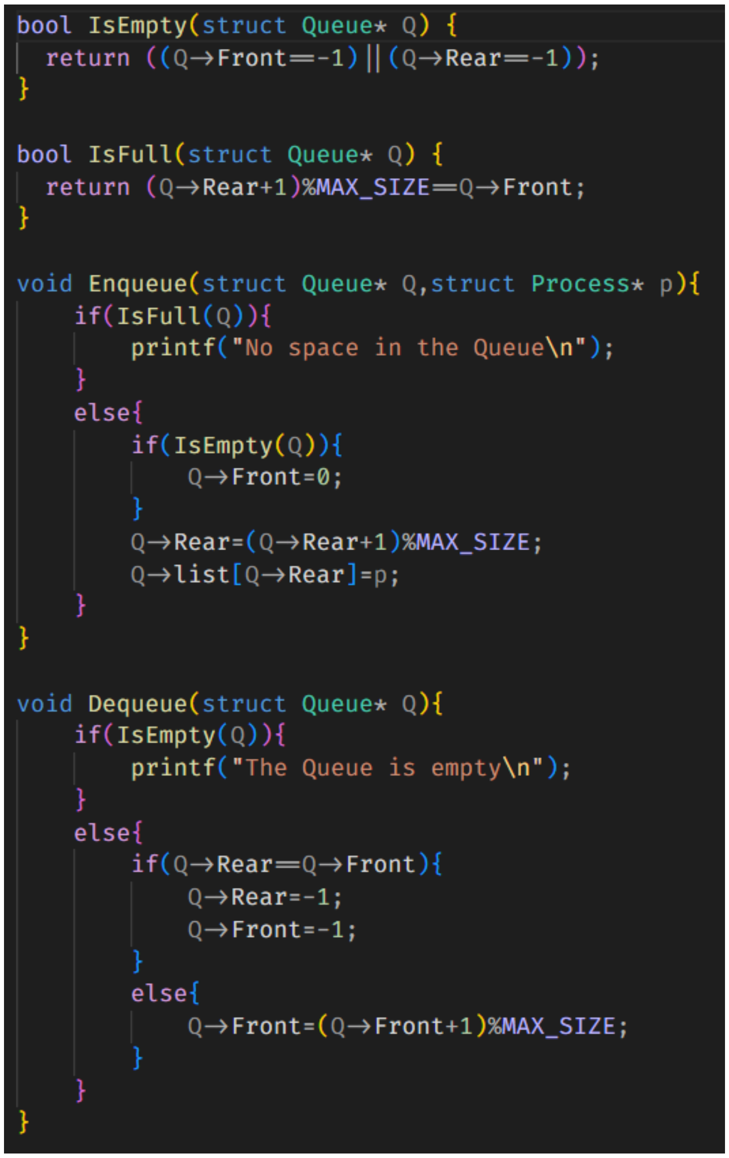


```
#define MAX_SIZE 100

struct Process{
    int Id;
    int BurstTime;
    int RemainingTime;
    int WaiteTime;
    int TurnaroundTime;
};

struct Queue{
    int Rear;
    int Front;
    struct Process* list[MAX_SIZE];
};
```

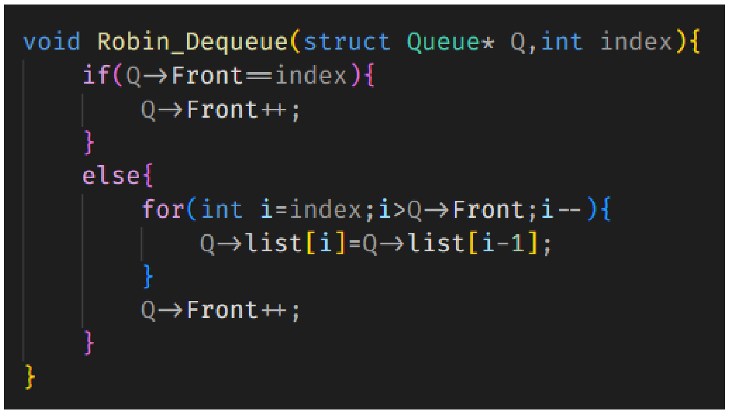
Figure 1: Structures

2.2 Queue Operations

The queue operations such as enqueue, dequeue, and checking if the queue is empty or full are implemented as functions. The scheduling algorithms for each queue are implemented as separate functions. The the functions IsFull and IsEmpty returns true or false and the Enqueue and Dequeue operation do the inserting and deletion in the queue



```c
bool IsEmpty(struct Queue* Q) {
  return ((Q->Front==-1)||(Q->Rear==-1));
}

bool IsFull(struct Queue* Q) {
  return (Q->Rear+1)%MAX_SIZE==Q->Front;
}

void Enqueue(struct Queue* Q,struct Process* p){
    if(IsFull(Q)){
        printf("No space in the Queue\n");
    }
    else{
        if(IsEmpty(Q)){
            Q->Front=0;
        }
        Q->Rear=(Q->Rear+1)%MAX_SIZE;
        Q->list[Q->Rear]=p;
    }
}

void Dequeue(struct Queue* Q){
    if(IsEmpty(Q)){
        printf("The Queue is empty\n");
    }
    else{
        if(Q->Rear==Q->Front){
            Q->Rear=-1;
            Q->Front=-1;
        }
        else{
            Q->Front=(Q->Front+1)%MAX_SIZE;
        }
    }
}
```



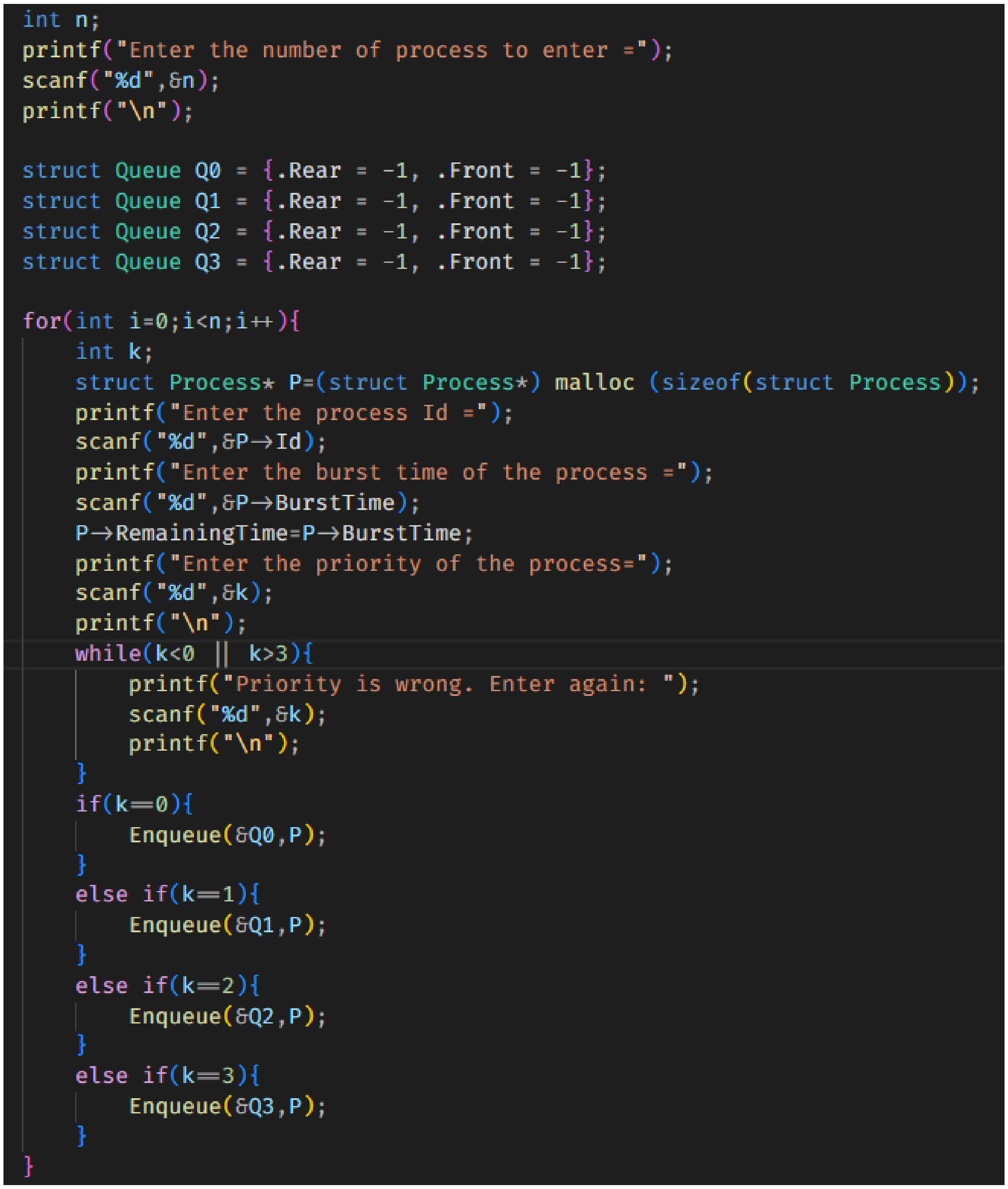
```c
void Robin_Dequeue(struct Queue* Q,int index){
    if(Q->Front==index){
        Q->Front++;
    }
    else{
        for(int i=index;i>Q->Front;i--){
            Q->list[i]=Q->list[i-1];
        }
        Q->Front++;
    }
}
```

Figure 2: Queue

Enqueue and Dequeue are the same operations following a circular queue.delete is a variation of dequeue, where the dequeued process will be enqueued to the same queue.

2.3 Process initialization

For the code to work without any interrupts we ask for the user to enter the count of the process that will be processed In c++ we cant do that without user inputs The code for initialization of the process after the detail for the process has been input we store the process temporarily to a user define variable and than that process is added to the corresponding queue



```
int n;
printf("Enter the number of process to enter =");
scanf("%d",&n);
printf("\n");

struct Queue Q0 = {.Rear = -1, .Front = -1};
struct Queue Q1 = {.Rear = -1, .Front = -1};
struct Queue Q2 = {.Rear = -1, .Front = -1};
struct Queue Q3 = {.Rear = -1, .Front = -1};

for(int i=0;i<n;i++){
    int k;
    struct Process* P=(struct Process*) malloc (sizeof(struct Process));
    printf("Enter the process Id =");
    scanf("%d",&P->Id);
    printf("Enter the burst time of the process =");
    scanf("%d",&P->BurstTime);
    P->RemainingTime=P->BurstTime;
    printf("Enter the priority of the process=");
    scanf("%d",&k);
    printf("\n");
    while(k<0 || k>3){
        printf("Priority is wrong. Enter again: ");
        scanf("%d",&k);
        printf("\n");
    }
    if(k==0){
        Enqueue(&Q0,P);
    }
    else if(k==1){
        Enqueue(&Q1,P);
    }
    else if(k==2){
        Enqueue(&Q2,P);
    }
    else if(k==3){
        Enqueue(&Q3,P);
    }
}
```

Figure 3: Queue

This part of the code initializes four separate queues (Q0, Q1, Q2, and Q3) representing different priority levels in a multilevel queue scheduling algorithm. It first prompts the user to input the number of processes to enter. Then, it iterates n times to input details for each process, such as process ID, burst time, and priority. During each iteration, memory is dynamically allocated for a new Process. The user is prompted to enter the process details, and the entered information is assigned to the respective members of the Process. The priority of the process is checked to ensure it falls within the valid range of 0 to 3. If the priority is valid, the process is enqueued into the corresponding queue (Q0, Q1, Q2, or Q3) using the Enqueue() function. Overall, this section of the code facilitates the dynamic creation and enqueuing of processes into their respective priority-based queues based on user input.

3 Scheduling Algorithm

3.1 RR (Round Robin) Scheduling Algorithm

- The function determines the quantum slice by dividing the total number of processes in the queue by the time quantum of the multilevel queue. This ensures fair distribution of time among processes.
- if the processes in the queue is greater than 20 seconds we cant use decimal time to execute a process there is a exception when that happens
- It enters a loop that iterates until the quantum slice is exhausted. This loop handles cases where the quantum slice is smaller than the Robin slice or vice versa.
- If a process's remaining time is less than or equal to the Robin slice, the process completes its execution. It is removed from the queue, and the global time is updated. A message indicating the process ID and exit time is printed
- If a process's remaining time exceeds the Robin slice, the Robin slice is subtracted from its remaining time, and the global time is updated accordingly. The process remains in the queue for further execution
- This process continues until the quantum slice is exhausted or the queue becomes empty.

3.2 SJF (Shortest Job First) Scheduling Algorithm

- According to the arrival time and the shortest job in the queue at the moment gets executed without being preemption.
- Selecting the Shortest Job: The scheduler selects the process with the shortest burst time from the pool of available processes.
- It prioritizes processes based on their burst time, where the process with the shortest burst time is given preference for execution.
- Executing the Process: Once selected, the chosen process is dispatched for execution on the CPU. The CPU executes the process until it completes its entire burst time.

3.3 FIFO (First In First Out) Scheduling Algorithm

- The process are executed in the order they are enqueued into the Queue just how the Queue algorithm works

4 Pros and Cons of Scheduling Algorithms

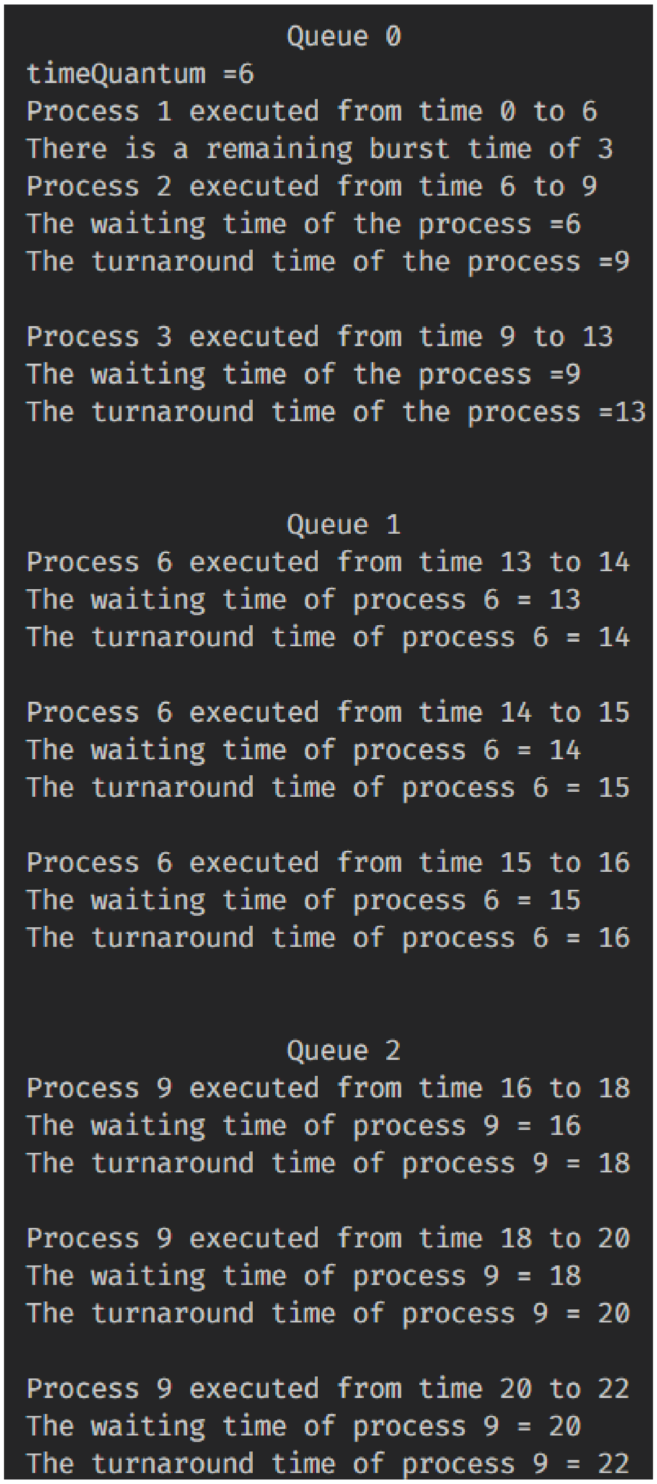
Algorithm	Pros	Cons
RR	• No starvation. • Time-sharing systems. • Every process gets a chance to execute.	• High context switches. • Poor performance for long running processes. • Becomes FIFO if the time quantum is very high.
SJF	• Minimizes average waiting time. • Provides better system throughput.	• Requires knowledge of burst times in advance. • May suffer from high overhead in the presence of frequent arrivals of short jobs.
FIFO	• Simple and easy to implement. • Suitable for batch processing systems. • Ensures fairness in First Come First serve idea.	• May suffer from convoy effect. • Poor performance in interactive systems with varying job sizes. • potentially higher average waiting times.

5 Results

This example covers the execution of the 4 queues. The time quantum in the example is taken as 8 for the sake of simplicity

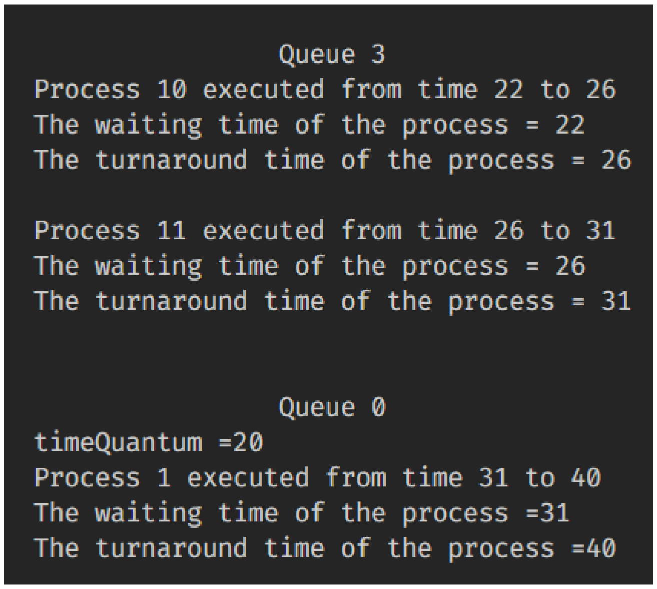
Process ID	Priority Number	Burst Time (s)
P1	0	9
P2	0	3
P3	1	4
P4	1	3
P5	1	2
P6	2	1
P7	2	13
P8	2	6
P9	3	2
P10	3	4
P11	3	5

Table 1: Process Information



```
                 Queue 0
timeQuantum =6
Process 1 executed from time 0 to 6
There is a remaining burst time of 3
Process 2 executed from time 6 to 9
The waiting time of the process =6
The turnaround time of the process =9

Process 3 executed from time 9 to 13
The waiting time of the process =9
The turnaround time of the process =13


                 Queue 1
Process 6 executed from time 13 to 14
The waiting time of process 6 = 13
The turnaround time of process 6 = 14

Process 6 executed from time 14 to 15
The waiting time of process 6 = 14
The turnaround time of process 6 = 15

Process 6 executed from time 15 to 16
The waiting time of process 6 = 15
The turnaround time of process 6 = 16


                 Queue 2
Process 9 executed from time 16 to 18
The waiting time of process 9 = 16
The turnaround time of process 9 = 18

Process 9 executed from time 18 to 20
The waiting time of process 9 = 18
The turnaround time of process 9 = 20

Process 9 executed from time 20 to 22
The waiting time of process 9 = 20
The turnaround time of process 9 = 22
```

(a) Queue 1



```
                 Queue 3
Process 10 executed from time 22 to 26
The waiting time of the process = 22
The turnaround time of the process = 26

Process 11 executed from time 26 to 31
The waiting time of the process = 26
The turnaround time of the process = 31


                 Queue 0
timeQuantum =20
Process 1 executed from time 31 to 40
The waiting time of the process =31
The turnaround time of the process =40
```

(b) Queue 2

Figure 4: Code output

6 Conclusion

6.1 Round Robin Scheduling Algorithm

- Setting the quantum too short produces the worst results, it increases the overhead.
- Higher average turnaround time than than SJF but better response.
- Turnaround time heavily depends on time quantum. It decreases if majority of burst time is lesser than the time quantum.
- Can be used for batch processing in multilevel queue.
- No starvation

6.2 Shortest Job First Scheduling Algorithm

- Produces the optimal average turnaround time among these scheduling algorithms.
- If shorter jobs keep getting enqueued then aging has to be implemented to prevent starvation.
- Turnaround time heavily depends on time quantum. It decreases if majority of burst time is lesser than thetime quantum.
- Cannot be implemented at the level of short-term CPU scheduling.
- Can be used for interactive processing in multilevel queue.

6.3 First In First Out Scheduling Algorithm

- Has low scheduling overhead because it does not require frequent context switches or involve complex scheduling decisions.
- Processes with shorter burst time will have to wait for an unreasonable time if processes with largest burst time arrive before it.
- Favours CPU bound processes than I/O bound processes.
- Perfect for processes with workloads with no strict time constraint.
- Can be used for system processing in multilevel queue.

6.4 Multi Level Queue Scheduling Algorithm

- Overhead is low because the scheduler needs to only select the appropriate queue.
- Fairness is present in this algorithm.
- Processes in the lower levels face the starvation problem.

7 Program Limitations

The current version of the program assumes that all process arrivals occur at time 0. However, the program does not account for instances where processes can arrive at different times. This limitation prevents us from obtaining accurate results, such as waiting time and turnaround time, when different processes are enqueued at different times.

The quantum decision during the selection of the Round Robin time quantum has a significant impact. Choosing a static time quantum can lead to worse performance. Instead, we implemented a dynamic approach by using the time quantum from the multilevel queues. Although not perfect, this dynamic implementation produces better results compared to the static implementation. When the Queue for round robin has more than 20 process only the first 20 process get 1 second to execute other process doesn't gets executed