Assignment 8-Multilevel Queue Scheduling

22000021

March 27, 2024

Contents

1 Introduction

Your introduction is quite comprehensive and covers all the essential aspects of the code. However, if you'd like to enhance it further, you might consider incorporating a bit more detail or emphasizing specific points. Here's a slightly refined version of your introduction:

This program serves as a simulation of a multilevel queue scheduling algorithm, a critical component in the realm of operating system design. It organizes processes into priority-based queues, with four distinct queues labeled q0 to q3. Each queue is assigned a unique scheduling strategy tailored to its priority level:

Round Robin (RR): Scheduling for q0

Shortest Job First (SJF): Scheduling for q1 and q2

First-In-First-Out (FIFO): Scheduling for q3

The time quantum for each queue is set at 20 seconds, after which the CPU switches between queues to execute processes. By prompting users to input the number of processes and their priorities for each queue, the program facilitates dynamic simulation scenarios. The primary objective of this simulation is to analyze the efficiency and performance of various scheduling algorithms under varying workload conditions. This analysis aims to provide valuable insights into optimizing system scheduling strategies for enhanced operational efficiency.

2 Methodology

The Methodology section of this report delves into the implementation details of the provided C code, which revolves around process and queue management. The code is designed to simulate a multilevel queue scheduling algorithm, a fundamental aspect of operating system design. The methodology is organized into several subsections, each focusing on different aspects of the code's implementation. These subsections include Process Implementation, which covers the structure and properties of processes; Queue Operations, detailing the functions for queue management such as enqueue, dequeue, and queue status checks; and Process Initialization, explaining how processes are initialized and added to the corresponding queues based on user input. Through a comprehensive exploration of these components, this report aims to provide a thorough understanding of the code's functionality and its relevance in the context of process and queue management in operating systems.

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];
};

int time=0;
```

1. `MAX_SIZE`: Defines the maximum number of processes that the program can handle.
2. `struct Process`:
   - `Id`: Represents the unique identifier or ID of the process.
   - `BurstTime`: Indicates the time taken by the process to execute or its burst time.
   - `RemainingTime`: Represents the remaining time needed for the process to complete its execution.
   - `WaitTime`: Stores the total waiting time of the process.
   - `TurnaroundTime`: Stores the total time taken for the process from submission to completion.
3. `struct Queue`:
   - `Rear`: Represents the rear index of the queue.
   - `Front`: Represents the front index of the queue.
   - `list[MAX_SIZE]`: An array of pointers to `Process` structures.
4. `int time`: Keeps track of the global time in the program.

2.2 Queue Operations

The code segment encapsulates essential functionalities for managing processes within a queue and scheduling algorithms. The IsEmpty and IsFull functions ascertain the status of the queue, determining whether it is empty or full, crucial for maintaining queue integrity during enqueue and dequeue operations.
Enqueue and Dequeue functions handle the insertion and deletion of processes from the queue, respectively, ensuring proper queue management. Additionally.
The MIN function identifies the process with the minimum remaining time in the queue, facilitating scheduling decisions.
Index_Dequeue function further refines dequeue operations by allowing removal of processes at specific index in the queue.
These functions collectively form the backbone of process and queue management, essential for implementing efficient scheduling strategies and ensuring smooth program execution.

```
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;
        }
    }
}


int MIN(struct Queue* Q){
    if(IsEmpty(Q)){
        return -1;
    }
    int min_Index=Q->Front;
    for(int i=Q->Front;i<=Q->Rear;i++){
        if(Q->list[min_Index]->RemainingTime>Q->list[i]->RemainingTime){
            min_Index=i;
        }
    }
    return min_Index;
}


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

Listing 1: Queue Operations

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>4){

           //To check the priority of the process
            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);
        }
    }
```

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.

```
1
2  void RR(struct Queue* Q){
3    int timeQuantum=20/size;
4    printf("timeQuantum=%d\n",timeQuantum);
5    for(int i=Q->Front;i<=Q->Rear;i++){
6      if(Q->list[i]->RemainingTime<=timeQuantum){
7        int t=time;
8        time=time+Q->list[i]->RemainingTime;
9        printf("Process %d executed from time %d to %d\n", Q->list[i]->Id,t,time);
10       Q->list[i]->TurnaroundTime=time;
11       Q->list[i]->WaiteTime=Q->list[i]->TurnaroundTime-Q->list[i]->BurstTime;
12       printf("The waiting time of the process =%d\n",Q->list[i]->WaiteTime);
13       printf("The turnaround time of the process =%d\n\n",Q->list[i]->TurnaroundTime);
14       Index_Dequeue(Q,i);
15     }
16     else{
17       int t=time;
18       time=time+timeQuantum;
19       printf("Process %d executed from time %d to %d\n", Q->list[i]->Id,t,time);
20       Q->list[i]->RemainingTime=Q->list[i]->RemainingTime-timeQuantum;
21       printf("There is a remaining burst time of %d\n\n",Q->list[i]->RemainingTime);
22     }
23   }
24 }
```

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.

```
void SJF(struct Queue *Q){
    int k=time;
    while(time<(k+20)){
        int index=MIN(Q);
        if(index>-1){
            int temp=Q->list[index]->RemainingTime;
            if(k+20-time>=temp){
                printf("Process %d executed from time %d to %d\n", Q->list[index]->Id, time, time + temp);
                time=time+temp;
                Q->list[index]->TurnaroundTime=time;
                Q->list[index]->WaiteTime=Q->list[index]->TurnaroundTime-Q->list[index]->BurstTime;
                printf("The waiting time of the process =%d\n",Q->list[index]->WaiteTime);
                printf("The turnaround time of the process =%d\n\n",Q->list[index]->TurnaroundTime);
                Index_Dequeue(Q,index);
                if(IsEmpty(Q) || time==k+20){
                    return;
                }
            }
            else{
                printf("Process %d executed from time %d to %d\n", Q->list[index]->Id,time,k+20);
                Q->list[index]->RemainingTime-=(k+20-time);
                printf("There is a remaining burst time of %d\n",Q->list[index]->RemainingTime);
                time=k+20;
            }
        }
    }
    return;
}
```

3.3 FIFO (First In First Out) Scheduling Algorithm

- The process are executed in the order they are Entered into the queue.
- if the Execution time exceeds the available time the process only runs the available time.
- the process waits until the control is given to process again in the processor

```
void FIFO(struct Queue* Q){
    int k=time;
    while (time<(k+20) && !IsEmpty(Q)) {
        int temp = Q->list[Q->Front]->RemainingTime;
        if (temp <= 20) {
            printf("Process %d executed from time %d to %d\n", Q->list[Q->Front]->Id, time, time + temp);
            time += temp;
            Q->list[Q->Front]->TurnaroundTime = time;
            Q->list[Q->Front]->WaiteTime = Q->list[Q->Front]->TurnaroundTime -
                Q->list[Q->Front]->BurstTime;
            printf("The waiting time of the process =%d\n", Q->list[Q->Front]->WaiteTime);
            printf("The turnaround time of the process =%d\n\n", Q->list[Q->Front]->TurnaroundTime);
            Dequeue(Q);
        } else {
            printf("Process %d executed from time %d to %d\n", Q->list[Q->Front]->Id, time,k+ 20);
            Q->list[Q->Front]->RemainingTime -=(k+20-time);
            printf("There is a remaining burst time of %d\n\n", Q->list[Q->Front]->RemainingTime);
            time=k+20;
        }
    }
}
```

4 Results

In this section lets see how different scheduling algorithms works for same processes using different criteria.

4.1 First In First Out

Process ID	Burst Time
P1	9
P2	11
P3	4
P4	8
P5	20
P6	1
P7	13
P8	6
P9	2
P10	21
P11	5
P12	12

Metric	Value
Average waiting Time	1
Average Turnaround Time	2
Average response Time	3
Total Time	4
Throughput	5
Fairness	No

Cycle No	Process	Execution Time	Waiting Time	Turnaround Time	Remaining Burst
1	P1	0 - 9	0	9	0
	P2	9 - 20	9	20	0
2	P3	20 - 24	20	24	0
	P4	24 - 32	24	32	0
	P5	32 - 40	32	40	12
3	P5	40 - 52	40	52	0
	P6	52 - 53	52	53	0
	P7	53 - 60	53	60	6
4	P7	60 - 66	60	66	0
	P8	66 - 72	66	72	0
	P9	72 - 74	72	74	0
	P10	74 - 80	74	80	15
5	P10	80 - 95	80	95	0
	P11	95 - 100	95	100	0
6	P12	100 - 112	100	112	0

4.2 Shortest Job First

Process ID	Burst Time
P1	9
P2	11
P3	4
P4	8
P5	20
P6	1
P7	13
P8	6
P9	2
P10	21
P11	5
P12	12

Metric	Value
Average waiting Time	1
Average Turnaround Time	2
Average response Time	3
Total Time	4
Throughput	5
Fairness	No

Cycle No	Process	Execution Time	Waiting Time	Turnaround Time	Remaining Burst
1	P1	0 - 9	0	9	0
	P2	9 - 20	9	20	0
2	P3	20 - 24	20	24	0
	P4	24 - 32	24	32	0
	P5	32 - 40	32	40	12
3	P5	40 - 52	40	52	0
	P6	52 - 53	52	53	0
	P7	53 - 60	53	60	6
4	P7	60 - 66	60	66	0
	P8	66 - 72	66	72	0
	P9	72 - 74	72	74	0
	P10	74 - 80	74	80	15
5	P10	80 - 95	80	95	0
	P11	95 - 100	95	100	0
6	P12	100 - 112	100	112	0

4.3 Round Robin

Process ID	Burst Time
P1	9
P2	11
P3	4
P4	8
P5	20
P6	1
P7	13
P8	6
P9	2
P10	21
P11	5
P12	12

Metric	Value
Average waiting Time	1
Average Turnaround Time	2
Average response Time	3
Total Time	4
Throughput	5
Fairness	No

Cycle No	Process	Execution Time	Waiting Time	Turnaround Time	Remaining Burst
1	P1	0 - 9	0	9	0
	P2	9 - 20	9	20	0
2	P3	20 - 24	20	24	0
	P4	24 - 32	24	32	0
	P5	32 - 40	32	40	12
3	P5	40 - 52	40	52	0
	P6	52 - 53	52	53	0
	P7	53 - 60	53	60	6
4	P7	60 - 66	60	66	0
	P8	66 - 72	66	72	0
	P9	72 - 74	72	74	0
	P10	74 - 80	74	80	15
5	P10	80 - 95	80	95	0
	P11	95 - 100	95	100	0
6	P12	100 - 112	100	112	0

5 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.

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