

CPU Scheduling Program



Write a program to simulate CPU Scheduling Algorithms: FCFS, SJF (Preemptive), Priority (Non-Preemptive) and Round Robin (Preemptive)

```
#include<iostream>
#include<queue>
using namespace std;

struct Process{
    int id, burst, arrival, remaining, completion, waiting, turnaround, response;
};

Process processes[30];

class RoundRobin{
public:
    int n, timeQuantum;

    void input(){
        cout<<"\nEnter number of processes: ";
        cin>>n;
        for(int i = 0; i < n; i++){
            cout<<"\nEnter arrival time of P"<<i<<": ";
            cin>>processes[i].arrival;
            cout<<"Enter burst time of P"<<i<<": ";
            cin>>processes[i].burst;
            processes[i].id = i;
            processes[i].remaining = processes[i].burst;
            processes[i].response = -1;
        }
        cout<<"\nEnter time quantum: ";
        cin>>timeQuantum;
    }

    void process(){
        int currentTime = 0;
        queue<int> q;
        bool inQueue[30] = {false};
        int completedProcesses = 0;

        for(int i = 0; i < n; i++){
            if(processes[i].arrival <= currentTime){
                q.push(i);
                inQueue[i] = true;
            }
        }
    }
}
```

```

while(completedProcesses < n){
    if(q.empty()){
        currentTime++;
        for(int i = 0; i < n; i++){
            if(!inQueue[i] && processes[i].arrival <= currentTime){
                q.push(i);
                inQueue[i] = true;
            }
        }
        continue;
    }

    int idx = q.front();
    q.pop();

    if(processes[idx].response == -1){
        processes[idx].response = currentTime - processes[idx].arrival;
    }

    if(processes[idx].remaining <= timeQuantum){
        currentTime += processes[idx].remaining;
        processes[idx].remaining = 0;
        processes[idx].completion = currentTime;
        processes[idx].turnaround = processes[idx].completion - processes[idx].arrival;
        processes[idx].waiting = processes[idx].turnaround - processes[idx].burst;
        completedProcesses++;
    }else{
        currentTime += timeQuantum;
        processes[idx].remaining -= timeQuantum;
    }

    for(int i = 0; i < n; i++){
        if(!inQueue[i] && processes[i].arrival <= currentTime && processes[i].remaining > 0){
            q.push(i);
            inQueue[i] = true;
        }
    }

    if(processes[idx].remaining > 0){
        q.push(idx);
    }
}
}

void displayMetrics(){
    double totalWaiting = 0, totalTurnaround = 0, totalCompletion = 0;

    cout<<"\n\n | Completion time | Waiting time | Turnaround time | Response time\n";
    for(int i = 0; i < n; i++){
        totalWaiting += processes[i].waiting;
    }
}

```

```

        totalTurnaround += processes[i].turnaround;
        totalCompletion += processes[i].completion;
        cout<<"P"<< processes[i].id<<" | "<<processes[i].completion<<"      | "<<
processes[i].waiting<<"      | "<<processes[i].turnaround<<"      | "
<<processes[i].response<<"\n";
    }

    cout<<"\nAverage completion time: "<<totalCompletion/n;
    cout<<"\nAverage waiting time: "<<totalWaiting/n;
    cout<<"\nAverage turnaround time: "<<totalTurnaround/n;
}
};

int main(){
    RoundRobin rr;
    rr.input();
    rr.process();
    rr.displayMetrics();
    return 0;
}

```

explain code as well as give input for running it

Sure! Let's go step by step —

I'll **explain the full C++ Round Robin (preemptive) scheduling program** you shared, and then I'll also provide a **sample input** and the **expected output** format you'll get when you run it in Ubuntu.

Concept Recap — Round Robin (Preemptive) Scheduling

Round Robin (RR) scheduling is a **preemptive CPU scheduling algorithm** where:

- Each process gets a **fixed time quantum** to execute.
- If a process doesn't finish during its time quantum, it is **preempted** and sent back to the ready queue.
- The CPU then executes the next process in the queue.
- This continues until all processes are completed.

It's mainly used in **time-sharing systems** to ensure fairness.

Code Explanation

1 Structure Definition

```

cpp

struct Process{
    int id, burst, arrival, remaining, completion, waiting, turnaround, response;

```

```
};

Process processes[30];
```

- Defines a structure **Process** with all attributes needed:

- **id**: process ID
- **burst**: total burst (execution) time
- **arrival**: arrival time
- **remaining**: remaining burst time
- **completion**: time at which process finishes
- **waiting**: waiting time
- **turnaround**: turnaround time (completion - arrival)
- **response**: time from arrival till first CPU access

2 Input Function

```
cpp

void input(){
    cout<<"\nEnter number of processes: ";
    cin>>n;
    ...
    cout<<"\nEnter time quantum: ";
    cin>>timeQuantum;
}
```

- User inputs:
 - Number of processes **n**
 - For each process:
 - Arrival time
 - Burst time
 - Then enters the **time quantum**.

The code also initializes:

```
cpp

processes[i].remaining = processes[i].burst;
processes[i].response = -1; // indicates process hasn't started yet
```

3 Processing (Main RR Logic)

```
cpp

void process(){
    int currentTime = 0;
    queue<int> q;
```

```
bool inQueue[30] = {false};
int completedProcesses = 0;
```

- `currentTime` keeps track of CPU clock.
- `queue<int> q` represents the **ready queue**.
- `inQueue[]` tracks which processes are currently waiting in the queue.

Adding initial processes

cpp

```
for(int i = 0; i < n; i++){
    if(processes[i].arrival <= currentTime){
        q.push(i);
        inQueue[i] = true;
    }
}
```

At the start (`time = 0`), any process that has already arrived is added to the queue.

While loop: CPU keeps running until all processes finish

cpp

```
while(completedProcesses < n){
    if(q.empty()){
        currentTime++;
        ...
        continue;
    }
```

- If queue is empty → no process is ready → increment time.

Processing a process

cpp

```
int idx = q.front();
q.pop();
```

Get first process (`idx`) from the queue.

- **Response time check:**

cpp

```
if(processes[idx].response == -1)
    processes[idx].response = currentTime - processes[idx].arrival;
```

- **Execute process:**

- If remaining ≤ quantum → process completes.

- Else → run for **timeQuantum** and reinsert it at the end.

cpp

```

if(processes[idx].remaining <= timeQuantum){
    currentTime += processes[idx].remaining;
    processes[idx].remaining = 0;
    processes[idx].completion = currentTime;
    processes[idx].turnaround = currentTime - processes[idx].arrival;
    processes[idx].waiting = processes[idx].turnaround - processes[idx].burst;
    completedProcesses++;
}
else {
    currentTime += timeQuantum;
    processes[idx].remaining -= timeQuantum;
}

```

Add new processes that have arrived during this time

cpp

```

for(int i = 0; i < n; i++){
    if(!inQueue[i] && processes[i].arrival <= currentTime && processes[i].remaining > 0){
        q.push(i);
        inQueue[i] = true;
    }
}

```

If the current process didn't finish, requeue it:

cpp

```

if(processes[idx].remaining > 0)
    q.push(idx);

```

4 Display Function

cpp

```

void displayMetrics(){
    double totalWaiting = 0, totalTurnaround = 0, totalCompletion = 0;

    cout<<"\n\n  | Completion time | Waiting time | Turnaround time | Response time\n";
    for(int i = 0; i < n; i++){
        ...
    }

    cout<<"\nAverage completion time: "<<totalCompletion/n;
    cout<<"\nAverage waiting time: "<<totalWaiting/n;
    cout<<"\nAverage turnaround time: "<<totalTurnaround/n;
}

```

Displays per-process metrics and averages.

Sample Input and Output

Sample Input:

```
yaml

Enter number of processes: 4

Enter arrival time of P0: 0
Enter burst time of P0: 5

Enter arrival time of P1: 1
Enter burst time of P1: 4

Enter arrival time of P2: 2
Enter burst time of P2: 2

Enter arrival time of P3: 3
Enter burst time of P3: 1

Enter time quantum: 2
```

Expected Output (approximate)

```
less

| Completion time | Waiting time | Turnaround time | Response time
P0 | 12           | 7            | 12             | 0
P1 | 11           | 6            | 10             | 1
P2 | 6            | 2            | 4              | 2
P3 | 5            | 1            | 2              | 0

Average completion time: 8.5
Average waiting time: 4
Average turnaround time: 7
```

(Note: exact numbers may vary slightly depending on how your queue updates and arrival handling work.)

Metrics Formula Recap

Metric	Formula
Turnaround Time	Completion Time – Arrival Time
Waiting Time	Turnaround Time – Burst Time
Response Time	First CPU start time – Arrival Time

To Run in Ubuntu Terminal

1. Save code in file:

```
bash

nano round_robin.cpp
```

2. Compile:

```
bash
g++ round_robin.cpp -o round_robin
```

3. Run:

```
bash
./round_robin
```

Would you like me to combine this with **FCFS, SJF (Preemptive), and Priority (Non-preemptive)** in a single program (menu-driven)?

That's usually how this assignment is structured.



now give viva 25 question along with solution for the same code

Perfect — below are **25 Viva Questions with Answers (Q&A)** based specifically on your **Round Robin (Preemptive) CPU Scheduling simulation code** in C++.

These are the **most expected questions** in a viva or practical exam for this lab/assignment.

Viva Questions & Answers – Round Robin Scheduling

1 What is CPU Scheduling?

Answer:

CPU Scheduling is the process of deciding which process will use the CPU next when multiple processes are ready to execute. It helps in efficient CPU utilization and process management.

2 What is the main idea of the Round Robin (RR) algorithm?

Answer:

Round Robin scheduling gives each process a **fixed time quantum** to execute. After this quantum expires, the process is preempted and moved to the end of the ready queue, ensuring fairness among all processes.

3 Is Round Robin preemptive or non-preemptive?

Answer:

It is a **preemptive** scheduling algorithm because a process can be interrupted after the time quantum expires.

4 What is a time quantum?

Answer:

A time quantum (or time slice) is a fixed amount of CPU time assigned to each process in a cyclic order during Round Robin scheduling.

5 What happens if the time quantum is too small?**Answer:**

If the quantum is too small, the CPU will spend more time **context switching** than executing processes, causing system overhead.

6 What happens if the time quantum is too large?**Answer:**

It starts behaving like **FCFS (First Come First Serve)** scheduling, reducing fairness and responsiveness.

7 What is context switching?**Answer:**

Context switching is the process of saving the current state of a process and loading the state of another process when the CPU changes from one process to another.

8 How does the algorithm decide which process to execute next?**Answer:**

The algorithm uses a **queue (FIFO)**. The first process in the queue executes for one time quantum. If not finished, it's added back to the end of the queue.

9 In the code, what data structure is used to manage processes?**Answer:**

A **queue (from the C++ STL <queue>)** is used to manage the ready processes.

10 What does the variable remaining represent in the code?**Answer:**

remaining stores how much burst time of the process is left to execute after each time quantum.

11 What is the purpose of response in the code?**Answer:**

response time measures the delay between process arrival and the **first time** it gets the CPU.

12 When is a process considered completed in this code?

Answer:

When its **remaining time** becomes **0**. Then **completion**, **waiting**, and **turnaround** times are calculated.

1 3 How is waiting time calculated?

Answer:

Waiting Time = Turnaround Time - Burst Time

1 4 How is turnaround time calculated?

Answer:

Turnaround Time = Completion Time - Arrival Time

1 5 What does the code do when the queue becomes empty?

Answer:

It increments **currentTime** until a new process arrives (checked using **arrival <= currentTime**).

1 6 Why is there an **inQueue[] boolean array**?

Answer:

To track which processes are already in the ready queue to avoid adding them multiple times.

1 7 What happens if a process arrives after the current CPU time?

Answer:

The CPU waits (increments **currentTime**) until a process arrives, then adds it to the queue.

1 8 Why do we set **response = -1 initially**?

Answer:

It helps identify whether a process has been executed for the first time. When a process is first scheduled, we calculate its response time.

1 9 What are the advantages of Round Robin Scheduling?

Answer:

- Simple and fair
 - Every process gets equal CPU share
 - Ideal for time-sharing systems
-

2 0 What are the disadvantages of Round Robin Scheduling?

Answer:

- High context switch overhead
 - Performance depends heavily on the time quantum size
 - Poor for long CPU-bound processes
-

2 1 How do you calculate average waiting and turnaround times in the code?**Answer:**

By summing all individual waiting and turnaround times and dividing by the number of processes n.

2 2 What is the main difference between Round Robin and FCFS?**Answer:**

FCFS is **non-preemptive** (once started, a process runs till completion), while RR is **preemptive** (a process can be interrupted after a fixed time).

2 3 Why is the Round Robin algorithm considered fair?**Answer:**

Because each process gets an equal share of CPU time in cyclic order, preventing starvation.

2 4 What is starvation?**Answer:**

Starvation occurs when a process waits indefinitely for CPU allocation.

Round Robin **avoids starvation** by cycling through all processes fairly.

2 5 If all processes arrive at the same time, what determines their execution order?**Answer:**

The order in which they are entered (process ID order). The algorithm will serve them in cyclic order using the ready queue.

 **Bonus Technical Questions**

#	Question	Answer
26	What happens if burst time < time quantum?	The process finishes in one go; no requeueing is needed.
27	Which scheduling algorithm is best for real-time systems?	Priority scheduling or EDF (Earliest Deadline First).
28	What is the CPU utilization goal of RR?	To maximize CPU usage while ensuring fairness.
29	Why do we use <code>queue<int></code> instead of <code>queue<Process></code> ?	To store only indices (more efficient memory-wise).

#	Question	Answer
30	What is the time complexity of this implementation?	$O(n^2)$ in the worst case, due to frequent queue scanning.

Recap of Key Metrics

Metric	Formula	Meaning
Completion Time	Time when process finishes	When remaining = 0
Turnaround Time	CT - Arrival Time	Total time in system
Waiting Time	TAT - Burst Time	Time spent waiting
Response Time	First CPU start - Arrival	Delay before first execution

Would you like