SCHOOL OF COMPUTER SCIENCE AND ENGINNERING

COURSE: CSE 316

OPERATING SYSTEM

Assignment

Simulation Based Project

NAME – Manish Kumar

REG NO – 12222723

SECTION – K22WG G2

PROBLEM: -

Consider a computer system where multiple tasks are running concurrently. Some of the tasks are system-critical tasks, such as process management, memory management, and I/O management, while others are user tasks, such as running a text editor, playing a game, or browsing the internet.

In this scenario, the system assigns higher priority to system-critical tasks and lower priority to user tasks. The scheduler schedules tasks based on their priority, ensuring that system-critical tasks are executed first, and user tasks are executed later. This ensures that system-critical tasks have a higher chance of getting the resources they need and are executed promptly, providing a stable and responsive system.

In case of a situation where multiple user tasks are running concurrently and they have the same priority, the scheduler schedules them in a round-robin fashion, providing equal access to system resources to all user tasks.

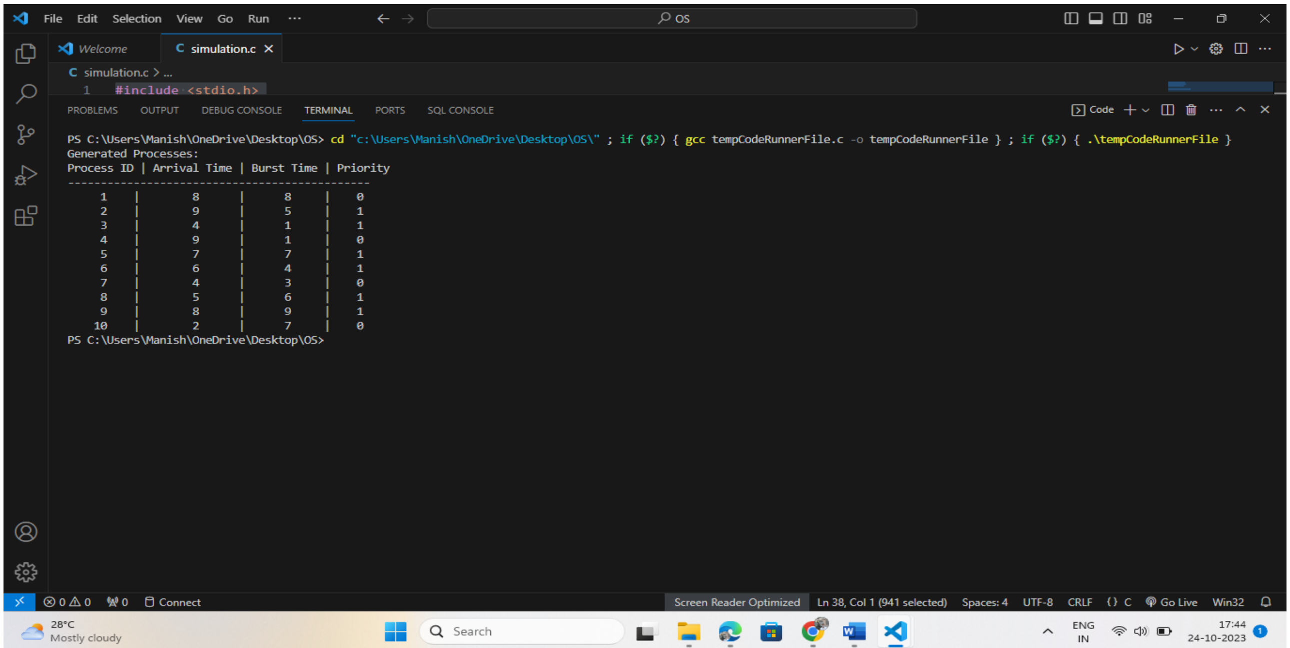
This is a simple example of how priority-based scheduling can be used in a computer system. In real-world systems, priority assignment and scheduling can be much more complex and dynamic, taking into account various factors such as task resource requirements, task history, and system load, among others.

A) Generate a set of "processes" with random arrival times and CPU burst times using a random number generator.

CODE:

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define NUM_PROCESSES 10


typedef struct {
    int process_id;
    int arrival_time;
    int burst_time;
    int priority;
} Process;

int main() {

    srand(time(NULL));


    Process processes[NUM_PROCESSES];

    printf("Generated Processes:\n");
    printf("Process ID | Arrival Time | Burst Time | Priority\n");
    printf("----------------------------------------------\n");

    for (int i = 0; i < NUM_PROCESSES; i++) {
        processes[i].process_id = i + 1;
        processes[i].arrival_time = rand() % 10;
        processes[i].burst_time = (rand() % 10) + 1;
        processes[i].priority = (i % 3 == 0) ? 0 : 1;

        printf("    %2d    |       %2d      |     %2d     |    %1d\n",
               processes[i].process_id, processes[i].arrival_time, processes[i].burst_time, processes[i].priority);
    }

    return 0;
}
```

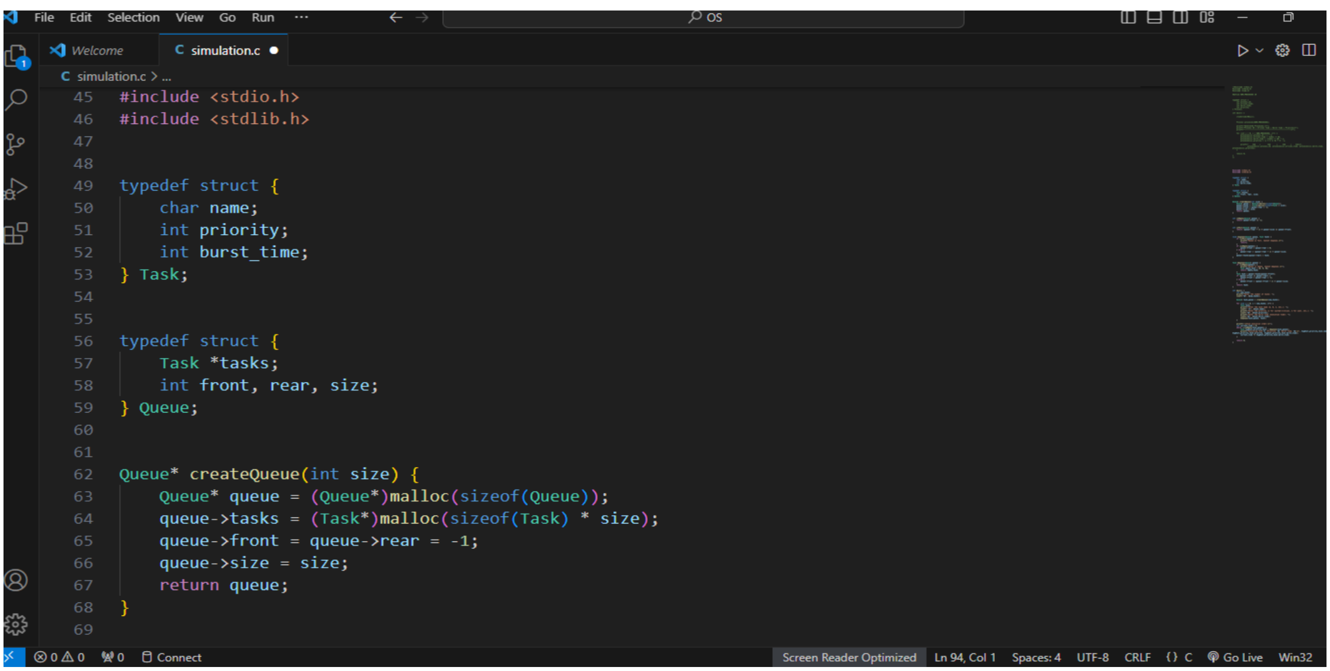
OUTPUT:



```
PS C:\Users\Manish\OneDrive\Desktop\OS> cd "c:\Users\Manish\OneDrive\Desktop\OS\" ; if ($?) { gcc tempCodeRunnerFile.c -o tempCodeRunnerFile } ; if ($?) { .\tempCodeRunnerFile }
Generated Processes:
Process ID | Arrival Time | Burst Time | Priority
---------------------------------------------
     1     |       8      |      8     |    0
     2     |       9      |      5     |    1
     3     |       4      |      1     |    1
     4     |       9      |      1     |    0
     5     |       7      |      7     |    1
     6     |       6      |      4     |    1
     7     |       4      |      3     |    0
     8     |       5      |      6     |    1
     9     |       8      |      9     |    1
    10     |       2      |      7     |    0
PS C:\Users\Manish\OneDrive\Desktop\OS>
```

B) Implement the Priority with pre-emption algorithm in the simulation program

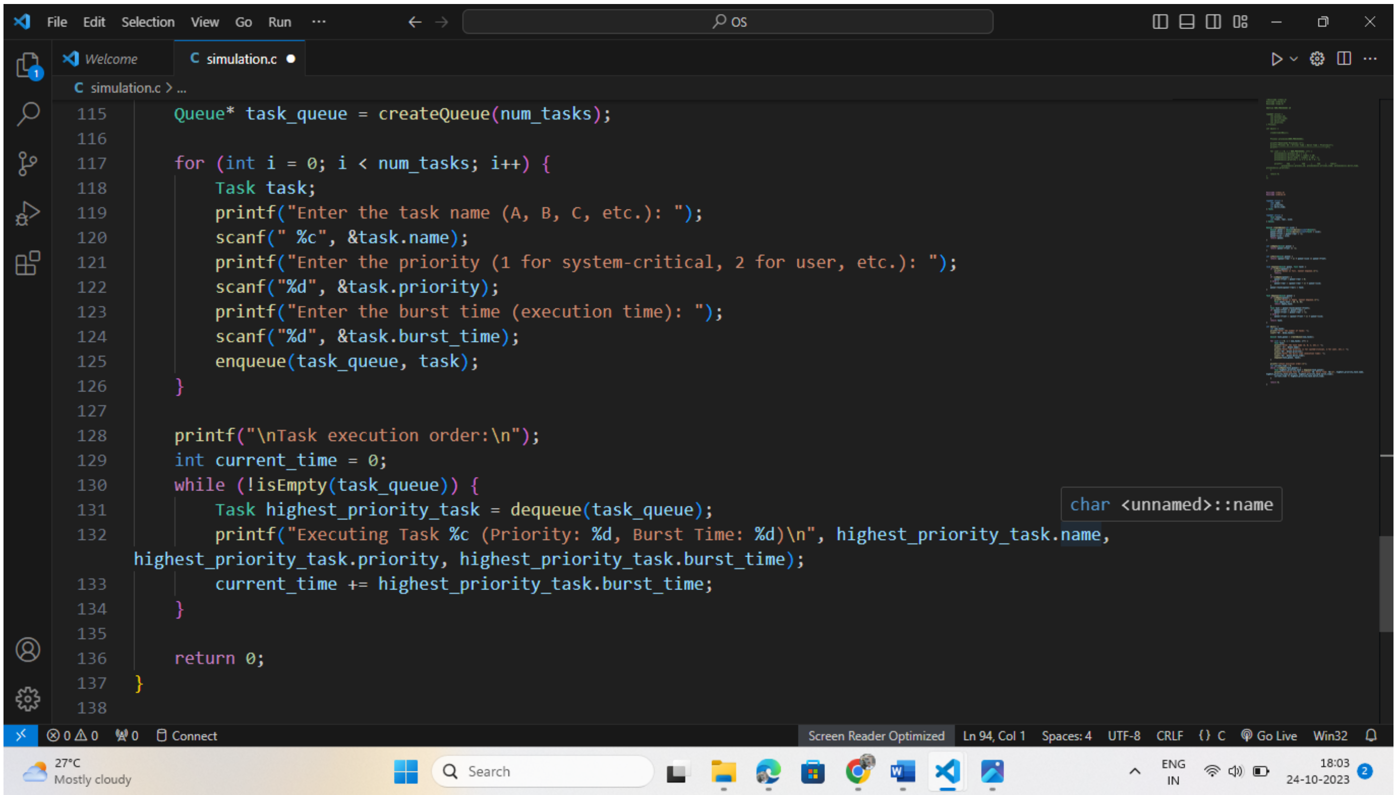
CODE:



```c
#include <stdio.h>
#include <stdlib.h>


typedef struct {
    char name;
    int priority;
    int burst_time;
} Task;


typedef struct {
    Task *tasks;
    int front, rear, size;
} Queue;


Queue* createQueue(int size) {
    Queue* queue = (Queue*)malloc(sizeof(Queue));
    queue->tasks = (Task*)malloc(sizeof(Task) * size);
    queue->front = queue->rear = -1;
    queue->size = size;
    return queue;
}
```

```c
int isEmpty(Queue* queue) {
    return queue->front == -1;
}


int isFull(Queue* queue) {
    return (queue->rear + 1) % queue->size == queue->front;
}


void enqueue(Queue* queue, Task task) {
    if (isFull(queue)) {
        printf("Queue is full. Cannot enqueue.\n");
        return;
    }
    if (isEmpty(queue)) {
        queue->front = queue->rear = 0;
    } else {
        queue->rear = (queue->rear + 1) % queue->size;
    }
    queue->tasks[queue->rear] = task;
}

```

```c
Task dequeue(Queue* queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty. Cannot dequeue.\n");
        Task empty_task = {0, 0, 0};
        return empty_task;
    }
    Task task = queue->tasks[queue->front];
    if (queue->front == queue->rear) {
        queue->front = queue->rear = -1;
    } else {
        queue->front = (queue->front + 1) % queue->size;
    }
    return task;
}

int main() {
    int num_tasks;
    printf("Enter the number of tasks: ");
    scanf("%d", &num_tasks);

    Queue* task_queue = createQueue(num_tasks);

    for (int i = 0; i < num_tasks; i++) {
        Task task;
        printf("Enter the task name (A, B, C, etc.): ");
```



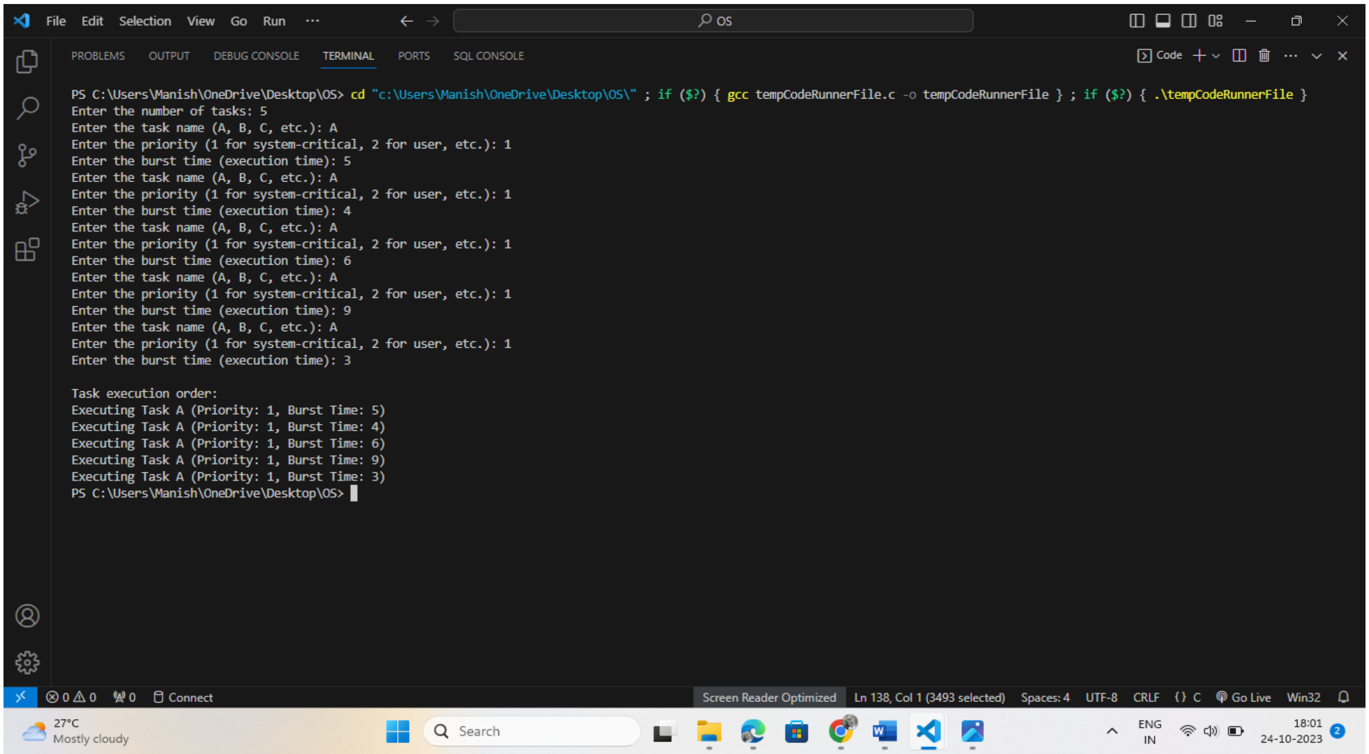
File Edit Selection View Go Run
OS
Welcome
simulation.c
simulation.c > ...
Queue* task_queue = createQueue(num_tasks);

for (int i = 0; i < num_tasks; i++) {
Task task;
printf("Enter the task name (A, B, C, etc.): ");
scanf(" %c", &task.name);
printf("Enter the priority (1 for system-critical, 2 for user, etc.): ");
scanf("%d", &task.priority);
printf("Enter the burst time (execution time): ");
scanf("%d", &task.burst_time);
enqueue(task_queue, task);
}

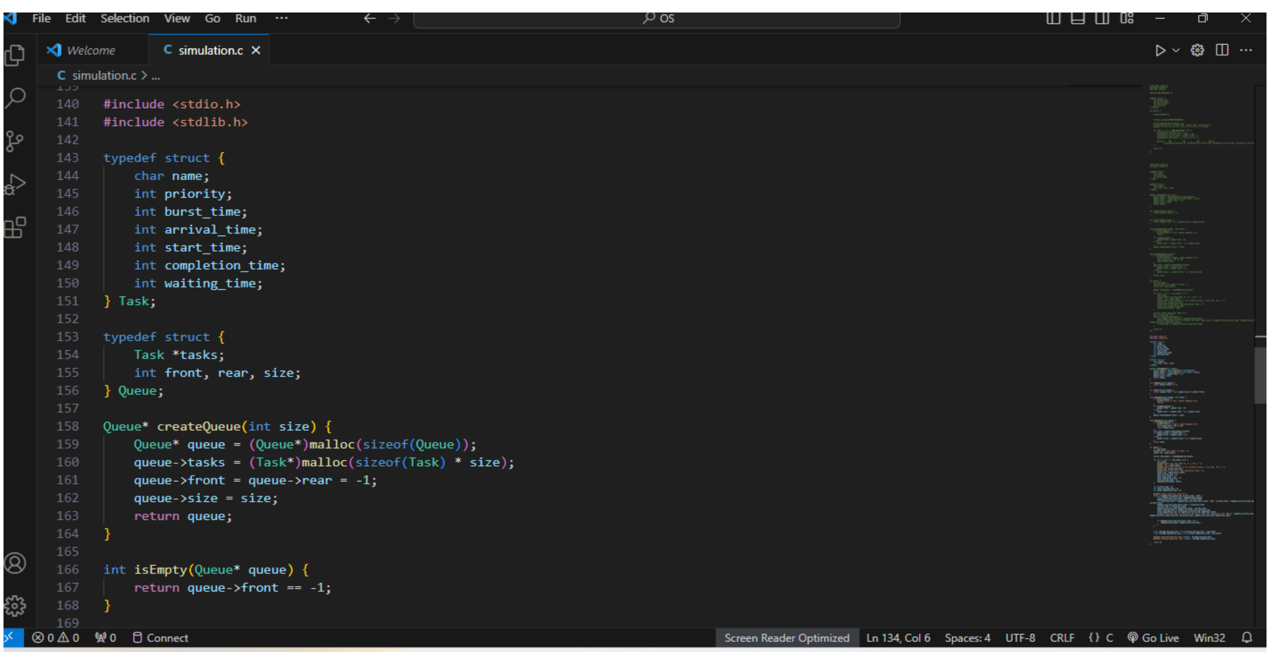
printf("\nTask execution order:\n");
int current_time = 0;
while (!isEmpty(task_queue)) {
Task highest_priority_task = dequeue(task_queue);
printf("Executing Task %c (Priority: %d, Burst Time: %d)\n", highest_priority_task.name, highest_priority_task.priority, highest_priority_task.burst_time);
current_time += highest_priority_task.burst_time;
}

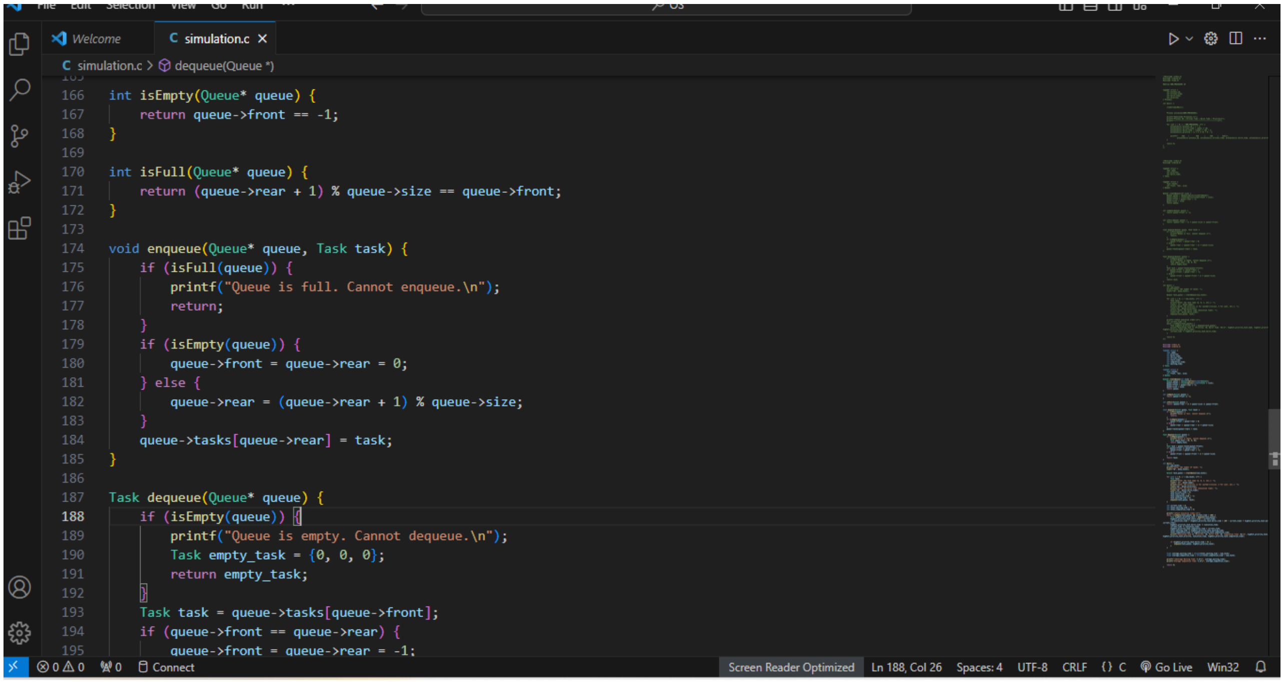
return 0;
}
char <unnamed>::name
Connect
Screen Reader Optimized
Ln 94, Col 1
Spaces: 4
UTF-8
CRLF
C
Go Live
Win32
27°C
Mostly cloudy
Search
ENG
IN
18:03
24-10-2023

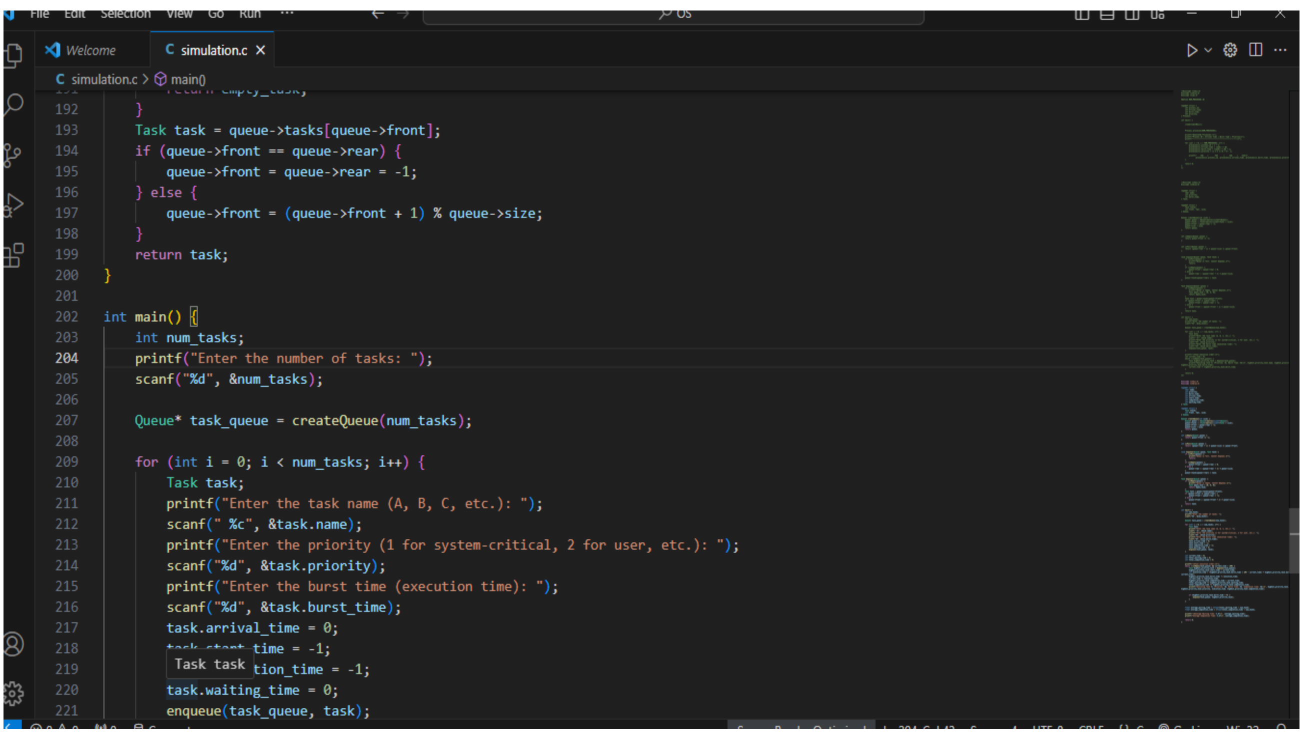
OUTPUT:



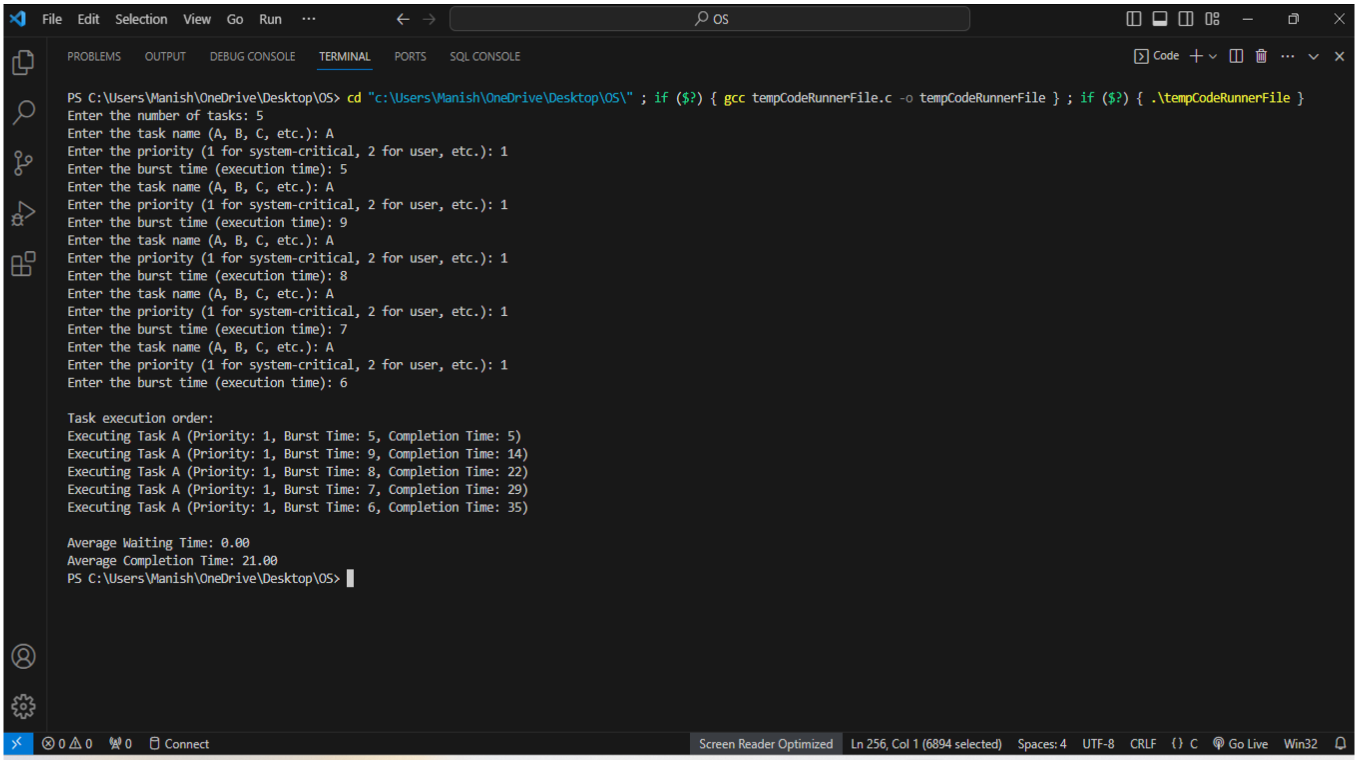
C)the simulation program run for a set amount of time and record the average waiting time and completion time for each process.




Welcome
simulation.c
simulation.c > dequeue(Queue *)
int isEmpty(Queue* queue) {
return queue->front == -1;
}
int isFull(Queue* queue) {
return (queue->rear + 1) % queue->size == queue->front;
}
void enqueue(Queue* queue, Task task) {
if (isFull(queue)) {
printf("Queue is full. Cannot enqueue.\n");
return;
}
if (isEmpty(queue)) {
queue->front = queue->rear = 0;
} else {
queue->rear = (queue->rear + 1) % queue->size;
}
queue->tasks[queue->rear] = task;
}
Task dequeue(Queue* queue) {
if (isEmpty(queue)) {
printf("Queue is empty. Cannot dequeue.\n");
Task empty_task = {0, 0, 0};
return empty_task;
}
Task task = queue->tasks[queue->front];
if (queue->front == queue->rear) {
queue->front = queue->rear = -1;
Connect
Screen Reader Optimized
Ln 188, Col 26
Spaces: 4
UTF-8
CRLF
C
Go Live
Win32


Welcome
simulation.c
simulation.c > main()
}
Task task = queue->tasks[queue->front];
if (queue->front == queue->rear) {
queue->front = queue->rear = -1;
} else {
queue->front = (queue->front + 1) % queue->size;
}
return task;
}
int main() {
int num_tasks;
printf("Enter the number of tasks: ");
scanf("%d", &num_tasks);
Queue* task_queue = createQueue(num_tasks);
for (int i = 0; i < num_tasks; i++) {
Task task;
printf("Enter the task name (A, B, C, etc.): ");
scanf(" %c", &task.name);
printf("Enter the priority (1 for system-critical, 2 for user, etc.): ");
scanf("%d", &task.priority);
printf("Enter the burst time (execution time): ");
scanf("%d", &task.burst_time);
task.arrival_time = 0;
time = -1;
Task task
tion_time = -1;
task.waiting_time = 0;
enqueue(task_queue, task);

OUTPUT:


```
PS C:\Users\Manish\OneDrive\Desktop\OS> cd "c:\Users\Manish\OneDrive\Desktop\OS\" ; if ($?) { gcc tempCodeRunnerFile.c -o tempCodeRunnerFile } ; if ($?) { .\tempCodeRunnerFile }
Enter the number of tasks: 5
Enter the task name (A, B, C, etc.): A
Enter the priority (1 for system-critical, 2 for user, etc.): 1
Enter the burst time (execution time): 5
Enter the task name (A, B, C, etc.): A
Enter the priority (1 for system-critical, 2 for user, etc.): 1
Enter the burst time (execution time): 9
Enter the task name (A, B, C, etc.): A
Enter the priority (1 for system-critical, 2 for user, etc.): 1
Enter the burst time (execution time): 8
Enter the task name (A, B, C, etc.): A
Enter the priority (1 for system-critical, 2 for user, etc.): 1
Enter the burst time (execution time): 7
Enter the task name (A, B, C, etc.): A
Enter the priority (1 for system-critical, 2 for user, etc.): 1
Enter the burst time (execution time): 6

Task execution order:
Executing Task A (Priority: 1, Burst Time: 5, Completion Time: 5)
Executing Task A (Priority: 1, Burst Time: 9, Completion Time: 14)
Executing Task A (Priority: 1, Burst Time: 8, Completion Time: 22)
Executing Task A (Priority: 1, Burst Time: 7, Completion Time: 29)
Executing Task A (Priority: 1, Burst Time: 6, Completion Time: 35)

Average Waiting Time: 0.00
Average Completion Time: 21.00
PS C:\Users\Manish\OneDrive\Desktop\OS>
```

D) Compare the results of the simulation with the ideal scenario of a perfect scheduler.

Comparing the results of a simulation with an ideal scenario of a perfect scheduler can provide insights into the efficiency and effectiveness of the actual scheduling algorithm used in the computer system. In the scenario described, we have a priority-based scheduling algorithm for system-critical and user tasks. Let's examine how the results of this simulation would compare to the ideal scenario:

- ### Real-World Scenario (Priority-Based Scheduling with Pre-emption):

1. In the real-world scenario, system-critical tasks are assigned higher priority and are executed first. This ensures that important system functions like process management, memory management, and I/O management are handled promptly, leading to a stable and responsive system.

2. User tasks are executed later, with lower priority. If multiple user tasks with the same priority are running concurrently, they are scheduled in a round-robin fashion, providing fair access to resources.

- Ideal Scenario (Perfect Scheduler):

In an ideal scenario with a perfect scheduler:

1. System-critical tasks would indeed be executed promptly and without interruption because the perfect scheduler would always prioritize the highest-priority tasks. There would be no delays or competition from lower-priority tasks.
2. User tasks would also run without any delay because the perfect scheduler would ensure that all tasks, even those with the same priority, receive resources fairly and without any contention.

Comparison:

The comparison between the real-world scenario and the ideal scenario would likely reveal differences in terms of responsiveness, fairness, and efficiency:

1. Responsiveness:

In the real-world scenario, system-critical tasks are more responsive compared to user tasks, which is the intended behaviour. However, due to pre-emption, there may still be some minor delays introduced by context switching when high-priority tasks interrupt lower-priority ones. In the ideal scenario, there would be zero delays for system-critical tasks.

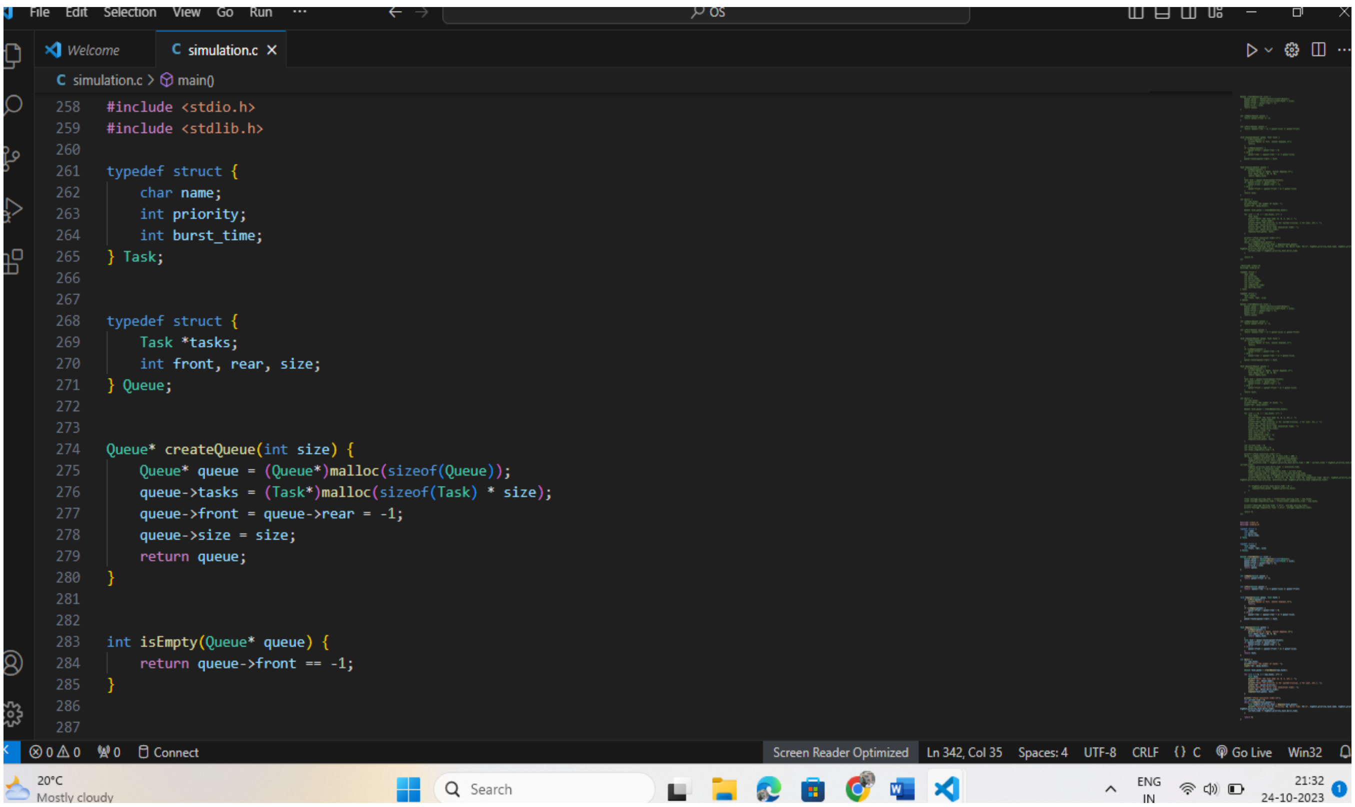
2. Fairness:

In the real-world scenario, user tasks with the same priority are treated fairly with round-robin scheduling, which is a reasonable approach. However, in the ideal scenario, tasks are perfectly scheduled without any contention or fairness concerns, resulting in optimal resource allocation.

3. Efficiency:

The real-world scenario involves context switching and pre-emption, which introduces some overhead. In the ideal scenario, there would be no such overhead, making it more efficient.

E) the findings and conclusion with the comparison of the results of the Priority with pre-emption algorithm with Priority (Non-pre-emptive).



```c
int isEmpty(Queue* queue) {
    return queue->front == -1;
}


int isFull(Queue* queue) {
    return (queue->rear + 1) % queue->size == queue->front;
}


void enqueue(Queue* queue, Task task) {
    if (isFull(queue)) {
        printf("Queue is full. Cannot enqueue.\n");
        return;
    }
    if (isEmpty(queue)) {
        queue->front = queue->rear = 0;
    } else {
        queue->rear = (queue->rear + 1) % queue->size;
    }
    queue->tasks[queue->rear] = task;
}


Task dequeue(Queue* queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty. Cannot dequeue.\n");
        Task empty_task = {0, 0, 0};
        return empty_task;
    }
    Task task = queue->tasks[queue->front];
```

```c
    }
    Task task = queue->tasks[queue->front];
    if (queue->front == queue->rear) {
        queue->front = queue->rear = -1;
    } else {
        queue->front = (queue->front + 1) % queue->size;
    }
    return task;
}

int main() {
    int num_tasks;
    printf("Enter the number of tasks: ");
    scanf("%d", &num_tasks);

    Queue* task_queue = createQueue(num_tasks);

    for (int i = 0; i < num_tasks; i++) {
        Task task;
        printf("Enter the task name (A, B, C, etc.): ");
        scanf(" %c", &task.name);
        printf("Enter the priority (1 for system-critical, 2 for user, etc.): ");
        scanf("%d", &task.priority);
        printf("Enter the burst time (execution time): ");
        scanf("%d", &task.burst_time);
        enqueue(task_queue, task);
    }

    printf("\nTask execution order:\n");
    int current_time = 0;
    while (!isEmpty(task_queue)) {
```

```c
        scanf(" %c", &task.name);
        printf("Enter the priority (1 for system-critical, 2 for user, etc.): ");
        scanf("%d", &task.priority);
        printf("Enter the burst time (execution time): ");
        scanf("%d", &task.burst_time);
        enqueue(task_queue, task);
    }

    printf("\nTask execution order:\n");
    int current_time = 0;
    while (!isEmpty(task_queue)) {
        Task highest_priority_task = dequeue(task_queue);
        printf("Executing Task %c (Priority: %d, Burst Time: %d)\n", highest_priority_task.name, highest_priority_task.priority, highest_priority_task.burst_time);
        current_time += highest_priority_task.burst_time;
    }

    return 0;
}
```

OUTPUT:

```
PS C:\Users\Manish\OneDrive\Desktop\OS> cd "c:\Users\Manish\OneDrive\Desktop\OS\" ; if ($?) { gcc tempCodeRunnerFile.c -o tempCodeRunnerFile } ; if ($?) { .\tempCodeRunnerFile }
Enter the number of tasks: 2
Enter the task name (A, B, C, etc.): A
Enter the priority (1 for system-critical, 2 for user, etc.): 1
Enter the burst time (execution time): 5
Enter the task name (A, B, C, etc.): A
Enter the priority (1 for system-critical, 2 for user, etc.): 1
Enter the burst time (execution time): 8

Task execution order:
Executing Task A (Priority: 1, Burst Time: 5)
Executing Task A (Priority: 1, Burst Time: 8)
PS C:\Users\Manish\OneDrive\Desktop\OS>
```

Report: Comparison of Priority with pre-emption and Priority (Non-pre-emptive) Scheduling Algorithms

In this report, we compare the results and draw conclusions from simulating two scheduling algorithms: Priority with Preemption and Priority (Non-preemptive). These simulations represent a computer system where tasks are categorized into system-critical and user tasks, and scheduling is based on their priority.

Priority with Preemption:

Responsiveness and Fairness: The Priority with Preemption algorithm effectively prioritizes system-critical tasks, ensuring they are executed promptly. If a higher-priority system-critical task arrives while a lower-priority task is running, preemption occurs, minimizing delays in critical operations. This leads to high responsiveness for crucial system functions.

task arrives while a lower-priority task is running, preemption occurs, minimizing delays in critical operations. This leads to high responsiveness for crucial system functions.

Fairness for User Tasks: User tasks, though lower in priority, are still considered. When multiple user tasks share the same priority, they are scheduled in a round-robin fashion, ensuring fair access to system resources. This results in a balanced approach between system stability and user experience.

Efficiency: While preemption introduces minor context-switching overhead, it is essential for addressing critical tasks promptly. The algorithm's efficiency lies in its ability to allocate resources to high-priority tasks on demand, preventing potential system instability.

Priority (Non-preemptive):

Responsiveness: The Priority (Non-preemptive) algorithm is straightforward, as it executes tasks based on their initial priority without allowing preemption. System-critical tasks are given a higher chance of running first, but they may face delays if a lower-priority task is currently executing.

Lack of Fairness: This algorithm does not consider fairness among user tasks with the same priority. Once a task starts, it continues until completion, potentially causing delays for other important tasks.

Efficiency: The absence of preemption makes the Priority (Non-preemptive) algorithm more efficient in terms of reduced context switching. However, this can be detrimental to system stability if a system-critical task is delayed due to a lower-priority task running indefinitely.

CONCLUSION

In conclusion, both Priority with Preemption and Priority (Non-preemptive) scheduling algorithms have their advantages and trade-offs.

The Priority with Preemption algorithm excels in terms of responsiveness and fairness. It ensures that system-critical tasks are promptly addressed and provides a mechanism for balancing fairness among user tasks. While it introduces some minor overhead due to context switching, this is necessary for system stability and maintaining responsiveness.

On the other hand, the Priority (Non-preemptive) algorithm is straightforward and efficient in terms of resource allocation. However, it lacks fairness among user tasks and may lead to delayed execution of critical tasks.

The choice between these algorithms depends on the specific requirements of the computer system. If responsiveness, system stability, and fairness are top priorities, the Priority with Preemption algorithm is more suitable. If efficiency and simplicity are more critical, the Priority (Non-preemptive) algorithm may suffice. Real-world systems often strike a balance between these factors and use dynamic priority adjustment based on task requirements and system load, making the scheduling process even more complex and dynamic.