



**L** OVELY  
**P** ROFESSIONAL  
**U** NIVERSITY

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

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.

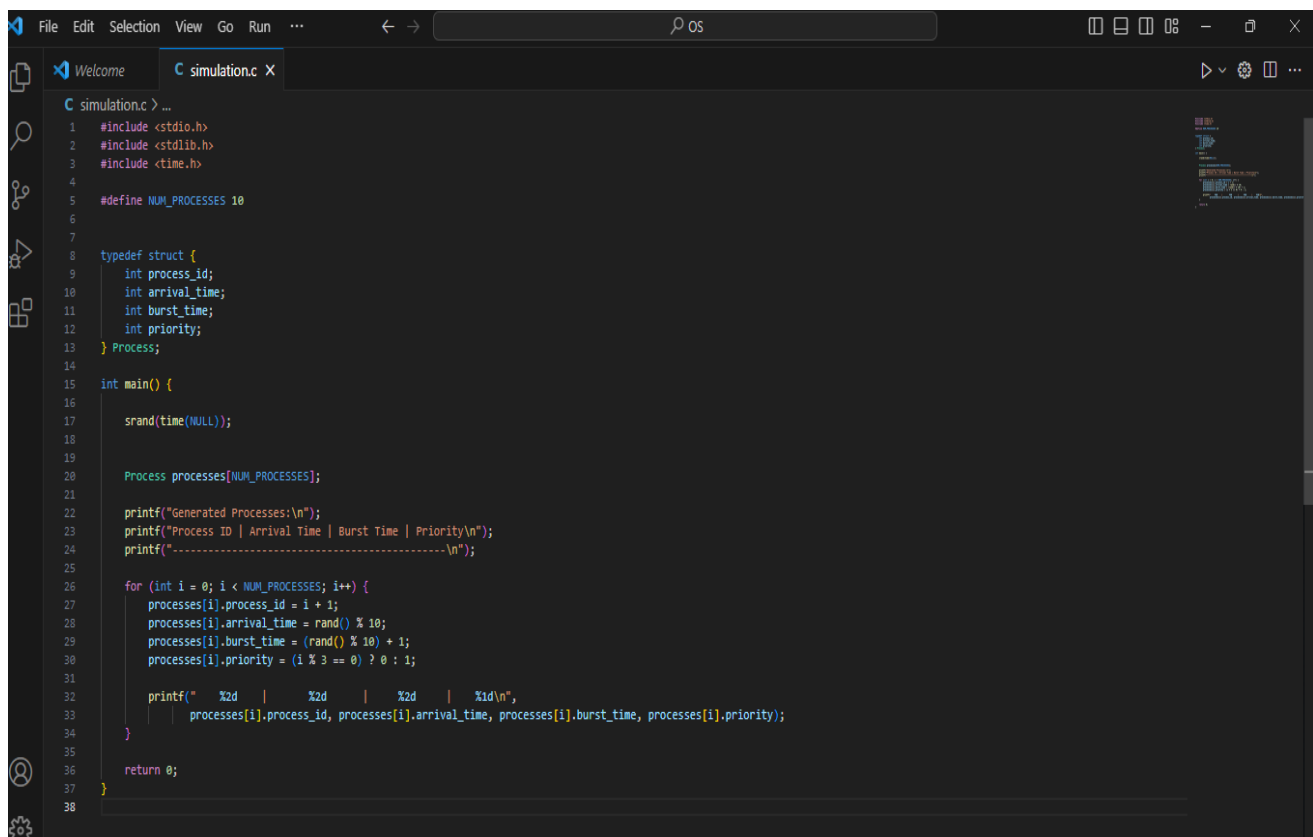
In this scenario, the system assigns higher priority to system-critical tasks and lower priority to user tasks. The scheduler schedules tasks based on their priority, ensuring that system-critical tasks are executed first, and user tasks are executed later. This ensures that system-critical tasks have 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.

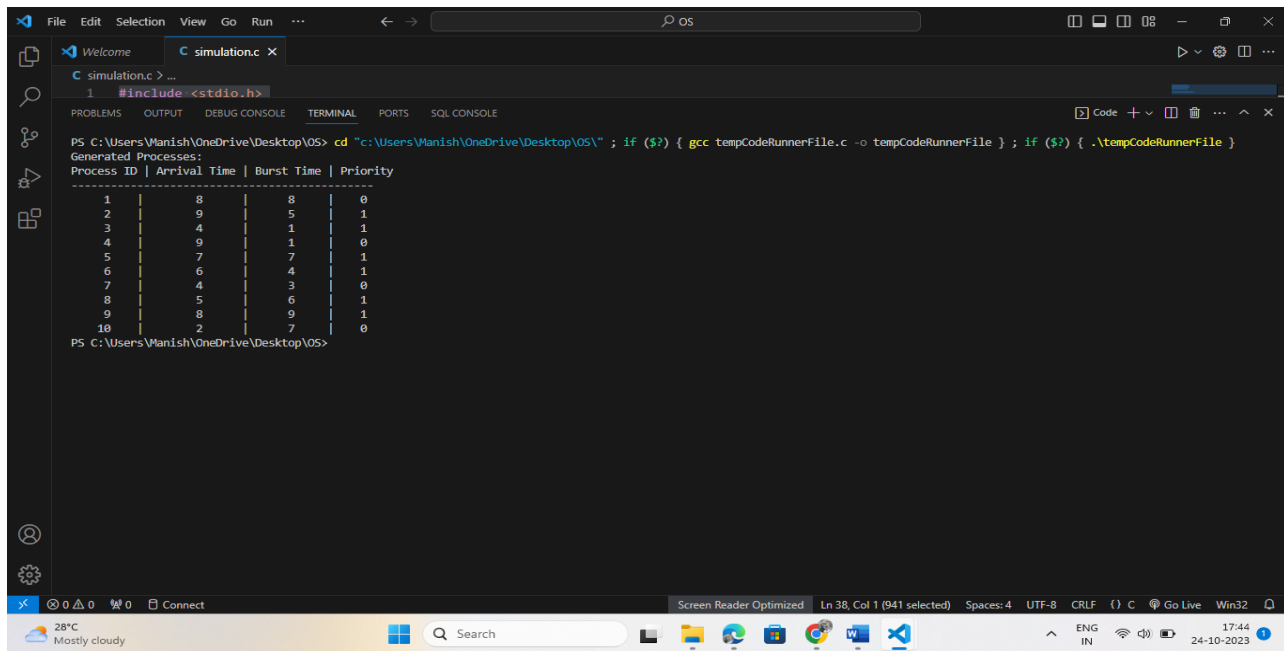
A) Generate a set of "processes" with random arrival times and CPU burst times using a random number generator.

CODE:

A screenshot of a code editor window with a dark theme. The window title is "simulation.c". The code is in C and defines a structure for a process, initializes an array of 10 processes, and generates random values for their arrival times, burst times, and priorities. The code is as follows:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  #define NUM_PROCESSES 10
6
7
8  typedef struct {
9      int process_id;
10     int arrival_time;
11     int burst_time;
12     int priority;
13 } Process;
14
15 int main() {
16
17     srand(time(NULL));
18
19     Process processes[NUM_PROCESSES];
20
21     printf("Generated Processes:\n");
22     printf("Process ID | Arrival Time | Burst Time | Priority\n");
23     printf("-----\n");
24
25     for (int i = 0; i < NUM_PROCESSES; i++) {
26         processes[i].process_id = i + 1;
27         processes[i].arrival_time = rand() % 10;
28         processes[i].burst_time = (rand() % 10) + 1;
29         processes[i].priority = (i % 3 == 0) ? 0 : 1;
30
31         printf("    %2d |      %2d |      %2d | %2d\n",
32             processes[i].process_id, processes[i].arrival_time, processes[i].burst_time, processes[i].priority);
33     }
34
35     return 0;
36 }
37
38
```

OUTPUT:



```
1 #include <stdio.h>

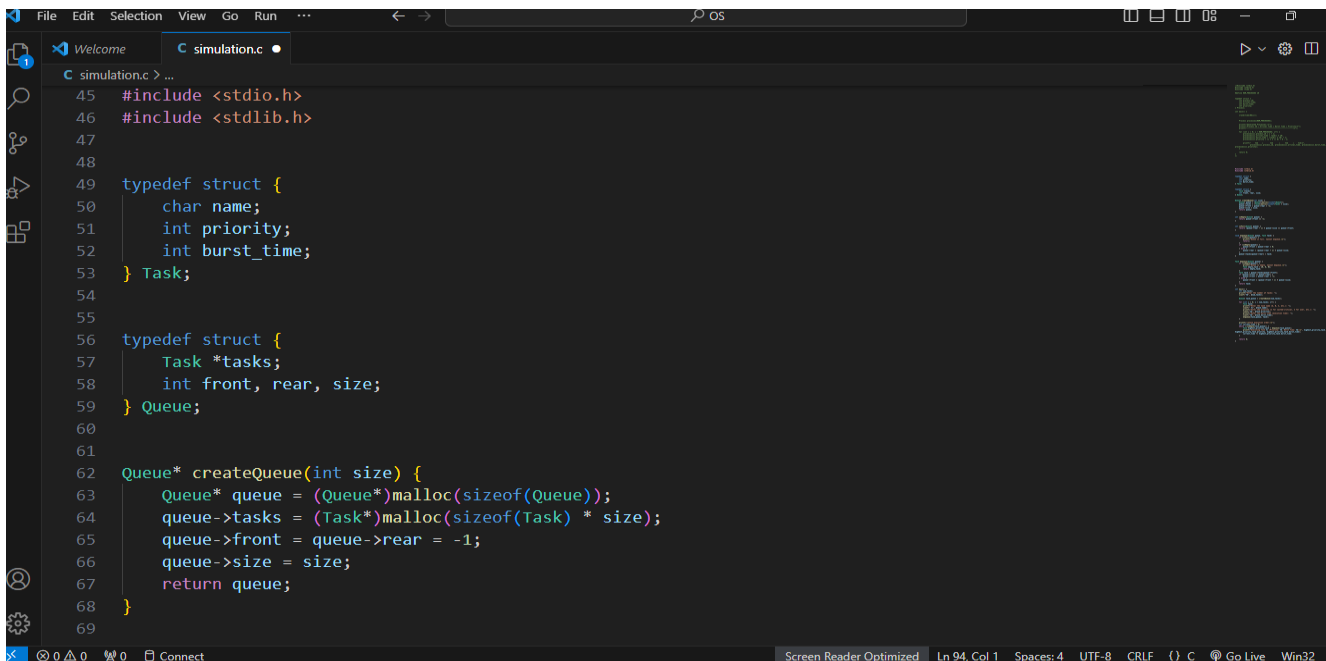
PS C:\Users\Manish\OneDrive\Desktop\OS> cd "c:\Users\Manish\OneDrive\Desktop\OS" ; if ($?) { gcc tempCodeRunnerFile.c -o tempCodeRunnerFile } ; if ($?) { .\tempCodeRunnerFile }

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:



```
45 #include <stdio.h>
46 #include <stdlib.h>
47
48
49 typedef struct {
50     char name;
51     int priority;
52     int burst_time;
53 } Task;
54
55
56 typedef struct {
57     Task *tasks;
58     int front, rear, size;
59 } Queue;
60
61
62 Queue* createQueue(int size) {
63     Queue* queue = (Queue*)malloc(sizeof(Queue));
64     queue->tasks = (Task*)malloc(sizeof(Task) * size);
65     queue->front = queue->rear = -1;
66     queue->size = size;
67     return queue;
68 }
69
```

```
File Edit Selection View Go Run ... OS
Welcome C simulation.c
simulation.c > ...
70
71 int isEmpty(Queue* queue) {
72     return queue->front == -1;
73 }
74
75
76 int isFull(Queue* queue) {
77     return (queue->rear + 1) % queue->size == queue->front;
78 }
79
80
81 void enqueue(Queue* queue, Task task) {
82     if (isFull(queue)) {
83         printf("Queue is full. Cannot enqueue.\n");
84         return;
85     }
86     if (isEmpty(queue)) {
87         queue->front = queue->rear = 0;
88     } else {
89         queue->rear = (queue->rear + 1) % queue->size;
90     }
91     queue->tasks[queue->rear] = task;
92 }
93
94 Task dequeue(Queue* queue) {
95     if (isEmpty(queue)) {
96         printf("Queue is empty. Cannot dequeue.\n");
97         Task empty_task = {0, 0, 0};
98         return empty_task;
99     }
100     Task task = queue->tasks[queue->front];
101     if (queue->front == queue->rear) {
102         queue->front = queue->rear = -1;
103     } else {
104         queue->front = (queue->front + 1) % queue->size;
105     }
106     return task;
107 }
108
109
110 int main() {
111     int num_tasks;
112     printf("Enter the number of tasks: ");
113     scanf("%d", &num_tasks);
114
115     Queue* task_queue = createQueue(num_tasks);
116
117     for (int i = 0; i < num_tasks; i++) {
118         Task task;
119         printf("Enter the task name (A, B, C, etc.): ");
```

```
File Edit Selection View Go Run ... OS
Welcome C simulation.c
simulation.c > ...
95 Task dequeue(Queue* queue) {
96     if (isEmpty(queue)) {
97         printf("Queue is empty. Cannot dequeue.\n");
98         Task empty_task = {0, 0, 0};
99         return empty_task;
100     }
101     Task task = queue->tasks[queue->front];
102     if (queue->front == queue->rear) {
103         queue->front = queue->rear = -1;
104     } else {
105         queue->front = (queue->front + 1) % queue->size;
106     }
107     return task;
108 }
109
110 int main() {
111     int num_tasks;
112     printf("Enter the number of tasks: ");
113     scanf("%d", &num_tasks);
114
115     Queue* task_queue = createQueue(num_tasks);
116
117     for (int i = 0; i < num_tasks; i++) {
118         Task task;
119         printf("Enter the task name (A, B, C, etc.): ");
```

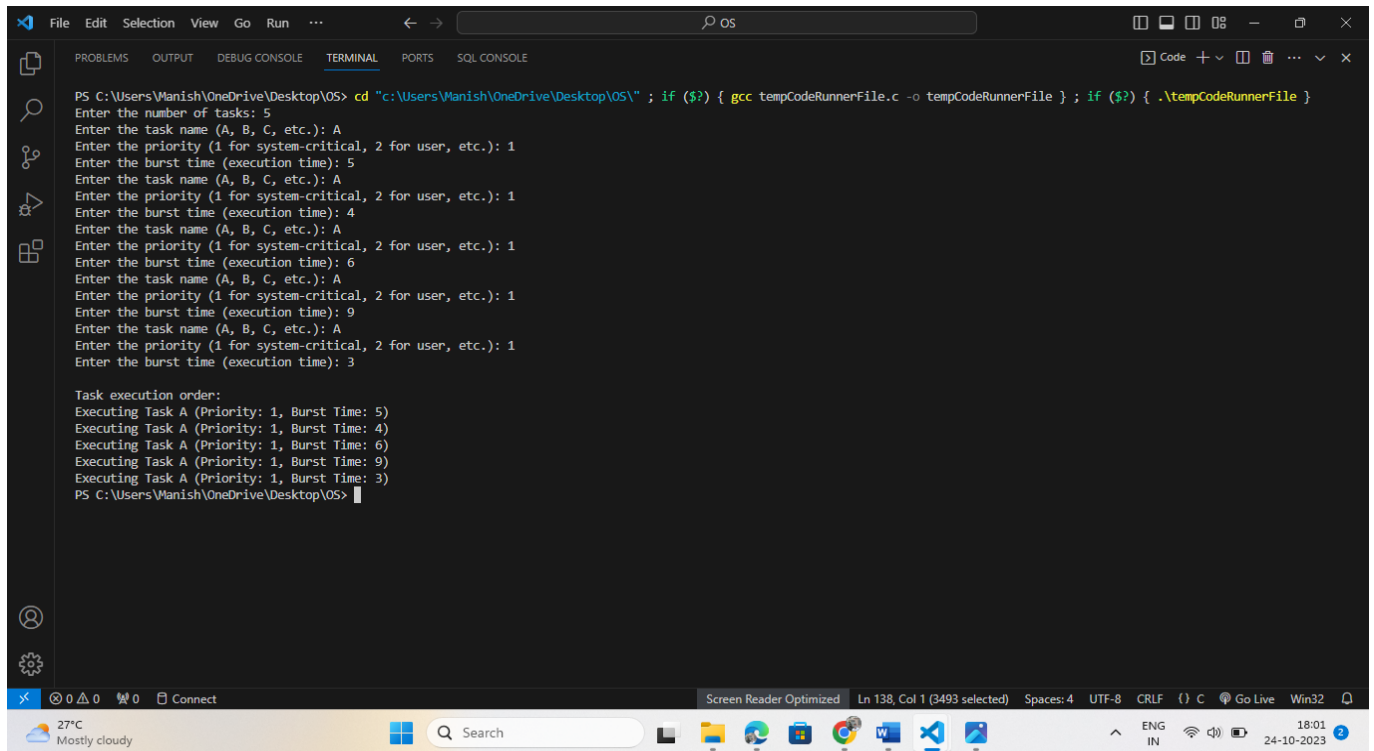
```
File Edit Selection View Go Run ... OS
C simulation.c
115 Queue* task_queue = createQueue(num_tasks);
116
117 for (int i = 0; i < num_tasks; i++) {
118     Task task;
119     printf("Enter the task name (A, B, C, etc.): ");
120     scanf("%c", &task.name);
121     printf("Enter the priority (1 for system-critical, 2 for user, etc.): ");
122     scanf("%d", &task.priority);
123     printf("Enter the burst time (execution time): ");
124     scanf("%d", &task.burst_time);
125     enqueue(task_queue, task);
126 }
127
128 printf("\nTask execution order:\n");
129 int current_time = 0;
130 while (!isEmpty(task_queue)) {
131     Task highest_priority_task = dequeue(task_queue);
132     printf("Executing Task %c (Priority: %d, Burst Time: %d)\n", highest_priority_task.name,
highest_priority_task.priority, highest_priority_task.burst_time);
133     current_time += highest_priority_task.burst_time;
134 }
135
136 return 0;
137 }
138
```

char <unnamed>::name

Screen Reader Optimized Ln 94, Col 1 Spaces: 4 UTF-8 CRLF {} C Go Live Win32

27°C Mostly cloudy Search 18:03 24-10-2023

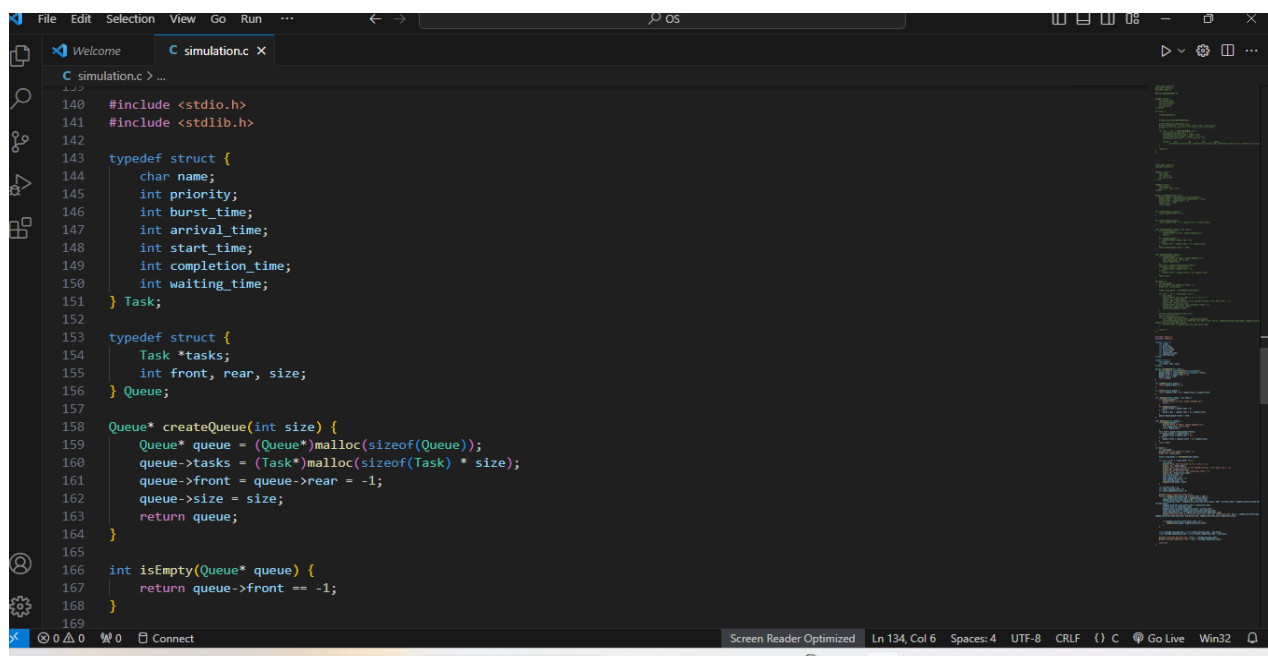
## OUTPUT:



```
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): 4
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
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): 3

Task execution order:
Executing Task A (Priority: 1, Burst Time: 5)
Executing Task A (Priority: 1, Burst Time: 4)
Executing Task A (Priority: 1, Burst Time: 6)
Executing Task A (Priority: 1, Burst Time: 9)
Executing Task A (Priority: 1, Burst Time: 3)
PS C:\Users\Manish\OneDrive\Desktop\OS>
```

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



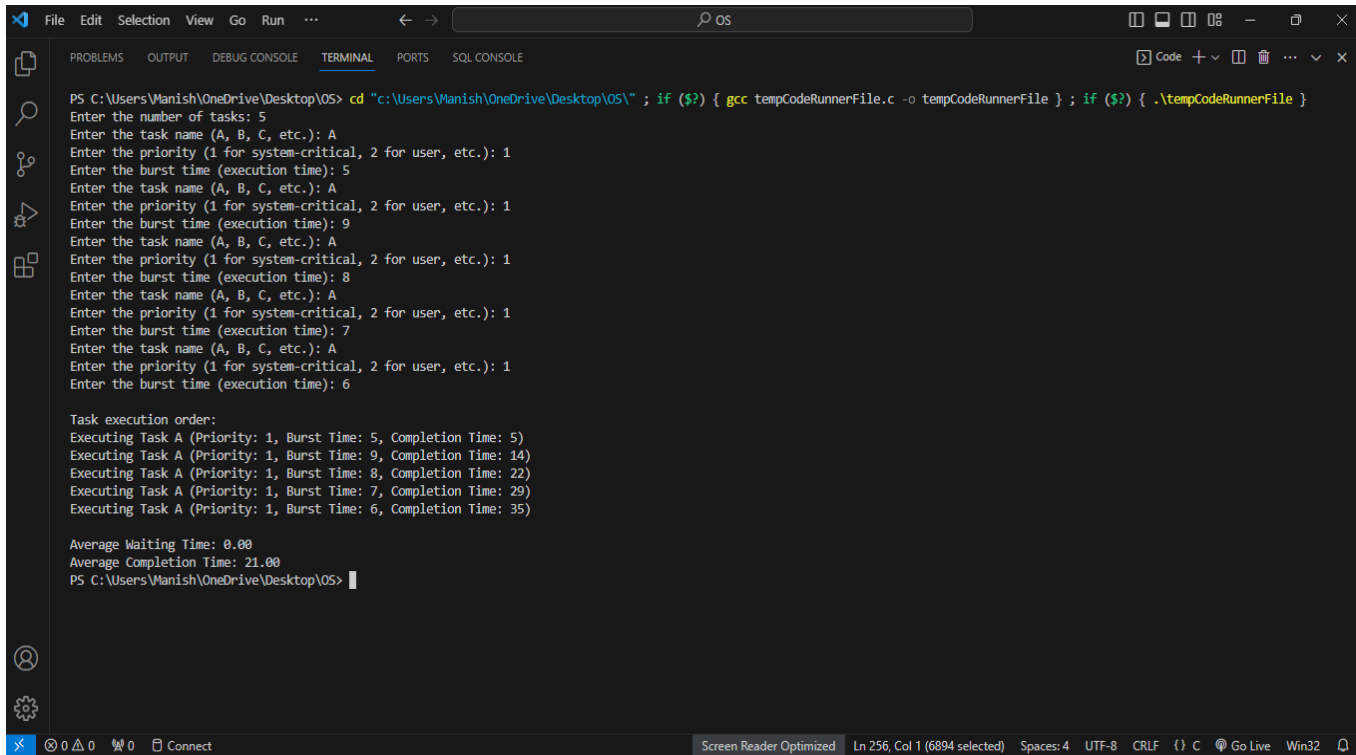
```
140 #include <stdio.h>
141 #include <stdlib.h>
142
143 typedef struct {
144     char name;
145     int priority;
146     int burst_time;
147     int arrival_time;
148     int start_time;
149     int completion_time;
150     int waiting_time;
151 } Task;
152
153 typedef struct {
154     Task *tasks;
155     int front, rear, size;
156 } Queue;
157
158 Queue* createQueue(int size) {
159     Queue* queue = (Queue*)malloc(sizeof(Queue));
160     queue->tasks = (Task*)malloc(sizeof(Task) * size);
161     queue->front = queue->rear = -1;
162     queue->size = size;
163     return queue;
164 }
165
166 int isEmpty(Queue* queue) {
167     return queue->front == -1;
168 }
```

```
166 int isEmpty(Queue* queue) {
167     return queue->front == -1;
168 }
169
170 int isFull(Queue* queue) {
171     return (queue->rear + 1) % queue->size == queue->front;
172 }
173
174 void enqueue(Queue* queue, Task task) {
175     if (isFull(queue)) {
176         printf("Queue is full. Cannot enqueue.\n");
177         return;
178     }
179     if (isEmpty(queue)) {
180         queue->front = queue->rear = 0;
181     } else {
182         queue->rear = (queue->rear + 1) % queue->size;
183     }
184     queue->tasks[queue->rear] = task;
185 }
186
187 Task dequeue(Queue* queue) {
188     if (isEmpty(queue)) {
189         printf("Queue is empty. Cannot dequeue.\n");
190         Task empty_task = {0, 0, 0};
191         return empty_task;
192     }
193     Task task = queue->tasks[queue->front];
194     if (queue->front == queue->rear) {
195         queue->front = queue->rear = -1;
196     }
197     return task;
198 }
```

```
192 }
193 Task task = queue->tasks[queue->front];
194 if (queue->front == queue->rear) {
195     queue->front = queue->rear = -1;
196 } else {
197     queue->front = (queue->front + 1) % queue->size;
198 }
199 return task;
200 }
201
202 int main() {
203     int num_tasks;
204     printf("Enter the number of tasks: ");
205     scanf("%d", &num_tasks);
206
207     Queue* task_queue = createQueue(num_tasks);
208
209     for (int i = 0; i < num_tasks; i++) {
210         Task task;
211         printf("Enter the task name (A, B, C, etc.): ");
212         scanf("%c", &task.name);
213         printf("Enter the priority (1 for system-critical, 2 for user, etc.): ");
214         scanf("%d", &task.priority);
215         printf("Enter the burst time (execution time): ");
216         scanf("%d", &task.burst_time);
217         task.arrival_time = 0;
218         task.start_time = -1;
219         Task task tion_time = -1;
220         task.waiting_time = 0;
221         enqueue(task_queue, task);
222     }
```



## 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.

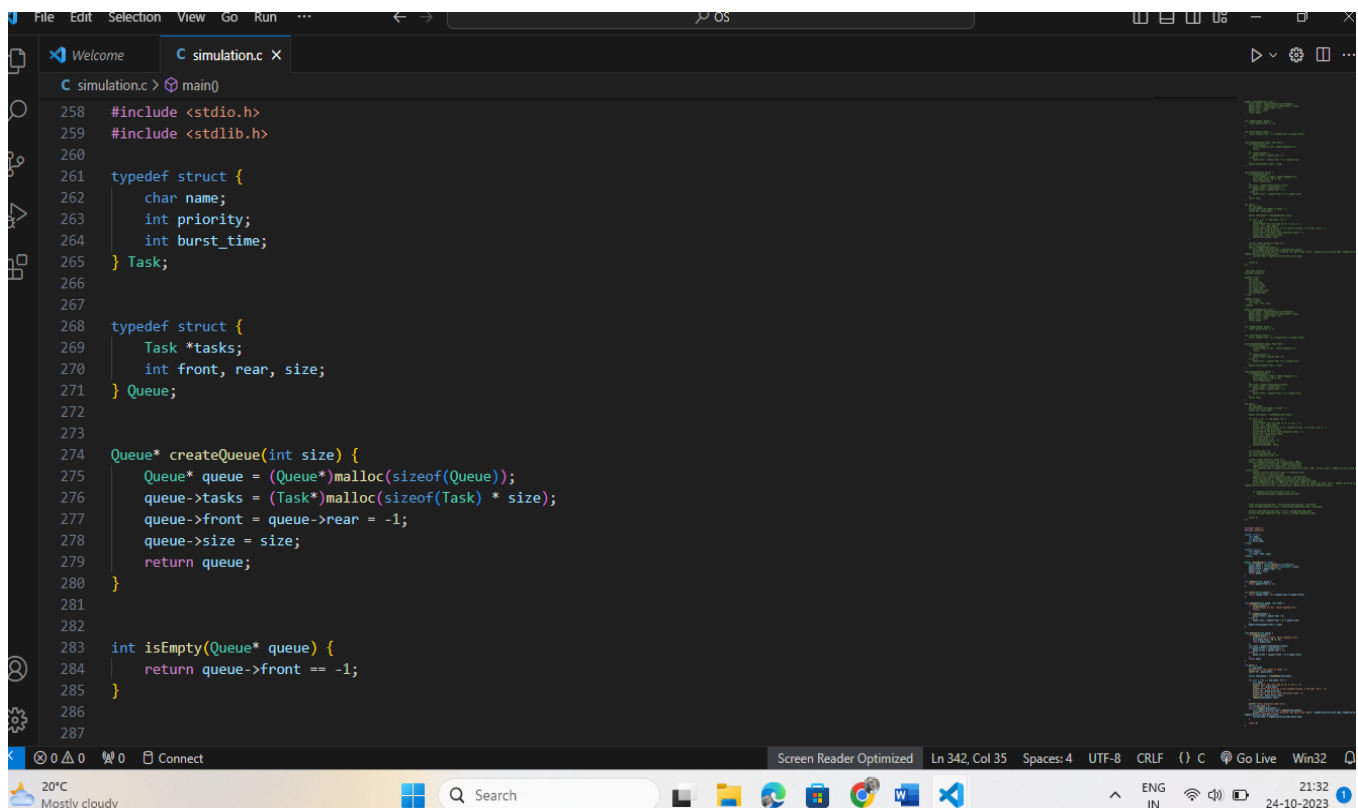
### **2. Fairness:**

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).



The screenshot displays a code editor window with a dark theme. The active file is 'simulation.c' and the cursor is at line 287. The code defines a 'Task' struct with fields for name, priority, and burst\_time, and a 'Queue' struct with an array of tasks and pointers for front, rear, and size. It includes functions for creating a queue and checking if it's empty. The Windows taskbar at the bottom shows the system clock as 21:32 on 24-10-2023, along with weather and search icons.

```
258 #include <stdio.h>
259 #include <stdlib.h>
260
261 typedef struct {
262     char name;
263     int priority;
264     int burst_time;
265 } Task;
266
267
268 typedef struct {
269     Task *tasks;
270     int front, rear, size;
271 } Queue;
272
273
274 Queue* createQueue(int size) {
275     Queue* queue = (Queue*)malloc(sizeof(Queue));
276     queue->tasks = (Task*)malloc(sizeof(Task) * size);
277     queue->front = queue->rear = -1;
278     queue->size = size;
279     return queue;
280 }
281
282
283 int isEmpty(Queue* queue) {
284     return queue->front == -1;
285 }
286
287
```

```
File Edit Selection View Go Run ...
C simulation.c X
C simulation.c > main()
283 int isEmpty(Queue* queue) {
284     return queue->front == -1;
285 }
286
287
288 int isFull(Queue* queue) {
289     return (queue->rear + 1) % queue->size == queue->front;
290 }
291
292
293 void enqueue(Queue* queue, Task task) {
294     if (isFull(queue)) {
295         printf("Queue is full. Cannot enqueue.\n");
296         return;
297     }
298     if (isEmpty(queue)) {
299         queue->front = queue->rear = 0;
300     } else {
301         queue->rear = (queue->rear + 1) % queue->size;
302     }
303     queue->tasks[queue->rear] = task;
304 }
305
306
307 Task dequeue(Queue* queue) {
308     if (isEmpty(queue)) {
309         printf("Queue is empty. Cannot dequeue.\n");
310         Task empty_task = {0, 0, 0};
311         return empty_task;
312     }
313     Task task = queue->tasks[queue->front];
```

```
File Edit Selection View Go Run ...
C simulation.c X
C simulation.c > main()
312 }
313 Task task = queue->tasks[queue->front];
314 if (queue->front == queue->rear) {
315     queue->front = queue->rear = -1;
316 } else {
317     queue->front = (queue->front + 1) % queue->size;
318 }
319 return task;
320 }
321
322 int main() {
323     int num_tasks;
324     printf("Enter the number of tasks: ");
325     scanf("%d", &num_tasks);
326
327     Queue* task_queue = createQueue(num_tasks);
328
329     for (int i = 0; i < num_tasks; i++) {
330         Task task;
331         printf("Enter the task name (A, B, C, etc.): ");
332         scanf("%c", &task.name);
333         printf("Enter the priority (1 for system-critical, 2 for user, etc.): ");
334         scanf("%d", &task.priority);
335         printf("Enter the burst time (execution time): ");
336         scanf("%d", &task.burst_time);
337         enqueue(task_queue, task);
338     }
339
340     printf("\nTask execution order:\n");
341     int current_time = 0;
342     while (!isEmpty(task_queue)) {
343         Task task = dequeue(task_queue);
344         if (task.priority == 1) {
345             printf("%c ", task.name);
346             current_time += task.burst_time;
347         } else {
348             printf("%c ", task.name);
349             current_time += task.burst_time;
350         }
351     }
352     printf("\n");
353 }
```

```
332     scanf("%c", &task.name);
333     printf("Enter the priority (1 for system-critical, 2 for user, etc.): ");
334     scanf("%d", &task.priority);
335     printf("Enter the burst time (execution time): ");
336     scanf("%d", &task.burst_time);
337     enqueue(task_queue, task);
338 }
339
340 printf("\nTask execution order:\n");
341 int current_time = 0;
342 while (!isEmpty(task_queue)) {
343     Task highest_priority_task = dequeue(task_queue);
344     printf("Executing Task %c (Priority: %d, Burst Time: %d)\n", highest_priority_task.name, highest_priority_task.priority,
highest_priority_task.burst_time);
345     current_time += highest_priority_task.burst_time;
346 }
347
348 return 0;
349 }
350
351
352
```

## 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.