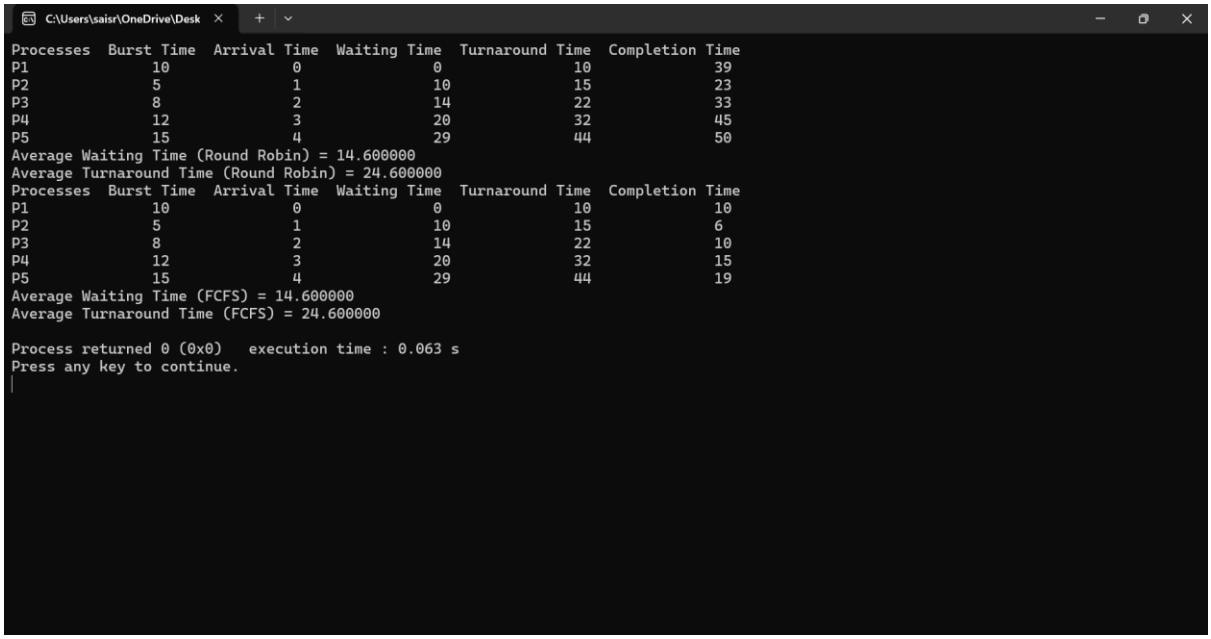
3. Write a C program to simulate multi-level queue scheduling algorithm considering the following scenario. All the processes in the system are divided into two categories – system processes and user processes. System processes are to be given higher priority than user processes. Use FCFS scheduling for the processes in each queue.

```
#include <stdio.h>

// Function to find waiting time for FCFS

void findWaitingTime(int processes[], int n, int bt[], int at[], int wt[]) {

   wt[0] = 0;

   for (int i = 1; i < n; i++) {

      wt[i] = bt[i-1] + wt[i-1] - at[i-1];

      if (wt[i] < 0)

         wt[i] = 0;

   }

}

// Function to find turnaround time

void findTurnaroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {

   for (int i = 0; i < n; i++) {

      tat[i] = bt[i] + wt[i];

   }

}

// Function to implement Round Robin scheduling

void roundRobin(int processes[], int n, int bt[], int at[], int quantum) {

   int wt[n], tat[n], ct[n], total_wt = 0, total_tat = 0;

   int remaining_bt[n];

   int completed = 0;

   int time = 0;

   for (int i = 0; i < n; i++) {

      remaining_bt[i] = bt[i];

   }

   while (completed < n) {

      for (int i = 0; i < n; i++) {

         if (remaining_bt[i] > 0 && at[i] <= time) {
```

```
            if (remaining_bt[i] <= quantum) {
                time += remaining_bt[i];
                remaining_bt[i] = 0;
                ct[i] = time;
                completed++;
            } else {
                time += quantum;
                remaining_bt[i] -= quantum;
            }
        }
    }
    findWaitingTime(processes, n, bt, at, wt);
    findTurnaroundTime(processes, n, bt, wt, tat);
    printf("Processes  Burst Time  Arrival Time  Waiting Time  Turnaround Time  Completion Time\n");
    for (int i = 0; i < n; i++) {
        printf("P%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i], bt[i], at[i], wt[i], tat[i], ct[i]);
        total_wt += wt[i];
        total_tat += tat[i];
    }
    printf("Average Waiting Time (Round Robin) = %f\n", (float)total_wt / n);
    printf("Average Turnaround Time (Round Robin) = %f\n", (float)total_tat / n);
}
// Function to implement FCFS scheduling
void fcfs(int processes[], int n, int bt[], int at[]) {
    int wt[n], tat[n], ct[n], total_wt = 0, total_tat = 0;
    findWaitingTime(processes, n, bt, at, wt);
    findTurnaroundTime(processes, n, bt, wt, tat);
    printf("Processes  Burst Time  Arrival Time  Waiting Time  Turnaround Time  Completion Time\n");
    for (int i = 0; i < n; i++) {
        ct[i] = at[i] + bt[i];
```

```
        printf("P%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i], bt[i], at[i], wt[i], tat[i], ct[i]);
        total_wt += wt[i];
        total_tat += tat[i];
    }
    printf("Average Waiting Time (FCFS) = %f\n", (float)total_wt / n);
    printf("Average Turnaround Time (FCFS) = %f\n", (float)total_tat / n);
}
int main() {
    int processes[] = {1, 2, 3, 4, 5};
    int n = sizeof(processes) / sizeof(processes[0]);
    int bt[] = {10, 5, 8, 12, 15};
    int at[] = {0, 1, 2, 3, 4};
    int quantum = 2;
    roundRobin(processes, n, bt, at, quantum);
    fcfs(processes, n, bt, at);
    return 0;
}
```

output:



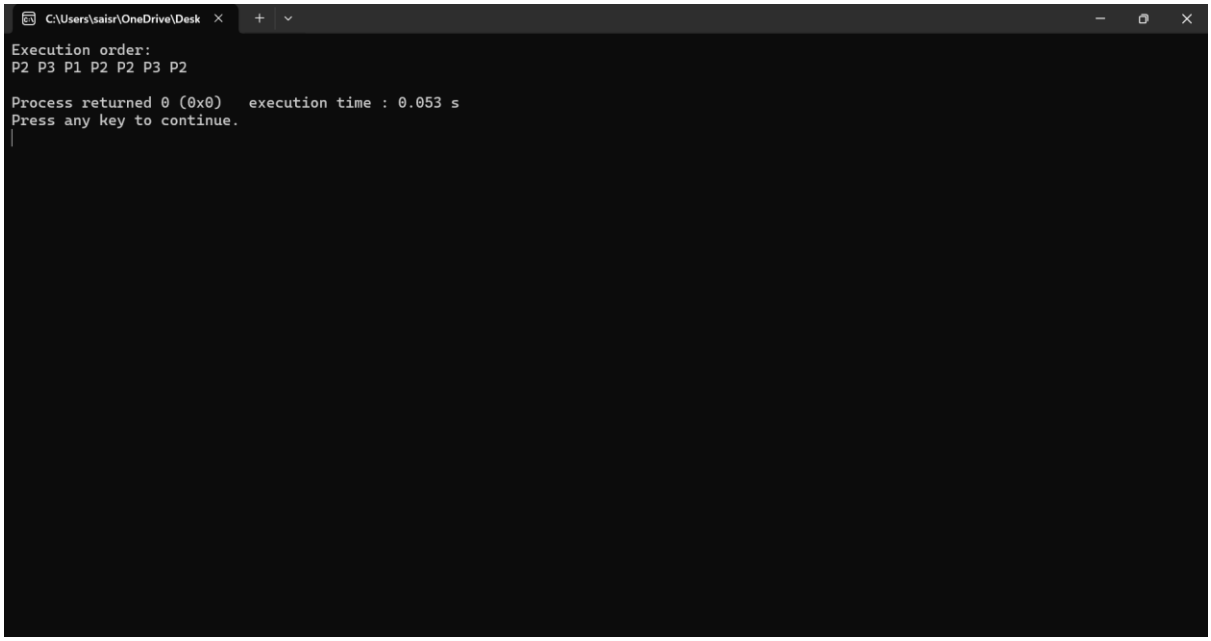
4. Write a C program to simulate Real-Time CPU Scheduling algorithms:

a) Rate- Monotonic

```
#include <stdio.h>
// Structure to represent a process
struct Process {
int execution_time;
int time_period;
};
// Function to calculate the least common multiple (LCM)
int lcm(int a, int b) {
int max = (a > b) ? a : b;
while (1) {
if (max % a == 0 && max % b == 0)
return max;
max++;
}
}
// Function to check if the set of processes is schedulable
int is_schedulable(struct Process processes[], int n) {
   float utilization = 0.0;
   for (int i = 0; i < n; i++) {
      utilization += (float)processes[i].execution_time / processes[i].time_period;
   }
   return utilization <= 1.0;
}
int main() {
   struct Process processes[] = {
      {3, 20}, // P1
      {2, 5},  // P2
      {2, 10}  // P3
   };
   int n = sizeof(processes) / sizeof(processes[0]);
```

```
    // Check if the processes are schedulable
    if (!is_schedulable(processes, n)) {
        printf("The given set of processes is not schedulable.\n");
        return 0;
    }
    // Calculate the scheduling time (LCM of time periods)
    int scheduling_time = lcm(processes[0].time_period, processes[1].time_period);
    scheduling_time = lcm(scheduling_time, processes[2].time_period);
    // Display the execution order
    printf("Execution order:\n");
    for (int t = 0; t < scheduling_time; t++) {
        if (t % processes[1].time_period == 0)
            printf("P2 ");
        if (t % processes[2].time_period == 0)
printf("P3 ");
if (t % processes[0].time_period == 0)
printf("P1 ");
}
printf("\n");
return 0;
}
```

output:



b) Earliest-deadline First

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
void
sort (int proc[], int d[], int b[], int pt[], int n)
{
  int temp = 0;
  for (int i = 0; i < n; i++)
          {
            for (int j = i; j < n; j++)
                    {
                      if (d[j] < d[i])
                              {
                                temp = d[j];
                                d[j] = d[i];
                                d[i] = temp;
                                temp = pt[i];
                                pt[i] = pt[j];
                                pt[j] = temp;
```

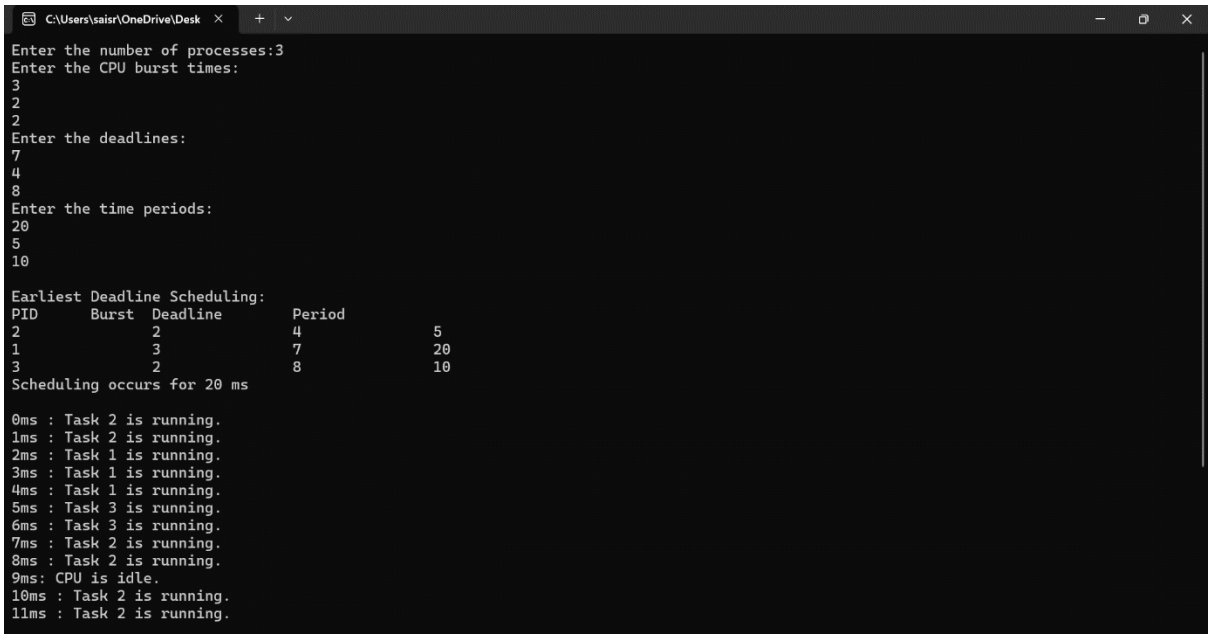
```
                                temp = b[j];
                                b[j] = b[i];
                                b[i] = temp;
                                temp = proc[i];
                                proc[i] = proc[j];
                                proc[j] = temp;
                               }
                        }
                }
}
int gcd (int a, int b)
{
 int r;
 while (b > 0)
                {
                 r = a % b;
                 a = b;
                 b = r;
                }
 return a;
}
int lcmul (int p[], int n)
{
 int lcm = p[0];
 for (int i = 1; i < n; i++)
                {
                 lcm = (lcm * p[i]) / gcd (lcm, p[i]);
                }
 return lcm;
}
```

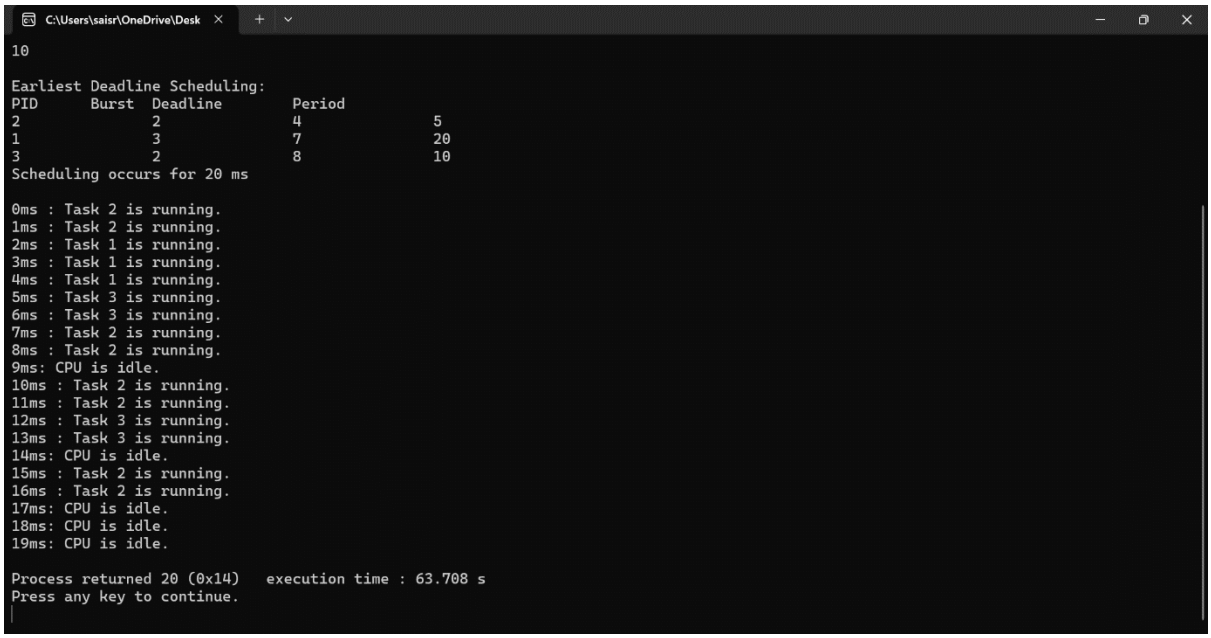
```
void main ()
{
  int n;
  printf ("Enter the number of processes:");
  scanf ("%d", &n);
  int proc[n], b[n], pt[n], d[n], rem[n];
  printf ("Enter the CPU burst times:\n");
  for (int i = 0; i < n; i++)
            {
              scanf ("%d", &b[i]);
              rem[i] = b[i];
            }
  printf ("Enter the deadlines:\n");
  for (int i = 0; i < n; i++)
            scanf ("%d", &d[i]);
  printf ("Enter the time periods:\n");
  for (int i = 0; i < n; i++)
            scanf ("%d", &pt[i]);
  for (int i = 0; i < n; i++)
            proc[i] = i + 1;

  sort (proc, d, b, pt, n);
  //LCM
  int l = lcmul (pt, n);
  printf ("\nEarliest Deadline Scheduling:\n");
  printf ("PID\t Burst\tDeadline\tPeriod\n");
  for (int i = 0; i < n; i++)
            printf ("%d\t\t%d\t\t%d\t\t%d\n", proc[i], b[i], d[i], pt[i]);
  printf ("Scheduling occurs for %d ms\n\n", l);
  //EDF
```

```
int time = 0, prev = 0, x = 0;
int nextDeadlines[n];
for (int i = 0; i < n; i++)
        {
         nextDeadlines[i] = d[i];
         rem[i] = b[i];
        }
while (time < l)
        {
         for (int i = 0; i < n; i++)
                {
                 if (time % pt[i] == 0 && time != 0)
                        {
                         nextDeadlines[i] = time + d[i];
                         rem[i] = b[i];
                        }
                }
         int minDeadline = l + 1;
         int taskToExecute = -1;
         for (int i = 0; i < n; i++)
                {
                 if (rem[i] > 0 && nextDeadlines[i] < minDeadline)
                        {
                         minDeadline = nextDeadlines[i];
                         taskToExecute = i;
                        }
                }
         if (taskToExecute != -1)
                {
                 printf ("%dms : Task %d is running.\n", time, proc[taskToExecute]);
                 rem[taskToExecute]--;
```

```
                }
            else
                {
                  printf ("%dms: CPU is idle.\n", time);
                }
            time++;
          }
}
```

output:





c) Proportional scheduling

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

#define MAX_TASKS 10
#define MAX_TICKETS 100
#define TIME_UNIT_DURATION_MS 100 // Duration of each time unit in milliseconds

struct Task {
   int tid;
   int tickets;
};
void schedule(struct Task tasks[], int num_tasks, int *time_span_ms) {
   int total_tickets = 0;

   for (int i = 0; i < num_tasks; i++) {
      total_tickets += tasks[i].tickets;
   }

   srand(time(NULL));

   int current_time = 0;
   int completed_tasks = 0;

      printf("Process Scheduling:\n");

   while (completed_tasks < num_tasks) {
      int winning_ticket = rand() % total_tickets;
      int cumulative_tickets = 0;

      for (int i = 0; i < num_tasks; i++) {
         cumulative_tickets += tasks[i].tickets;

         if (winning_ticket < cumulative_tickets) {
            printf("Time %d-%d: Task %d is running\n", current_time, current_time + 1, tasks[i].tid);
            current_time++;
            break;
         }
      }
      completed_tasks++;
   }

   // Calculate time span in milliseconds
   *time_span_ms = current_time * TIME_UNIT_DURATION_MS;
}

int main() {
   struct Task tasks[MAX_TASKS];
   int num_tasks;
   int time_span_ms;

   printf("Enter the number of tasks: ");
```

```
    scanf("%d", &num_tasks);

    if (num_tasks <= 0 || num_tasks > MAX_TASKS) {
        printf("Invalid number of tasks. Please enter a number between 1 and %d.\n", MAX_TASKS);
        return 1;
    }

    printf("Enter number of tickets for each task:\n");
    for (int i = 0; i < num_tasks; i++) {
        tasks[i].tid = i + 1;
        printf("Task %d tickets: ", tasks[i].tid);
        scanf("%d", &tasks[i].tickets);
    }

    printf("\nRunning tasks:\n");
    schedule(tasks, num_tasks, &time_span_ms);

    printf("\nTime span of the Gantt chart: %d milliseconds\n", time_span_ms);

    return 0;
}
```

output: