Output:

```
Enter the number of processes : 3
Enter the arrival time for Process 1 : 0
Enter burst time for Process 1 : 5
Enter priority for Process 1 : 3
Enter arrival time for Process 2 : 2
Enter burst time for Process 2 : 6
Enter priority for Process 2 : 1
Enter arrival time for Process 3 : 3
Enter burst time for Process 3 : 3
Enter priority for Process 3 : 2
Simulation Start :
Execution Timeline :
P1 will be executed from 0 to 5
P2 will be executed from 5 to 11
P3 will be execute from 11 to 14
```

Priority	Process	Arrival Time	Burst Time	Start Time	Waiting Time	Finish Time	Turnaround time
3	P_1	0	5	0	0	5	5
1	P_2	2	6	5	3	11	9
2	P_3	3	3	11	8	14	11

```
Average Waiting Time : 3.67
Average Turnaround Time : 8.33

Simulation End.
```

Experiment : 9

Date : 26/09/2023

Aim : To implement priority and round robin scheduling algo.

Priority Scheduling Algorithm

Program:

```
#include <stdio.h>
#include <stdlib.h>
struct Process{  // Define a structure to represent a process
    int id;   // Process ID
    int arrivalTime;  // Arrival time of the process
    int burstTime;  // Burst time (time needed for execution)
    int priority;  // Priority of the process
    int startTime;  // Start time of execution
    int finishTime;  // Finish time of execution
    int waitingTime;  // Waiting time in the queue
    int turnaroundTime;  // Turnaround time
    int remainingTime;  // Remaining time for execution
};
// Function to compare processes based on priority
int comparePriority (const void *a, const void *b){
    return ((struct Process *)a)->priority - ((struct Process *)b) ->
    priority;
}  // Function to compare processes based on ID
int compareID (const void *a, const void *b){
    return ((struct Process *)a)->id - ((struct Process *)b)->id;
}
// Function to execute priority scheduling
void executePriority (struct Process processes[], int numProcesses){
    int currentTime = 0;  // Initialize current time to 0
    int completedProcesses = 0;
    int totalWaitingTime = 0;
    int totalTurnaroundTime = 0;
    qsort (processes, numProcesses, sizeof(struct Process), comparePriority);
    printf ("Execution Timeline :\n");
    while (completedProcesses < numProcesses){
        int highestPriorityIndex = -1;
        int highestPriority = 9999;
        for (int i=0; i< numProcesses; i++){
            if (processes[i].arrivalTime <= currentTime &&
                processes[i].remainingTime >0 && processes[i].priority
                < highestPriority){
                highestPriority = processes[i].priority;
                highestPriorityIndex = i;
            }
    }
```

```
if (highestPriorityIndex == -1){
    currentTime++;
}else {
    struct Process *currentProcess = &processes[highestPriorityIndex];
    currentProcess->startTime = currentTime;
    int unitsExecuted = 0;
    while (unitsExecuted < currentProcess->remainingTime) {
        unitsExecuted++;
    }       currentTime++;
    currentProcess->remainingTime = 0;
    currentProcess->finishTime = currentTime;
    currentProcess->waitingTime = currentProcess->finishTime -
    currentProcess->arrivalTime - currentProcess->burstTime;
    currentProcess->turnaroundTime = currentProcess->finishTime -
                                     currentProcess->arrivalTime;
    completedProcesses++;
    printf("P%d will be executed from %d to %d.\n",
    currentProcess->id, currentProcess->startTime, currentProcess->finishTime);
    totalWaitingTime += currentProcess->waitingTime;
    totalTurnaroundTime += currentProcess->turnaroundTime;
}
}
// Sort processes by ID for displaying the final results
qsort(processes, numProcesses, sizeof(struct Process), compareID);
printf("Priority | Process | Arrival Time | Burst Time | Start Time |
        Waiting Time | Finish Time | Turnaround Time \n");
for (int i=0; i<numProcesses; i++) {
printf("%2d | P%2d | %2d | %2d | %2d | %2d | %2d | %2d\n",
processes[i].priority, processes[i].id, processes[i].arrivalTime,
processes[i].burstTime, processes[i].startTime, processes[i].waitingTime,
processes[i].finishTime, processes[i].turnaroundTime);
}

double avgWaitingTime = (double)totalWaitingTime / numProcesses;
double avgTurnaroundTime = (double)totalTurnaroundTime / numProcesses;
printf("\nAverage Waiting Time: %.2lf\n", avgWaitingTime);
printf("Average Turnaround Time: %.2lf\n", avgTurnaroundTime);
}
```

```
int main (){
    int numProcesses ;
    printf ("Enter the number of processes : ");
    scanf ("%d", & numProcesses) ;
    struct Process processes [numProcesses];
    for (int i=0; i < numProcesses ; i++) {
        processes[i]. id = i+1 ;
        printf ("Enter Arrival time for Process %d :", i+1);
        scanf ("%d", & processes[i]. arrivalTime );
        printf ("Enter burst time for Process %d:", i+1);
        scanf ("%d", & processes [i].burstTime);
        printf ("Enter priority for Process %d : ", i+1);
        scanf ("%d", & processes [i]. priority);
        processes[i]. remainingTime = processes[i]. burstTime ;
    }
    print f ("\n Simulation Start : \n \n ");
    execute Priority (processes , numProcesses);
    printf ("\n Simulation End. \n");
    return 0;
}
```

Output :

```
Enter the number of processes : 3
Enter the time quantum for Round Robin (in ms) : 2
Enter arrival time for Process 1 (in ms): 0
Enter burst time for Process 1 (in ms): 3
Enter arrival time for Process 2 (in ms): 1
Enter burst time for Process 2 (in ms): 2
Enter arrival time for Process 3 (in ms): 2
Enter burst time for Process 3 (in ms): 1
Simulation Start :
Executing Process 1 for 2 ms.
Executing Process 2 for 2 ms.
Executing Process 3 for 1 ms.
Executing Process 1 for 1 ms.
Simulation End.
```

Results :

Process	Arrival Time	Burst Time	Start Time	Waiting Time	Finish Time	Turn around Time
1	0	3	0	3	6	6
2	1	2	2	1	4	3
3	2	1	4	2	5	3

Round Robin Scheduling Algorithm

Program :

```
#include <stdio.h>
# include <stdlib.h>
struct Process {    // Define a structure to represent a process
    int id;         // Process ID
    int arrivalTime; // Arrival time of the process
    int burstTime; // Burst time required for execution
    int remainingTime; // Remaining time to complete the process
    int startTime; // Start time of execution
    int finishTime; // Finish time of execution
    int waitingTime; // finish time - arrival time - burst time
    int turnaroundTime; // finish time - arrival time
};
void executeRoundRobin(struct Process processes[], int numProcesses, int
                       timeQuantum){
    int currentTime = 0;
    int queueSize = numProcesses;
    while (queueSize > 0) {
        for (int i=0; i < numProcesses; i++){
            struct Process *currentProcess = &processes[i];
            if (currentProcess->arrivalTime <= currentTime &&
                currentProcess->remainingTime > 0){
                if (currentProcess->startTime == -1) {
                    currentProcess->startTime = currentTime;
                }
                int executionTime = (currentProcess->remainingTime < timeQuantum)
                ? currentProcess->remainingTime : timeQuantum;
                printf("Executing Process %d for %d ms.\n",
                currentProcess->id, executionTime);
                currentProcess->remainingTime -= executionTime;
                currentTime += executionTime;
                if (currentProcess->remainingTime == 0){
                    currentProcess->finishTime = currentTime;
                    currentProcess->turnaroundTime = currentProcess->
                    finishTime - currentProcess->arrivalTime;
                    currentProcess->waitingTime = currentProcess->
                    turnaroundTime - currentProcess->burstTime;
                    queueSize--;
                }
            }
        }
    }
}
```

```
int main(){
    int numProcesses, timeQuantum;
    printf("Enter the number of processes: ");
    scanf("%d", &numProcesses);
    printf("Enter the time quantum for Round Robin (in ms): ");
    scanf("%d", &timeQuantum);
    struct Process processes[numProcesses];
    for (int i=0; i<numProcesses; i++){
        processes[i].id = i+1;
        printf("Enter arrival time for Process %d (in ms): ", i+1);
        scanf("%d", &processes[i].arrivalTime);
        printf("Enter burst time for Process %d (in ms): ", i+1);
        scanf("%d", &processes[i].burstTime);
        processes[i].remainingTime = processes[i].burstTime;
        processes[i].startTime = -1;
        processes[i].finishTime = -1;
        processes[i].waitingTime = 0;
        processes[i].turnaroundTime = 0;
    }
    printf("\n Simulation Start :\n\n");
    executeRoundRobin(processes, numProcesses, timeQuantum);
    printf("\n Simulation End.\n");
    printf("Results :\n\n");
    for (int i=0; i<numProcesses; i++){
        printf("Process %d Arrival Time: %d ms Burst Time: %d ms -
        Start Time: %d ms Waiting Time: %d ms Finish Time: %d ms
        Turnaround Time: %d ms\n", processes[i].id, processes[i].
        arrivalTime, processes[i].burstTime, processes[i].startTime,
        processes[i].waitingTime, processes[i].finishTime, processes[i].turnaround
                                                            Time);
    }
    return 0;
}
```

Results :

Priority and Round Robin CPU scheduling algorithms are implemented.

(successfully).
