ASSIGNMENT – 1

CPU Scheduling

Name – Ashutosh Soni
Id – 2018ucp1505

Question: Simplified version of CPU Scheduling. (We shall be dealing with only one CPU burst). Write a C program (preferably for Linux gcc compiler) to simulate CPU scheduling. Following CPU scheduling mechanisms need to be implemented:

1. FCFS
2. SJF
3. Priority
4. MLFQ
5. Round Robin
6. Lottery (proportional share)

The process traces should be read from a file called "process.dat". Format for this file is as follows:

<number of processes> <pid> <arrival time> <priority> <share> <burst>
.....
<pid> <arrival time> <priority> <share> <burst>

So first line describes number of processes (say N).
Each of next N lines describe process id <pid>, time this process is put into ready queue <arrival time>. <priority> is the priority assigned to a process. A lower value means higher priority. <share> means number of tickets assigned in lottery (proportional share) scheduling.
We are assuming one CPU.
<burst> shall be an integer value between 1 - 20
For Round Robin scheduling, quantum shall be one.
For MLFQ scheduling, we shall assume that there are three queues - high, medium and low priority. Also process priority (except for those in high priority queue) is incremented every 10 cycles.
For priority queue, both preemptive and non-preemptive versions need to be implemented.
In lottery scheduling, it can be assumed that system is aware of total number of tickets (irrespective of process's arrival time).
Output should be process wise Gantt chart, CPU wise Gantt chart
And
Turnaround Time: process-wise, total and average
Waiting Time: process-wise, total and average
Response Time: process-wise, total and average

Answer:

1) FCFS

Program for FCFS CPU scheduling mechanism:

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

// Structure of process
typedef struct process{
        char name[5];
```

```
        int pid;
        int arrival_time;
        int priority;
        int share;
        int burst_time;
        int waiting_time;
        int response_time;
        int turn_around_time;
}processes;

// Sorting the process according to arrival time
void arrival_sort(processes temp[],int n){
        processes t;
        for(int i=0;i<n;i++){
                for(int j=0;j<n-i;j++){
                        if(temp[j].arrival_time >temp[j+1].arrival_time){
                                t=temp[j];
                                temp[j]=temp[j+1];
                                temp[j+1]=t;
                        }
                }
        }
}

// Print Table
void print_table(processes p[], int n)
{
   int i;

   printf("---------+-----+--------------+-----------+-------------+----------------+----------------+\n");
   printf("| Pname  | PID | Arrival Time  | Burst Time | Waiting Time | Turnaround Time | Response
Time  |\n");
   printf("---------+-----+--------------+-----------+-------------+----------------+----------------+\n");

   for(i=0; i<n; i++) {
      printf("|   %s   |  %d |      %d      |     %d    |      %d     |      %d       |      %d      |\n",

         p[i].name,p[i].pid,p[i].arrival_time,p[i].burst_time,p[i].waiting_time,p[i].turn_around_time,p[i].
response_time);
      printf("---------+-----+--------------+-----------+-------------+----------------+----------------+\n");
   }

}

// Printing gantt chart
void print_gantt_chart(processes p[], int n)
{
```

```
    int i, j;
    // print top bar
    printf(" ");
    for(i=0; i<n; i++) {
        for(j=0; j<p[i].burst_time; j++) printf("--");
        printf(" ");
    }
    printf("\n|");

    // printing process id in the middle
    for(i=0; i<n; i++) {
        for(j=0; j<p[i].burst_time - 1; j++) printf(" ");
        printf("%s", p[i].name);
        for(j=0; j<p[i].burst_time - 1; j++) printf(" ");
        printf("|");
    }
    printf("\n ");
    // printing bottom bar
    for(i=0; i<n; i++) {
        for(j=0; j<p[i].burst_time; j++) printf("--");
        printf(" ");
    }
    printf("\n");

    // printing the time line
    int val =0;
    printf("0");
    for(i=1; i<=n; i++) {
            val =val+ p[i-1].burst_time;
        for(j=0; j<p[i-1].burst_time; j++) printf("  ");
        // if(p[i].turn_around_time > 9) printf("\b"); // backspace : remove 1 space
        printf("%d", val);

    }
    printf("\n");

}

// Implementation of FCFS Aldorithm
void FIFS(processes P[],int n){
        processes temp[n];
        for(int i=0;i<n;i++){
                temp[i]=P[i];
        }
        // Sort according to arrival_time
        arrival_sort(temp,n);
```

```
	// Calculating waiting ,turnaround time and response time
	int sum_waiting=0, sum_turnaround=0,sum_response=0;
	sum_waiting=temp[0].waiting_time=0;
	temp[0].turn_around_time=temp[0].burst_time - temp[0].arrival_time;
	sum_turnaround = temp[0].turn_around_time;
	sum_response=sum_waiting;

	for(int i=1;i<n;i++){
		temp[i].waiting_time=(temp[i-1].burst_time + temp[i-1].arrival_time+temp[i-
1].waiting_time) - temp[i].arrival_time;
		temp[i].turn_around_time = (temp[i].waiting_time+temp[i].burst_time);
		temp[i].response_time= temp[i].waiting_time;
		sum_waiting += temp[i].waiting_time;
		sum_turnaround += temp[i].turn_around_time;
		sum_response =sum_waiting;
	}

	// Printing table
	printf("Table showing info about the processes: \n");
	print_table(temp,n);

	// Calculating Average waiting, Average turnaround and average response time
	float average_waiting_time, average_turnaround_time, average_response_time;
	average_waiting_time = (float)sum_waiting/n;
	average_turnaround_time = (float)sum_turnaround/n;
	average_response_time = (float)sum_response/n;

	printf("\n");
	printf("Average waiting time : %f\n",average_waiting_time );
	printf("Average turnaround time : %f\n",average_turnaround_time );
	printf("Average response time: %f\n",average_response_time );
	printf("\n");

	// Printing CPU Gantt chart
	printf("Gantt chart of CPU for these processes:  \n");
	print_gantt_chart(temp,n);
	printf("\n");

}

int main(int args,char *argv[]){
	FILE *fp = fopen("process.dat","r");
	if(fp==NULL){
		printf("No such file exists.. Unable to open the file....\n");
		exit(-1);
```

```
        }
        // Taking number of process thorugh file
        int n;
        fscanf(fp,"%d",&n);
        processes P[n];
        for(int i=0;i<n;i++){
                fscanf(fp,"%s",P[i].name);
                fscanf(fp,"%d",&P[i].pid);
                fscanf(fp,"%d",&P[i].arrival_time);
                fscanf(fp,"%d",&P[i].priority);
                fscanf(fp,"%d",&P[i].share);
                fscanf(fp,"%d",&P[i].burst_time);
                P[i].waiting_time = P[i].response_time = P[i].turn_around_time = 0;
        }

        // FIFS Algorithm
        FIFS(P,n);

        fclose(fp);
        return 0;
}
```

2) SJF

Program for SJF CPU scheduling mechanism:

```
// Non preemptive approach

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

// Structure of process
typedef struct process{
        char name[5];
        int pid;
        int arrival_time;
        int priority;
        int share;
        int burst_time;
        int waiting_time;
        int response_time;
        int turn_around_time;
}processes;

// Sorting the process according to arrival time
void arrival_sort(processes temp[],int n){
        processes t;
        for(int i=0;i<n;i++){
```

```
                for(int j=0;j<n-i;j++){
                        if(temp[j].arrival_time >temp[j+1].arrival_time){
                                t=temp[j];
                                temp[j]=temp[j+1];
                                temp[j+1]=t;
                        }
                }
        }
}

// Print Table
void print_table(processes p[], int n)
{
   int i;

   printf("---------+-----+---------------+------------+--------------+----------------+----------------+\n");
   printf("| Pname  | PID | Arrival Time  | Burst Time | Waiting Time | Turnaround Time | Response Time  |\n");
   printf("---------+-----+---------------+------------+--------------+----------------+----------------+\n");

   for(i=0; i<n; i++) {
      printf("|   %s   |  %d |      %d      |     %d     |      %d      |      %d        |      %d      |\n",

          p[i].name,p[i].pid,p[i].arrival_time,p[i].burst_time,p[i].waiting_time,p[i].turn_around_time,p[i].response
_time);
      printf("---------+-----+---------------+------------+--------------+----------------+----------------+\n");
   }

}

// Printing gantt chart
void print_gantt_chart(processes p[], int n)
{
   int i, j;
   // print top bar
   printf(" ");
   for(i=0; i<n; i++) {
      for(j=0; j<p[i].burst_time; j++) printf("--");
      printf(" ");
   }
   printf("\n|");

   // printing process id in the middle
   for(i=0; i<n; i++) {
      for(j=0; j<p[i].burst_time - 1; j++) printf(" ");
      printf("%s", p[i].name);
      for(j=0; j<p[i].burst_time - 1; j++) printf(" ");
      printf("|");
   }
   printf("\n ");
   // printing bottom bar
```

```
    for(i=0; i<n; i++) {
        for(j=0; j<p[i].burst_time; j++) printf("--");
        printf(" ");
    }
    printf("\n");

    // printing the time line
    int val =0;
    printf("0");
    for(i=1; i<=n; i++) {
            val =val+ p[i-1].burst_time;
        for(j=0; j<p[i-1].burst_time; j++) printf("  ");
        // if(p[i].turn_around_time > 9) printf("\b"); // backspace : remove 1 space
        printf("%d", val);

    }
    printf("\n");

}


// Algorithm of Short Job First
void SJF(processes P[],int n){
        processes temp[n];
        for(int i=0;i<n;i++){
                temp[i]=P[i];
        }
        // Sorting on the basis of arrival time
        arrival_sort(temp,n);
        for(int i=0;i<n;i++){
                printf("%s  ",temp[i].name );
        }
        printf("\n");

        // Sorting on the basis of minimum job
        processes t;
        for(int i=2;i<n;i++){
                for(int j=1;j<n-i+1;j++){
                        if(temp[j].burst_time > temp[j+1].burst_time){
                                t = temp[j];
                                temp[j] = temp[j+1];
                                temp[j+1] = t;
                        }
                }
        }

        for(int i=0;i<n;i++){
                printf("%s  ",temp[i].name );
        }
```

```
	int sum_waiting=0 , sum_turnaround=0, sum_response=0;

	//Calculating waiting, turnaround and response time
	sum_waiting = temp[0].waiting_time = 0;
	sum_turnaround = temp[0].turn_around_time = temp[0].burst_time - temp[0].arrival_time;
	sum_response = sum_waiting;

	for(int i=1;i<n;i++){
		temp[i].waiting_time = (temp[i-1].burst_time + temp[i-1].arrival_time+temp[i-1].waiting_time) -
temp[i].arrival_time;
		temp[i].turn_around_time = temp[i].waiting_time+ temp[i].burst_time;
		temp[i].response_time = temp[i].waiting_time;
		sum_waiting += temp[i].waiting_time;
		sum_turnaround +=temp[i].turn_around_time;
		sum_response=sum_waiting;
	}

	// Printing table
	printf("Table showing info about the processes: \n");
	print_table(temp,n);

	// Calculating Average waiting , Average turnaround and average response time
	float average_waiting_time, average_turnaround_time, average_response_time;
	average_waiting_time = (float)sum_waiting/n;
	average_turnaround_time = (float)sum_turnaround/n;
	average_response_time = (float)sum_response/n;

	printf("\n");
	printf("Average waiting time : %f\n",average_waiting_time );
	printf("Average turnaround time : %f\n",average_turnaround_time );
	printf("Average response time: %f\n",average_response_time );
	printf("\n");

	// Printing CPU Gantt chart
	printf("Gantt chart of CPU for these processes:  \n");
	print_gantt_chart(temp,n);
	printf("\n");
}


int main(int args,char *argv[]){
	FILE *fp = fopen("process.dat","r");
	if(fp==NULL){
		printf("No such file exists.. Unable to open the file....\n");
		exit(-1);
	}
	// Taking number of process thorugh file
	int n;
	fscanf(fp,"%d",&n);
	processes P[n];
```

```
        for(int i=0;i<n;i++){
                fscanf(fp,"%s",P[i].name);
                fscanf(fp,"%d",&P[i].pid);
                fscanf(fp,"%d",&P[i].arrival_time);
                fscanf(fp,"%d",&P[i].priority);
                fscanf(fp,"%d",&P[i].share);
                fscanf(fp,"%d",&P[i].burst_time);
                P[i].waiting_time = P[i].response_time = P[i].turn_around_time = 0;
        }
        fclose(fp);

        // SJF Algorithm
        SJF(P,n);
        return 0;
}
```

3) Priority

Program for Priority CPU scheduling mechanism:

```
// Priority Scheduling

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

// Structure of process
typedef struct process{
        char name[5];
        int pid;
        int arrival_time;
        int priority;
        int share;
        int burst_time;
        int waiting_time;
        int response_time;
        int turn_around_time;
        int flag;
}processes;

// Sorting the process according to arrival time
void arrival_sort(processes temp[],int n){
        processes t;
        for(int i=0;i<n;i++){
                for(int j=0;j<n-i;j++){
                        if(temp[j].arrival_time >temp[j+1].arrival_time){
                                t=temp[j];
                                temp[j]=temp[j+1];
```

```
                                    temp[j+1]=t;
                                }
                            }
                        }
}

// Print Table
void print_table(processes p[], int n)
{
   int i;

   printf("---------+-----+---------------+------------+--------------+--------------+-----------------+----------------+\n");
   printf("| Pname  | PID | Arrival Time  | Burst Time |    Priority  |  Waiting Time | Turnaround Time | Response
Time  |\n");
   printf("---------+-----+---------------+------------+--------------+--------------+-----------------+----------------+\n");

   for(i=0; i<n; i++) {
      printf("|   %s   |  %d |      %d      |     %d     |     %d      |      %d      |      %d        |      %d      |\n",

           p[i].name,p[i].pid,p[i].arrival_time,p[i].burst_time,p[i].priority,p[i].waiting_time,p[i].turn_around_time,
p[i].response_time);
      printf("---------+-----+---------------+------------+--------------+--------------+-----------------+----------------+\n");
   }

}

// Printing gantt chart
void print_gantt_chart(processes p[], int n)
{
   int i, j;
   // print top bar
   printf(" ");
   for(i=0; i<n; i++) {
      for(j=0; j<p[i].burst_time; j++) printf("--");
      printf(" ");
   }
   printf("\n|");

   // printing process id in the middle
   for(i=0; i<n; i++) {
      for(j=0; j<p[i].burst_time - 1; j++) printf(" ");
      printf("%s", p[i].name);
      for(j=0; j<p[i].burst_time - 1; j++) printf(" ");
      printf("|");
   }
   printf("\n ");
   // printing bottom bar
   for(i=0; i<n; i++) {
      for(j=0; j<p[i].burst_time; j++) printf("--");
      printf(" ");
   }
```

```
    printf("\n");

    // printing the time line
    int val =0;
    printf("0");
    for(i=1; i<=n; i++) {
            val =val+ p[i-1].burst_time;
       for(j=0; j<p[i-1].burst_time; j++) printf("  ");
       // if(p[i].turn_around_time > 9) printf("\b"); // backspace : remove 1 space
       printf("%d", val);

    }
    printf("\n");

}

// Priority Scheduling pre-emptive approach
void PRT_P(processes P[],int n){

        int t_total =0; // Total time of CPU burst
        processes temp[n];
        for(int i=0;i<n;i++){
                temp[i] = P[i];
                t_total += temp[i].burst_time;
                temp[i].response_time=-1;
        }
        // Sorting on the basis of arrival time
        arrival_sort(temp,n);

        int burst[n];
        for(int i=0;i<n;i++){
                burst[i]=temp[i].burst_time;
        }

        int tcurr=0;  // Current time
        int i=0,j=0;
        int min_pr; // Minimum priority
        int sum_waiting=0,sum_turnaround=0,sum_response=0;

        printf("\n Gantt chart of CPU for these processes: \n\n %d %s",i,temp[i].name);
        for(tcurr=0;tcurr<t_total;tcurr++)
        {

                if(burst[i] > 0 && temp[i].arrival_time <= tcurr)
                        burst[i]--;

                if(i!=j)
                        printf(" %d %s",tcurr,temp[i].name);

                if(burst[i]<=0 && temp[i].flag != 1)
```

```
                {
                        temp[i].flag = 1;
                        temp[i].waiting_time = (tcurr+1) - temp[i].burst_time - temp[i].arrival_time;
                        temp[i].turn_around_time = (tcurr+1) - temp[i].arrival_time;
                        sum_waiting+=temp[i].waiting_time;
                        sum_turnaround+=temp[i].turn_around_time;
                }
                if(temp[i].response_time == -1){
                        temp[i].response_time = tcurr- temp[i].arrival_time;
                        sum_response +=temp[i].response_time;
                }
                j=i;
                min_pr = 999;
                for(int x=0;x<n;x++){

                        if(temp[x].arrival_time <= (tcurr+1) && temp[x].flag != 1){

                                if(min_pr != temp[x].priority && min_pr > temp[x].priority){
                                        min_pr = temp[x].priority;
                                        i=x;
                                }
                        }
                }

        }
        printf(" %d",tcurr);
        printf("\n");
        printf("\n");
        printf("\n");
        printf("Table showing info about the processes: \n");
        print_table(temp,n);

        // Calculating Average waiting , Average turnaround and average response time
        float average_waiting_time, average_turnaround_time, average_response_time;
        average_waiting_time = (float)sum_waiting/n;
        average_turnaround_time = (float)sum_turnaround/n;
        average_response_time = (float)sum_response/n;

        printf("\n");
        printf("Average waiting time : %f\n",average_waiting_time );
        printf("Average turnaround time : %f\n",average_turnaround_time );
        printf("Average response time: %f\n",average_response_time );
        printf("\n");
}


// Priority Scheduling using non-pre-emptive approach

void PRT_NP(processes P[],int n){
        processes temp[n];
```

```
	for(int i=0;i<n;i++){
		temp[i]=P[i];
	}

	// Sorting on the basis of arrival time
	arrival_sort(P,n);

	// Now Sorting on the basis if priority
	processes t;
	for(int i=2;i<n;i++){
		for(int j=1;j<n-i+1;j++){
			if(temp[j].priority > temp[j+1].priority){
				t=temp[j];
				temp[j]=temp[j+1];
				temp[j+1]=t;
			}
		}
	}

	int sum_waiting=0, sum_turnaround=0, sum_response=0;
	sum_waiting = temp[0].waiting_time =0;
	sum_turnaround = temp[0].turn_around_time = temp[0].burst_time - temp[0].arrival_time;
	sum_response = sum_waiting;

	for(int i=1;i<n;i++){
		temp[i].waiting_time = (temp[i-1].burst_time + temp[i-1].arrival_time+temp[i-1].waiting_time) -
temp[i].arrival_time;
		temp[i].turn_around_time = temp[i].waiting_time+ temp[i].burst_time;
		temp[i].response_time = temp[i].waiting_time;
		sum_waiting += temp[i].waiting_time;
		sum_turnaround +=temp[i].turn_around_time;
		sum_response=sum_waiting;
	}

	printf("Table showing info about the processes: \n");
	print_table(temp,n);

	// Calculating Average waiting , Average turnaround and average response time
	float average_waiting_time, average_turnaround_time, average_response_time;
	average_waiting_time = (float)sum_waiting/n;
	average_turnaround_time = (float)sum_turnaround/n;
	average_response_time = (float)sum_response/n;

	printf("\n");
	printf("Average waiting time : %f\n",average_waiting_time );
	printf("Average turnaround time : %f\n",average_turnaround_time );
	printf("Average response time: %f\n",average_response_time );
	printf("\n");

	// Printing CPU Gantt chart
```

```
        printf("Gantt chart of CPU for these processes:  \n");
        print_gantt_chart(temp,n);
        printf("\n");

}


int main(int args,char *argv[]){
        FILE *fp = fopen("process.dat","r");
        if(fp==NULL){
                printf("No such file exists.. Unable to open the file....\n");
                exit(-1);
        }
        // Taking number of process thorugh file
        int n;
        fscanf(fp,"%d",&n);
        processes P[n];
        for(int i=0;i<n;i++){
                fscanf(fp,"%s",P[i].name);
                fscanf(fp,"%d",&P[i].pid);
                fscanf(fp,"%d",&P[i].arrival_time);
                fscanf(fp,"%d",&P[i].priority);
                fscanf(fp,"%d",&P[i].share);
                fscanf(fp,"%d",&P[i].burst_time);
                P[i].waiting_time = P[i].response_time = P[i].turn_around_time = 0;
        }
        fclose(fp);

        // SJF Algorithm
        printf("\nFrom pre-emptive Priority Scheduling: \n");
        printf("----------------------------------------\n");
        PRT_P(P,n);

        printf("From non-pre-emptive Priority Scheduling: \n");
        printf("\n");
        printf("----------------------------------------------\n");
        PRT_NP(P,n);
        return 0;
}
```

4) MLFQ

Program for MLFQ CPU scheduling mechanism:

```
// Non preemptive approach
```

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

// Structure of process
typedef struct process{
	char name[5];
	int pid;
	int arrival_time;
	int priority;
	int share;
	int burst_time;
	int waiting_time;
	int response_time;
	int turn_around_time;
	int flag;
}processes;

// Sorting the process according to arrival time
void arrival_sort(processes temp[],int n){
	processes t;
	for(int i=0;i<n;i++){
		for(int j=0;j<n-i;j++){
			if(temp[j].arrival_time >temp[j+1].arrival_time){
				t=temp[j];
				temp[j]=temp[j+1];
				temp[j+1]=t;
			}
		}
	}
}

// Print Table
void print_table(processes p[], int n)
{
   int i;

   printf("---------+-----+---------------+------------+--------------+----------------+----------------+\n");
   printf("| Pname  | PID | Arrival Time  | Burst Time | Waiting Time | Turnaround Time | Response Time  |\n");
   printf("---------+-----+---------------+------------+--------------+----------------+----------------+\n");

   for(i=0; i<n; i++) {
      printf("|   %s   |  %d |      %d      |     %d     |      %d      |      %d        |      %d       |\n",

	p[i].name,p[i].pid,p[i].arrival_time,p[i].burst_time,p[i].waiting_time,p[i].turn_around_time,p[i].response
_time);
      printf("---------+-----+---------------+------------+--------------+----------------+----------------+\n");
   }

}
```

```
// Printing gantt chart
void print_gantt_chart(processes p[], int n)
{
   int i, j;
   // print top bar
   printf(" ");
   for(i=0; i<n; i++) {
      for(j=0; j<p[i].burst_time; j++) printf("--");
      printf(" ");
   }
   printf("\n|");

   // printing process id in the middle
   for(i=0; i<n; i++) {
      for(j=0; j<p[i].burst_time - 1; j++) printf(" ");
      printf("%s", p[i].name);
      for(j=0; j<p[i].burst_time - 1; j++) printf(" ");
      printf("|");
   }
   printf("\n ");
   // printing bottom bar
   for(i=0; i<n; i++) {
      for(j=0; j<p[i].burst_time; j++) printf("--");
      printf(" ");
   }
   printf("\n");

   // printing the time line
   int val =0;
   printf("0");
   for(i=1; i<=n; i++) {
         val =val+ p[i-1].burst_time;
      for(j=0; j<p[i-1].burst_time; j++) printf("  ");
      // if(p[i].turn_around_time > 9) printf("\b"); // backspace : remove 1 space
      printf("%d", val);

   }
   printf("\n");

}

// Algorithm for Round Robin scheduling

void RR(processes P[],int n){
         processes temp1[n],temp2[n];
         for(int i=0;i<n;i++){
                  temp1[i]=P[i];
                  temp1[i].response_time =-1;
         }
```

```
// Sorting on the basis of arrival time
arrival_sort(temp1,n);

for(int i=0;i<n;i++){
	temp2[i] = temp1[i];
}

int quantum_time = 1;

int sum_waiting=0, sum_turnaround=0, sum_response=0;
printf("Quantum time as given it is 1 . \n\n");

// Printing Gantt chart and calculating others
printf("Gantt chart of CPU for these processes:  \n\n");
int t,tcurr=0,pflag=0;
for(int k=0; ; k++){
	if(k>n-1){
		k=0;
	}
	if(temp1[k].burst_time >0){
		printf("  %d  %s ",tcurr , temp1[k].name);
	}
	t=0;
	while(t<quantum_time && temp1[k].burst_time>0){
		t++;
		tcurr++;
		temp1[k].burst_time--;
	}
	if(temp1[k].burst_time <= 0 && temp1[k].flag !=1 ){
		temp1[k].waiting_time = tcurr - temp2[k].burst_time -temp1[k].arrival_time;
		temp1[k].turn_around_time = tcurr - temp1[k].arrival_time;
		pflag++;
		temp1[k].flag=1;
		sum_waiting += temp1[k].waiting_time;
		sum_turnaround += temp1[k].turn_around_time;
	}
	if(temp1[k].response_time == -1){
		temp1[k].response_time = tcurr -1 - temp1[k].arrival_time;
		sum_response +=temp1[k].response_time;
	}
	if(pflag==n){
		break;
	}
}
printf("  %d\n\n\n",tcurr );

printf("Table showing info about the processes: \n\n");
for(int i=0;i<n;i++){
	temp1[i].burst_time = temp2[i].burst_time;
}
```

```
	print_table(temp1,n);

	// Calculating Average waiting , Average turnaround and average response time
	float average_waiting_time, average_turnaround_time, average_response_time;
	average_waiting_time = (float)sum_waiting/n;
	average_turnaround_time = (float)sum_turnaround/n;
	average_response_time = (float)sum_response/n;

	printf("\n");
	printf("Average waiting time : %f\n",average_waiting_time );
	printf("Average turnaround time : %f\n",average_turnaround_time );
	printf("Average response time: %f\n",average_response_time );
	printf("\n");

}

// Algorithm for Multi Level Feedback Scheduling
// Lets suppose the process having
// quantum time less than or equal to 2 go to first queue
// quantum time less than or equal to 4 go to second queue
// rest go to third queue
void MLFQ(processes P[],int n){
	processes temp[n],temp1[n],temp2[n],temp3[n];
	for(int i=0;i<n;i++){
		temp[i] = P[i];
	}
	arrival_sort(temp,n);

	int tcurr=0;
	int size1=0,size2=0,size3=0;
	for(int i=0;i<n;i++){
		if(temp[i].burst_time <=2){
			temp1[size1]=temp[i];
			size1++;
		}
		else if(temp[i].burst_time <=4){
			temp2[size2]=temp[i];
			size2++;
		}
		else{
			temp3[size3]=temp[i];
			size3++;
		}
	}
	printf("size1 = %d  size2 = %d  size3 = %d\n",size1,size2,size3 );
	// RR(temp1,size1);
	// RR(temp2,size2);
	// RR(temp3,size3);

}
```

```
int main(int args,char *argv[]){
	FILE *fp = fopen("process.dat","r");
	if(fp==NULL){
		printf("No such file exists.. Unable to open the file....\n");
		exit(-1);
	}
	// Taking number of process thorugh file
	int n;
	fscanf(fp,"%d",&n);
	processes P[n];
	for(int i=0;i<n;i++){
		fscanf(fp,"%s",P[i].name);
		fscanf(fp,"%d",&P[i].pid);
		fscanf(fp,"%d",&P[i].arrival_time);
		fscanf(fp,"%d",&P[i].priority);
		fscanf(fp,"%d",&P[i].share);
		fscanf(fp,"%d",&P[i].burst_time);
		P[i].waiting_time = P[i].response_time = P[i].turn_around_time = 0;
	}
	fclose(fp);

	// SJF Algorithm
	printf("\n");
	MLFQ(P,n);
	return 0;
}
```

5) Round Robin

Program for Round Robin CPU scheduling mechanism:

```
// Non preemptive approach

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

// Structure of process
typedef struct process{
	char name[5];
	int pid;
	int arrival_time;
	int priority;
	int share;
	int burst_time;
	int waiting_time;
	int response_time;
	int turn_around_time;
```

```
        int flag;
}processes;

// Sorting the process according to arrival time
void arrival_sort(processes temp[],int n){
        processes t;
        for(int i=0;i<n;i++){
                for(int j=0;j<n-i;j++){
                        if(temp[j].arrival_time >temp[j+1].arrival_time){
                                t=temp[j];
                                temp[j]=temp[j+1];
                                temp[j+1]=t;
                        }
                }
        }
}

// Print Table
void print_table(processes p[], int n)
{
   int i;

   printf("---------+-----+---------------+------------+--------------+----------------+----------------+\n");
   printf("| Pname  | PID | Arrival Time  | Burst Time | Waiting Time | Turnaround Time | Response Time  |\n");
   printf("---------+-----+---------------+------------+--------------+----------------+----------------+\n");

   for(i=0; i<n; i++) {
      printf("|   %s   |  %d |      %d      |     %d     |      %d      |      %d        |      %d       |\n",

         p[i].name,p[i].pid,p[i].arrival_time,p[i].burst_time,p[i].waiting_time,p[i].turn_around_time,p[i].response
_time);
      printf("---------+-----+---------------+------------+--------------+----------------+----------------+\n");
   }

}

// Printing gantt chart
void print_gantt_chart(processes p[], int n)
{
   int i, j;
   // print top bar
   printf(" ");
   for(i=0; i<n; i++) {
      for(j=0; j<p[i].burst_time; j++) printf("--");
      printf(" ");
   }
   printf("\n|");

   // printing process id in the middle
   for(i=0; i<n; i++) {
```

```
        for(j=0; j<p[i].burst_time - 1; j++) printf(" ");
        printf("%s", p[i].name);
        for(j=0; j<p[i].burst_time - 1; j++) printf(" ");
        printf("|");
    }
    printf("\n ");
    // printing bottom bar
    for(i=0; i<n; i++) {
        for(j=0; j<p[i].burst_time; j++) printf("--");
        printf(" ");
    }
    printf("\n");

    // printing the time line
    int val =0;
    printf("0");
    for(i=1; i<=n; i++) {
            val =val+ p[i-1].burst_time;
        for(j=0; j<p[i-1].burst_time; j++) printf("  ");
        // if(p[i].turn_around_time > 9) printf("\b"); // backspace : remove 1 space
        printf("%d", val);

    }
    printf("\n");

}


// Algorithm for Round Robin scheduling

void RR(processes P[],int n){
            processes temp1[n],temp2[n];
            for(int i=0;i<n;i++){
                        temp1[i]=P[i];
                        temp1[i].response_time =-1;
            }
            // Sorting on the basis of arrival time
            arrival_sort(temp1,n);

            for(int i=0;i<n;i++){
                        temp2[i] = temp1[i];
            }

            int quantum_time = 1;

            int sum_waiting=0, sum_turnaround=0, sum_response=0;
            printf("Quantum time as given it is 1 . \n\n");

            // Printing Gantt chart and calculating others
            printf("Gantt chart of CPU for these processes:  \n\n");
```

```
int t,tcurr=0,pflag=0;
for(int k=0; ; k++){
        if(k>n-1){
                k=0;
        }
        if(temp1[k].burst_time >0){
                printf("  %d  %s ",tcurr , temp1[k].name);
        }
        t=0;
        while(t<quantum_time && temp1[k].burst_time>0){
                t++;
                tcurr++;
                temp1[k].burst_time--;
        }
        if(temp1[k].burst_time <= 0 && temp1[k].flag !=1 ){
                temp1[k].waiting_time = tcurr - temp2[k].burst_time -temp1[k].arrival_time;
                temp1[k].turn_around_time = tcurr - temp1[k].arrival_time;
                pflag++;
                temp1[k].flag=1;
                sum_waiting += temp1[k].waiting_time;
                sum_turnaround += temp1[k].turn_around_time;
        }
        if(temp1[k].response_time == -1){
                temp1[k].response_time = tcurr -1 - temp1[k].arrival_time;
                sum_response +=temp1[k].response_time;
        }
        if(pflag==n){
                break;
        }
}
printf("  %d\n\n\n",tcurr );

printf("Table showing info about the processes: \n\n");
for(int i=0;i<n;i++){
        temp1[i].burst_time = temp2[i].burst_time;
}
print_table(temp1,n);

// Calculating Average waiting , Average turnaround and average response time
float average_waiting_time, average_turnaround_time, average_response_time;
average_waiting_time = (float)sum_waiting/n;
average_turnaround_time = (float)sum_turnaround/n;
average_response_time = (float)sum_response/n;

printf("\n");
printf("Average waiting time : %f\n",average_waiting_time );
printf("Average turnaround time : %f\n",average_turnaround_time );
printf("Average response time: %f\n",average_response_time );
printf("\n");
```

```
}

int main(int args,char *argv[]){
	FILE *fp = fopen("process.dat","r");
	if(fp==NULL){
		printf("No such file exists.. Unable to open the file....\n");
		exit(-1);
	}
	// Taking number of process thorugh file
	int n;
	fscanf(fp,"%d",&n);
	processes P[n];
	for(int i=0;i<n;i++){
		fscanf(fp,"%s",P[i].name);
		fscanf(fp,"%d",&P[i].pid);
		fscanf(fp,"%d",&P[i].arrival_time);
		fscanf(fp,"%d",&P[i].priority);
		fscanf(fp,"%d",&P[i].share);
		fscanf(fp,"%d",&P[i].burst_time);
		P[i].waiting_time = P[i].response_time = P[i].turn_around_time = 0;
	}
	fclose(fp);

	// SJF Algorithm
	printf("\n");
	RR(P,n);
	return 0;
}
```

6) Lottery (Proportional Share)

Program for Lottery CPU scheduling mechanism:

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

// Structure of process
typedef struct process{
	char name[5];
	int pid;
	int arrival_time;
	int priority;
	int share;
	int burst_time;
	int waiting_time;
	int response_time;
	int turn_around_time;
	int flag;
```

```
        int ticket_start;
        int ticket_end;
}processes;

// Sorting the process according to arrival time
void arrival_sort(processes temp[],int n){
        processes t;
        for(int i=0;i<n;i++){
                for(int j=0;j<n-i;j++){
                        if(temp[j].arrival_time >temp[j+1].arrival_time){
                                t=temp[j];
                                temp[j]=temp[j+1];
                                temp[j+1]=t;
                        }
                }
        }
}

// Print Table
void print_table(processes p[], int n)
{
   int i;

   printf("---------+-----+---------------+------------+--------------+-----------------+----------------+\n");
   printf("| Pname  | PID | Arrival Time  | Burst Time | Waiting Time | Turnaround Time | Response
Time  |\n");
   printf("---------+-----+---------------+------------+--------------+-----------------+----------------+\n");

   for(i=0; i<n; i++) {
      printf("|   %s   |  %d |      %d      |     %d     |     %d     |      %d       |      %d      |\n",

         p[i].name,p[i].pid,p[i].arrival_time,p[i].burst_time,p[i].waiting_time,p[i].turn_around_time,p[i].
response_time);
      printf("---------+-----+---------------+------------+--------------+-----------------+----------------+\n");
   }

}


// Implementation of FCFS Aldorithm
void lottery(processes P[],int n){
        processes temp[n],temp1[n];
        int total_ticket=0;
        int start=0;
        for(int i=0;i<n;i++){
                temp[i]=P[i];
                total_ticket+=temp[i].burst_time;
```

```
        temp[i].response_time=-1;
}
total_ticket+=temp[n-1].burst_time;

// Sort according to arrival time
arrival_sort(temp,n);

// Now Sorting on the basis if priority
processes t;
for(int i=2;i<n;i++){
        for(int j=1;j<n-i+1;j++){
                if(temp[j].priority > temp[j+1].priority){
                        t=temp[j];
                        temp[j]=temp[j+1];
                        temp[j+1]=t;
                }
        }
}

for(int i=0;i<n;i++){
        temp[i].ticket_start=start;
        start+=temp[i].burst_time+1;
        temp[i].ticket_end=start-1;
        temp1[i]=temp[i];
}

int tcurr=0;
int sum_waiting=0, sum_turnaround=0, sum_response=0;
int count=0;

// Gantt Chart for lottery scheduling
printf("\nGantt chart of CPU for these processes:  \n");
printf("CPU taking time per second as follow\n\n");
for(int i=0;;i++){
        int random= rand()%(total_ticket);
        // printf("%d\n",random );
        for(int j=0;j<n;j++){
                if(random>temp1[j].ticket_start && random<temp1[j].ticket_end){
                        if(temp1[j].burst_time>0){
                                temp1[j].burst_time--;
                                if(temp[j].response_time==-1){
                                        temp[j].response_time=tcurr;
                                        sum_response+=temp[j].response_time;
                                }
                                tcurr++;
                                printf("%s ",temp[j].name );
```

```
                        }
                        else if(temp1[j].burst_time==0 && temp[j].flag!=1){
                                temp[j].flag=1;
                                temp[j].turn_around_time= tcurr - temp[j].arrival_time;
                                temp[j].waiting_time=temp[j].turn_around_time-
temp[j].burst_time;
                                sum_waiting+=temp[j].waiting_time;
                                sum_turnaround+=temp[j].turn_around_time;
                                count++;
                        }
                }
        }
        if(count==n){
                break;
        }
    }
    printf("\n");
    printf("\n");
    printf("Table showing info about the processes: \n\n");
    print_table(temp,n);
    printf("\n");
    printf("\n");

    // Calculating Average waiting , Average turnaround and average response time
    float average_waiting_time, average_turnaround_time, average_response_time;
    average_waiting_time = (float)sum_waiting/n;
    average_turnaround_time = (float)sum_turnaround/n;
    average_response_time = (float)sum_response/n;

    printf("\n");
    printf("Average waiting time : %f\n",average_waiting_time );
    printf("Average turnaround time : %f\n",average_turnaround_time );
    printf("Average response time: %f\n",average_response_time );
    printf("\n");


}

int main(int args,char *argv[]){
    FILE *fp = fopen("process.dat","r");
    if(fp==NULL){
        printf("No such file exists.. Unable to open the file....\n");
        exit(-1);
    }
    // Taking number of process thorugh file
    int n;
```

```
        fscanf(fp,"%d",&n);
        processes P[n];
        for(int i=0;i<n;i++){
                fscanf(fp,"%s",P[i].name);
                fscanf(fp,"%d",&P[i].pid);
                fscanf(fp,"%d",&P[i].arrival_time);
                fscanf(fp,"%d",&P[i].priority);
                fscanf(fp,"%d",&P[i].share);
                fscanf(fp,"%d",&P[i].burst_time);
                P[i].waiting_time = P[i].response_time = P[i].turn_around_time = 0;
        }

        // FIFS Algorithm
        lottery(P,n);

        fclose(fp);
        return 0;
}
```