```
#include<st
dio.h>
                #include<stdlib.h>

                int p_count; //total process count
                int q1_size=-1;
                int q2_size=-1;
                int q2_p_count=0;

                struct process{
                int pid;
                int arr_time;
                int bur_time;
                int priority;
                int wait_time;
                int turnaround_time;
                }*p1;

                int *burst; //temp burst time storage
                int *q1;
                int *q2;

                void insert1(int process_id)
                {
                q1[++q1_size]=process_id;
                }

                void insert2(int process_id)
                {
                q2[++q2_size]=process_id;
                }

                void delete1(int process_id)
                {
                int i;
                for(i=0;i<=q1_size;i++)
                {
                if(q1[i]==process_id)
                break;
                }
                for(int j=i;j<=q1_size-1;j++)
                {
                q1[j]=q1[j+1];
                }
                q1_size--;
                }

                void delete2()
                {
                int j;
                for(int j=0;j<=q2_size-1;j++)
                {
                q2[j]=q2[j+1];
                }
                q2_size--;
                }

                void wait(int process_id)
                {
                for(int i=0;i<=q1_size;i++)
```

```c
{
if(process_id!=q1[i])
{
p1[q1[i]-1].wait_time++; //incrementing wait time of processes in the queue.
}
}
for(int j=0;j<=q2_size;j++)
{
p1[q2[j]-1].wait_time++;
}
}

void wait2()
{
for(int i=0;i<=q2_size;i++)
{
p1[q2[i]-1].wait_time+=1; //incrementing wait time of processes in the queue.
}
}

int process_arrival(int time)
{
int *id=(int*)malloc(sizeof(int)*p_count); //array to store processes arriving at same time
int pos=-1;
for(int i=0;i<p_count;i++)
{
if(p1[i].arr_time==time)
{
id[++pos]=p1[i].pid;
insert1(id[pos]);
}
}
if(pos==-1)
{ //no process
return 0;
}
else if(pos==0) //only one process
{
return id[pos];
}
else //more than one process at given time
{
int max_p=id[0];
for(int i=1;i<=pos;i++)
{
if(p1[id[i]-1].priority<p1[max_p-1].priority) //checking priority
{
max_p=id[i];
}
}
return max_p;
}
}

int process_arrival2(int time)
{
int i=0;
int id=0;
for(;i<p_count;i++)
```

```
{
if(time==p1[i].arr_time)
{
id=p1[i].pid;
}
}
return id;
}

int allocate_p()
{
if(q1_size==0) //if only one process in queue
return q1[0];
else
{
int max_p=q1[0];
for(int i=1;i<=q1_size;i++)
{
if(p1[q1[i]-1].priority<p1[max_p-1].priority)
{
max_p=q1[i];
}
else if(p1[q1[i]-1].priority==p1[max_p-1].priority) //checking arrival time of equal priority
processes
{
if(p1[q1[i]-1].arr_time<p1[max_p-1].arr_time)
{
max_p=q1[i];
}
}
}
return max_p;
}
}

int round_robin(int time,int flag)
{
int time_q=4;
int counter=0;
int run_id=0;
static int finish_process=0;
while(finish_process!=q2_p_count)
{
if(flag==1)
{
int new_id=process_arrival2(time);
if(new_id!=0)
{
if(run_id!=0)
{
insert2(run_id);
}
return time;
}
}
if(run_id==0)
{
run_id=q2[0];
delete2();
}
time+=1;
burst[run_id-1]-=1;
wait2();
counter+=1;
if(burst[run_id-1]==0)
{
finish_process++;
run_id=0;
counter=0;
}
else if(counter==time_q)
{
insert1(run_id);
counter=0;
run_id=0;
}
}
return time;
}

int preemptive()
{
int temp_process_count=p_count;
int finished_process=0;
int time=0;
int run_id=0;
int new_id;
while(finished_process!=temp_process_count)
{
new_id=process_arrival(time); //process arriving at current time
if(new_id!=0)
{
if(run_id!=0)
{
if(p1[new_id-1].priority<p1[run_id-1].priority) //checking priority
{
delete1(run_id);
insert2(run_id); //swap into queue2
q2_p_count++;
run_id=new_id;
temp_process_count--;
}
}
else
{
run_id=new_id; //first running process or no process was running before
}
}
else
{
if(q2_size>-1 &&q1_size==-1)
{
time=round_robin(time,1);
continue;
}
}
```

```
if(q1_size>-1) //if queue1 not empty
{
wait(run_id);
burst[run_id-1]=burst[run_id-1]-1;
if(burst[run_id-1]==0) //if a process is finished
{
delete1(run_id);
finished_process++;
run_id=0; //resetting running process id to NULL
if(q1_size!=-1)
{
run_id=allocate_p(); //select new process from queue1 if not empty
}
}

}
time++;
}
return time;
}

void display()
{
int time=preemptive();
float avg_wait_time,avg_turnaround_time;
int wait_s=0,turn_s=0;
if(q2_size!=-1)
time=round_robin(time,0);
printf("\nPID\tPriority\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time");
for(int i=0;i<p_count;i++)
{
p1[i].turnaround_time=p1[i].bur_time+p1[i].wait_time;
printf("\n%-8d%-16d%-16d%-16d%-16d%d",p1[i].pid,p1[i].priority,p1[i].arr_time,p1[i].bur_time
,p1[i].wait_time,p1[i].turnaround_time);
}
for(int i=0;i<p_count;i++)
{
wait_s+=p1[i].wait_time;
turn_s+=p1[i].turnaround_time;
}
avg_wait_time=wait_s*1.0/p_count;
avg_turnaround_time=turn_s*1.0/p_count;
printf("\nAverage Waiting time:%.3f",avg_wait_time);
printf("\nAverage Turnaround time:%.3f",avg_turnaround_time);
printf("\n");
}
int main()
{
int temp;
while(1)
{
printf("Enter number of Processes:");
scanf("%d",&temp);
if(temp>0)
{
p_count=temp;
break;
}
```

```
else
{
printf("Error:Only enter positive number greater than 0.\n\n");
}
}
p1=(struct process*)malloc(sizeof(struct process)*p_count);
q1=(int*)malloc(sizeof(int)*p_count);
q2=(int*)malloc(sizeof(int)*(p_count-1));
burst=(int*)malloc(sizeof(int)*p_count);
printf("\n\t\tDetails of Processes\n");
for(int i=0;i<p_count;i++)
{
p1[i].pid=i+1;
while(1)
{
printf("\nEnter Arrival time of Process %d:",i+1);
scanf("%d",&temp);
if(temp>=0)
{
p1[i].arr_time=temp;
break;
}
else
{
printf("Error:Only enter positive numbers.\n");
}
}
while(1)
{
printf("Enter Burst time of Process %d:",i+1);
scanf("%d",&temp);
if(temp>0 && temp%2==0)
{
p1[i].bur_time=temp;
break;
}
else
{
printf("Error:Only enter positive numbers more than 0 which are multiples of 2.\n\n");
}
}
burst[i]=p1[i].bur_time;
while(1)
{
printf("Enter Priority of Process %d:",i+1);
scanf("%d",&temp);
if(temp>=0)
{
p1[i].priority=temp;
break;
}
else
{
printf("Error:Only enter positive numbers.\n\n");
}
}
p1[i].wait_time=0;
}
```

```c
display();
free(p1);
free(q1);
free(burst);
free(q2);
}
```