**Student Name:** Manoj Suthar

**Student ID :** 11803111

**Email Address:** [manojsuthar2000@gmail.com](mailto:manojsuthar2000@gmail.com)

**GitHub Link:**

<https://github.com/Manojsuthar2000/Multilevel-Queue-Scheduling-Algorithm>

**Code:**

#include<stdio.h>

#include<stdlib.h>

#include<stdbool.h>

#define Quantum 2

struct process

{

int at,copy\_bt,bt,pr,pno,st,ct,position;

bool flag;

};

int comp(const void \*a,const void \*b)

{

struct process \*ia=(struct process \*)a;

struct process \*ib=(struct process \*)b;

if(ia->at == ib->at)

{

return ia->pr-ib->pr;

}

else

{

return ia->at-ib->at;

}

}

int pr\_comp(const void \*a,const void \*b)

{

struct process \*ia=(struct process \*)a;

struct process \*ib=(struct process \*)b;

if(ia->pr == ib->pr)

{

return ia->at-ib->at;

}

else

{

return ia->pr-ib->pr;

}

}

int comp2(const void \*a,const void \*b)

{

struct process \*ia=(struct process \*)a;

struct process \*ib=(struct process \*)b;

return ia->position-ib->position;

}

void priority\_st\_ct\_time(struct process Q[],int n,int \*t)

{

int pos=0;

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

{

Q[i].flag=false;

}

\*t=Q[0].at;

Q[0].st=\*t;

int j=1,k=0,count=0;

for((\*t)++;\*t<=Q[n-1].at;(\*t)++)

{

Q[k].bt--;

count++;

if(\*t==Q[j].at)

{

if(Q[j].pr<Q[k].pr)

{

count=0;

Q[k].flag=true;

Q[k].position=pos++;

k=j;

Q[k].st=\*t;

}

j++;

}

if(count==Quantum || Q[k].bt<=0)

{

if(Q[k].bt==0)

{

Q[k].ct=\*t-1;

}

count=0;

Q[k].flag=true;

Q[k].position=pos++;

int min=Q[k+1].pr,b=k+1;

for(int a=k+2;a<j;a++)

{

if(Q[a].pr<min){

min=Q[a].pr;

b=a;

}

}

k=b;

Q[k].st=\*t;

j=k+1;

}

}

qsort(Q,n,sizeof(struct process),pr\_comp);

k=0;

int count1=0;

for(;count1<n;(\*t)++)

{

if(Q[k].flag)

{

k++;

(\*t)--;

count1++;

continue;

}

Q[k].bt--;

count++;

if(count==Quantum || Q[k].bt<=0)

{

count=0;count1++;

if(Q[k].bt==0)

{

Q[k].ct=\*t-1;

Q[k].st=\*t-Q[k].copy\_bt;

}

else

{

Q[k].st=\*t-Quantum;

}

Q[k].flag=true;

Q[k].position=pos++;

k++;

}

}

}

void round\_robin\_st\_ct\_time(struct process Q[],int n,int \*t)

{

qsort(Q,n,sizeof(struct process),comp2);

while (1)

{

bool done = true;

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

{

if (Q[i].bt > 0)

{

done = false;

if (Q[i].bt > Quantum)

{

\*t += Quantum;

Q[i].bt -= Quantum;

}

else

{

\*t+=Q[i].bt;

Q[i].ct=\*t-1;

Q[i].bt = 0;

}

}

}

if (done == true)

break;

}

}

void get\_wt\_time(struct process Q[],int tat[],int wt[],int n)

{

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

wt[i]=tat[i]-Q[i].copy\_bt;

}

void get\_tat\_time(struct process Q[],int tat[],int n)

{

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

tat[i]=Q[i].ct-Q[i].at;

}

void findgc(struct process Q[],int n)

{

int wt[n],tat[n],t=0;

double wavg=0,tavg=0;

//Fixed priority preemptive scheduling

priority\_st\_ct\_time(Q,n,&t);

//round robin scheduling

round\_robin\_st\_ct\_time(Q,n,&t);

get\_tat\_time(Q,tat,n);

get\_wt\_time(Q,tat,wt,n);

printf("Process\_no\tStart\_time\tComplete\_time\tTurn\_Around\_Time\tWaiting\_Time\n");

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

{

wavg += wt[i];

tavg += tat[i];

printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n",Q[i].pno,Q[i].st,Q[i].ct,tat[i],wt[i]);

}

printf("Average waiting time is : %f\n",wavg/(float)n);

printf("average turnaround time : %f\n",tavg/(float)n);

}

int main()

{

int n;

printf("Enter the no. of process count : ");

scanf("%d",&n);

struct process Q1[n];

printf("Enter the Arrival\_time, Burst\_time & Priority : e.g. (4 7 2)\n");

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

{

Q1[i].pno=i+1;

printf("For Process %d : ",Q1[i].pno);

scanf("%d %d %d",&Q1[i].at,&Q1[i].bt,&Q1[i].pr);

Q1[i].copy\_bt=Q1[i].bt;

}

qsort(Q1,n,sizeof(struct process),comp);

findgc(Q1,n);

return 0;

}

1. **Description:**

In this project, I implemented Multilevel Feedback Queue in **c** with two levels:

**Level 1:** Fixed priority preemptive Scheduling

**Level 2:** Round Robin Scheduling

1. Fixed priority preemptive Scheduling (Queue 1)

* Priority 0 is highest priority.
* Quantum : 2 unit time
* Preemptive :

If one process e.g. P1 is scheduled and running, now another process with higher priority comes e.g. P2. New process (high priority) process P2 preempts currently running process P1 and process P1 will go to second level queue.

1. Round Robin Scheduling (Queue 2)

* Quantum : 2 unit time
* All the processes in second level queue will complete their execution according to round robin scheduling.
* Queue 2 will be processed after Queue 1 becomes empty.
* Priority of Queue 2 has lower priority than in Queue 1.

Suppose Queue 1 is empty and currently process from Queue 2 is being executed. Now, if at this a new process arrives then new process will be part of Queue 1. So, new process should be scheduled as Queue 1 has higher priority than Queue2. Again after Queue 1 becomes empty Queue 2 will resume execution.

**2. Algorithm:**

1. Gobal variable Quantum=2.
2. Define a struct process consist of - int arrival\_time, burst\_time, temp\_burst\_time, priority, process\_no, start\_time, complete\_time, position(for round robin) & Boolean flag.
3. Define a comparator comp for qsort() function to sort according to arrival\_time.
4. Define a comparator pr\_comp for qsort() function to sort according to priority.
5. Define a comparator comp2 for qsort() function to sort according to arriving position in round robin scheduling.
6. Define a function priority\_st\_ct\_time (struct process Q[], int n, int \*t)

{

1. p=0.
2. for i - from 1 to n initialize Q[i].flag=false.

[End of for loop]

1. \*t=Q[0].arrival\_time.
2. Q[0].start\_time=\*t.
3. j=1, k=0,count=0.
4. for \*t – from \*t to last process arrival\_time repeat step 7 to 9 & 17 .
5. decrement Q[k].burst\_time.
6. increment count.
7. if \*t is equal to Q[j].arrival\_time then execute step 10 & 16.
8. if Q[j].priority is less than Q[k].priority then run step 11 to 15 .
9. count=0.
10. Q[k].flag=true.
11. Q[k].position= p & increment p.
12. k=j.
13. Q[k].start\_time=\*t.

[End of if condition]

1. increment j.

[End of if condition]

1. if count is equal to Quantum or Q[k].burst\_time is equal to zero then run step 18 to 26 .
2. if Q[k].burst\_time is zero then Q[k].complete time= \*t-1.

[End of if condition]

1. count=0.
2. Q[k].flag=true.
3. Q[k].position=p & increment p.
4. min=Q[k+1].priority & b=k+1.
5. for a – from k+2 to j repeat step 24.
6. if Q[a].priority is less than min then min=Q[a].priority & b=a.

[End of if condition]

[End of for loop]

1. k=b.
2. Q[k].start\_time=\*t & j=k+1.

[End of if condition]

1. Call qsort(Q, n, sizeof (struct process), pr\_comp.
2. k=0 & count1=0.
3. for \*t- till count1<n repeat step 30, 33, 34.
4. if Q[k].flag is true then run step 31 & 32.
5. increment k , decrement \*t & increment count1.
6. continue.

[End of if condition]

1. decrement Q[k].burst\_time & increment count.
2. if count is equal to Quantum or Q[k].burst\_time is zero then run step 35 & 36.
3. count=0 & increment count1.
4. if Q[k].burst\_time is zero then run step 37 to 38.
5. Q[k].complete\_time=\*t-1.
6. Q[k].start\_time=\*t-Q[k].temp\_burst\_time.
7. else Q[k].start\_time=\*t-Quantum.

[End of if else]

1. Q[k].flag=true.
2. Q[k].position=p , increment p & increment k.

[End of if condition]

[End of for loop]

}

1. Define round\_robin\_st\_ct\_time(struct process Q[], int n, int \*t) function.

{

1. Call qsort (Q, n ,sizeof (struct process), comp2) function.
2. repeat step 3,4,& 14.
3. done=true.
4. for i – from 0 to n repeat step 5.
5. if Q[i].burst\_time is greater than zero then run step 6 & 7.
6. done=false.
7. if Q[i].burst\_time is greater then Quantum then run step 8 & 9.
8. \*t=\*t+Quantum.
9. Q[i].burst\_time=Q[i].burst\_time-Quantum.
10. else repeat step 11 to 13.
11. \*t+=Q[i].burst\_time.
12. Q[i].complete=\*t-1.
13. Q[i].burst\_time = 0.

[End of if else]

[End of if condition]

[End of for loop]

1. if done is true then stop the loop.

[End of while loop]

}

1. Define get\_wt\_time (struct process Q[], int tat[], int wt[], int n) function.

for i – from 0 to n : wt[i]=tat[i]-Q[i].temp\_burst\_time.

1. Define get\_tat\_time (struct process Q[], int tat[], int n) function.

for i – from 0 to n : tat[i]=Q[i].complete\_time-Q[i].arrival\_time.

1. Define findgc (struct process Q[], int n) function.

{

1. int wt[n], tat[ n], t=0.
2. double wavg=0,tavg=0.
3. Call priority\_st\_ct\_time (Q, n, &t) function.
4. Call round\_robin\_st\_ct\_time (Q, n, &t) function.
5. Call get\_tat\_time (Q, tat, n) function.
6. Call get\_wt\_time (Q, tat, wt, n) function.
7. for i – from 0 to n : repeat step 8 to 10.
8. wavg+=wt[i].
9. tavg+=tat[i].
10. print Q[i].process\_no, Q[i].start\_time, Q[i].complete\_time, tat[i] & wt[i].
11. print wavg & tavg.

}

1. Define main() function.

{

1. int n;
2. get input n.
3. struct process Q[n].
4. for i – from 0 to n: repeat 5 to 7.
5. Q[i].process\_no=i+1.
6. get input Q[i].arrival\_time, Q[i].burst\_time & Q[i].priority.
7. Q[i].temp\_burst\_time=Q[i].burst\_time.
8. Call qsort(Q, n, sizeof(struct process), comp) function.
9. Call findgc(Q, n) function.

}

1. End.
2. **Time Complexity**

#define Quantum 2

struct process

{

int at,copy\_bt,bt,pr,pno,st,ct,position;

bool flag;

};

int comp(const void \*a,const void \*b)

{

struct process \*ia=(struct process \*)a;

struct process \*ib=(struct process \*)b;

if(ia->at == ib->at) ………….O(2)

{

return ia->pr-ib->pr;

}

else

{

return ia->at-ib->at;

}

}

int pr\_comp(const void \*a,const void \*b)

{

struct process \*ia=(struct process \*)a;

struct process \*ib=(struct process \*)b;

if(ia->pr == ib->pr) ………….O(n)

{

return ia->at-ib->at;

}

else

{

return ia->pr-ib->pr;

}

}

int comp2(const void \*a,const void \*b)

{

struct process \*ia=(struct process \*)a;

struct process \*ib=(struct process \*)b;

return ia->position-ib->position; ………….O(1)

}

void priority\_st\_ct\_time(struct process Q[],int n,int \*t)

{

int pos=0;

for(int i=0;i<n;i++) ………….O(n)

{

Q[i].flag=false;

}

\*t=Q[0].at;

Q[0].st=\*t;

int j=1,k=0,count=0;

for((\*t)++;\*t<=Q[n-1].at;(\*t)++) ………….O(n-1)

{

Q[k].bt--;

count++;

if(\*t==Q[j].at) ………….O(1)

{

if(Q[j].pr<Q[k].pr) ………….O(1)

{

count=0;

Q[k].flag=true;

Q[k].position=pos++;

k=j;

Q[k].st=\*t;

}

j++;

}

if(count==Quantum || Q[k].bt<=0) ………….O(1)

{

if(Q[k].bt==0) ………….O(1)

{

Q[k].ct=\*t-1;

}

count=0;

Q[k].flag=true;

Q[k].position=pos++;

int min=Q[k+1].pr,b=k+1;

for(int a=k+2;a<j;a++) ………….O(n-1)

{

if(Q[a].pr<min){ ………….O(1)

min=Q[a].pr;

b=a;

}

}

k=b;

Q[k].st=\*t;

j=k+1;

}

}

qsort(Q,n,sizeof(struct process),pr\_comp); ………….O(nlogn)

k=0;

int count1=0;

for(;count1<n;(\*t)++) ………….O((Q[0].bt +…+ Q[n-1].bt)\*2)

{

if(Q[k].flag) ………….O(1)

{

k++;

(\*t)--;

count1++;

continue;

}

Q[k].bt--;

count++;

if(count==Quantum || Q[k].bt<=0) ………….O(2\*1)

{

count=0;count1++;

if(Q[k].bt==0) ………….O(2)

{

Q[k].ct=\*t-1;

Q[k].st=\*t-Q[k].copy\_bt;

}

else

{

Q[k].st=\*t-Quantum;

}

Q[k].flag=true;

Q[k].position=pos++;

k++;

}

}

}

void round\_robin\_st\_ct\_time(struct process Q[],int n,int \*t)

{

qsort(Q,n,sizeof(struct process),comp2); ………….O(nlogn)

while (1) …………..O(2n²)

{

bool done = true;

for (int i = 0 ; i < n; i++) ………….O(2n)

{

if (Q[i].bt > 0) ………….O(2\*1)

{

done = false;

if (Q[i].bt > Quantum) ………….O(2)

{

\*t += Quantum;

Q[i].bt -= Quantum;

}

else

{

\*t+=Q[i].bt;

Q[i].ct=\*t-1;

Q[i].bt = 0;

}

}

}

if (done == true) …………….O(1)

break;

}

}

void get\_wt\_time(struct process Q[],int tat[],int wt[],int n)

{

for(int i=0;i<n;i++) …………….O(n)

wt[i]=tat[i]-Q[i].copy\_bt;

}

void get\_tat\_time(struct process Q[],int tat[],int n)

{

for(int i=0;i<n;i++) …………….O(n)

tat[i]=Q[i].ct-Q[i].at;

}

void findgc(struct process Q[],int n)

{

int wt[n],tat[n],t=0;

double wavg=0,tavg=0;

//Fixed priority preemptive scheduling

priority\_st\_ct\_time(Q,n,&t); …………….O(1)

//round robin scheduling

round\_robin\_st\_ct\_time(Q,n,&t); …………….O(1)

get\_tat\_time(Q,tat,n); …………….O(1)

get\_wt\_time(Q,tat,wt,n); …………….O(1)

printf(“Process\_no\tStart\_time\tComplete\_time\tTurn\_Around\_Time\tWaiting\_Time\n”);

…………….O(1)

for(int i=0;i<n;i++) …………….O(n)

{

wavg += wt[i];

tavg += tat[i];

printf(“%d\t\t%d\t\t%d\t\t%d\t\t%d\n”,Q[i].pno,Q[i].st,Q[i].ct,tat[i],wt[i]); ……..O(1)

}

printf(“Average waiting time is : %f\n”,wavg/(float)n); …………….O(1)

printf(“average turnaround time : %f\n”,tavg/(float)n); …………….O(1)

}

int main()

{

int n;

printf(“Enter the no. of process count : “); …………….O(1)

scanf(“%d”,&n); …………….O(1)

struct process Q1[n];

printf(“Enter the Arrival\_time, Burst\_time & Priority : e.g. (4 7 2)\n”); …………….O(1)

for(int i=0;i<n;i++) …………….O(n)

{

Q1[i].pno=i+1;

printf(“For Process %d : “,Q1[i].pno);

scanf(“%d %d %d”,&Q1[i].at,&Q1[i].bt,&Q1[i].pr);

Q1[i].copy\_bt=Q1[i].bt;

}

qsort(Q1,n,sizeof(struct process),comp); …………….O(nlog)

findgc(Q1,n); …………….O(1)

return 0;

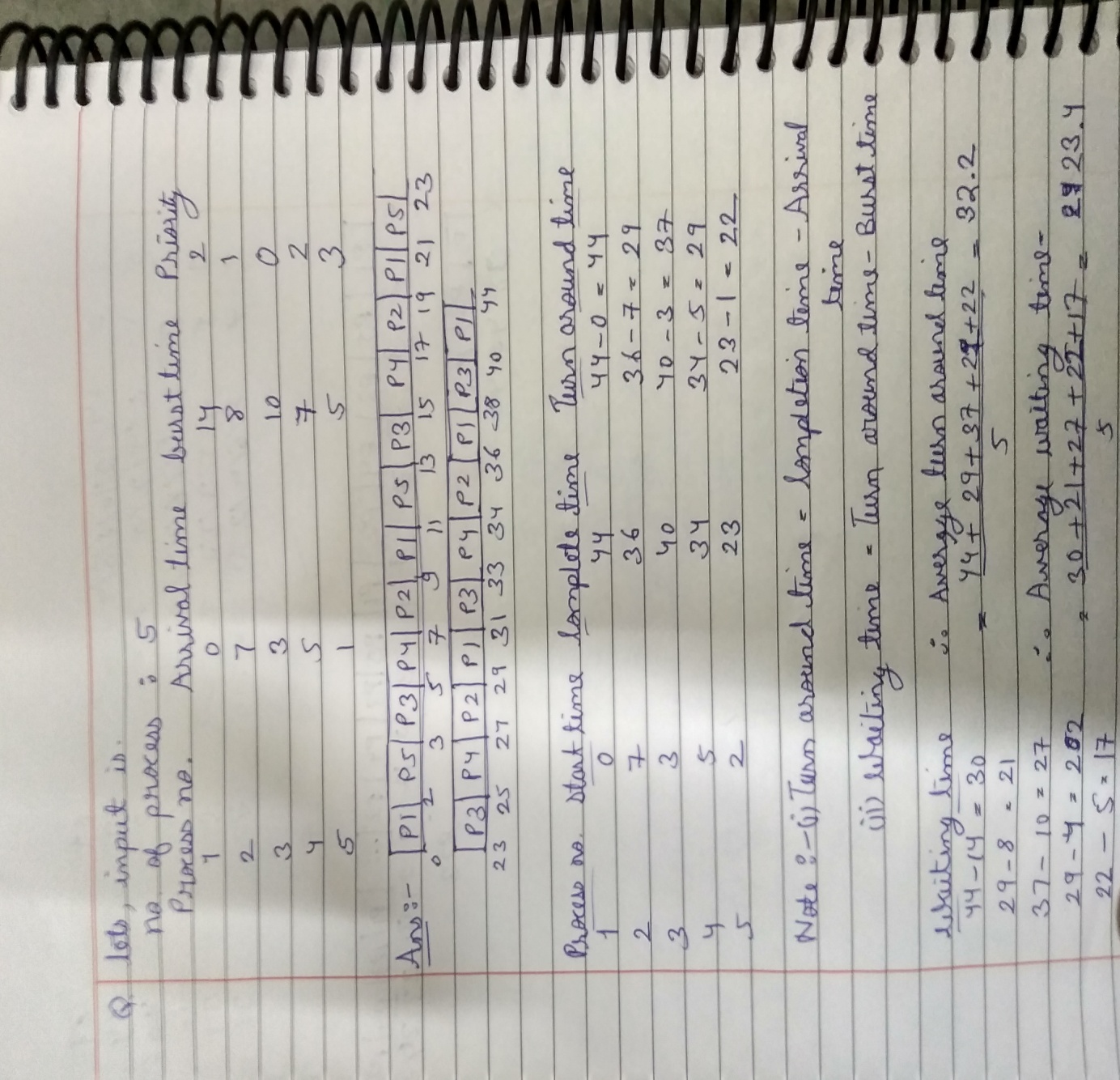
}

**Overall complexity : …O(1)+…+O(2)+…+O(n)+…+O(2n)+…+O(nlogn)+..+O(2n²) = O(n²)**

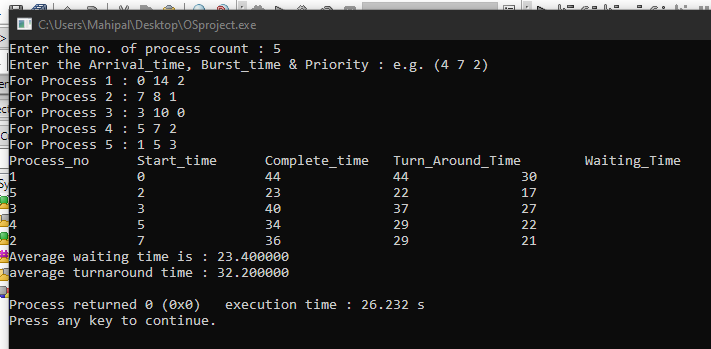
1. **Constraints & Boundary conditions**

* **2<=n<=107 where n is the no. of process i.e. length of Queue**
* **0<=at<=107 where at is the arrival time.**
* **1<=bt<=107 where bt is the burst time.**
* **0<=pr<=107 where pr is the priority.**
* **As Quantum period is 2. So, done give large burst time value which only increases the overhead of the CPU.**

1. **Test cases**
2. Actual solution of the test case 1

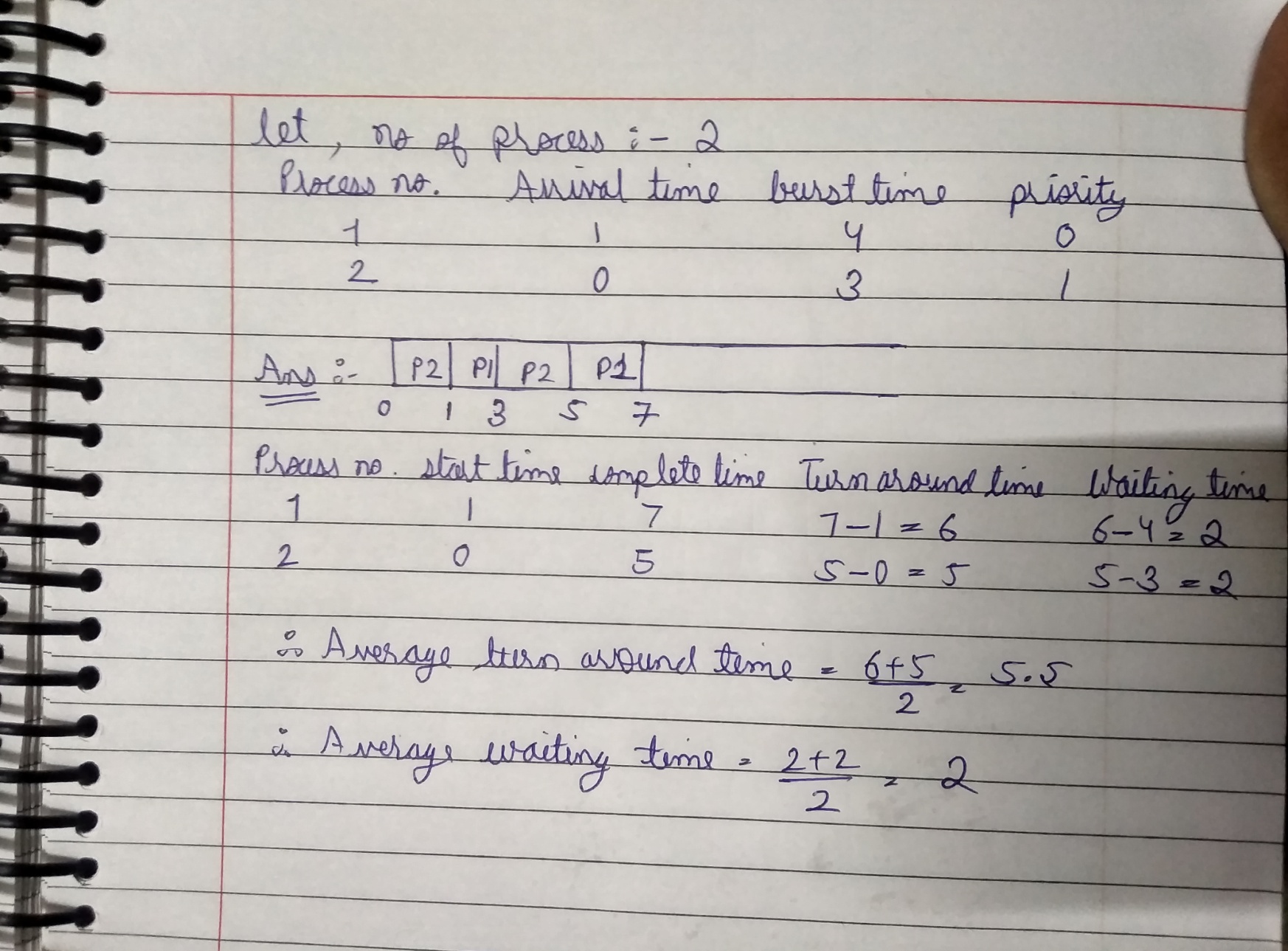


The output of the implemented code is:

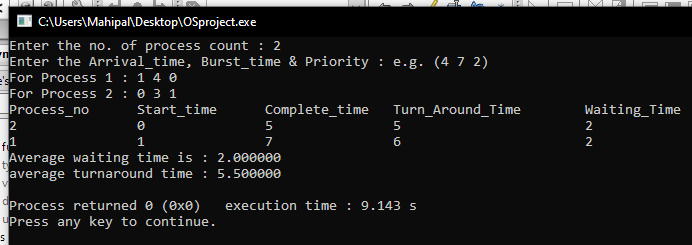


From both the pictures, output is similar which satisfy the implemented code.

1. Actual solution of test cases 2



The output of the implemented code is:



From both the pictures, output is similar which satisfy the implemented code.

1. **Have you made minimum 5 revisions of solution on GitHub?**

**GitHub Link:**

<https://github.com/Manojsuthar2000/Multilevel-Queue-Scheduling-Algorithm>