**CPU Scheduling Algorithms:**

1. **FCFS:**

**GUI Class:**private void **addRowActionPerformed**(java.awt.event.ActionEvent evt) {

//Table of Processes

DefaultTableModel model= (DefaultTableModel) Table.getModel();

int time=0;

try{

time=Integer.parseInt(BurstTime.getText().trim()); // for getting Integer from string textfield }

catch(Exception e){

e.printStackTrace();

}

if(ProcessName.getText()!=null && BurstTime.getText()!=null)

{

int input= JOptionPane.showConfirmDialog(null,"Add Or Not");

if(input==0)//for yes

{

JOptionPane.showMessageDialog(null,"Added Successfully", "Process Insertion Alert", JOptionPane.INFORMATION\_MESSAGE);

model.addRow(new Object[]{ProcessName.getText(),Integer.parseInt(BurstTime.getText())});

q1.insert(ProcessName.getText(),Integer.parseInt(BurstTime.getText()));

}

else if(input==1 || input==2) // for cancels or No option selected

JOptionPane.showMessageDialog(null,"You have canceled", "Process Insertion Alert", JOptionPane.INFORMATION\_MESSAGE);

ProcessName.setText(null);

BurstTime.setText(null);

}

else if(ProcessName.getText()==null && BurstTime.getText()==null){

JOptionPane.showMessageDialog(null,"Process Name or Burst Time is null", "Process Insertion Alert", JOptionPane.ERROR\_MESSAGE);

}

}

private void **ExecuteActionPerformed**(java.awt.event.ActionEvent evt) {

DefaultTableModel model= (DefaultTableModel) Table2.getModel();

//String step=(String) Mode.getSelectedItem(); // for Selection of Scheduling Algo TYpe

int c=Table.getRowCount(); //table inserted rows count

if(c==1 || c==0){

JOptionPane.showMessageDialog(null,"Not applicable on less than 2 processes","Process Alert",JOptionPane.ERROR\_MESSAGE);

}

//q1.counter=Table.getRowCount();

else {

// q1.print();

q1.chartForFCFS();

DecimalFormat dc = new DecimalFormat("0.00"); //to get in 2 decimal places

String formattedText = dc.format(q1.avgTime);

WaitTime.setText(formattedText);

int[] wait=q1.ganttArray;

String[] process=q1.processGantt;

for(int i=0;i<q1.counter-1;i++)

{

model.addRow(new Object[]{process[i],wait[i]});

}

r.apw=q1.ganttArray;

r.p=q1.processGantt;

r.frame.setVisible(true);

r.paint(r.g);

}

}

1. **Main Class for FCFS and SJF:**

import java.applet.Applet;

class Node{

Node next;

int startdata;

int enddata;

String name;

Node prev;

public Node(String name,int data) {

this.name=name;

this.startdata=data;

}

public Node() {}

}

public class Queue extends Applet{

int array[];

public Node head,last;

int counter=1;

Queue gantt; //for displaying gantt chart

int[] ganttArray;

String[] processGantt;

double avgTime=0.0;

public Queue() {

array=new int[200];

}

public Node dequeue() {

Node node = head;

if (node == null) {

return node;

} else {

node = head;

head = head.next;

if (head == null)// means if the list has become empty after deleting one node

{

last = null;// then tail should also point to null

}

}

return node;

}

void insert(String name,int data) { // for process insertions

Node n=new Node(name,data);

if(head==null) {

array[counter]=data;

n.prev=null;

this.head=last=n;

counter++;

}

else

{

if(search(name,data)==true) {

array[counter]=data;

n.prev=last;

last.next=n;

last=n;

counter++;

}

}

}

void insert(String name,int data,int endvalue) { // for operations like FCFS and SJF saving their waiting times used as Queue q2

int counter = 1;

Node n=new Node(name,data);

if(head==null) {

array[counter]=data;

n.enddata=endvalue;

n.prev=null;

this.head=last=n;

counter++;

}

else

{

if(search(name,data)==true) {

array[counter]=data;

n.prev=last;

n.enddata=endvalue;

last.next=n;

last=n;

counter++;

}

}

}

boolean search(String name,int data) {

Node n=head;

while(n!=null) {

if(n.startdata==data && n.name==name) {

return false;

}

n=n.next;

}

return true;

}

double waitingTime(Queue q){

Node n=q.dequeue();

int processNo=1;

double waitTime=0.0;

while(n!=null){

waitTime=waitTime+n.startdata;

processNo++;

n=n.next;

}

return waitTime/processNo;

}

int[] sortOFArray(int arr[])

{

int n = counter;

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

{

// Find the minimum element in unsorted array

int min\_idx = i;

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

if (arr[j] < arr[min\_idx])

min\_idx = j;

// Swap the found minimum element with the first

// element

int temp = arr[min\_idx];

arr[min\_idx] = arr[i];

arr[i] = temp;

}

return arr;

}

void chartForSJF() {

ganttArray=new int[counter]; // for gantt chart display only

processGantt=new String[counter];

Node hh=head;

Node nn=InsertSort(hh);

int[]array2=sortOFArray(array);// for waiting time calculation only

Queue q2=new Queue();

int step=1;

q2.insert((String) nn.name,0,array2[step]);

step++;

nn=nn.next;

while(nn!=null && step!=counter+1) {

array2[step]=array2[step]+array2[step-1];

q2.insert((String) nn.name,array2[step-1],array2[step]);

step++;

nn=nn.next;

}

if(counter>2) { //two processes compulsory

int i=0;

Node del=q2.dequeue();

ganttArray[0]=0;

processGantt[0]=(String)del.name;

i++;

del=del.next;

while(del!=null) {

ganttArray[i]=del.startdata;

processGantt[i]=(String)del.name;

//System.out.println(""+del.name+" starts from "+del.startdata+"and ends on "+del.enddata);

if(del.next==null)

{

ganttArray[i]=del.startdata;

processGantt[i+1]="P4"; //for garbage only

ganttArray[i+1]=del.enddata;

break;

}

i++;

del=del.next;

}

avgTime= waitingTime(q2);

}

}

void chartForFCFS() {

ganttArray=new int[counter];

processGantt=new String[counter];

gantt=new Queue();

System.out.println("In chart function");

Node nn=head;

Node n=head;

while(n!=null){

System.out.println(n.name);

n=n.next;

}

Queue q2=new Queue();

int step=1;

q2.insert((String) nn.name,0,array[step]);

step++;

nn=nn.next;

while(nn!=null && step!=counter+1) {

array[step]=array[step]+array[step-1];

q2.insert((String) nn.name,array[step-1],array[step]);

step++;

nn=nn.next;

}

if(counter>2) {

Node del=q2.dequeue();

gantt.insert((String) del.name, 0,del.enddata);

int i=0;

ganttArray[0]=0;

processGantt[0]=(String)del.name;

i++;

System.out.println(""+del.name+" starts from "+0+"and ends on "+del.enddata);

del=del.next;

while(del!=null) {

ganttArray[i]=del.startdata;

processGantt[i]=(String)del.name;

gantt.insert((String) del.name, del.startdata,del.enddata);

if(del.next==null){

ganttArray[i]=del.startdata;

processGantt[i+1]="P4";//for garbage only

ganttArray[i+1]=del.enddata;

break;

}

i++;

del=del.next;

}

avgTime= waitingTime(q2);

}

}

public static Node SortedInsert(Node head, Node newNode)

{

Node dummy = new Node();

Node current = dummy;

dummy.next = head;

while (current.next != null && current.next.startdata <= newNode.startdata) {

current = current.next;

}

newNode.next = current.next;

current.next = newNode;

return dummy.next;

}

public static Node InsertSort(Node head)

{

Node result = null;

Node current = head;

Node next;

while (current != null)

{

next = current.next;

result = SortedInsert(result, current);

current = next;

}

return result;

}

}

1. **SRTF:**

class NewClass {

int n; // for size of array.

int processes[][]; // for input of arrival times and burst times

int i;

String processGantt[]; // for gantt chart printing

int timesGantt[];// for gantt chart printing

float waitTime; // avg wait time

String ProcessTable[]; // for printing table of waiting time

int WaitTable[];// for printing table of waiting time

public void insert(int arrival,int burst){

System.out.println(processes.length);

System.out.println("Please enter the Arrival Time for Process " + i + ": ");

processes[i][0] = arrival;

System.out.println("Please enter the Burst Time for Process " + i + ": ");

processes[i][1] = burst;

i++;

}

public NewClass(int nt){

this.n=nt;

this.i=1;

processes = new int[nt + 1][4];

}

int step;

void Gantt(){ // for gantt chart

processGantt=new String[20];

timesGantt=new int[20];

waitTime=0;

step=0;

}

public void init(){ // working

// Calculation of Total Time and Initialization of Time Chart array

int total\_time = 0;

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

total\_time += processes[i][1];

}

int time\_chart[] = new int[total\_time];

for (int i = 0; i < total\_time; i++) {

// Selection of shortest process which has arrived

int sel\_proc = 1;

int min = 99999;

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

if (processes[j][0] <= i)// Condition to check if Process has arrived

{

if (processes[j][1] < min && processes[j][1] != 0) {

min = processes[j][1];

sel\_proc = j;

}

}

} // processes set according to their arrival time.

time\_chart[i] = sel\_proc; // select process and save it in array

// process remaining time decrement since it takes 1 sec to allocate to CPU(COntext Switch)

processes[sel\_proc][1]--;

// WT Calculation

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

if (processes[j][0] <= i) {

if (processes[j][1] != 0) { //if its burst time is not zero

processes[j][3]++;// If process has arrived and it has not already completed execution its TT is

// incremented by 1

if (j != sel\_proc)// If the process has not been currently assigned the CPU and has arrived its

processes[j][2]++;

} else if (j == sel\_proc)// This is a special case in which the process has been assigned CPU and

{ // has completed its execution

processes[j][3]++;}

}

}

if (i != 0) {

if (sel\_proc != time\_chart[i - 1])

// in between execution after cpu allocation

{

timesGantt[step]=i;

System.out.print("--" + i + "--P" + sel\_proc);

processGantt[step]="P"+sel\_proc;

step++;

}

} else{ // starts from 0

timesGantt[step]=i;

processGantt[step]="P"+sel\_proc;

step++;

}

if (i == total\_time - 1)// last print

{

timesGantt[step]=i+1;

}

}

WaitTable=new int[n+1];

ProcessTable=new String[n+1];

// Printing the WT.

int s=0;

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

WaitTable[s]=processes[i][2];

ProcessTable[s]="P"+i;

s++;

}

float WT = 0;

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

WT += processes[i][2];

//TT += proc[i][3];

}

WT /= n;

waitTime=WT;

}}

1. **Priority Scheduling:**

class Node2{

int starttime;

int endtime;

String process;

Node2 prev;

int pr;

public Node2() {}

public Node2(String proc,int e)

{

this.process=proc;

this.endtime=e;

}

Node2 next;

}

public class PrioritySchedulingNonPreemptive {

int[] waitingTimes;

void insert2nd(String d,int s,int e) {

Node2 n=new Node2(d,e);

n.starttime=s;

if(head==null)

{

head=tail=n;

counter++;

}

else

{

//tail=n;

n.prev=tail;

tail.next=n;

tail=n;

counter++;

}

}

String processes[];

public void work() {

processes=new String[counter];

int prevTime[]=new int[counter+1];

System.out.println("counter is "+counter);

Node2 nn=head;

Node2 h=InsertSort(nn);

int i=0;

waitingTimes=new int[counter+1];

q2=new PrioritySchedulingNonPreemptive();

waitingTimes[0]=0;

prevTime[0]=h.endtime;

waitingTimes[1]=h.endtime;

q2.insert2nd(h.process,0,h.endtime);

i=i+2;

h=h.next;

while(h!=null)

{

//waitingTimes[i]=h.endtime;

System.out.println("\nprocess is"+h.process);

System.out.println("end time is "+h.endtime);

System.out.println("i is "+i);

System.out.println("previous waiting time is"+waitingTimes[i-1]);

waitingTimes[i]=h.endtime+waitingTimes[i-1];

prevTime[i]=h.endtime;

q2.insert2nd(h.process,waitingTimes[i-1],waitingTimes[i]);

i++;

h=h.next;

}

int step=0;

Node2 n=q2.head;

while(n!=null)

{

processes[step]=n.process;

System.out.println("Process Name is "+n.process+"starts on "+n.starttime+"ends on"+n.endtime);

step++;

n=n.next;

}

for(int it=0;it<counter;it++)

{

wait=wait+waitingTimes[it];

}

System.out.println("wait is"+wait);

wait=wait/(counter);

System.out.println("average waiting time is"+wait);

}

float wait=0;

public static Node2 SortedInsert(Node2 head, Node2 newNode)

{

Node2 dummy = new Node2();

Node2 current = dummy;

dummy.next = head;

while (current.next != null && current.next.pr < newNode.pr) {

current = current.next;

}

newNode.next = current.next;

current.next = newNode;

return dummy.next;

}

// Given a list, change it to be in sorted order (using SortedInsert())

public static Node2 InsertSort(Node2 head)

{

Node2 result = null;

Node2 current = head;

Node2 next;

while (current != null)

{

next = current.next;

result = SortedInsert(result, current);

current = next;

}

return result;

}

PrioritySchedulingNonPreemptive q2;

int counter;

public Node2 head,tail;

Node2 searchPriority(int p) {

Node2 h=head;

while(h!=null)

{

if(h.pr==p)

{

return h;

}

h=h.next;

}

return null;

}

void insert(String d,int e,int p) {

Node2 n=new Node2(d,e);

n.pr=p;

if(head==null)

{

n.prev=null;

this.head=this.tail=n;

counter++;

}

else

{

//tail=n;

n.prev=tail;

tail.next=n;

tail=n;

counter++;

}

}

} **5. Round Robin:**public class RR {

int time; // time slice

int N; // total processes

int arr[]; // for burst time

int rear; // for circular queue

String process[]; // saving inputnprocesses

int front; // for circular queue

int arrival[]; // for input arrivals

int c; // current counter

int bt[]; // for burst times array used in gantt chart and computing waiting Times

int ww[];

String pp[];

int WaitGantt[];

String ProcessGantt[];

Node head, tail;

int counter = 0;

public RR(int N) {

this.N = N;

arr = new int[N + 1];

process = new String[N + 1];

rear = -1;

front = -1;

bt=new int[N+1];

}

public RR() {}

void insertion(String a, int ar, int e) {

arrival = new int[N + 1];

if (front == -1 && rear == -1) {

rear = 0;

front = 0;

arr[rear] = e;

bt[rear]=e;

process[rear] = a;

arrival[rear] = ar;

c++;

} else {

if (front == 0 && rear == 0) {

rear++;

bt[rear]=e;

arr[rear] = e;

process[rear] = a;

c++;

} else {

rear = (rear + 1) % N;

arr[rear] = e;

bt[rear]=e;

process[rear] = a;

arrival[rear] = ar;

c++;

}

}

}

boolean isEmpt() {

int i = 0;

int itt = front;

while (true) {

if (arr[itt] == 0)

i = -1;

if (itt == front)

break;

itt = (itt + 1) % N;

}

if (i == -1)

return true;

else

return false;

}

int NextIndex(int val) { // find next free index

int ii = val + 1;

int m = -1;

if (ii == arr.length - 1)

ii = 0;

while (ii != val) {

if ((int) arr[ii] != 0 && (int) arr[ii] > 0) {

m = ii;

break;

}

ii = (ii + 1) % N;

}

return m;

}

RR cq;

int wait[]; // for saving times for gantt chart

void printt() {

cq = new RR(N);

int i = 0;

int index = front;

wait = new int[200];

wait[0] = 0;

i++;

while (true) {

if ((int) arr[index] > time) {

int tt = (int) arr[index] - time;

arr[index] = tt;

wait[i] = time + wait[i - 1];

cq.insert(process[index], wait[i - 1], time + wait[i - 1]);

i++;

index = (index + 1) % N;

} else if ((int) arr[index] <= time && (int) arr[index] > 0) {

int tt = (int) arr[index];

arr[index] = -1;

wait[i] = tt + wait[i - 1];

cq.insert(process[index], wait[i - 1], tt + wait[i - 1]);

i++;

index = (index + 1) % N;

} else if ((int) arr[index] == -1 && isEmpt() == false) {

if (NextIndex(index) != -1)

index = NextIndex(index);

else

break;

} else if (isEmpt() == true) {

break;

}

}

ww =new int[cq.counter+1]; // for Gantt chart

pp=new String[cq.counter+1]; // for Gantt chart

WaitGantt=wait;

Node hh = cq.head;

int step=0;

int step2=0;

ww[step]=hh.startdata;

ww[step+1]=hh.enddata;

pp[step2]=hh.name;

step++;

step2++;

hh=hh.next;

while (hh.next != null) {

ww[step]=hh.enddata;

pp[step2]=hh.name;

step++;

step2++;

hh = hh.next;

}

ww[step+1]=hh.enddata;

pp[step2]=hh.name;

ProcessGantt=pp;

}

void insert(String d, int s, int e) { // simple insertion

Node n = new Node(d, e);

n.startdata = s;

n.enddata = e;

if (head == null) {

head = tail = n;

counter++;

} else {

tail.next = n;

tail = n;

counter++;

}

}

int i = 0;

int wt[]; // to find waiting time

void findWaitingTime()

{

wt=new int[N]; // for saving waiting times

int[] bt1=bt;

int quantum=time;

// Make a copy of burst times bt[] to store remaining

// burst times.

int rem\_bt[] = new int[N];

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

rem\_bt[i] = bt1[i];

int t = 0; // Current time

while(true)

{

boolean done = true;

// Traverse all processes one by one repeatedly

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

{

if (rem\_bt[i] > 0)

{

done = false; // process current status

if (rem\_bt[i] > quantum)

{

// processing time increases

t += quantum;

// decrease burst time.

rem\_bt[i] -= quantum;

}

else

{

// increase value of processed time

t = t + rem\_bt[i];

// waiting time is current time- burst time

wt[i] = t - bt1[i];

// if its fully executed then set burst time to the zero

rem\_bt[i] = 0;

}

}

}

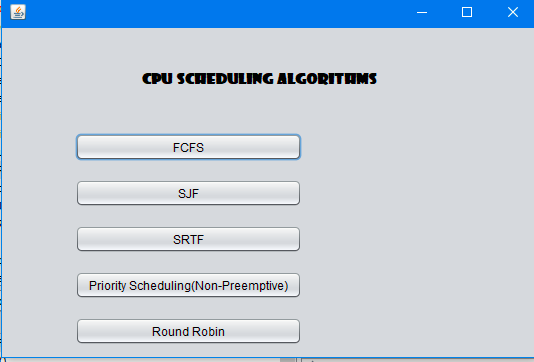
// If all processes are done

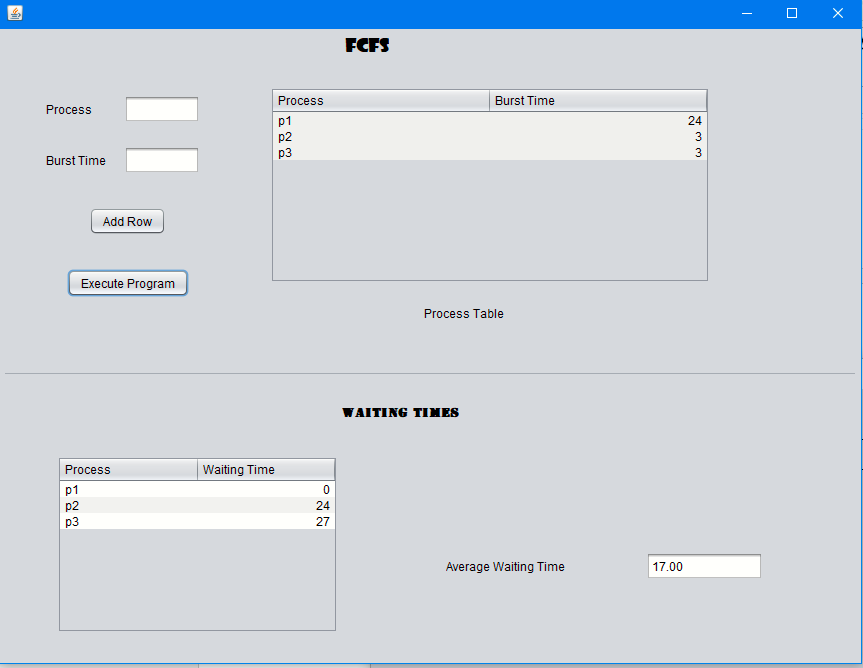
if (done == true){

break;}

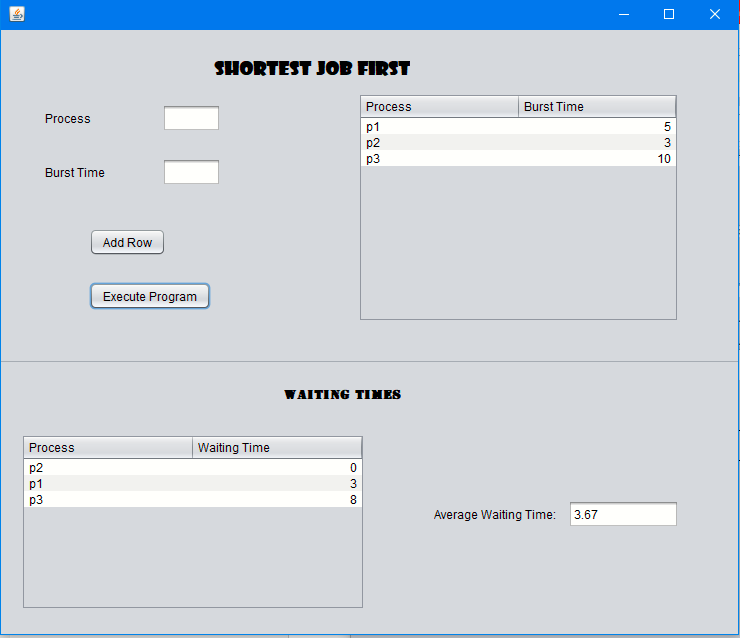
}}}

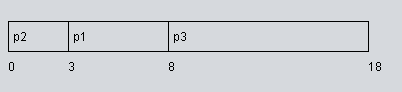
**Main Form:**



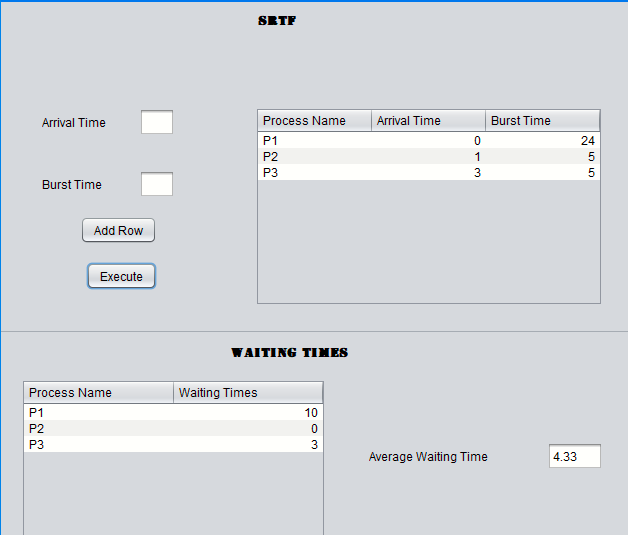
**FCFS:**



**SJF:**

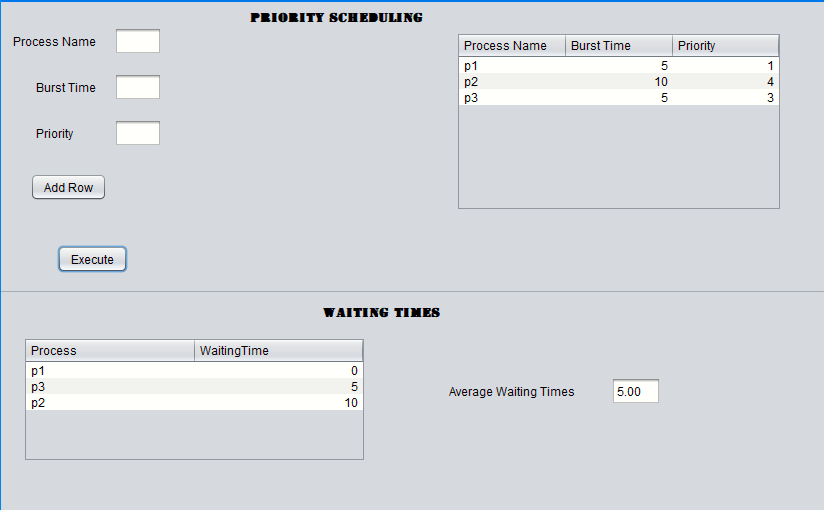


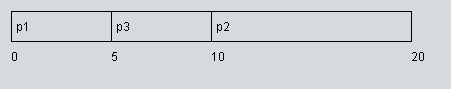
**SRTF:**



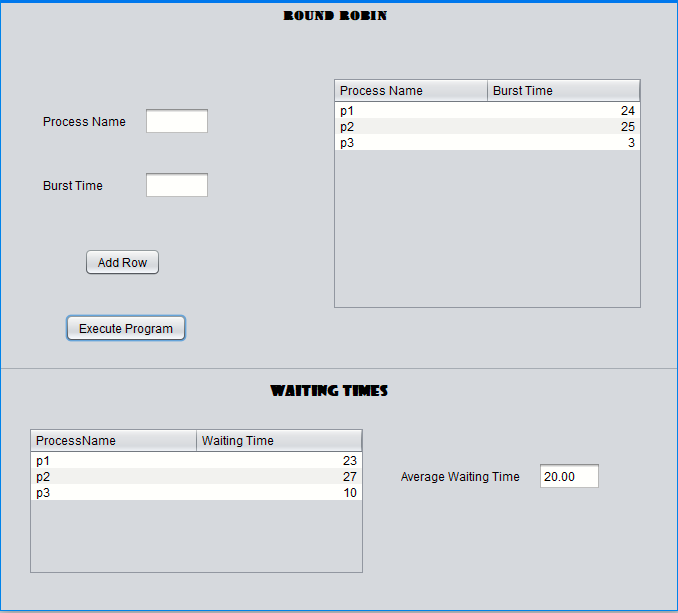


**Priority Scheduling:**





**RR:**



**With time Slice=5**

