**Lab 7: CPU Scheduling**

1. **First Come First Serve (FCFS)**

|  |
| --- |
| import java.util.Scanner;  public class FCFS {  public static void main(String[] args) {  Scanner input = new Scanner(System.in);  // Take the number of processes as input  System.out.print("Enter number of processes: ");  int n = input.nextInt();  System.out.println();  // Initialize arrays to store arrival time, burst time, waiting time, turn around time, and completion status of each process  int[] at = new int[n];  int[] bt = new int[n];  int[] tat = new int[n];  int[] wt = new int[n];  boolean[] completed = new boolean[n];  // Take the arrival time and burst time of each process as input  for(int i = 0; i < n; i ++) {  System.out.print("Enter Arrival Time of P" + (i + 1) + ": ");  at[i] = input.nextInt();  System.out.print("Enter Burt Time of P" + (i + 1) + ": ");  bt[i] = input.nextInt();  }  int currentTime = 0;  int completedProcesses = 0;  // Loop until all processes have been completed  while(completedProcesses < n) {  int Job = -1;  int shortestArrivalTime = Integer.MAX\_VALUE;  // Find the process with the shortest arrival time that has arrived and has not yet been completed  for(int i = 0; i < n; i ++) {  if(at[i] <= currentTime && completed[i] == false && at[i] < shortestArrivalTime) {  Job = i;  shortestArrivalTime = at[i];  }  }  // If there are no such processes, increment the current time by 1  if(Job == -1) {  currentTime ++;  }  // If there is such a process, calculate the waiting time and turn around time of the process, update the current time, mark the process as completed, and increment the number of completed processes  else {  wt[Job] = currentTime - at[Job];  tat[Job] = bt[Job] + wt[Job];  currentTime = currentTime + bt[Job];  completed[Job] = true;  completedProcesses ++;  }  }  // Calculate the average waiting time and average turn around time  double avgwt = 0.0;  double avgtat = 0.0;  // System.out.println("\nP \tAT\tBT\tET\tWT\tTT");  for(int i = 0; i < n; i ++) {  // System.out.printf("P%d\t%d\t%d\t%d\t%d\t%d\n", i+1, at[i], bt[i], tat[i] + bt[i], tat[i], wt[i]);  avgwt = avgwt + wt[i];  avgtat = avgtat + tat[i];  }  avgwt = avgwt / n;  avgtat = avgtat / n;  // Print the average waiting time and average turn around time  System.out.println("\nAverage Waiting Time is " + avgwt);  System.out.println("Average Turn Around Time is " + avgtat);  input.close();  }  }  **Output:** |

1. **Shortest Job First (SJF)**

|  |
| --- |
| import java.util.Scanner;  public class SJF {  public static void main(String[] args) {  Scanner input = new Scanner(System.in);  // Take the number of processes as input  System.out.print("Enter number of processes: ");  int n = input.nextInt();  System.out.println();  // Initialize arrays to store arrival time, burst time, waiting time, turn around time, and completion status of each process  int[] at = new int[n];  int[] bt = new int[n];  int[] tat = new int[n];  int[] wt = new int[n];  boolean[] completed = new boolean[n];  // Take the arrival time and burst time of each process as input  for(int i = 0; i < n; i ++) {  System.out.print("Enter Arrival Time of P" + (i + 1) + ": ");  at[i] = input.nextInt();  System.out.print("Enter Burt Time of P" + (i + 1) + ": ");  bt[i] = input.nextInt();  }  int currentTime = 0;  int completedProcesses = 0;  // Loop until all processes have been completed  while(completedProcesses < n) {  int shortestJob = -1;  int shortestBurstTime = Integer.MAX\_VALUE;  // Find the process with the shortest burst time that has arrived and has not yet been completed  for(int i = 0; i < n; i++) {  if(at[i] <= currentTime && completed[i] == false && bt[i] < shortestBurstTime) {  shortestJob = i;  shortestBurstTime = bt[i];  }  }  // If there are no such processes, increment the current time by 1  if(shortestJob == -1) {  currentTime ++;  }    // If there is such a process  // Calculate the waiting time and turn around time of the process  // Udate the current time  // Mark the process as completed  // Increment the number of completed processes  else {  wt[shortestJob] = currentTime - at[shortestJob];  tat[shortestJob] = bt[shortestJob] + wt[shortestJob];  currentTime = currentTime + bt[shortestJob];  completed[shortestJob] = true;  completedProcesses++;  }  }  // Calculate the average waiting time and average turn around time  double avgwt = 0.0;  double avgtat = 0.0;  for(int i = 0; i < n; i ++) {  avgwt = avgwt + wt[i];  avgtat = avgtat + tat[i];  }  avgwt = avgwt / n;  avgtat = avgtat / n;  // Print the average waiting time and average turn around time  System.out.println("\nAverage Waiting Time is " + avgwt);  System.out.println("Average Turn Around Time is " + avgtat);  input.close();  }  }  **Output:** |

1. **Shortest Remaining Time First (SRTF)**

|  |
| --- |
| import java.util.Scanner;  public class SRTF {  public static void main(String[] args) {  Scanner input = new Scanner(System.in);    // Take the number of processes as input  System.out.print("Enter number of processes: ");  int n = input.nextInt();  System.out.println();    // Initialize arrays to store arrival time, burst time, remaining time, completion time, waiting time, turn around time, and completion status of each process  int[] at = new int[n];  int[] bt = new int[n];  int[] rt = new int[n];  int[] ct = new int[n];  int[] tat = new int[n];  int[] wt = new int[n];  boolean[] completed = new boolean[n];  // Take the arrival time and burst time of each process as input and initialize the remaining time of each process to its burst time  for(int i = 0; i < n; i ++) {  System.out.print("Enter Arrival Time of P" + (i + 1) + ": ");  at[i] = input.nextInt();  System.out.print("Enter Burst Time of P" + (i + 1) + ": ");  bt[i] = input.nextInt();  rt[i] = bt[i];  }  int currentTime = 0;  int completedProcesses = 0;  // Loop until all processes have been completed  while(completedProcesses < n) {  int shortestJob = -1;  int shortestRemainingTime = Integer.MAX\_VALUE;  // Find the process with the shortest remaining time that has arrived and has not yet been completed  for(int i = 0; i < n; i ++) {  if(at[i] <= currentTime && completed[i] == false && rt[i] < shortestRemainingTime) {  shortestJob = i;  shortestRemainingTime = rt[i];  }  }  // If there are no such processes, increment the current time by 1  if(shortestJob == -1) {  currentTime ++;  }  // If there is such a process  else {  if(shortestRemainingTime == 1) {  ct[shortestJob] = currentTime + 1;  completed[shortestJob] = true;  completedProcesses ++;  }  rt[shortestJob] = rt[shortestJob] - 1;  currentTime ++;  }  }  // Calculate the waiting time and turn around time of each process  for(int i = 0; i < n; i ++) {  tat[i] = ct[i] - at[i];  wt[i] = tat[i] - bt[i];  }  // Calculate the average waiting time and average turn around time  double avgwt = 0.0;  double avgtat = 0.0;  for(int i = 0; i < n; i ++) {  avgwt = avgwt + wt[i];  avgtat = avgtat + tat[i];  }  avgwt = avgwt / n;  avgtat = avgtat / n;  // Print the average waiting time and average turn around time  System.out.println("\nAverage Waiting Time is " + avgwt);  System.out.println("Average Turn Around Time is " + avgtat);  input.close();  }  }  **Output:** |

1. **Round Robin**

|  |
| --- |
| import java.util.\*;  public class Round\_Robbin {  public static void main(String[] args) {  Scanner input = new Scanner(System.in);  // Input number of processes  System.out.print("Enter the number of processes: ");  int n = input.nextInt();  // Creating an "Array of Arraylist" to store data of each process  ArrayList<int[]> processes = new ArrayList<int[]>();  // Input Arrival Time & Burst Time  for (int i = 0; i < n; i++) {  System.out.printf("Enter arrival time for process %d: ", i+1);  int at = input.nextInt();  System.out.printf("Enter burst time for process %d: ", i+1);  int bt = input.nextInt();  processes.add(new int[] {at, bt, bt, 0, 0, 0});  }  // Input Time Quantum  System.out.print("Enter time quantum: ");  int quantum = input.nextInt();  // Declarations  int time = 0;  double averageWaitingTime = 0;  double averageTurnaroundTime = 0;  // Sorting list according to Arrival time  processes.sort(Comparator.comparingInt(process -> process[0]));  // Creating a Ready Queue  Queue <Integer> readyQueue = new LinkedList<>();  readyQueue.add(0);  while (!readyQueue.isEmpty()) {  // Ready the first ready process  int i = readyQueue.poll();  // If Remaining burst time of process is less than or equal to Time Quantum  if (processes.get(i)[2] <= quantum) {  time += processes.get(i)[2];  processes.get(i)[2] = 0;  processes.get(i)[3] = time;  processes.get(i)[4] = processes.get(i)[3] - processes.get(i)[0];  processes.get(i)[5] = processes.get(i)[4] - processes.get(i)[1];  }    // If Remaining burst time of process is greater than Time Quantum  else {  time += quantum;  processes.get(i)[2] -= quantum;  // Create a new ArrayList of integers to store the indices of the processes that are ready to be executed  ArrayList<Integer> temp = new ArrayList<>();  // Loop through all processes and check if they are ready to be executed based on their arrival time and remaining burst time  for (int j = 0; j < n; j++) {  if (processes.get(j)[0] <= time && j != i && !readyQueue.contains(j) && processes.get(j)[2] != 0) {  // If a process is ready to be executed, add its index to the temporary ArrayList  temp.add(j);  }  }  // Add all the processes that are ready to be executed to the queue  readyQueue.addAll(temp);  // Add the current process to the queue  readyQueue.offer(i);  }  }  // Calculating Average Waiting Time & Average Turnaround Time  System.out.println("\nP \tAT\tBT\tET\tWT\tTT");  for (int i = 0; i < n; i++) {  System.out.printf("P%d\t%d\t%d\t%d\t%d\t%d\n", i+1, processes.get(i)[0], processes.get(i)[1], processes.get(i)[3], processes.get(i)[4], processes.get(i)[5]);  averageWaitingTime += processes.get(i)[5];  averageTurnaroundTime += processes.get(i)[4];  }  averageWaitingTime /= n;  averageTurnaroundTime /= n;  // Printing Final Results  System.out.printf("\nAverage Waiting Time: %.2f\n", averageWaitingTime);  System.out.printf("Average Turnaround Time: %.2f\n", averageTurnaroundTime);  input.close();  }  }  **Output:** |