# LP - 1 Pratical

## Experiment no. 1 (Assembler : Pass 1 and Pass 2)

### Code :

### Pass-1:

import java.io.\*;

import java.util.Scanner;

import java.util.StringTokenizer;

class AssemblerPassOne {

static Scanner in = new Scanner(System.in);

static String[] is = { "STOP", "ADD", "SUB", "MULT", "MOVER", "MOVEM", "COMP", "BC", "DIV", "READ", "PRINT" };

static String[] ad = { "START", "END", "ORIGIN", "EQU", "LTORG" };

static String[] dl = { "DC", "DS" };

static String[] cc = { "LT", "LE", "EQ", "GT", "GE", "ANY" };

static int symCounter = 0;

static int litCounter = 0;

static String[][] sym = new String[100][2];

static String[][] lit = new String[100][2];

static String[][] ptab = new String[100][2];

public static void main(String args[]) throws Exception {

int locate = 0;

int litCount = 0;

BufferedReader reader = new BufferedReader(new FileReader("input.asm"));

BufferedWriter writer = new BufferedWriter(new FileWriter("intermediate.txt"));

BufferedWriter writer1 = new BufferedWriter(new FileWriter("SYMTAB.txt"));

BufferedWriter writer2 = new BufferedWriter(new FileWriter("LITTAB.txt"));

String st, y, prev = null;

int stp = 0, k = 0;

String ans, buffer = "", buffer1 = "", buffer2 = "";

while ((st = reader.readLine()) != null) {

int isFlag = 0;

k++;

StringTokenizer splitted = new StringTokenizer(st);

ans = "";

while (splitted.hasMoreTokens()) {

y = splitted.nextToken();

if (y.equals("START")) {

locate = Integer.parseInt(splitted.nextToken());

ans = "(AD,01)(C," + locate + ")";

break;

} else {

if (searchIs(y)) {

if (y.equals("STOP")) stp = 1;

ans += "(IS," + indexIs(y) + ")";

isFlag = 1;

locate++;

} else if (searchAd(y)) {

processAd(y, splitted, locate, litCount, ans);

} else if (searchDl(y)) {

processDl(y, splitted, locate, ans);

} else {

processSymbols(y, isFlag, stp, locate, splitted, ans, prev);

}

}

}

ans += "\n";

buffer += ans;

}

System.out.println(buffer + "\n");

// Symbol Table

for (int i = 0; i < symCounter; i++) {

buffer1 += sym[i][0] + "\t" + sym[i][1] + "\n";

}

// Literal Table

for (int i = 0; i < litCounter; i++) {

buffer2 += lit[i][0] + "\t" + lit[i][1] + "\n";

}

writer.write(buffer);

writer1.write(buffer1);

writer2.write(buffer2);

reader.close();

writer.close();

writer1.close();

writer2.close();

System.out.println("Program finished...");

}

public static boolean searchIs(String s) {

for (String item : is) if (item.equals(s)) return true;

return false;

}

public static boolean searchAd(String s) {

for (String item : ad) if (item.equals(s)) return true;

return false;

}

public static boolean searchDl(String s) {

for (String item : dl) if (item.equals(s)) return true;

return false;

}

public static boolean searchSym(String s) {

for (String[] entry : sym) if (entry[0].equals(s)) return true;

return false;

}

public static boolean searchCc(String s) {

for (String item : cc) if (item.equals(s)) return true;

return false;

}

public static int indexSym(String s) {

for (int i = 0; i < symCounter; i++) if (sym[i][0].equals(s)) return i;

return -1;

}

public static int indexLit(String s) {

for (int i = 0; i < litCounter; i++) if (lit[i][0].equals(s)) return i;

return -1;

}

public static int indexIs(String s) {

for (int i = 0; i < is.length; i++) if (is[i].equals(s)) return i;

return -1;

}

public static int indexAd(String s) {

for (int i = 0; i < ad.length; i++) if (ad[i].equals(s)) return i;

return -1;

}

public static int indexDl(String s) {

for (int i = 0; i < dl.length; i++) if (dl[i].equals(s)) return i;

return -1;

}

public static int indexCc(String s) {

for (int i = 0; i < cc.length; i++) if (cc[i].equals(s)) return i;

return -1;

}

private static void processAd(String y, StringTokenizer splitted, int locate, int litCount, String ans) {

if (y.equals("LTORG")) {

locate += litCount;

ans = "(AD,05)\n";

while (litCount > 0) {

lit[litCounter - litCount][1] = Integer.toString(locate - litCount);

String temp = lit[litCounter - litCount][0].substring(2, lit[litCounter - litCount][0].length() - 1);

ans += "(DL,02)(C," + temp + ")";

litCount--;

if (litCount != 0) ans += "\n";

}

}

}

private static void processDl(String y, StringTokenizer splitted, int locate, String ans) {

if (y.equals("DS")) {

ans += "(DL,1)(C," + splitted.nextToken() + ")";

} else if (y.equals("DC")) {

ans += "(DL,2)(C," + splitted.nextToken() + ")";

}

locate++;

}

private static void processSymbols(String y, int isFlag, int stp, int locate, StringTokenizer splitted, String ans, String prev) {

if (!searchSym(y) && isFlag == 0 && stp == 0) {

sym[symCounter][0] = y;

sym[symCounter++][1] = Integer.toString(locate);

ans += "(S," + (indexSym(y) + 1) + ")";

if (splitted.hasMoreTokens()) ans = "";

} else if (!searchSym(y) && isFlag == 1 && stp == 0) {

sym[symCounter++][0] = y;

ans += "(S," + (indexSym(y) + 1) + ")";

} else if (searchSym(y) && isFlag == 0) {

sym[indexSym(y)][1] = Integer.toString(locate);

ans += "(S," + (indexSym(y) + 1) + ")";

if (splitted.hasMoreTokens()) ans = "";

prev = y;

} else {

if (!splitted.hasMoreTokens()) ans += "(S," + (indexSym(y) + 1) + ")";

}

}

}

### Output :

INPUT

START 200

MOVER AREG , ='5'

MOVEM AREG , X

L1 MOVER BREG , ='2'

ORIGIN L1+3

LTORG

NEXT ADD AREG , ='1'

SUB BREG , ='2'

BC LT , BACK

LTORG

BACK EQU L1

ORIGIN NEXT+5

MULT CREG , ='4'

STOP

X DS 1

END

OUTPUT :-

INTERMEDIATE CODE:

(AD,01)(C,200)

(IS,4)(R,1)(L,1)

(IS,5)(R,1)(S,1)

(IS,4)(R,2)(L,2)

(AD,03)(S,2)+3

(AD,05)

(DL,02)(C,5)

(DL,02)(C,2)

(IS,1)(R,1)(L,3)

(IS,2)(R,2)(L,4)

(IS,7)(1)(S,4)

(AD,05)

(DL,02)(C,1)

(DL,02)(C,2)

(AD,03)(S,3)+5

(IS,3)(R,3)(L,5)

(IS,0)

(DL,1)(C,1)

(AD,02)

(DL,02)(C,4)

LITERAL TABLE:

='5' 205

='2' 206

='1' 210

='2' 211

='4' 215

SYMBOL TABLE

X 214

L1 202

NEXT 207

BACK 202

### Pass-2:

import java.io.\*;

public class AssemblerPassTwo {

public static void main(String[] args) throws Exception {

BufferedReader inputReader = new BufferedReader(new FileReader("intermediate.txt"));

BufferedReader symReader = new BufferedReader(new FileReader("SYMTAB.txt"));

BufferedReader litReader = new BufferedReader(new FileReader("LITTAB.txt"));

BufferedWriter outputWriter = new BufferedWriter(new FileWriter("machine\_code.txt"));

String inputLine;

String symLine = null;

String litLine = null;

while ((inputLine = inputReader.readLine()) != null) {

String[] tokens = inputLine.split("\\)\\(");

StringBuilder outputLine = new StringBuilder();

for (String token : tokens) {

token = token.replaceAll("[\\(\\)]", "");

if (token.startsWith("S,")) {

if (symLine == null) {

symLine = symReader.readLine();

}

if (symLine != null) {

String[] symTokens = symLine.split("\t");

if (symTokens.length > 1) {

int address = Integer.parseInt(symTokens[1]);

token = token.replace("S,", "");

token = getFormattedMachineCode(token, address);

}

}

} else if (token.startsWith("L,")) {

if (litLine == null) {

litLine = litReader.readLine();

}

if (litLine != null) {

String[] litTokens = litLine.split("\t");

if (litTokens.length > 1) {

int address = Integer.parseInt(litTokens[1]);

token = token.replace("L,", "");

token = getFormattedMachineCode(token, address);

}

}

}

outputLine.append(token.replace(",", "")).append(" ");

}

outputLine = new StringBuilder(outputLine.toString().replaceAll("[A-Za-z]", ""));

outputWriter.write(outputLine.toString().trim());

outputWriter.newLine();

}

inputReader.close();

symReader.close();

litReader.close();

outputWriter.close();

System.out.println("Machine code generated successfully.");

}

private static String getFormattedMachineCode(String instruction, int address) {

String[] parts = instruction.split(",");

String opcode = parts[0];

String[] operands = parts.length > 1 ? parts[1].split(" ") : new String[0];

String registerNumbers = "";

for (String operand : operands) {

int regIndex = Integer.parseInt(operand);

registerNumbers += getRegisterNumber(regIndex);

}

return opcode + " " + registerNumbers + " " + address;

}

private static String getRegisterNumber(int index) {

String[] regNumbers = { "01", "02", "03", "04" };

if (index >= 0 && index < regNumbers.length) {

return regNumbers[index];

}

return "";

}

}

### Output :

04 01 205

05 01 214

04 02 206

00 00 005

00 00 002

01 01 210

02 02 211

07 02 202

00 00 001

00 00 002

03 03 215

00 00 000

00 00 004

## Experiment no. 2 (Macro : Pass 1 and Pass 2)

### Code :

### Pass-1:

import java.io.\*;

import java.util.\*;

class Pass1 {

static String mnt[][] = new String[5][3]; // Macro Name Table (MNT)

static String ala[][] = new String[10][2]; // Argument List Array (ALA)

static String mdt[][] = new String[20][1]; // Macro Definition Table (MDT)

static int mntc = 0, mdtc = 0, alac = 0;

public static void main(String args[]) {

pass1();

System.out.println("Macro Name Table (MNT)");

display(mnt, mntc, 3);

System.out.println("Argument List Array (ALA) for Pass1");

display(ala, alac, 2);

System.out.println("Macro Definition Table (MDT)");

display(mdt, mdtc, 1);

}

static void pass1() {

int index = 0;

String s, prev = "", substring;

try {

BufferedReader inp = new BufferedReader(new FileReader("input.txt"));

while ((s = inp.readLine()) != null) {

if (s.equalsIgnoreCase("MACRO")) {

prev = s;

while (!(s = inp.readLine()).equalsIgnoreCase("MEND")) {

mdtc++;

prev = s;

if (prev.equalsIgnoreCase("MACRO")) {

StringTokenizer st = new StringTokenizer(s);

String str[] = new String[st.countTokens()];

for (int i = 0; i < str.length; i++) {

str[i] = st.nextToken();

}

// Forming MNT entry

mnt[mntc][0] = Integer.toString(mntc + 1);

mnt[mntc][1] = str[0];

mnt[mntc++][2] = Integer.toString(++mdtc);

// Tokenizing the arguments

st = new StringTokenizer(str[1], ",");

String args[] = new String[st.countTokens()];

for (int i = 0; i < args.length; i++) {

args[i] = st.nextToken();

ala[alac][0] = Integer.toString(alac);

index = args[i].indexOf("=");

if (index != -1) {

ala[alac++][1] = args[i].substring(0, index);

} else {

ala[alac++][1] = args[i];

}

}

} else {

// Replacing arguments with ALA index in MDT entries

index = s.indexOf("&");

if (index != -1) {

substring = s.substring(index);

for (int i = 0; i < alac; i++) {

if (ala[i][1].equals(substring)) {

s = s.replaceAll(substring, "#" + ala[i][0]);

}

}

}

}

mdt[mdtc - 1][0] = s;

}

mdt[mdtc - 1][0] = s; // Adding "MEND" to MDT

}

}

} catch (FileNotFoundException ex) {

System.out.println("Unable to find file ");

} catch (IOException e) {

e.printStackTrace();

}

}

static void display(String[][] table, int rowCount, int colCount) {

for (int i = 0; i < rowCount; i++) {

for (int j = 0; j < colCount; j++) {

System.out.print(table[i][j] + " ");

}

System.out.println();

}

}

}

**INPUT**

MACRO

ADDITION &arg1,&arg2,&arg3

MOV ax,&arg1

ADD ax,&arg2

ADD ax,&arg3

MEND

ADDITION 34,45,44

END

**OUTPUT**

Macro Name Table(MNT)

1 ADDITION 1

Argument List Array(ALA) for Pass1

0 &arg1

1 &arg2

2 &arg3

Macro Definition Table(MDT)

ADDITION &arg1,&arg2,&arg3

MOV ax,#0

ADD ax,#1

ADD ax,#2

MEND

### Pass-2:

import java.io.\*;

import java.util.\*;

class MacroPass2 {

public static void main(String args[]) {

pass2();

System.out.println("Argument List Array (ALA) for Pass2");

display(Pass1.ala, Pass1.alac, 2);

System.out.println("Note: All tables are displayed here whereas the expanded output is stored in the file pass2\_output.txt");

}

static void pass2() {

int alap = 0, index, mdtp, flag = 0;

String s, temp;

try {

BufferedReader inp = new BufferedReader(new FileReader("pass1\_output.txt"));

File op = new File("pass2\_output.txt");

if (!op.exists()) {

op.createNewFile();

}

BufferedWriter output = new BufferedWriter(new FileWriter(op.getAbsoluteFile()));

while ((s = inp.readLine()) != null) {

flag = 0;

StringTokenizer st = new StringTokenizer(s);

String str[] = new String[st.countTokens()];

for (int i = 0; i < str.length; i++) {

str[i] = st.nextToken();

}

for (int j = 0; j < Pass1.mntc; j++) {

if (str[0].equals(Pass1.mnt[j][1])) {

mdtp = Integer.parseInt(Pass1.mnt[j][2]);

st = new StringTokenizer(str[1], ",");

String arg[] = new String[st.countTokens()];

for (int i = 0; i < arg.length; i++) {

arg[i] = st.nextToken();

Pass1.ala[alap++][1] = arg[i];

}

// Expanding the macro until "MEND"

for (int i = mdtp; !Pass1.mdt[i][0].equalsIgnoreCase("MEND"); i++) {

index = Pass1.mdt[i][0].indexOf("#");

temp = Pass1.mdt[i][0].substring(0, index);

temp += Pass1.ala[Integer.parseInt("" + Pass1.mdt[i][0].charAt(index + 1))][1]; // Replace # argument with ALA value

output.write(temp);

output.newLine();

}

flag = 1;

}

}

if (flag == 0) { // When the line is not a macro

output.write(s);

output.newLine();

}

}

output.close();

} catch (FileNotFoundException ex) {

System.out.println("Unable to find file.");

} catch (IOException e) {

e.printStackTrace();

}

}

static void display(String a[][], int n, int m) {

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

for (int j = 0; j < m; j++) {

System.out.print(a[i][j] + " ");

}

System.out.println();

}

}

}

**INPUT:**

PRG2 START

USING \*,BASE

INCR1 DATA1,DATA2

INCR2 DATA3,DATA4

FOUR DC F'4'

FIVE DC F'5'

BASE EQU 8

TEMP DS 1F

DROP 8

END

**Output:**

Argument List Array(ALA) for Pass2

0 DATA1

1 DATA2

2 DATA3

3 DATA4

PRG2 START

USING \*,BASE

A 1,DATA1

L 2,DATA2

L 3,DATA3

ST 4,DATA4

FOUR DC F'4'

FIVE DC F'5'

BASE EQU 8

TEMP DS 1F

DROP 8

END

## Experiment no. 3 (CPU Scheduling : FCFS)

### Code :

import java.util.Scanner;

class Process {

int pid, burstTime, arrivalTime, waitingTime, turnaroundTime;

public Process(int pid, int burstTime, int arrivalTime) {

this.pid = pid;

this.burstTime = burstTime;

this.arrivalTime = arrivalTime;

}

}

public class FCFS {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter number of processes: ");

int n = sc.nextInt();

Process[] processes = new Process[n];

// Input process details

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

System.out.print("Enter arrival time and burst time for process " + (i + 1) + ": ");

int at = sc.nextInt();

int bt = sc.nextInt();

processes[i] = new Process(i + 1, bt, at);

}

// Sort processes based on arrival time

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (processes[j].arrivalTime > processes[j + 1].arrivalTime) {

Process temp = processes[j];

processes[j] = processes[j + 1];

processes[j + 1] = temp;

}

}

}

// Calculate waiting time and turnaround time

int totalTime = 0, totalWT = 0, totalTAT = 0;

for (Process p : processes) {

// Calculate waiting time for the current process

p.waitingTime = Math.max(0, totalTime - p.arrivalTime);

totalTime += p.burstTime;

// Calculate turnaround time for the current process

p.turnaroundTime = p.waitingTime + p.burstTime;

// Accumulate total waiting and turnaround times

totalWT += p.waitingTime;

totalTAT += p.turnaroundTime;

}

// Print Gantt Chart

System.out.println("\nGantt Chart: ");

for (Process p : processes) {

System.out.print("P" + p.pid + " ");

}

System.out.println("\n");

// Print process details

System.out.println("Process\tArrival\tBurst\tWaiting\tTurnaround");

for (Process p : processes) {

System.out.println("P" + p.pid + "\t" + p.arrivalTime + "\t" + p.burstTime + "\t" + p.waitingTime + "\t" + p.turnaroundTime);

}

// Print average waiting and turnaround times

System.out.printf("Average Waiting Time: %.2f\n", totalWT / (float) n);

System.out.printf("Average Turnaround Time: %.2f\n", totalTAT / (float) n);

sc.close();

}

}

## Output :

Enter number of processes: 3

Enter arrival time and burst time for process 1: 0 24

Enter arrival time and burst time for process 2: 0 3

Enter arrival time and burst time for process 3: 0 4

Gantt Chart:

P1 P2 P3

Process Arrival Burst Waiting Turnaround

P1 0 24 0 24

P2 0 3 24 27

P3 0 4 27 31

Average Waiting Time: 17.0

Average Turnaround Time: 27.333334

### Experiment no. 3 (CPU Scheduling : Priority)

### Code :

#### Preemptive :

import java.util.Scanner;

public class PriorityPreemptive {

static class Process {

int id, burst, priority, arrival, waitingTime, turnaroundTime;

int remainingBurst;

public Process(int id, int burst, int priority, int arrival) {

this.id = id;

this.burst = burst;

this.priority = priority;

this.arrival = arrival;

this.remainingBurst = burst;

}

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter number of processes: ");

int n = sc.nextInt();

Process[] processes = new Process[n];

// Input process details

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

System.out.print("Enter arrival time, burst time, and priority for process " + (i + 1) + ": ");

int at = sc.nextInt();

int bt = sc.nextInt();

int priority = sc.nextInt();

processes[i] = new Process(i + 1, bt, priority, at);

}

int completed = 0, time = 0;

String ganttChart = "";

// Main loop for scheduling

while (completed < n) {

int idx = -1;

int highestPriority = Integer.MAX\_VALUE;

// Find the process with the highest priority that is ready to execute

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

if (processes[i].arrival <= time && processes[i].remainingBurst > 0 && processes[i].priority < highestPriority) {

highestPriority = processes[i].priority;

idx = i;

}

}

if (idx != -1) {

// Execute the selected process for 1 unit of time

processes[idx].remainingBurst--;

ganttChart += "P" + processes[idx].id + " ";

// If the process completes, calculate its turnaround and waiting times

if (processes[idx].remainingBurst == 0) {

completed++;

processes[idx].turnaroundTime = time + 1 - processes[idx].arrival;

processes[idx].waitingTime = processes[idx].turnaroundTime - processes[idx].burst;

}

} else {

ganttChart += "idle "; // CPU is idle

}

time++;

}

// Print Gantt Chart

System.out.println("Gantt Chart: " + ganttChart);

// Calculate and print average waiting time and turnaround time

int totalWT = 0, totalTAT = 0;

System.out.println("Process\tArrival\tBurst\tPriority\tWaiting\tTurnaround");

for (Process p : processes) {

totalWT += p.waitingTime;

totalTAT += p.turnaroundTime;

System.out.println("P" + p.id + "\t" + p.arrival + "\t" + p.burst + "\t" + p.priority + "\t\t" + p.waitingTime + "\t" + p.turnaroundTime);

}

System.out.printf("Average Waiting Time: %.2f\n", totalWT / (float) n);

System.out.printf("Average Turnaround Time: %.2f\n", totalTAT / (float) n);

sc.close();

}

}

OUTPUT :

Enter number of processes: 5

Enter arrival time, burst time, and priority for process 1: 10 2 1

Enter arrival time, burst time, and priority for process 2: 10 2 2

Enter arrival time, burst time, and priority for process 3: 11 1 3

Enter arrival time, burst time, and priority for process 4: 13 3 4

Enter arrival time, burst time, and priority for process 5: 14 1 5

Gantt Chart: idle idle idle idle idle idle idle idle idle idle P1 P1 P2 P2 P3 P4 P4 P4 P5

Process Arrival Burst Priority Waiting Turnaround

P1 10 2 1 0 2

P2 10 2 2 2 4

P3 11 1 3 3 4

P4 13 3 4 2 5

P5 14 1 5 4 5

Average Waiting Time: 2.2

Average Turnaround Time: 4.0

#### Non-preemptive :

import java.util.Arrays;

import java.util.Comparator;

import java.util.Scanner;

class PriorityProcess {

int pid, burstTime, priority, waitingTime, turnaroundTime;

public PriorityProcess(int pid, int burstTime, int priority) {

this.pid = pid;

this.burstTime = burstTime;

this.priority = priority;

}

}

public class Main {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter number of processes: ");

int n = sc.nextInt();

PriorityProcess[] processes = new PriorityProcess[n];

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

System.out.print("Enter burst time and priority for process " + (i + 1) + ": ");

int bt = sc.nextInt();

int priority = sc.nextInt();

processes[i] = new PriorityProcess(i + 1, bt, priority);

}

// Sort processes by priority in ascending order (lower priority number = higher priority)

Arrays.sort(processes, Comparator.comparingInt(p -> p.priority));

int totalTime = 0, totalWT = 0, totalTAT = 0;

// Calculate waiting time and turnaround time for each process

for (PriorityProcess p : processes) {

p.waitingTime = totalTime;

totalTime += p.burstTime;

p.turnaroundTime = p.waitingTime + p.burstTime;

totalWT += p.waitingTime;

totalTAT += p.turnaroundTime;

}

// Print Gantt Chart

System.out.println("Gantt Chart: ");

for (PriorityProcess p : processes) {

System.out.print("P" + p.pid + " ");

}

System.out.println("\n");

// Print process details

System.out.println("Process\tBurst\tPriority\tWaiting\tTurnaround");

for (PriorityProcess p : processes) {

System.out.println("P" + p.pid + "\t" + p.burstTime + "\t" + p.priority + "\t\t" + p.waitingTime + "\t\t" + p.turnaroundTime);

}

System.out.printf("Average Waiting Time: %.2f\n", totalWT / (float) n);

System.out.printf("Average Turnaround Time: %.2f\n", totalTAT / (float) n);

sc.close();

}

}

OUTPUT :

Enter number of processes: 3

Enter burst time and priority for process 1: 10 2

Enter burst time and priority for process 2: 5 0

Enter burst time and priority for process 3: 8 1

Gantt Chart:

P2 P3 P1

Process Burst time Priority Waiting time Turnaround time

P2 5 0 0 5

P3 8 1 5 13

P1 10 2 13 23

Average Waiting Time: 6.0

Average Turnaround Time: 13.666667

## Experiment no. 3 (CPU Scheduling : Round Robin)

### Code :

#### Preemptive :

import java.util.Scanner;

public class RoundRobin {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter number of processes: ");

int n = sc.nextInt();

int[] burstTime = new int[n];

int[] remainingTime = new int[n];

int[] waitingTime = new int[n];

int[] turnaroundTime = new int[n];

int[] arrivalTime = new int[n];

System.out.print("Enter time quantum: ");

int quantum = sc.nextInt();

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

System.out.print("Enter arrival time and burst time for process " + (i + 1) + ": ");

arrivalTime[i] = sc.nextInt();

burstTime[i] = sc.nextInt();

remainingTime[i] = burstTime[i];

}

int currentTime = 0;

int completedProcesses = 0;

StringBuilder ganttChart = new StringBuilder();

while (completedProcesses < n) {

boolean processExecuted = false;

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

if (remainingTime[i] > 0 && arrivalTime[i] <= currentTime) {

processExecuted = true;

if (remainingTime[i] > quantum) {

currentTime += quantum;

remainingTime[i] -= quantum;

ganttChart.append("P").append(i + 1).append(" ");

} else {

currentTime += remainingTime[i];

ganttChart.append("P").append(i + 1).append(" ");

waitingTime[i] = currentTime - burstTime[i] - arrivalTime[i];

turnaroundTime[i] = waitingTime[i] + burstTime[i];

remainingTime[i] = 0;

completedProcesses++;

}

}

}

if (!processExecuted) {

currentTime++;

ganttChart.append("idle ");

}

}

// Output Gantt Chart

System.out.println("Gantt Chart: " + ganttChart);

// Calculate and output average waiting and turnaround time

float totalWT = 0, totalTAT = 0;

System.out.println("Process\tArrival\tBurst\tWaiting\tTurnaround");

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

totalWT += waitingTime[i];

totalTAT += turnaroundTime[i];

System.out.println("P" + (i + 1) + "\t" + arrivalTime[i] + "\t" + burstTime[i] + "\t" + waitingTime[i] + "\t" + turnaroundTime[i]);

}

System.out.printf("Average Waiting Time: %.2f\n", totalWT / n);

System.out.printf("Average Turnaround Time: %.2f\n", totalTAT / n);

sc.close();

}

}

OUTPUT :

Enter number of processes: 3

Enter time quantum: 5

Enter arrival time and burst time for process 1: 20 27

Enter arrival time and burst time for process 2: 3 8

Enter arrival time and burst time for process 3: 4 12

Gantt Chart: idle idle idle P2 P3 P2 P3 P1 P3 P1 P1 P1 P1 P1

Process Arrival Burst Waiting Turnaround

P1 20 27 3 30

P2 3 8 5 13

P3 4 12 12 24

Average Waiting Time: 6.6666665

Average Turnaround Time: 22.333334

#### Non-preemptive :

import java.util.Scanner;

public class RoundRobin {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter number of processes: ");

int n = sc.nextInt();

int[] burstTime = new int[n];

int[] remainingTime = new int[n];

int[] waitingTime = new int[n];

int[] turnaroundTime = new int[n];

System.out.print("Enter time quantum: ");

int quantum = sc.nextInt();

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

System.out.print("Enter burst time for process " + (i + 1) + ": ");

burstTime[i] = sc.nextInt();

remainingTime[i] = burstTime[i];

}

int time = 0, completed = 0;

StringBuilder ganttChart = new StringBuilder();

// Round Robin Execution Loop

while (completed < n) {

boolean processExecuted = false;

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

if (remainingTime[i] > 0) {

processExecuted = true;

if (remainingTime[i] > quantum) {

// Execute for the time quantum

time += quantum;

remainingTime[i] -= quantum;

ganttChart.append("P").append(i + 1).append(" ");

} else {

// Process completes

time += remainingTime[i];

remainingTime[i] = 0;

ganttChart.append("P").append(i + 1).append(" ");

completed++;

waitingTime[i] = time - burstTime[i];

turnaroundTime[i] = time;

}

}

}

// If no process was ready, indicate idle time

if (!processExecuted) {

ganttChart.append("idle ");

time++;

}

}

// Print Gantt Chart

System.out.println("Gantt Chart: " + ganttChart);

// Calculate and print average waiting and turnaround times

int totalWT = 0, totalTAT = 0;

System.out.println("Process\tBurst\tWaiting\tTurnaround");

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

totalWT += waitingTime[i];

totalTAT += turnaroundTime[i];

System.out.println("P" + (i + 1) + "\t" + burstTime[i] + "\t" + waitingTime[i] + "\t" + turnaroundTime[i]);

}

System.out.printf("Average Waiting Time: %.2f\n", totalWT / (float) n);

System.out.printf("Average Turnaround Time: %.2f\n", totalTAT / (float) n);

sc.close();

}

}

OUTPUT :

Enter number of processes: 5

Enter time quantum: 2

Enter burst time for process 1: 3

Enter burst time for process 2: 5

Enter burst time for process 3: 2

Enter burst time for process 4: 5

Enter burst time for process 5: 5

Gantt Chart: P1 P2 P3 P4 P5 P1 P2 P4 P5 P2 P4 P5

Process Burst Waiting Turnaround

P1 3 8 11

P2 5 13 18

P3 2 4 6

P4 5 14 19

P5 5 15 20

Average Waiting Time: 10.8

Average Turnaround Time: 14.8

## Experiment no. 3 (CPU Scheduling : SFJ )

### Code :

#### Preemptive :

import java.util.Scanner;

public class SJFPreemptive {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter the number of processes: ");

int n = sc.nextInt();

int[] bt = new int[n]; // Burst time

int[] at = new int[n]; // Arrival time

int[] rt = new int[n]; // Remaining time

int[] wt = new int[n]; // Waiting time

int[] tat = new int[n]; // Turnaround time

boolean[] completed = new boolean[n]; // Completion status of processes

System.out.println("Enter Arrival Time and Burst Time of the processes:");

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

System.out.print("P" + (i + 1) + ": ");

at[i] = sc.nextInt();

bt[i] = sc.nextInt();

rt[i] = bt[i]; // Initialize remaining time with burst time

}

int completedProcesses = 0, currentTime = 0, shortest = 0;

boolean found;

StringBuilder ganttChart = new StringBuilder(); // Using StringBuilder for efficient concatenation

while (completedProcesses < n) {

found = false;

// Find the process with the shortest remaining time that has arrived

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

if (!completed[i] && at[i] <= currentTime && (found == false || rt[i] < rt[shortest])) {

shortest = i;

found = true;

}

}

if (found) {

rt[shortest]--; // Execute for one unit of time

ganttChart.append("P").append(shortest + 1).append(" "); // Append to gantt chart

currentTime++; // Increase current time

if (rt[shortest] == 0) { // If process completes

completed[shortest] = true;

completedProcesses++;

tat[shortest] = currentTime - at[shortest]; // Turnaround time = completion time - arrival time

wt[shortest] = tat[shortest] - bt[shortest]; // Waiting time = turnaround time - burst time

}

} else {

currentTime++; // If no process is found to execute, increment time

ganttChart.append("idle "); // Append idle if no process is executing

}

}

// Output Gantt Chart

System.out.println("Gantt Chart: " + ganttChart);

// Calculate Average Waiting Time and Turnaround Time

float avgWT = 0, avgTAT = 0;

System.out.println("Process\tArrival\tBurst\tWaiting\tTurnaround");

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

avgWT += wt[i];

avgTAT += tat[i];

System.out.println("P" + (i + 1) + "\t" + at[i] + "\t" + bt[i] + "\t" + wt[i] + "\t" + tat[i]);

}

avgWT /= n; // Average waiting time

avgTAT /= n; // Average turnaround time

System.out.println("Average Waiting Time: " + avgWT);

System.out.println("Average Turnaround Time: " + avgTAT);

sc.close();

}

}

OUTPUT :

Enter the number of processes: 5

Enter Arrival Time and Burst Time of the processes:

P1: 2 6

P2: 5 2

P3: 1 8

P4: 0 3

P5: 4 4

Gantt Chart: P4 P4 P4 P1 P5 P2 P2 P5 P5 P5 P1 P1 P1 P1 P1 P3 P3 P3 P3 P3 P3 P3 P3

Process Arrival Burst Waiting Turnaround

P1 2 6 7 13

P2 5 2 0 2

P3 1 8 14 22

P4 0 3 0 3

P5 4 4 2 6

Average Waiting Time: 4.6

Average Turnaround Time: 9.2

#### Non-preemptive :

import java.util.Scanner;

class SJFNProcess {

int pid, burstTime, arrivalTime, waitingTime, turnaroundTime, completionTime;

public SJFNProcess(int pid, int burstTime, int arrivalTime) {

this.pid = pid;

this.burstTime = burstTime;

this.arrivalTime = arrivalTime;

}

}

public class SJFNonPreemptive {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter number of processes: ");

int n = sc.nextInt();

SJFNProcess[] processes = new SJFNProcess[n];

// Input process details

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

System.out.print("Enter arrival time and burst time for process " + (i + 1) + ": ");

int at = sc.nextInt();

int bt = sc.nextInt();

processes[i] = new SJFNProcess(i + 1, bt, at);

}

// Sort processes by arrival time, and then by burst time

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (processes[j].arrivalTime > processes[j + 1].arrivalTime ||

(processes[j].arrivalTime == processes[j + 1].arrivalTime && processes[j].burstTime > processes[j + 1].burstTime)) {

SJFNProcess temp = processes[j];

processes[j] = processes[j + 1];

processes[j + 1] = temp;

}

}

}

int totalTime = 0, totalWT = 0, totalTAT = 0;

for (SJFNProcess p : processes) {

// Waiting time: Current time minus arrival time (time when the process is scheduled to start)

p.waitingTime = totalTime - p.arrivalTime;

// Add burst time to total time to simulate the execution

totalTime += p.burstTime;

// Turnaround time = waiting time + burst time

p.turnaroundTime = p.waitingTime + p.burstTime;

// Add waiting time and turnaround time for averages

totalWT += p.waitingTime;

totalTAT += p.turnaroundTime;

}

// Print Gantt Chart

System.out.println("Gantt Chart: ");

for (SJFNProcess p : processes) {

System.out.print("P" + p.pid + " ");

}

System.out.println("\n");

// Print process details

System.out.println("Process\tArrival\tBurst\tWaiting\tTurnaround");

for (SJFNProcess p : processes) {

System.out.println("P" + p.pid + "\t" + p.arrivalTime + "\t" + p.burstTime + "\t" + p.waitingTime + "\t" + p.turnaroundTime);

}

System.out.println("Average Waiting Time: " + (totalWT / (float) n));

System.out.println("Average Turnaround Time: " + (totalTAT / (float) n));

sc.close();

}

}

OUTPUT :

Enter number of processes: 5

Enter arrival time and burst time for process 1: 2 6

Enter arrival time and burst time for process 2: 5 2

Enter arrival time and burst time for process 3: 1 8

Enter arrival time and burst time for process 4: 0 3

Enter arrival time and burst time for process 5: 4 4

Gantt Chart:

P2 P4 P5 P1 P3

Process Arrival Burst Waiting Turnaround

P2 5 2 -5 -3

P4 0 3 2 5

P5 4 4 1 5

P1 2 6 7 13

P3 1 8 14 22

Average Waiting Time: 3.8

Average Turnaround Time: 8.4

## Experiment no. 4 (Page Replacement : FIFO)

### Code :

import java.util.\*;

public class Fifo {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter the number of frames: ");

int numberOfFrames = scanner.nextInt();

System.out.print("Enter the number of pages: ");

int numberOfPages = scanner.nextInt();

System.out.print("Enter the page reference string (space-separated): ");

int[] pageReferenceString = new int[numberOfPages];

for (int i = 0; i < numberOfPages; i++) {

pageReferenceString[i] = scanner.nextInt();

}

int[] frames = new int[numberOfFrames];

Arrays.fill(frames, -1);

int pageFaults = 0;

int currentIndex = 0;

System.out.println("Page Reference String: " + Arrays.toString(pageReferenceString));

for (int i = 0; i < numberOfPages; i++) {

int page = pageReferenceString[i];

boolean pageHit = false;

// Check if page is already in one of the frames

for (int frame : frames) {

if (frame == page) {

pageHit = true;

break;

}

}

// If page is not found in frames, it's a page fault

if (!pageHit) {

frames[currentIndex] = page; // Replace the page

currentIndex = (currentIndex + 1) % numberOfFrames; // Move to the next frame

pageFaults++;

}

// Print the current state of the frames

System.out.print("Frames: ");

for (int frame : frames) {

System.out.print(frame + " ");

}

System.out.println();

}

// Calculate page fault ratio

double pageFaultRatio = (double) pageFaults / numberOfPages;

System.out.println("Total Page Faults: " + pageFaults);

System.out.println("Page Fault Ratio: " + String.format("%.2f", pageFaultRatio));

scanner.close();

}

}

OUTPUT :

Enter the number of frames: 3

Enter the number of pages: 7

Enter the page reference string (space-separated): 1 3 0 3 5 6 3

Frames: 1 -1 -1

Frames: 1 3 -1

Frames: 1 3 0

Frames: 1 3 0

Frames: 5 3 0

Frames: 5 6 0

Frames: 5 6 3

Total Page Faults: 6

Page Faults ratio: 6:7

## Experiment no. 4 (Page Replacement : LRU)

### Code :

import java.util.\*;

public class Lru {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter the number of frames: ");

int numberOfFrames = scanner.nextInt();

System.out.print("Enter the number of pages: ");

int numberOfPages = scanner.nextInt();

System.out.print("Enter the page reference string (space-separated): ");

int[] pageReferenceString = new int[numberOfPages];

for (int i = 0; i < numberOfPages; i++) {

pageReferenceString[i] = scanner.nextInt();

}

LinkedList<Integer> frames = new LinkedList<>();

int pageFaults = 0;

for (int page : pageReferenceString) {

if (!frames.contains(page)) {

// Page fault occurs, add page to the frames

if (frames.size() >= numberOfFrames) {

frames.removeFirst(); // Remove least recently used page

}

frames.addLast(page);

pageFaults++;

} else {

// Page is already in memory, move it to the end (most recently used)

frames.remove(frames.indexOf(page));

frames.addLast(page);

}

// Print the current state of frames

System.out.print("Frames: ");

for (int frame : frames) {

System.out.print(frame + " ");

}

System.out.println();

}

// Calculate and print the page fault ratio

double pageFaultRatio = (double) pageFaults / numberOfPages;

System.out.println("Total Page Faults: " + pageFaults);

System.out.println("Page Fault Ratio: " + String.format("%.2f", pageFaultRatio));

scanner.close();

}

}

OUTPUT:

Enter the number of frames: 3

Enter the number of pages: 12

Enter the page reference string (space-separated): 2 3 2 1 5 2 4 5 3 2 5 2

Frames: 2

Frames: 2 3

Frames: 3 2

Frames: 3 2 1

Frames: 2 1 5

Frames: 1 5 2

Frames: 5 2 4

Frames: 2 4 5

Frames: 4 5 3

Frames: 5 3 2

Frames: 3 2 5

Frames: 3 5 2

Total Page Faults: 7

Page Fault ratio: 7:12

## Experiment no. 4 (Page Replacement : Optimal)

### Code :

import java.util.Scanner;

import java.io.IOException;

public class Optimal

{

public static void main(String[] args) throws IOException

{

Scanner in = new Scanner(System.in);

int frames = 0;

int pointer = 0;

int numFault = 0;

int ref\_len;

boolean isFull = false;

int[] buffer;

boolean[] hit;

int[] fault;

int[] reference;

int[][] mem\_layout;

System.out.println("Please enter the number of frames: ");

frames = Integer.parseInt(in.nextLine());

System.out.println("Please enter the length of the reference string: ");

ref\_len = Integer.parseInt(in.nextLine());

reference = new int[ref\_len];

mem\_layout = new int[ref\_len][frames];

buffer = new int[frames];

hit = new boolean[ref\_len];

fault = new int[ref\_len];

for (int j = 0; j < frames; j++) {

buffer[j] = -1;

}

System.out.println("Please enter the reference string (hit Enter/Return after each number in the string): ");

for (int i = 0; i < ref\_len; i++) {

reference[i] = Integer.parseInt(in.nextLine());

}

System.out.println();

for (int i = 0; i < ref\_len; i++) {

int search = -1;

// Check if the current page is already in memory

for (int j = 0; j < frames; j++) {

if (buffer[j] == reference[i]) {

search = j;

hit[i] = true;

fault[i] = numFault;

break;

}

}

if (search == -1) { // Page fault occurs

if (isFull) {

// Find which page to replace using the Optimal algorithm

int[] index = new int[frames];

boolean[] index\_flag = new boolean[frames];

for (int j = i + 1; j < ref\_len; j++) {

for (int k = 0; k < frames; k++) {

if (reference[j] == buffer[k] && !index\_flag[k]) {

index[k] = j;

index\_flag[k] = true;

break;

}

}

}

int max = index[0];

pointer = 0;

if (max == 0) {

max = Integer.MAX\_VALUE; // Set a large value if no future reference

}

// Find the page with the furthest future reference

for (int j = 0; j < frames; j++) {

if (index[j] == 0) {

index[j] = Integer.MAX\_VALUE;

}

if (index[j] > max) {

max = index[j];

pointer = j;

}

}

}

buffer[pointer] = reference[i];

numFault++;

fault[i] = numFault;

if (!isFull) {

pointer++;

if (pointer == frames) {

pointer = 0;

isFull = true;

}

}

}

// Update memory layout

for (int j = 0; j < frames; j++) {

mem\_layout[i][j] = buffer[j];

}

}

// Display the memory layout and page faults

for (int i = 0; i < ref\_len; i++) {

System.out.print(reference[i] + ": Memory is: ");

for (int j = 0; j < frames; j++) {

if (mem\_layout[i][j] == -1) {

System.out.printf("%3s ", "\*");

} else {

System.out.printf("%3d ", mem\_layout[i][j]);

}

}

System.out.print(": ");

if (hit[i]) {

System.out.print("Hit");

} else {

System.out.print("Page Fault");

}

System.out.print(": (Number of Page Faults: " + fault[i] + ")");

System.out.println();

}

// Final results

System.out.println("Total Number of Page Faults: " + numFault);

double pageFaultRatio = (double) numFault / ref\_len;

System.out.println("Page Faults ratio: " + String.format("%.2f", pageFaultRatio));

in.close();

}

}

OUTPUT :

Please enter the number of frames:

3

Please enter the length of the reference string:

12

Please enter the reference string (hit Enter/Return after each number in the string):

2

3

2

1

5

2

4

5

3

2

5

2

2: Memory is: 2 \* \* : Page Fault: (Number of Page Faults: 1)

3: Memory is: 2 3 \* : Page Fault: (Number of Page Faults: 2)

2: Memory is: 2 3 \* : Hit: (Number of Page Faults: 2)

1: Memory is: 2 3 1 : Page Fault: (Number of Page Faults: 3)

5: Memory is: 2 3 5 : Page Fault: (Number of Page Faults: 4)

2: Memory is: 2 3 5 : Hit: (Number of Page Faults: 4)

4: Memory is: 4 3 5 : Page Fault: (Number of Page Faults: 5)

5: Memory is: 4 3 5 : Hit: (Number of Page Faults: 5)

3: Memory is: 4 3 5 : Hit: (Number of Page Faults: 5)

2: Memory is: 2 3 5 : Page Fault: (Number of Page Faults: 6)

5: Memory is: 2 3 5 : Hit: (Number of Page Faults: 6)

2: Memory is: 2 3 5 : Hit: (Number of Page Faults: 6)

Total Number of Page Faults: 6

Page Faults ratio : 6:12