**Design suitable Data structures and implement Pass-I of a two-pass assembler for pseudo-machine. Implementation should consist of a few instructions from each category and few assembler directives.**

import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;

import java.util.HashMap;

import java.util.Map;

class SymbolTableEntry {

String name;

int address;

public SymbolTableEntry(String name, int address) {

this.name = name;

this.address = address;

}

}

class LiteralTableEntry {

String value;

int address;

public LiteralTableEntry(String value, int address) {

this.value = value;

this.address = address;

}

}

class TwoPassAssembler {

private Map<String, SymbolTableEntry> symbolTable = new HashMap<>();

private Map<String, LiteralTableEntry> literalTable = new HashMap<>();

private int locationCounter = 0;

public void processPass1(String fileName) {

try (BufferedReader br = new BufferedReader(new FileReader(fileName))) {

String line;

while ((line = br.readLine()) != null) {

String[] tokens = line.split("\\s+");

String opcode = tokens[0];

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

locationCounter = Integer.parseInt(tokens[1], 16);

} else if (opcode.equals("END")) {

break;

} else if (opcode.equals("LTORG")) {

processLtorg();

} else if (!opcode.equals("")) {

processOpcode(opcode, tokens);

}

}

} catch (IOException e) {

e.printStackTrace();

}

}

private void processOpcode(String opcode, String[] tokens) {

// Handle different opcodes and update symbol table, literal table, etc.

// This is a placeholder and needs to be expanded based on the specific instruction set.

// You may need to handle different addressing modes, operand types, etc.

// For simplicity, let's assume each instruction occupies one memory location.

symbolTable.put(tokens[1], new SymbolTableEntry(tokens[1], locationCounter));

locationCounter++;

}

private void processLtorg() {

// Process literals and update literal table

// This is a placeholder and needs to be expanded based on the specific requirements.

}

public void printSymbolTable() {

System.out.println("Symbol Table:");

for (SymbolTableEntry entry : symbolTable.values()) {

System.out.println(entry.name + ": " + entry.address);

}

}

public void printLiteralTable() {

System.out.println("Literal Table:");

for (LiteralTableEntry entry : literalTable.values()) {

System.out.println(entry.value + ": " + entry.address);

}

}

// Example usage

public static void main(String[] args) {

TwoPassAssembler assembler = new TwoPassAssembler();

assembler.processPass1("source\_code.asm");

assembler.printSymbolTable();

assembler.printLiteralTable();

}

}

**Design suitable Data structures and implement Pass-II of a two-pass assembler for pseudo-machine. Implementation should consist of a few instructions from each category and few assembler directives. Intermediate code file and symbol table should be input for Pass-II.**

import java.util.HashMap;

import java.util.List;

import java.util.Map;

class SymbolTableEntry {

String name;

int address;

public SymbolTableEntry(String name, int address) {

this.name = name;

this.address = address;

}

}

class Instruction {

String opcode;

List<String> operands;

public Instruction(String opcode, List<String> operands) {

this.opcode = opcode;

this.operands = operands;

}

}

class AssemblerDirective {

String directive;

String value;

public AssemblerDirective(String directive, String value) {

this.directive = directive;

this.value = value;

}

}

import java.util.ArrayList;

public class PassTwoAssembler {

public static void main(String[] args) {

// Example symbol table from Pass-I

Map<String, SymbolTableEntry> symbolTable = new HashMap<>();

symbolTable.put("A", new SymbolTableEntry("A", 100));

symbolTable.put("B", new SymbolTableEntry("B", 200));

// Example intermediate code from Pass-I

List<Object> intermediateCode = new ArrayList<>();

intermediateCode.add(new Instruction("ADD", List.of("A", "B")));

intermediateCode.add(new AssemblerDirective("ORIGIN", "300"));

intermediateCode.add(new Instruction("SUB", List.of("A", "100")));

intermediateCode.add(new AssemblerDirective("EQU", "C 500"));

// Pass-II

List<Integer> machineCode = passTwo(symbolTable, intermediateCode);

System.out.println(machineCode);

}

public static List<Integer> passTwo(Map<String, SymbolTableEntry> symbolTable, List<Object> intermediateCode) {

List<Integer> machineCode = new ArrayList<>();

for (Object line : intermediateCode) {

if (line instanceof Instruction) {

machineCode.addAll(generateMachineCode((Instruction) line, symbolTable));

} else if (line instanceof AssemblerDirective) {

handleDirective((AssemblerDirective) line, symbolTable);

}

}

return machineCode;

}

public static List<Integer> generateMachineCode(Instruction instruction, Map<String, SymbolTableEntry> symbolTable) {

List<Integer> machineCode = new ArrayList<>();

String opcode = instruction.opcode;

List<String> operands = instruction.operands;

// Replace symbols with addresses from the symbol table

for (String operand : operands) {

if (symbolTable.containsKey(operand)) {

operands.set(operands.indexOf(operand), String.valueOf(symbolTable.get(operand).address));

}

}

// Generate machine code based on the opcode and operands

// This is a simplified example; you would need to replace it with the actual logic

machineCode.add(getOpcodeValue(opcode));

for (String operand : operands) {

machineCode.add(Integer.parseInt(operand));

}

return machineCode;

}

public static void handleDirective(AssemblerDirective directive, Map<String, SymbolTableEntry> symbolTable) {

// Handle assembler directives (e.g., ORIGIN, EQU, etc.)

// This is a simplified example; you would need to replace it with the actual logic

if (directive.directive.equals("ORIGIN")) {

// Set the current address to the specified value

int address = Integer.parseInt(directive.value);

// You need to implement the logic to update the current address

} else if (directive.directive.equals("EQU")) {

// Add or update symbol table entry with the specified value

String[] parts = directive.value.split(" ");

String symbolName = parts[0];

int symbolValue = Integer.parseInt(parts[1]);

symbolTable.put(symbolName, new SymbolTableEntry(symbolName, symbolValue));

}

}

public static int getOpcodeValue(String opcode) {

// Replace this with the actual opcode-to-value mapping

// For simplicity, assume a direct mapping for this example

return Integer.parseInt(opcode);

}

}

**Design suitable data structures and implement Pass-I of a two-pass macro processor.**

import java.util.HashMap;

import java.util.Map;

class Macro {

String name;

String definition;

int parameterCount;

public Macro(String name, String definition, int parameterCount) {

this.name = name;

this.definition = definition;

this.parameterCount = parameterCount;

}

}

public class PassOneMacroProcessor {

public static void main(String[] args) {

// Example source code

String sourceCode = "MACRO ADD A, B\n" +

" LDA A\n" +

" ADD B\n" +

" STA RESULT\n" +

"MEND\n" +

"ADD X, Y";

// Pass-I

Map<String, Macro> macroTable = new HashMap<>();

processPassOne(sourceCode, macroTable);

// Print the macro table

System.out.println("Macro Table:");

for (Macro macro : macroTable.values()) {

System.out.println("Name: " + macro.name + ", Definition: " + macro.definition +

", Parameter Count: " + macro.parameterCount);

}

}

public static void processPassOne(String sourceCode, Map<String, Macro> macroTable) {

String[] lines = sourceCode.split("\n");

for (String line : lines) {

if (line.startsWith("MACRO")) {

// Process macro definition

String[] parts = line.split(" ");

String macroName = parts[1];

int parameterCount = parts.length - 2;

StringBuilder macroDefinition = new StringBuilder();

// Read the macro definition until MEND

int i = Arrays.asList(parts).indexOf("MACRO") + 1;

while (i < lines.length && !lines[i].trim().equals("MEND")) {

macroDefinition.append(lines[i]).append("\n");

i++;

}

// Add the macro to the macro table

Macro macro = new Macro(macroName, macroDefinition.toString().trim(), parameterCount);

macroTable.put(macroName, macro);

}

}

}

}

**Design suitable data structures and implement Pass-II of a two-pass macro processor. MNT, MDT and intermediate code file without any macro definitions should be input for Pass-II.**

import java.util.HashMap;

import java.util.Map;

class Macro {

String name;

int index;

public Macro(String name, int index) {

this.name = name;

this.index = index;

}

}

public class PassTwoMacroProcessor {

public static void main(String[] args) {

// Example Intermediate Code without macro definitions

String intermediateCode = "START\n" +

"LDA A\n" +

"ADD B\n" +

"STA RESULT\n" +

"STOP";

// Example Macro Name Table (MNT)

Map<String, Macro> mnt = new HashMap<>();

mnt.put("ADD", new Macro("ADD", 1));

// Example Macro Definition Table (MDT)

Map<Integer, String> mdt = new HashMap<>();

mdt.put(1, "LDA &1\nADD &2\nSTA &3");

// Pass-II

String expandedCode = processPassTwo(intermediateCode, mnt, mdt);

System.out.println(expandedCode);

}

public static String processPassTwo(String intermediateCode, Map<String, Macro> mnt, Map<Integer, String> mdt) {

StringBuilder expandedCode = new StringBuilder();

String[] lines = intermediateCode.split("\n");

for (String line : lines) {

String[] words = line.split(" ");

// Check if the opcode is a macro

if (mnt.containsKey(words[0])) {

Macro macro = mnt.get(words[0]);

String[] arguments = words[1].split(",");

// Check if the number of arguments matches the expected count

if (arguments.length != mdt.get(macro.index).split("&").length - 1) {

System.out.println("Error: Incorrect number of arguments for macro " + macro.name);

continue;

}

// Expand the macro by replacing parameters with arguments

String expandedMacro = mdt.get(macro.index);

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

expandedMacro = expandedMacro.replace("&" + (i + 1), arguments[i]);

}

// Append the expanded macro to the result

expandedCode.append(expandedMacro).append("\n");

} else {

// If not a macro, append the line as is

expandedCode.append(line).append("\n");

}

}

return expandedCode.toString();

}

}

**Write a program to simulate CPU Scheduling Algorithms: Priority (Preemptive) and Round Robin (Preemptive)**

import java.util.ArrayList;

import java.util.Comparator;

import java.util.LinkedList;

import java.util.List;

import java.util.Queue;

class Process {

int id;

int arrivalTime;

int burstTime;

int priority;

public Process(int id, int arrivalTime, int burstTime, int priority) {

this.id = id;

this.arrivalTime = arrivalTime;

this.burstTime = burstTime;

this.priority = priority;

}

}

public class CPUSchedulingSimulation {

public static void main(String[] args) {

// Example processes

List<Process> processes = new ArrayList<>();

processes.add(new Process(1, 0, 5, 3));

processes.add(new Process(2, 2, 7, 2));

processes.add(new Process(3, 4, 2, 1));

processes.add(new Process(4, 6, 8, 4));

// Priority (Preemptive) Scheduling Simulation

priorityPreemptive(processes);

// Round Robin (Preemptive) Scheduling Simulation

roundRobinPreemptive(processes, 3);

}

public static void priorityPreemptive(List<Process> processes) {

processes.sort(Comparator.comparingInt(p -> p.arrivalTime));

List<Process> priorityQueue = new ArrayList<>();

int currentTime = 0;

int totalExecutionTime = 0;

while (!processes.isEmpty() || !priorityQueue.isEmpty()) {

// Add arriving processes to the priority queue

while (!processes.isEmpty() && processes.get(0).arrivalTime <= currentTime) {

priorityQueue.add(processes.remove(0));

}

if (!priorityQueue.isEmpty()) {

// Find the process with the highest priority

Process currentProcess = priorityQueue.stream()

.min(Comparator.comparingInt(p -> p.priority))

.orElse(null);

if (currentProcess != null) {

// Execute the process for one time unit

System.out.println("Time " + currentTime + ": Process " + currentProcess.id + " is running");

currentProcess.burstTime--;

// Remove the process if it is complete

if (currentProcess.burstTime == 0) {

priorityQueue.remove(currentProcess);

totalExecutionTime += currentTime - currentProcess.arrivalTime + 1;

}

}

}

currentTime++;

}

double averageTurnaroundTime = (double) totalExecutionTime / priorityQueue.size();

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

}

public static void roundRobinPreemptive(List<Process> processes, int timeQuantum) {

processes.sort(Comparator.comparingInt(p -> p.arrivalTime));

Queue<Process> roundRobinQueue = new LinkedList<>();

int currentTime = 0;

int totalExecutionTime = 0;

while (!processes.isEmpty() || !roundRobinQueue.isEmpty()) {

// Add arriving processes to the round robin queue

while (!processes.isEmpty() && processes.get(0).arrivalTime <= currentTime) {

roundRobinQueue.add(processes.remove(0));

}

if (!roundRobinQueue.isEmpty()) {

Process currentProcess = roundRobinQueue.poll();

System.out.println("Time " + currentTime + ": Process " + currentProcess.id + " is running");

currentProcess.burstTime--;

// Remove the process if it is complete

if (currentProcess.burstTime == 0) {

totalExecutionTime += currentTime - currentProcess.arrivalTime + 1;

} else {

// Add the process back to the queue if it still has burst time left

roundRobinQueue.add(currentProcess);

}

}

currentTime = Math.min(currentTime + timeQuantum, processes.isEmpty() ? Integer.MAX\_VALUE : processes.get(0).arrivalTime);

}

double averageTurnaroundTime = (double) totalExecutionTime / processes.size();

System.out.println("Average Turnaround Time (Round Robin): " + averageTurnaroundTime);

}

}

**Write a program to simulate CPU Scheduling Algorithms: Priority (Preemptive) and Round Robin (Preemptive).**

import java.util.ArrayList;

import java.util.Comparator;

import java.util.LinkedList;

import java.util.List;

import java.util.Queue;

class Process {

int id;

int arrivalTime;

int burstTime;

int priority;

public Process(int id, int arrivalTime, int burstTime, int priority) {

this.id = id;

this.arrivalTime = arrivalTime;

this.burstTime = burstTime;

this.priority = priority;

}

}

public class CPUSchedulingSimulation {

public static void main(String[] args) {

// Example processes

List<Process> processes = new ArrayList<>();

processes.add(new Process(1, 0, 5, 3));

processes.add(new Process(2, 2, 7, 2));

processes.add(new Process(3, 4, 2, 1));

processes.add(new Process(4, 6, 8, 4));

// Priority (Preemptive) Scheduling Simulation

priorityPreemptive(processes);

// Round Robin (Preemptive) Scheduling Simulation

roundRobinPreemptive(processes, 3);

}

public static void priorityPreemptive(List<Process> processes) {

processes.sort(Comparator.comparingInt(p -> p.arrivalTime));

List<Process> priorityQueue = new ArrayList<>();

int currentTime = 0;

int totalExecutionTime = 0;

while (!processes.isEmpty() || !priorityQueue.isEmpty()) {

// Add arriving processes to the priority queue

while (!processes.isEmpty() && processes.get(0).arrivalTime <= currentTime) {

priorityQueue.add(processes.remove(0));

}

if (!priorityQueue.isEmpty()) {

// Find the process with the highest priority

Process currentProcess = priorityQueue.stream()

.min(Comparator.comparingInt(p -> p.priority))

.orElse(null);

if (currentProcess != null) {

// Execute the process for one time unit

System.out.println("Time " + currentTime + ": Process " + currentProcess.id + " is running");

currentProcess.burstTime--;

// Remove the process if it is complete

if (currentProcess.burstTime == 0) {

priorityQueue.remove(currentProcess);

totalExecutionTime += currentTime - currentProcess.arrivalTime + 1;

}

}

}

currentTime++;

}

double averageTurnaroundTime = (double) totalExecutionTime / priorityQueue.size();

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

}

public static void roundRobinPreemptive(List<Process> processes, int timeQuantum) {

processes.sort(Comparator.comparingInt(p -> p.arrivalTime));

Queue<Process> roundRobinQueue = new LinkedList<>();

int currentTime = 0;

int totalExecutionTime = 0;

while (!processes.isEmpty() || !roundRobinQueue.isEmpty()) {

// Add arriving processes to the round robin queue

while (!processes.isEmpty() && processes.get(0).arrivalTime <= currentTime) {

roundRobinQueue.add(processes.remove(0));

}

if (!roundRobinQueue.isEmpty()) {

Process currentProcess = roundRobinQueue.poll();

System.out.println("Time " + currentTime + ": Process " + currentProcess.id + " is running");

currentProcess.burstTime--;

// Remove the process if it is complete

if (currentProcess.burstTime == 0) {

totalExecutionTime += currentTime - currentProcess.arrivalTime + 1;

} else {

// Add the process back to the queue if it still has burst time left

roundRobinQueue.add(currentProcess);

}

}

currentTime = Math.min(currentTime + timeQuantum, processes.isEmpty() ? Integer.MAX\_VALUE : processes.get(0).arrivalTime);

}

double averageTurnaroundTime = (double) totalExecutionTime / processes.size();

System.out.println("Average Turnaround Time (Round Robin): " + averageTurnaroundTime);

}

}

**Write a program to simulate CPU Scheduling Algorithms: FCFS(Non-Preemptive), SJF (non -Preemptive)**

import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

import java.util.List;

class Process {

int id;

int arrivalTime;

int burstTime;

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

this.id = id;

this.arrivalTime = arrivalTime;

this.burstTime = burstTime;

}

}

public class CPUSchedulingSimulation {

public static void main(String[] args) {

// Example processes

List<Process> processes = new ArrayList<>();

processes.add(new Process(1, 0, 5));

processes.add(new Process(2, 2, 7));

processes.add(new Process(3, 4, 2));

processes.add(new Process(4, 6, 8));

// FCFS (Non-Preemptive) Scheduling Simulation

fcfsNonPreemptive(processes);

// SJF (Non-Preemptive) Scheduling Simulation

sjfNonPreemptive(processes);

}

public static void fcfsNonPreemptive(List<Process> processes) {

processes.sort(Comparator.comparingInt(p -> p.arrivalTime));

int currentTime = 0;

int totalExecutionTime = 0;

for (Process process : processes) {

// Wait until the process arrives (if necessary)

currentTime = Math.max(currentTime, process.arrivalTime);

// Execute the process

System.out.println("Time " + currentTime + ": Process " + process.id + " is running");

totalExecutionTime += currentTime - process.arrivalTime + process.burstTime;

// Move the current time to the end of the process execution

currentTime += process.burstTime;

}

double averageTurnaroundTime = (double) totalExecutionTime / processes.size();

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

}

public static void sjfNonPreemptive(List<Process> processes) {

processes.sort(Comparator.comparingInt(p -> p.arrivalTime));

List<Process> waitingQueue = new ArrayList<>(processes);

int currentTime = 0;

int totalExecutionTime = 0;

while (!waitingQueue.isEmpty()) {

// Select the process with the shortest burst time

Process shortestJob = Collections.min(waitingQueue, Comparator.comparingInt(p -> p.burstTime));

// Wait until the process arrives (if necessary)

currentTime = Math.max(currentTime, shortestJob.arrivalTime);

// Execute the process

System.out.println("Time " + currentTime + ": Process " + shortestJob.id + " is running");

totalExecutionTime += currentTime - shortestJob.arrivalTime + shortestJob.burstTime;

// Move the current time to the end of the process execution

currentTime += shortestJob.burstTime;

// Remove the executed process from the waiting queue

waitingQueue.remove(shortestJob);

}

double averageTurnaroundTime = (double) totalExecutionTime / processes.size();

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

}

}

**Write a program to simulate CPU Scheduling Algorithms: FCFS, SJF (Preemptive), Priority (Non-Preemptive) and Round Robin (Preemptive)**

import java.util.\*;

class Process {

int id;

int arrivalTime;

int burstTime;

int priority;

int remainingBurstTime;

public Process(int id, int arrivalTime, int burstTime, int priority) {

this.id = id;

this.arrivalTime = arrivalTime;

this.burstTime = burstTime;

this.priority = priority;

this.remainingBurstTime = burstTime;

}

}

public class CPUSchedulingSimulation {

public static void main(String[] args) {

// Example processes

List<Process> processes = new ArrayList<>();

processes.add(new Process(1, 0, 5, 3));

processes.add(new Process(2, 2, 7, 2));

processes.add(new Process(3, 4, 2, 1));

processes.add(new Process(4, 6, 8, 4));

// FCFS (Non-Preemptive) Scheduling Simulation

fcfsNonPreemptive(new ArrayList<>(processes));

// SJF (Preemptive) Scheduling Simulation

sjfPreemptive(new ArrayList<>(processes));

// Priority (Non-Preemptive) Scheduling Simulation

priorityNonPreemptive(new ArrayList<>(processes));

// Round Robin (Preemptive) Scheduling Simulation

roundRobinPreemptive(new ArrayList<>(processes), 3);

}

public static void fcfsNonPreemptive(List<Process> processes) {

processes.sort(Comparator.comparingInt(p -> p.arrivalTime));

int currentTime = 0;

int totalExecutionTime = 0;

for (Process process : processes) {

// Wait until the process arrives (if necessary)

currentTime = Math.max(currentTime, process.arrivalTime);

// Execute the process

System.out.println("Time " + currentTime + ": Process " + process.id + " is running");

totalExecutionTime += currentTime - process.arrivalTime + process.burstTime;

// Move the current time to the end of the process execution

currentTime += process.burstTime;

}

double averageTurnaroundTime = (double) totalExecutionTime / processes.size();

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

}

public static void sjfPreemptive(List<Process> processes) {

processes.sort(Comparator.comparingInt(p -> p.arrivalTime));

PriorityQueue<Process> priorityQueue = new PriorityQueue<>(Comparator.comparingInt(p -> p.remainingBurstTime));

int currentTime = 0;

int totalExecutionTime = 0;

while (!processes.isEmpty() || !priorityQueue.isEmpty()) {

// Add arriving processes to the priority queue

while (!processes.isEmpty() && processes.get(0).arrivalTime <= currentTime) {

priorityQueue.add(processes.remove(0));

}

if (!priorityQueue.isEmpty()) {

Process currentProcess = priorityQueue.poll();

System.out.println("Time " + currentTime + ": Process " + currentProcess.id + " is running");

currentProcess.remainingBurstTime--;

// Remove the process if it is complete

if (currentProcess.remainingBurstTime == 0) {

totalExecutionTime += currentTime - currentProcess.arrivalTime + 1;

} else {

// Add the process back to the queue if it still has burst time left

priorityQueue.add(currentProcess);

}

}

currentTime++;

}

double averageTurnaroundTime = (double) totalExecutionTime / processes.size();

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

}

public static void priorityNonPreemptive(List<Process> processes) {

processes.sort(Comparator.comparingInt(p -> p.arrivalTime));

PriorityQueue<Process> priorityQueue = new PriorityQueue<>(Comparator.comparingInt(p -> p.priority));

int currentTime = 0;

int totalExecutionTime = 0;

while (!processes.isEmpty() || !priorityQueue.isEmpty()) {

// Add arriving processes to the priority queue

while (!processes.isEmpty() && processes.get(0).arrivalTime <= currentTime) {

priorityQueue.add(processes.remove(0));

}

if (!priorityQueue.isEmpty()) {

Process currentProcess = priorityQueue.poll();

System.out.println("Time " + currentTime + ": Process " + currentProcess.id + " is running");

totalExecutionTime += currentTime - currentProcess.arrivalTime + currentProcess.burstTime;

currentTime += currentProcess.burstTime;

} else {

currentTime++;

}

}

double averageTurnaroundTime = (double) totalExecutionTime / processes.size();

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

}

public static void roundRobinPreemptive(List<Process> processes, int timeQuantum) {

Queue<Process> roundRobinQueue = new LinkedList<>(processes);

int currentTime = 0;

int totalExecutionTime = 0;

while (!roundRobinQueue.isEmpty()) {

Process currentProcess = roundRobinQueue.poll();

System.out.println("Time " + currentTime + ": Process " + currentProcess.id + " is running");

int remainingBurstTime = Math.min(currentProcess.remainingBurstTime, timeQuantum);

currentProcess.remainingBurstTime -= remainingBurstTime;

currentTime += remainingBurstTime;

// Move the current time to the end of the process execution

if (currentProcess.remainingBurstTime > 0) {

roundRobinQueue.add(currentProcess);

} else {

totalExecutionTime += currentTime - currentProcess.arrivalTime;

}

}

double averageTurnaroundTime = (double) totalExecutionTime / processes.size();

System.out.println("Average Turnaround Time (Round Robin): " + averageTurnaroundTime);

}

}

**Write a program to simulate Memory placement strategies – best fit, first fit, next fit and worst fit**

import java.util.\*;

class MemoryBlock {

int id;

int size;

boolean allocated;

public MemoryBlock(int id, int size) {

this.id = id;

this.size = size;

this.allocated = false;

}

}

public class MemoryPlacementSimulation {

public static void main(String[] args) {

// Example memory blocks

List<MemoryBlock> memoryBlocks = new ArrayList<>();

memoryBlocks.add(new MemoryBlock(1, 100));

memoryBlocks.add(new MemoryBlock(2, 50));

memoryBlocks.add(new MemoryBlock(3, 200));

memoryBlocks.add(new MemoryBlock(4, 80));

memoryBlocks.add(new MemoryBlock(5, 120));

// Example processes

List<Integer> processSizes = Arrays.asList(60, 100, 150, 80, 200);

// Best Fit Simulation

System.out.println("Best Fit:");

memoryPlacementBestFit(new ArrayList<>(memoryBlocks), new ArrayList<>(processSizes));

// First Fit Simulation

System.out.println("\nFirst Fit:");

memoryPlacementFirstFit(new ArrayList<>(memoryBlocks), new ArrayList<>(processSizes));

// Next Fit Simulation

System.out.println("\nNext Fit:");

memoryPlacementNextFit(new ArrayList<>(memoryBlocks), new ArrayList<>(processSizes));

// Worst Fit Simulation

System.out.println("\nWorst Fit:");

memoryPlacementWorstFit(new ArrayList<>(memoryBlocks), new ArrayList<>(processSizes));

}

public static void memoryPlacementBestFit(List<MemoryBlock> memoryBlocks, List<Integer> processSizes) {

for (int processSize : processSizes) {

MemoryBlock bestFitBlock = null;

for (MemoryBlock block : memoryBlocks) {

if (!block.allocated && block.size >= processSize) {

if (bestFitBlock == null || block.size < bestFitBlock.size) {

bestFitBlock = block;

}

}

}

if (bestFitBlock != null) {

allocateMemory(bestFitBlock, processSize);

} else {

System.out.println("Process with size " + processSize + " cannot be allocated.");

}

printMemoryStatus(memoryBlocks);

}

}

public static void memoryPlacementFirstFit(List<MemoryBlock> memoryBlocks, List<Integer> processSizes) {

for (int processSize : processSizes) {

for (MemoryBlock block : memoryBlocks) {

if (!block.allocated && block.size >= processSize) {

allocateMemory(block, processSize);

break;

}

}

printMemoryStatus(memoryBlocks);

}

}

public static void memoryPlacementNextFit(List<MemoryBlock> memoryBlocks, List<Integer> processSizes) {

int lastAllocatedBlockIndex = 0;

for (int processSize : processSizes) {

MemoryBlock block = memoryBlocks.get(lastAllocatedBlockIndex);

while (lastAllocatedBlockIndex < memoryBlocks.size() && (block.allocated || block.size < processSize)) {

lastAllocatedBlockIndex++;

if (lastAllocatedBlockIndex < memoryBlocks.size()) {

block = memoryBlocks.get(lastAllocatedBlockIndex);

}

}

if (lastAllocatedBlockIndex < memoryBlocks.size()) {

allocateMemory(block, processSize);

} else {

System.out.println("Process with size " + processSize + " cannot be allocated.");

}

printMemoryStatus(memoryBlocks);

}

}

public static void memoryPlacementWorstFit(List<MemoryBlock> memoryBlocks, List<Integer> processSizes) {

for (int processSize : processSizes) {

MemoryBlock worstFitBlock = null;

for (MemoryBlock block : memoryBlocks) {

if (!block.allocated && block.size >= processSize) {

if (worstFitBlock == null || block.size > worstFitBlock.size) {

worstFitBlock = block;

}

}

}

if (worstFitBlock != null) {

allocateMemory(worstFitBlock, processSize);

} else {

System.out.println("Process with size " + processSize + " cannot be allocated.");

}

printMemoryStatus(memoryBlocks);

}

}

public static void allocateMemory(MemoryBlock block, int processSize) {

block.allocated = true;

System.out.println("Allocated Process with size " + processSize + " to Memory Block " + block.id);

}

public static void printMemoryStatus(List<MemoryBlock> memoryBlocks) {

System.out.print("Memory Status: ");

for (MemoryBlock block : memoryBlocks) {

System.out.print("Block " + block.id + " (" + block.size + "): ");

if (block.allocated) {

System.out.print("Allocated ");

} else {

System.out.print("Unallocated ");

}

}

System.out.println();

}

}

**Write a program to simulate Memory placement strategies – best fit, first fit**

import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

import java.util.List;

class MemoryBlock {

int id;

int size;

boolean allocated;

public MemoryBlock(int id, int size) {

this.id = id;

this.size = size;

this.allocated = false;

}

}

public class MemoryPlacementSimulation {

public static void main(String[] args) {

// Example memory blocks

List<MemoryBlock> memoryBlocks = new ArrayList<>();

memoryBlocks.add(new MemoryBlock(1, 100));

memoryBlocks.add(new MemoryBlock(2, 50));

memoryBlocks.add(new MemoryBlock(3, 200));

memoryBlocks.add(new MemoryBlock(4, 80));

memoryBlocks.add(new MemoryBlock(5, 120));

// Example process sizes

List<Integer> processSizes = List.of(60, 100, 150, 80, 200);

// Best Fit Simulation

System.out.println("Best Fit:");

memoryPlacementBestFit(new ArrayList<>(memoryBlocks), new ArrayList<>(processSizes));

// First Fit Simulation

System.out.println("\nFirst Fit:");

memoryPlacementFirstFit(new ArrayList<>(memoryBlocks), new ArrayList<>(processSizes));

}

public static void memoryPlacementBestFit(List<MemoryBlock> memoryBlocks, List<Integer> processSizes) {

for (int processSize : processSizes) {

MemoryBlock bestFitBlock = null;

for (MemoryBlock block : memoryBlocks) {

if (!block.allocated && block.size >= processSize) {

if (bestFitBlock == null || block.size < bestFitBlock.size) {

bestFitBlock = block;

}

}

}

if (bestFitBlock != null) {

allocateMemory(bestFitBlock, processSize);

} else {

System.out.println("Process with size " + processSize + " cannot be allocated using Best Fit.");

}

printMemoryStatus(memoryBlocks);

}

}

public static void memoryPlacementFirstFit(List<MemoryBlock> memoryBlocks, List<Integer> processSizes) {

for (int processSize : processSizes) {

for (MemoryBlock block : memoryBlocks) {

if (!block.allocated && block.size >= processSize) {

allocateMemory(block, processSize);

break;

}

}

printMemoryStatus(memoryBlocks);

}

}

public static void allocateMemory(MemoryBlock block, int processSize) {

block.allocated = true;

System.out.println("Allocated Process with size " + processSize + " to Memory Block " + block.id);

}

public static void printMemoryStatus(List<MemoryBlock> memoryBlocks) {

System.out.print("Memory Status: ");

for (MemoryBlock block : memoryBlocks) {

System.out.print("Block " + block.id + " (" + block.size + "): ");

if (block.allocated) {

System.out.print("Allocated ");

} else {

System.out.print("Unallocated ");

}

}

System.out.println();

}

}