Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики і обчислювальної техніки

Кафедра обчислювальної техніки

Лабораторна робота №3

з курсу: «Сучасні операційні системи»

*Виконав:*

студент групи ІС-72

Гороховський І.О.

*Перевірив:*

Сімоненко А.В.

Київ, 2020 р.

**Тема**: Дослідження дисциплін обслуговування заявок при обмежених ресурсах

**Мета**: розробити програму обслуговування заявок за алгоритмом Fbn (Foregraund Backgraund).

**Теорія:**

(Foregraund Backgraund). Алгоритм має N черг. Вхідний потік заявок поступає

в першу чергу. Із черг заявки поступають на виконання. Якщо заявка за відведений

квант часу не встигла завершитися, то вона повертається в чергу i+1, де i – черга

з якої заявка була взята. З найбільш високим пріоритетом черга No1. Черга i

обслуговується, якщо порожні всі черги котрі менші за i. Починаючи з 2-ої черги

заявки сортуються за пріоритетом (0 пріоритет < 1 пріоритету). Квант часу для заявки

визначається по формулі 2i-1, заявка з останньої черги обслуговується стільки часу,

скільки їй необхідно до завершення.

**Лістинг:**

import java.io.\*;  
import java.util.Arrays;  
  
public class GraphicsBuilder {  
 public static void main(String[] args) {  
 int N = 32;  
 int length = 5000;  
 double intensity = 0.01;  
 double step = 0.01;  
 int mint = 9;  
 int maxt = 10;  
 int maxp = 10;  
 int cntr = 0;  
 double[] intensities = new double[100];  
 double[] koefs = new double[100];  
 double[] averagesWait = new double[100];  
 while(intensity <= 1){  
 cntr++;  
 Processor processor = new Processor(N, length, intensity, mint, maxt, maxp);  
 processor.modelStepsUntilQueueNotEmpty(1000);  
 int countBusy = 0;  
 int coutnFree = 0;  
 int all = processor.procStatistics.size();  
 for(boolean b : processor.procStatistics){  
 if(b){  
 countBusy++;  
 }else{  
 coutnFree++;  
 }  
 }  
  
 double koef = (double)(coutnFree)/all;  
 koef = Math.round(koef\*1000);  
 koef = koef/1000;  
 koefs[cntr-1] = koef;  
 int allWait = 0;  
 for(Task t : processor.taskStatistics){  
 allWait += t.overallWaitTime;  
 }  
 double averageWait = (double)(allWait)/all;  
 averageWait = Math.round(averageWait\*1000);  
 averageWait = averageWait/1000;  
 averagesWait[cntr-1] = averageWait;  
 intensity = Math.round(intensity\*1000);  
 intensity = intensity/1000;  
 intensities[cntr-1] = intensity;  
  
 intensity += step;  
 }  
  
 try {  
 PrintWriter writer = new PrintWriter("Graph1.txt");  
 writer.print("");  
 writer.close();  
 PrintWriter pw = new PrintWriter(new BufferedWriter(new FileWriter("Graph1.txt", true)));  
 for(int i =0; i < koefs.length; i++){  
 String s = Double.toString(koefs[i]);  
 s = s.replace('.', ',');  
 pw.println(s);  
 }  
 pw.println();  
 for(int i =0; i < averagesWait.length; i++){  
 String s = Double.toString(averagesWait[i]);  
 s = s.replace('.', ',');  
 pw.println(s);  
 }  
 pw.println();  
 for(int i =0; i < intensities.length; i++){  
 String s = Double.toString(intensities[i]);  
 s = s.replace('.', ',');  
 pw.println(s);  
 }  
 pw.close();  
 } catch (FileNotFoundException e) {  
 e.printStackTrace();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
  
  
 intensity = 0.1;  
  
 Processor processor = new Processor(N, length, intensity, mint, maxt, maxp);  
 int minTakts = 10000;  
 processor.modelStepsUntilQueueNotEmpty(minTakts);  
 double[] averagesWaitByPriority = new double[maxp];  
 int[] countersOfTaskTheSamePriority = new int[maxp];  
 Arrays.fill(averagesWaitByPriority, 0);  
 Arrays.fill(countersOfTaskTheSamePriority, 0);  
 for(Task t : processor.taskStatistics){  
 averagesWaitByPriority[t.priority] += t.overallWaitTime;  
 countersOfTaskTheSamePriority[t.priority]++;  
 }  
 for(int i = 0; i < maxp; i++){  
 if(countersOfTaskTheSamePriority[i] != 0)  
 averagesWaitByPriority[i] /= countersOfTaskTheSamePriority[i];  
 }  
  
 try {  
 PrintWriter writer = new PrintWriter("Graph2.txt");  
 writer.print("");  
 writer.close();  
 PrintWriter pw = new PrintWriter(new BufferedWriter(new FileWriter("Graph2.txt", true)));  
 for(int i =0; i < maxp; i++){  
 double d = averagesWaitByPriority[i];  
 d = Math.round(d\*1000)/(double)1000;  
 String s = Double.toString(d);  
 s = s.replace('.', ',');  
 pw.println(s);  
 }  
 pw.close();  
 } catch (FileNotFoundException e) {  
 e.printStackTrace();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
  
  
  
 int countBusy = 0;  
 int coutnFree = 0;  
 int all = processor.procStatistics.size();  
 for(boolean b : processor.procStatistics){  
 if(b){  
 countBusy++;  
 }else{  
 coutnFree++;  
 }  
 }  
 System.out.println("count free: "+coutnFree);  
 System.out.println("count busy: "+countBusy);  
 System.out.println("all: " + all);  
 System.out.println("count of tasks: " + processor.taskStatistics.size());  
 System.out.println("tasks:");  
 System.out.println(processor.taskStatistics.toString());  
 int allWait = 0;  
 for(Task t : processor.taskStatistics){  
 allWait += t.overallWaitTime;  
 }  
 double averageWait = (double)(allWait)/processor.taskStatistics.size();  
 System.out.println("Average wait: " + averageWait);  
  
 try {  
 PrintWriter pw = new PrintWriter("RESULTS.txt");  
 pw.println(processor.taskStatistics.size());  
 for(Task t : processor.taskStatistics){  
 pw.println(t.priority);  
 }  
 for(Task t : processor.taskStatistics){  
 pw.println(t.startTime);  
 }  
 for(Task t : processor.taskStatistics){  
 pw.println(t.execTime);  
 }  
 for(Task t : processor.taskStatistics){  
 pw.println(t.overallWaitTime);  
 }  
 pw.close();  
 System.out.println("Writing finished");  
 } catch (FileNotFoundException e) {  
 e.printStackTrace();  
 }  
 }  
}

import java.util.LinkedList;  
  
public class Processor {  
 private ProcessorQueue queue;  
 private Generator generator;  
 private int systemTime = 0;  
 private int deltaTime = 1;  
 int interval = 1000;  
 public LinkedList<Task> taskStatistics = new LinkedList<Task>();  
 public LinkedList<Boolean> procStatistics = new LinkedList<Boolean>();  
  
 public Processor(int n, int length, double intensity, int mint, int maxt, int maxp){  
 queue = new ProcessorQueue(n, length);  
 generator = new Generator(intensity, mint, maxt, maxp);  
 }  
  
 public void modelStepsUntilQueueNotEmpty(int mt){  
 interval = mt;  
 while((systemTime < interval) || (queue.notEmpty())){  
 nextStep();  
 }  
 }  
  
 public void modelSteps(int n){  
 for(int i = 0; i < n; i++){  
 nextStep();  
 }  
 }  
  
 public void nextStep(){  
 if(systemTime < interval) {  
 while (deltaTime > 0) {  
 deltaTime--;  
 queue.addTask(generator.generate(systemTime));  
 }  
 }  
 ProcessorTask currentTask = queue.getTask();  
 if(currentTask != null){  
 if(currentTask.queueNumber == queue.N-1){  
 systemTime += currentTask.task.timeOfExecute;  
 deltaTime = currentTask.task.timeOfExecute;  
 currentTask.task.timeOfExecute = 0;  
 currentTask.task.overallWaitTime = systemTime - currentTask.task.startTime;  
 taskStatistics.add(currentTask.task);  
 } else {  
 if (currentTask.time < currentTask.task.timeOfExecute) {  
 systemTime += currentTask.time;  
 deltaTime = currentTask.time;  
 currentTask.task.timeOfExecute -= currentTask.time;  
 queue.addTaskInNextQueue(currentTask.task, currentTask.queueNumber);  
 } else {  
 systemTime +=currentTask.task.timeOfExecute;  
 deltaTime = currentTask.task.timeOfExecute;  
 currentTask.task.timeOfExecute = 0;  
 currentTask.task.overallWaitTime = systemTime - currentTask.task.startTime;  
 taskStatistics.add(currentTask.task);  
 }  
 }  
 addUsedTime(deltaTime);  
 } else {  
 systemTime++;  
 deltaTime++;  
 procStatistics.add(false);  
 }  
 }  
 private void addUsedTime(int counter){  
 for(int i = 0; i < counter; i++){  
 procStatistics.add(true);  
 }  
 }  
}

import java.util.LinkedList;  
import java.util.PriorityQueue;  
import java.util.Queue;  
  
public class ProcessorQueue {  
 public final int N;  
 private final int length;  
 private Queue<Task>[] queues;  
 private int[] times;  
  
  
 public ProcessorQueue(int n, int length){  
 N = n;  
 this.length = length;  
 init();  
 }  
  
 private void init(){  
 queues = new Queue[N];  
 queues[0] = new LinkedList<Task>();  
 for(int i = 1; i < queues.length; i++){  
 queues[i] = new PriorityQueue<Task>(length, new Task());  
 }  
  
 times = new int[N];  
 for(int i = 0; i < times.length; i++){  
 times[i] = 2\*(i+1) - 1;  
 }  
 times[times.length-1] = -1;  
 }  
  
 public void addTask(Task t){  
 if(t != null){  
 queues[0].add(t);  
 }  
 }  
  
 public void addTaskInNextQueue(Task t, int n){  
 queues[n+1].add(t);  
 }  
  
 public ProcessorTask getTask(){  
 ProcessorTask pt = null;  
 for(int i = 0; i < queues.length; i++){  
 if(queues[i].size() == 0){  
 continue;  
 } else {  
 Task res = queues[i].remove();  
 pt = new ProcessorTask();  
 pt.task = res;  
 pt.time = times[i];  
 pt.queueNumber = i;  
 return pt;  
 }  
 }  
 return pt;  
 }  
  
 public boolean notEmpty(){  
 boolean res = false;  
 for(int i = 0; i < queues.length; i++){  
 if(queues[i].size() == 0){  
 continue;  
 } else {  
 res = true;  
 return res;  
 }  
 }  
 return res;  
 }  
  
 @Override  
 public String toString() {  
 StringBuffer res = new StringBuffer();  
 res.append("---- Processor Queue ----- \n"+"number of queues: "+N+"\n");  
 for(int i = 0; i < N; i++){  
 res.append((i+1));  
 res.append(": " + queues[i].toString() + "\n");  
 }  
 res.append("---- /Processor Queue ----");  
  
 return res.toString();  
 }  
}

public class ProcessorTask {  
 public Task task;  
 public int time;  
 public int queueNumber;  
}

import java.util.Comparator;  
  
public class Task implements Comparator<Task>{  
 public int priority;  
 public int timeOfExecute;  
 public int overallWaitTime;  
 public int execTime;  
 public int startTime;  
  
 public Task(){  
  
 }  
  
 public Task(int pr, int te, int time){  
 priority = pr;  
 timeOfExecute = te;  
 execTime = timeOfExecute;  
 overallWaitTime = 0;  
 startTime = time;  
 }  
  
 @Override  
 public int compare(Task o1, Task o2) {  
 if(o1.priority > o2.priority){  
 return -1;  
 }  
 if(o1.priority < o2.priority){  
 return 1;  
 }  
 return 0;  
 }  
  
 @Override  
 public String toString() {  
 return ("(" + priority + ", " + execTime+", "+ timeOfExecute + ")"); //", "+overallWaitTime+", "+startTime+  
 }  
}

import java.util.Random;  
  
public class Generator {  
 private final int minTimeOfExecute;  
 private final int maxTimeOfExecute;  
  
 private final int maxPriority;  
 private final double intensity;  
 public Generator(double intensity, int mint, int maxt, int maxp){  
 this.intensity = intensity;  
 minTimeOfExecute = mint;  
 maxTimeOfExecute = maxt;  
 maxPriority = maxp;  
 }  
 public Task generate(int currentSystemTime){  
 Random r = new Random();  
 double posibility = r.nextDouble();  
 //System.out.println(posibility);  
 if(posibility < intensity){  
 int priority = r.nextInt(maxPriority);  
 int timeOfExecution = r.nextInt(maxTimeOfExecute - minTimeOfExecute) + minTimeOfExecute;  
 return new Task(priority, timeOfExecution, currentSystemTime);  
 }  
 return null;  
 }  
}