МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

Сучасні операційні системи

Лабораторна робота 3

На тему: “Дослідження дисциплін обслуговування заявок при обмежених ресурсах ”

*Виконав студент групи ІC-72*

*Цилюрик Дмитро*

Київ

НТУУ «КПІ»

2020

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

**Мета робот**: розробити програму обслуговування заявок за алгоритмом Fbn.

**Теорія:** 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.

**Посилання на GitHub проекту:**

https://github.com/lost-dt/os-course/tree/master/Task3

**Розроблене програмне забезпечення:**

* Файл *Generator.java*:

1. **import** java.util.Random;
3. **public** **class** Generator {
4. **private** **final** **int** minTimeOfExecute;
5. **private** **final** **int** maxTimeOfExecute;
7. **private** **final** **int** maxPriority;
8. **private** **final** **double** intensity;
9. **public** Generator(**double** intensity, **int** mint, **int** maxt, **int** maxp){
10. **this**.intensity = intensity;
11. minTimeOfExecute = mint;
12. maxTimeOfExecute = maxt;
13. maxPriority = maxp;
14. }
15. **public** Task generate(**int** currentSystemTime){
16. Random r = **new** Random();
17. **double** posibility = r.nextDouble();
18. **if**(posibility < intensity){
19. **int** priority = r.nextInt(maxPriority);
20. **int** timeOfExecution = r.nextInt(maxTimeOfExecute - minTimeOfExecute) + minTimeOfExecute;
21. **return** **new** Task(priority, timeOfExecution, currentSystemTime);
22. }
23. **return** **null**;
24. }
25. }

* Файл *Processor.java*:

1. **import** java.util.LinkedList;
3. **public** **class** Processor {
4. **private** ProcessorQueue queue;
5. **private** Generator generator;
6. **private** **int** systemTime = 0;
7. **private** **int** deltaTime = 1;
8. **int** interval = 1000;
9. **public** LinkedList<Task> taskStatistics = **new** LinkedList<Task>();
10. **public** LinkedList<Boolean> procStatistics = **new** LinkedList<Boolean>();
12. **public** Processor(**int** n, **int** length, **double** intensity, **int** mint, **int** maxt, **int** maxp){
13. queue = **new** ProcessorQueue(n, length);
14. generator = **new** Generator(intensity, mint, maxt, maxp);
15. }
17. **public** **void** modelStepsUntilQueueNotEmpty(**int** mt){
18. interval = mt;
19. **while**((systemTime < interval) || (queue.notEmpty())){
20. nextStep();
21. }
22. }
24. **public** **void** modelSteps(**int** n){
25. **for**(**int** i = 0; i < n; i++){
26. nextStep();
27. }
28. }
30. **public** **void** nextStep(){
31. **if**(systemTime < interval) {
32. **while** (deltaTime > 0) {
33. deltaTime--;
34. queue.addTask(generator.generate(systemTime));
35. }
36. }
37. ProcessorTask currentTask = queue.getTask();
38. **if**(currentTask != **null**){
39. **if**(currentTask.queueNumber == queue.N-1){
40. systemTime += currentTask.task.timeOfExecute;
41. deltaTime = currentTask.task.timeOfExecute;
42. currentTask.task.timeOfExecute = 0;
43. currentTask.task.overallWaitTime = systemTime - currentTask.task.startTime;
44. taskStatistics.add(currentTask.task);
45. } **else** {
46. **if** (currentTask.time < currentTask.task.timeOfExecute) {
47. systemTime += currentTask.time;
48. deltaTime = currentTask.time;
49. currentTask.task.timeOfExecute -= currentTask.time;
50. queue.addTaskInNextQueue(currentTask.task, currentTask.queueNumber);
51. } **else** {
52. systemTime +=currentTask.task.timeOfExecute;
53. deltaTime = currentTask.task.timeOfExecute;
54. currentTask.task.timeOfExecute = 0;
55. currentTask.task.overallWaitTime = systemTime - currentTask.task.startTime;
56. taskStatistics.add(currentTask.task);
57. }
58. }
59. addUsedTime(deltaTime);
60. } **else** {
61. systemTime++;
62. deltaTime++;
63. procStatistics.add(**false**);
64. }
65. }
66. **private** **void** addUsedTime(**int** counter){
67. **for**(**int** i = 0; i < counter; i++){
68. procStatistics.add(**true**);
69. }
70. }
71. }

* Файл *ProcessorQueue.java*:

1. **import** java.util.LinkedList;
2. **import** java.util.PriorityQueue;
3. **import** java.util.Queue;
5. **public** **class** ProcessorQueue {
6. **public** **final** **int** N;
7. **private** **final** **int** length;
8. **private** Queue<Task>[] queues;
9. **private** **int**[] times;

12. **public** ProcessorQueue(**int** n, **int** length){
13. N = n;
14. **this**.length = length;
15. init();
16. }
18. **private** **void** init(){
19. queues = **new** Queue[N];
20. queues[0] = **new** LinkedList<Task>();
21. **for**(**int** i = 1; i < queues.length; i++){
22. queues[i] = **new** PriorityQueue<Task>(length, **new** Task());
23. }
25. times = **new** **int**[N];
26. **for**(**int** i = 0; i < times.length; i++){
27. times[i] = 2\*(i+1) - 1;
28. }
29. times[times.length-1] = -1;
30. }
32. **public** **void** addTask(Task t){
33. **if**(t != **null**){
34. queues[0].add(t);
35. }
36. }
38. **public** **void** addTaskInNextQueue(Task t, **int** n){
39. queues[n+1].add(t);
40. }
42. **public** ProcessorTask getTask(){
43. ProcessorTask pt = **null**;
44. **for**(**int** i = 0; i < queues.length; i++){
45. **if**(queues[i].size() == 0){
46. **continue**;
47. } **else** {
48. Task res = queues[i].remove();
49. pt = **new** ProcessorTask();
50. pt.task = res;
51. pt.time = times[i];
52. pt.queueNumber = i;
53. **return** pt;
54. }
55. }
56. **return** pt;
57. }
59. **public** **boolean** notEmpty(){
60. **boolean** res = **false**;
61. **for**(**int** i = 0; i < queues.length; i++){
62. **if**(queues[i].size() == 0){
63. **continue**;
64. } **else** {
65. res = **true**;
66. **return** res;
67. }
68. }
69. **return** res;
70. }
72. @Override
73. **public** String toString() {
74. StringBuffer res = **new** StringBuffer();
75. res.append("---- Processor Queue ----- \n"+"number of queues: "+N+"\n");
76. **for**(**int** i = 0; i < N; i++){
77. res.append((i+1));
78. res.append(":   " + queues[i].toString() + "\n");
79. }
80. res.append("---- /Processor Queue ----");
82. **return** res.toString();
83. }
84. }

* Файл *ProcessorTask.java*:

1. **public** **class** ProcessorTask {
2. **public** Task task;
3. **public** **int** time;
4. **public** **int** queueNumber;
5. }

* Файл *Task.java*:

1. **import** java.util.Comparator;
3. **public** **class** Task **implements** Comparator<Task>{
4. **public** **int** priority;
5. **public** **int** timeOfExecute;
6. **public** **int** overallWaitTime;
7. **public** **int** execTime;
8. **public** **int** startTime;
10. **public** Task(){
12. }
14. **public** Task(**int** pr, **int** te, **int** time){
15. priority = pr;
16. timeOfExecute = te;
17. execTime = timeOfExecute;
18. overallWaitTime = 0;
19. startTime = time;
20. }
22. @Override
23. **public** **int** compare(Task o1, Task o2) {
24. **if**(o1.priority > o2.priority){
25. **return** -1;
26. }
27. **if**(o1.priority < o2.priority){
28. **return** 1;
29. }
30. **return** 0;
31. }
33. @Override
34. **public** String toString() {
35. **return** ("(" + priority + ", " + execTime+", "+ timeOfExecute + ")"); //", "+overallWaitTime+", "+startTime+
36. }
37. }

* Файл *Task3.java*:

1. **import** com.sun.org.apache.xpath.internal.SourceTree;
3. **import** java.io.\*;
4. **import** java.util.Arrays;
6. **public** **class** Task3 {
7. **public** **static** **void** main(String[] args) {
8. **int** N = 32;
9. **int** length = 5000;
10. **double** intensity = 0.01;
11. **double** step = 0.01;
12. **int** mint = 9;
13. **int** maxt = 10;
14. **int** maxp = 10;
15. **int** cntr = 0;
16. **double**[] intensities = **new** **double**[100];
17. **double**[] koefs = **new** **double**[100];
18. **double**[] averagesWait = **new** **double**[100];
19. **while**(intensity <= 1){
20. cntr++;
21. Processor processor = **new** Processor(N, length, intensity, mint, maxt, maxp);
22. processor.modelStepsUntilQueueNotEmpty(1000);
23. **int** countBusy = 0;
24. **int** coutnFree = 0;
25. **int** all = processor.procStatistics.size();
26. **for**(**boolean** b : processor.procStatistics){
27. **if**(b){
28. countBusy++;
29. }**else**{
30. coutnFree++;
31. }
32. }
34. **double** koef = (**double**)(coutnFree)/all;
35. koef = Math.round(koef\*1000);
36. koef = koef/1000;
37. koefs[cntr-1] = koef;
38. **int** allWait = 0;
39. **for**(Task t : processor.taskStatistics){
40. allWait += t.overallWaitTime;
41. }
42. **double** averageWait = (**double**)(allWait)/all;
43. averageWait = Math.round(averageWait\*1000);
44. averageWait = averageWait/1000;
45. averagesWait[cntr-1] = averageWait;
46. intensity = Math.round(intensity\*1000);
47. intensity = intensity/1000;
48. intensities[cntr-1] = intensity;
50. intensity += step;
51. }
53. **try** {
54. PrintWriter writer = **new** PrintWriter("Graph1.txt");
55. writer.print("");
56. writer.close();
57. PrintWriter pw = **new** PrintWriter(**new** BufferedWriter(**new** FileWriter("Graph1.txt", **true**)));
58. **for**(**int** i =0; i < koefs.length; i++){
59. String s = Double.toString(koefs[i]);
60. s = s.replace('.', ',');
61. pw.println(s);
62. }
63. pw.println();
64. **for**(**int** i =0; i < averagesWait.length; i++){
65. String s = Double.toString(averagesWait[i]);
66. s = s.replace('.', ',');
67. pw.println(s);
68. }
69. pw.println();
70. **for**(**int** i =0; i < intensities.length; i++){
71. String s = Double.toString(intensities[i]);
72. s = s.replace('.', ',');
73. pw.println(s);
74. }
75. pw.close();
76. } **catch** (FileNotFoundException e) {
77. e.printStackTrace();
78. } **catch** (IOException e) {
79. e.printStackTrace();
80. }

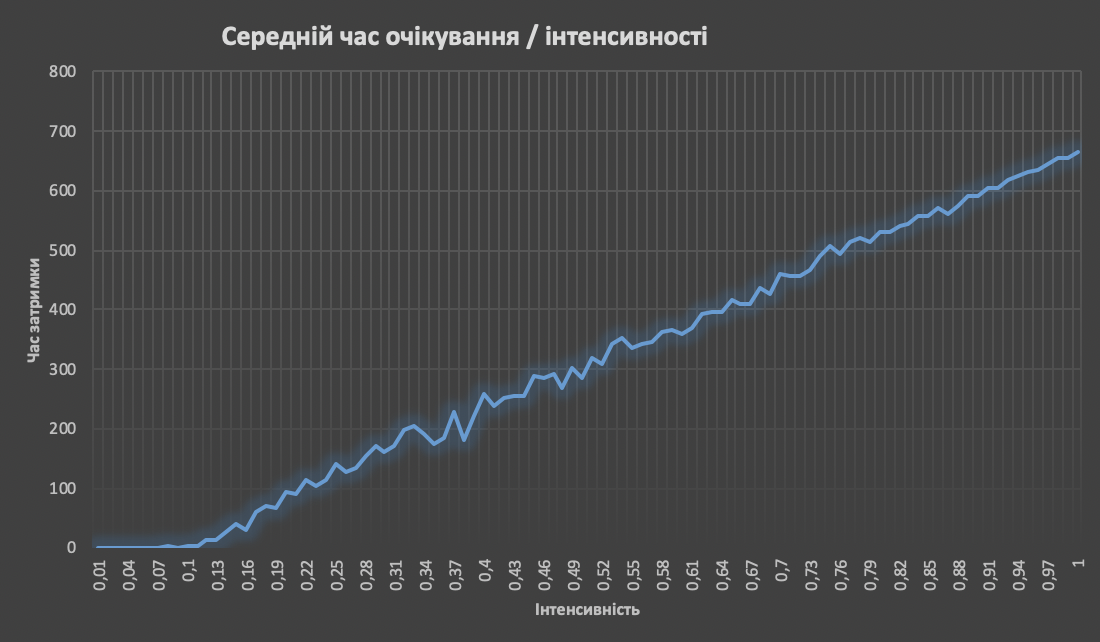
83. intensity = 0.1;
85. Processor processor = **new** Processor(N, length, intensity, mint, maxt, maxp);
86. **int** minTakts = 10000;
87. processor.modelStepsUntilQueueNotEmpty(minTakts);
88. **double**[] averagesWaitByPriority = **new** **double**[maxp];
89. **int**[] countersOfTaskTheSamePriority = **new** **int**[maxp];
90. Arrays.fill(averagesWaitByPriority, 0);
91. Arrays.fill(countersOfTaskTheSamePriority, 0);
92. **for**(Task t : processor.taskStatistics){
93. averagesWaitByPriority[t.priority] += t.overallWaitTime;
94. countersOfTaskTheSamePriority[t.priority]++;
95. }
96. **for**(**int** i = 0; i < maxp; i++){
97. **if**(countersOfTaskTheSamePriority[i] != 0)
98. averagesWaitByPriority[i] /= countersOfTaskTheSamePriority[i];
99. }
101. **try** {
102. PrintWriter writer = **new** PrintWriter("Graph2.txt");
103. writer.print("");
104. writer.close();
105. PrintWriter pw = **new** PrintWriter(**new** BufferedWriter(**new** FileWriter("Graph2.txt", **true**)));
106. **for**(**int** i =0; i < maxp; i++){
107. **double** d = averagesWaitByPriority[i];
108. d = Math.round(d\*1000)/(**double**)1000;
109. String s = Double.toString(d);
110. s = s.replace('.', ',');
111. pw.println(s);
112. }
113. pw.close();
114. } **catch** (FileNotFoundException e) {
115. e.printStackTrace();
116. } **catch** (IOException e) {
117. e.printStackTrace();
118. }


122. **int** countBusy = 0;
123. **int** coutnFree = 0;
124. **int** all = processor.procStatistics.size();
125. **for**(**boolean** b : processor.procStatistics){
126. **if**(b){
127. countBusy++;
128. }**else**{
129. coutnFree++;
130. }
131. }
132. System.out.println("count free: "+coutnFree);
133. System.out.println("count busy: "+countBusy);
134. System.out.println("all: " + all);
135. System.out.println("count of tasks: " + processor.taskStatistics.size());
136. System.out.println("tasks:");
137. System.out.println(processor.taskStatistics.toString());
138. **int** allWait = 0;
139. **for**(Task t : processor.taskStatistics){
140. allWait += t.overallWaitTime;
141. }
142. **double** averageWait = (**double**)(allWait)/processor.taskStatistics.size();
143. System.out.println("Average wait: " + averageWait);
145. **try** {
146. PrintWriter pw = **new** PrintWriter("RESULTS.txt");
147. pw.println(processor.taskStatistics.size());
148. **for**(Task t : processor.taskStatistics){
149. pw.println(t.priority);
150. }
151. **for**(Task t : processor.taskStatistics){
152. pw.println(t.startTime);
153. }
154. **for**(Task t : processor.taskStatistics){
155. pw.println(t.execTime);
156. }
157. **for**(Task t : processor.taskStatistics){
158. pw.println(t.overallWaitTime);
159. }
160. pw.close();
161. System.out.println("Writing finished");
162. } **catch** (FileNotFoundException e) {
163. e.printStackTrace();
164. }
165. }
166. }

**Результати роботи програми в вигляді діаграм:**

A screenshot of a computer

Description automatically generated



A picture containing clock

Description automatically generated