НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

КАФЕДРА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

Лабораторна робота №3

з дисципліни **«**Організація обчислювальних процесів**»**

Виконав:

студент 3 курсу

ФІОТ гр. ІО-21

Кузьменко Володимир

Перевірив:

Сімоненко В. П.

Київ – 2015 р.

***PRIORITY QUEUING - (PQ)***

Описані раніше стратегії можуть розглядатися як частковий випадок стратегії пріоритетного планування. Ця стратегія передбачає, що кожному процесу надається пріоритет, що визначає черговість надання йому CPU. Наприклад, стратегія FIFO передбачає, що всі процеси мають однакові пріоритети, а стратегія SJF передбачає, що пріоритет є величина, обернена часу наступного обслуговування.

Пріоритет — це ціле додатне число, що знаходиться в деякому діапазоні, наприклад від 0 до 7, від 0 до 4095. Будемо вважати, що чим менше значення числа, тим вищий пріоритет процесу.

|  |  |
| --- | --- |
| ***Приклад №3.*** | **пріоритет** |
| П1(10 мс) | 3 |
| П2(1 мс) | 1 |
| П3(2 мс) | 3 |
| П4(1 мс) | 4 |
| П5(5 мс) | 2 |

На малюнку наведена діаграма Ганта, виконування черги готових процесів при розташуванні процесів у черзі у відповідності зі стратегією пріоритетного планування

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| П2 | П5 | П1 | П3 | П4 |  |
| WT2=0 мс | WT5=1 мс | WT1=6 мс | WT3=16 мс | WT4=18 мс |  |

Пріоритети визначаються виходячи із сукупності внутрішніх та зовнішніх по відношенню до операційної системи факторів.

**Внутрішні фактори:**

1. вимоги до пам'яті
2. кількість відкритих файлів
3. відношення середнього часу вводу - виводу до середнього часу CPU і так далі

**Зовнішні фактори :**

1. важливість процесу
2. тип та величина файлів, що використовуються для оплати
3. відділ, що виконує роботи і так далі

Внутрішні фактори можуть використовуватися для автоматичного призначення пріоритетів самою операційною системою, а зовнішні для примусового, за допомогою оператора.

Головний недолік пріоритетного планування полягає у можливості блокування на невизначено довгий строк низько пріоритетних процесів.

Для виправлення відміченого недоліку використовуються наступні методи.: Процеси, час очікування котрих перевищує фіксовану величину, наприклад 15 хвилин, автоматично отримує одиничний приріст пріоритету.

Пріоритетне обслуговування (PQ) є ефективною і прямою формою управління перевантаженням. PQ у сучасних протоколах дозволяє виділити до чотирьох черг в мережевому трафіку. Передбачені черги високого, середнього, нормального і низького пріоритету. Маршрутизатор обробляє черги строго відповідно до їх пріоритету. Пакети з черги з високим пріоритетом обробляються, поки в черзі не залишиться жодного пакету, після цього починається обробка черги з середнім пріоритетом, паралельно здійснюється контроль появи пакетів в черзі з високим пріоритетом. Пакети з черги з низьким пріоритетом обробляються лише тоді, коли решта черг порожня. Низько пріоритетний трафік при певних обставинах може бути повністю блокований, а пакети втрачені. Зазвичай PQ використовується, коли прикладні програми, критичні до затримок, стикаються з проблемами.

Якщо високо пріоритетний трафік має високу інтенсивність, то існує висока ймовірність того, що решта складових трафіку буде заблокована. Пакети, не класифіковані як PQ, автоматично відносяться до черги з нормальним пріоритетом. Системна черга має найвищий пріоритет.

Варіант 123 => Алгоритм PQ (*Priority Queuing).*

Лістинг програми:

**package** lab3;

**import** java.util.Random;

1. **public** **class** Generator **implements** Runnable {
2. **public** **static** **final** **int** MAX\_TASK\_LENGTH = 100;
3. **public** **static** **final** **int** MIN\_TASK\_LENGTH = 0;
4. **public** **static** **final** **int** MAX\_DELAY\_BETWEEN\_TASKS = 60;
5. **public** **static** **final** **int** MIN\_DELAY\_BETWEEN\_TASKS = 60;
6. **public** **static** **final** **int** MAX\_TASK\_PRIORITY = 5;
7. **public** **static** **final** **int** MIN\_TASK\_PRIORITY = 1;
8. **private** Queue queue;
9. **private** Random random = **new** Random();
10. **public** **Generator**(Queue queue) {
11. **this**.queue = queue;
12. }
13. **public** **int** **generateTaskLength**() {
14. **return** random.nextInt(MAX\_TASK\_LENGTH - MIN\_TASK\_LENGTH + 1)
15. + MIN\_TASK\_LENGTH;
16. }
17. **public** **int** **generateDelayBetweenTasks**() {
18. **return** random.nextInt(MAX\_DELAY\_BETWEEN\_TASKS - MIN\_DELAY\_BETWEEN\_TASKS
19. + 1)
20. + MIN\_DELAY\_BETWEEN\_TASKS;
21. }
22. **public** **int** **generateTaskPriority**() {
23. **return** random.nextInt(MAX\_TASK\_PRIORITY - MIN\_TASK\_PRIORITY + 1)
24. + MIN\_TASK\_PRIORITY;
25. }
26. @Override
27. **public** **void** **run**() {
28. **while** (**true**) {
29. queue.addTask(**new** Task(generateTaskLength(), generateTaskPriority()));
30. **try** {
31. Thread.*sleep*(generateDelayBetweenTasks());
32. } **catch** (InterruptedException e) {
33. Thread.*currentThread*().interrupt();
34. }
35. }
36. }
37. }
38. **package** lab3;
39. **public** **class** Processor **implements** Runnable {
40. **private** String name;
41. **private** Queue inputTasksQueue;
42. **private** Task currentTask;
43. **private** **int** finishedTasksCounter = 0;
44. **private** **long** workingTime = 0;
45. **private** **long** startTime;
46. **private** **double** returnTime = 0;
47. **private** **double** waitingTime = 0;
48. **private** **double**[] wait = { 0, 0, 0, 0, 0 };
49. **private** **double**[] count = { 0, 0, 0, 0, 0 };
50. **public** **Processor**(String name, Queue inputQueue) {
51. **this**.name = name;
52. inputTasksQueue = inputQueue;
53. startTime = System.*currentTimeMillis*();
54. }
55. @Override
56. **public** **void** **run**() {
57. **while** (**true**) {
58. **try** {
59. currentTask = inputTasksQueue.removeTask();
60. **long** currentWaitingTime = System.*currentTimeMillis*()
61. - currentTask.getCreationTime();
62. waitingTime += currentWaitingTime;
63. **for** (**int** i = 0; i < wait.length; i++)
64. wait[i] += currentWaitingTime;
65. count[currentTask.getPriority() - 1]++;
66. Thread.*sleep*(currentTask.getLength());
67. finishedTasksCounter++;
68. workingTime += currentTask.getLength();
69. **double** lifeTime = System.*currentTimeMillis*() - startTime;
70. **double** cpu = (workingTime / lifeTime) \* 100;
71. **long** currentReturnTime = System.*currentTimeMillis*()
72. - currentTask.getCreationTime();
73. returnTime += currentReturnTime;
74. **double** throughput = finishedTasksCounter / (lifeTime / 1000);
75. System.out
76. .printf("%d\t%02.2f%%\t%03.2f\t%05.2f ms\t%05.2f ms\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\n",
77. finishedTasksCounter, cpu, throughput,
78. (returnTime / finishedTasksCounter),
79. (waitingTime / finishedTasksCounter), wait[4]
80. / count[4], wait[3] / count[3], wait[2]
81. / count[2], wait[1] / count[1], wait[0]
82. / count[0]);
83. } **catch** (InterruptedException e) {
84. System.out.println("interrupt on sleep");
85. Thread.*currentThread*().interrupt();
86. }
87. }
88. }
89. **private** **void** **stat**(Task task) {
90. }
91. }
92. package lab3;
93. import java.awt.event.ActionEvent;
94. import java.awt.event.ActionListener;
95. import java.util.Iterator;
96. import java.util.PriorityQueue;
97. public class Queue implements ActionListener {
98. private java.util.Queue<Task> queue;
99. private String name;
100. public Queue(String name) {
101. this.name = name;
102. queue = new PriorityQueue<>(10, new TaskPriorityComparator());
103. }
104. public synchronized void addTask(Task task) {
105. queue.add(task);
106. notifyAll();
107. }
108. public synchronized Task removeTask() throws InterruptedException {
109. while (queue.size() == 0) {
110. wait();
111. }
112. return queue.poll();
113. }
114. public synchronized int size() {
115. return queue.size();
116. }
117. public String toString() {
118. return name + " : " + size();
119. }
120. @Override
121. public void actionPerformed(ActionEvent e) {
122. synchronized (this) {
123. Iterator<Task> it = queue.iterator();
124. while (it.hasNext()) {
125. Task current = it.next();
126. if (current.getPriority() < Generator.MAX\_TASK\_PRIORITY)
127. current.setPriority(current.getPriority() + 1);
128. }
129. }
130. }
131. }
132. **package** lab3;
133. **public** **class** Task {
134. **private** **int** length;
135. **private** **long** creationTime;
136. **private** **int** priority;
137. **public** **Task**(**int** length, **int** priority) {
138. **this**.length = length;
139. creationTime = System.*currentTimeMillis*();
140. **this**.priority = priority;
141. }
142. **public** **int** **getLength**() {
143. **return** length;
144. }
145. **public** **int** **getPriority**() {
146. **return** priority;
147. }
148. **public** **void** **setPriority**(**int** priority) {
149. **this**.priority = priority;
150. }
151. **public** **long** **getCreationTime**() {
152. **return** creationTime;
153. }
154. **public** String **toString**() {
155. **return** length + " " + priority + "\n";
156. }
157. }
158. package lab3;
159. import javax.swing.Timer;
160. public class TestMain {
161. public static void main(String[] args) {
162. Queue queue = new Queue("queue");
163. Generator generator = new Generator(queue);
164. Processor processor = new Processor("cpu", queue);
165. Timer timer = new Timer(Generator.MAX\_TASK\_LENGTH \* 10, queue);
166. Thread generatorThread = new Thread(generator);
167. Thread cpuThread = new Thread(processor);
168. generatorThread.start();
169. cpuThread.start();
170. timer.start();
171. }
172. }
173. **package** lab3;
174. **import** java.util.Comparator;
175. **public** **class** TaskPriorityComparator **implements** Comparator<Task> {
176. @Override
177. **public** **int** **compare**(Task o1, Task o2) {
178. **return** Integer.*valueOf*(o2.getPriority()).compareTo(o1.getPriority());
179. }
180. }