Kpi-best

Міністерство освіти і науки України

Національний технічний університет України “Київський політехнічний інститут”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Лабораторна робота № 3

з курсу “Системне програмне забезпечення – 1” на тему:

“ Дослідження дисциплін обслуговування заявок при обмежених ресурсах”

Виконав: студент ІІІ курсу групи ІО – 01

Редько О.М.

Київ – 2013

# Варіант

№ залікової книжки **0115**.

115 mod 22 +1 = **6 - Алгоритм SJF ( *Shortest Job First*).**

# Опис дисципліни обслуговування

**SJF** — Shortest Job First алгоритм управління чергами, що полягає у виконанні найкоротшої заявки першою.

# Результати виконання

Для реалізації системи обслуговування був використаний вхідний потік заявок з експоненціальним розподілом (потік Пуассона).

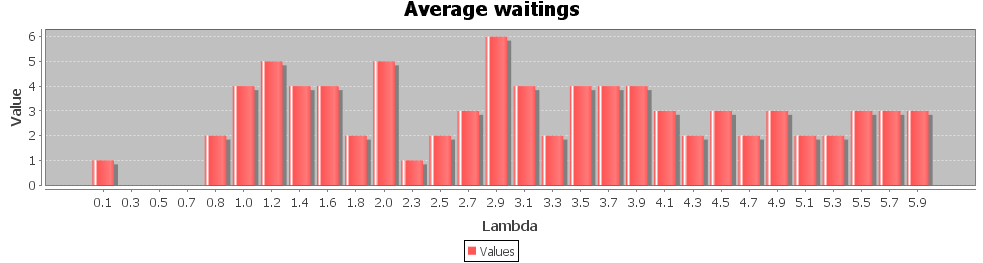


Рис. 1. Середній час очікування в черзі.

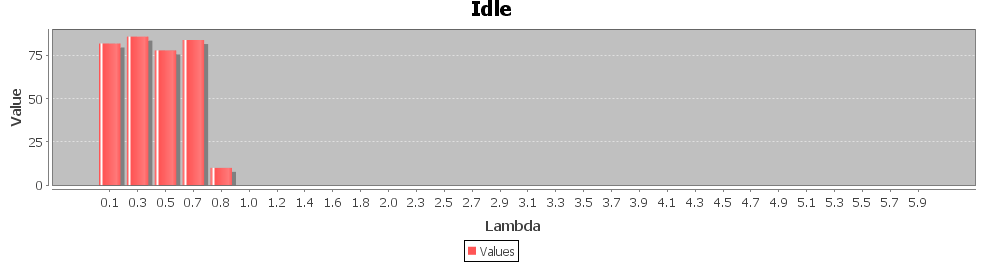
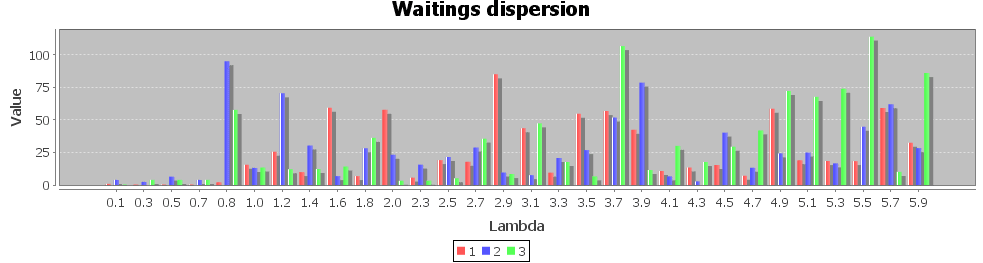
На графіку зображена залежність середнього часу очікування в черзі від інтенсивності вхідного потоку. При малій інтенсивності заявки майже не затримуються в системі. При зростанні інтенсивності зростає час очікування заявок в системі, так як вірогідність появи нової заявки у системі збільшуется.

Рис. 2. Час простою пристрою.

При малій інтенсивності пристрій простоює до 75% часу. При зростанні інтенсивності вхідного потоку, пристрій завжди зайнятий.

 Рис. 3. Дисперсія часу очікування.

При малій інтенсивності дисперсія часу очікування незначна. Це зумовлено тим, що заявки швидко виходять з системи, коли їх мало у черзі. Зі зростанням інтенсивності вхідного потоку короткі заявки швидко виходять з системи, а довгі затримуються більше, тому маємо різке зростання дисперсії.

# Лістинг програми

**package** shortjobsirst;

**public** **class** Task **implements** Comparable<Task> {

**private** **int** time;

**private** **int** waiting;

**public** Task(**int** time) {

**super**();

**this**.time = time;

waiting = 0;

}

@Override

**public** **int** compareTo(Task t) {

**return** getTime() - t.getTime();

}

**public** **void** decTime() {

time--;

}

**public** **int** getTime() {

**return** time;

}

/\*\*

\* **@return** the waiting

\*/

**public** **synchronized** **int** getWaiting() {

**return** waiting;

}

**public** **void** setTime(**int** time) {

**this**.time = time;

}

@Override

**public** String toString() {

**return** **new** Integer(time).toString();

}

**public** **void** waitOne() {

waiting++;

}

}

**package** shortjobsirst;

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.Random;

**public** **class** SJFSystem {

**public** **static** **void** main(String[] args) {

SJFSystem s = **new** SJFSystem(3, 1, 5);

**for** (**int** i = 0; i < 50; i++) {

s.nextIteration();

System.*out*.println("#" + (i + 1) + " CPU: " + s.getCPU() + " Q: "

+ s.getQueve());

}

System.*out*.println("idle: " + s.getIdle());

System.*out*.println("waitTimes: " + s.getWaitings());

}

**private** **double** lambda;

**private** **int** minTime, maxTime;

**private** ArrayList<Task> queve;

**private** Task CPU;

**private** **final** **double** range = 0.5;

Random rand = **new** Random();

**private** ArrayList<Integer> waitings;

**private** **int** idle;

**public** SJFSystem(**double** lambda, **int** minTime, **int** maxTime) {

**super**();

**this**.lambda = lambda;

**this**.minTime = minTime;

**this**.maxTime = maxTime;

queve = **new** ArrayList<Task>();

waitings = **new** ArrayList<Integer>();

idle = 0;

CPU = **new** Task(nextTime(minTime, maxTime));

queve.add(**new** Task(nextTime(minTime, maxTime)));

queve.add(**new** Task(nextTime(minTime, maxTime)));

}

**private** **double** exp(**double** lambda) {

**return** (1 - Math.*exp*(-lambda \* Math.*random*()));

}

/\*\*

\* **@return** the cPU

\*/

**public** **synchronized** Task getCPU() {

**return** CPU;

}

/\*\*

\* **@return** the idle

\*/

**public** **synchronized** **int** getIdle() {

**return** idle;

}

/\*\*

\* **@return** the lambda

\*/

**public** **synchronized** **double** getLambda() {

**return** lambda;

}

/\*\*

\* **@return** the maxTime

\*/

**public** **synchronized** **int** getMaxTime() {

**return** maxTime;

}

/\*\*

\* **@return** the minTime

\*/

**public** **synchronized** **int** getMinTime() {

**return** minTime;

}

/\*\*

\* **@return** the queve

\*/

**public** **synchronized** ArrayList<Task> getQueve() {

**return** queve;

}

/\*\*

\* **@return** the waitings

\*/

**public** **synchronized** ArrayList<Integer> getWaitings() {

**return** waitings;

}

**public** **void** nextIteration() {

**if** (CPU != **null** && CPU.getTime() != 0)

CPU.decTime();

**else** **if** (queve.size() > 0) {

**if** (CPU != **null**) {

waitings.add(CPU.getWaiting());

}

CPU = queve.get(0);

queve.remove(0);

} **else** {

CPU = **null**;

idle++;

}

queveWaits();

putTask();

}

**public** **int** nextTime(**int** minTime, **int** maxTime) {

**if** (minTime <= 0 || maxTime <= 0)

**throw** **new** IllegalArgumentException("Time must be positive");

**if** (minTime > maxTime)

**throw** **new** IllegalArgumentException("minTime must less than maxTime");

Random r = **new** Random();

**int** time = minTime;

time += r.nextInt(maxTime - minTime + 1);

**return** time;

}

**private** **boolean** putTask() {

Task t = **new** Task(nextTime(minTime, maxTime));

**boolean** p = exp(lambda) > range;

// System.out.println(p);

**if** (p) {

// List must be sorted in ascending order!!!

**int** i = Collections.*binarySearch*(queve, t);

**if** (i < 0)

i = -i - 1;

queve.add(i, t);

**return** **true**;

}

**return** **false**;

}

**public** **void** queveWaits() {

**for** (Task t : queve)

t.waitOne();

}

**public** **int** averageWaiting(){

**int** summ = 0;

**for** (Integer i : waitings)

summ+=i;

summ/=waitings.size();

**return** summ;

}

/\*\*

\* **@param** cpu

\* the cPU to set

\*/

**public** **synchronized** **void** setCPU(Task cpu) {

CPU = cpu;

}

/\*\*

\* **@param** idle

\* the idle to set

\*/

**public** **synchronized** **void** setIdle(**int** idle) {

**this**.idle = idle;

}

/\*\*

\* **@param** lambda

\* the lambda to set

\*/

**public** **synchronized** **void** setLambda(**double** lambda) {

**this**.lambda = lambda;

}

/\*\*

\* **@param** maxTime

\* the maxTime to set

\*/

**public** **synchronized** **void** setMaxTime(**int** maxTime) {

**this**.maxTime = maxTime;

}

/\*\*

\* **@param** minTime

\* the minTime to set

\*/

**public** **synchronized** **void** setMinTime(**int** minTime) {

**this**.minTime = minTime;

}

/\*\*

\* **@param** queve

\* the queve to set

\*/

**public** **synchronized** **void** setQueve(ArrayList<Task> queve) {

**this**.queve = queve;

}

**public** **boolean** task2CPU() {

**if** (queve.size() > 0) {

CPU = queve.get(0);

queve.remove(0);

**return** **true**;

}

**return** **false**;

}

}

**package** shortjobsirst;

**import** java.util.Random;

**import** org.jfree.chart.ChartFactory;

**import** org.jfree.chart.JFreeChart;

**import** org.jfree.chart.plot.PlotOrientation;

**import** org.jfree.data.category.CategoryDataset;

**import** org.jfree.data.general.DatasetUtilities;

**import** org.jfree.data.xy.IntervalXYDataset;

**public** **class** Controller {

**private** **int** m = 30;

**int** n = 3;

**double** lambdaInit = 0.1;

**double** lambdaInc = 0.2;

**private** Random r = **new** Random();

**private** **int** scale = 50;

**private** **double**[][] avgWaitings = **new** **double**[n][m];

**private** **double**[][] dispersions = **new** **double**[n][m];

**private** **double**[][] idles = **new** **double**[n][m];

**public** **void** test() {

**double** lambda = lambdaInit;

**int** minTime = 1;

**int** maxTime = 5;

**for** (**int** i = 0; i < m; i++) {

**for** (**int** j = 0; j < n; j++) {

SJFSystem s = **new** SJFSystem(lambda, minTime, maxTime);

**for** (**int** l = 0; l < scale; l++) {

s.nextIteration();

System.*out*.println("#" + (l + 1) + " CPU: " + s.getCPU()

+ " Q: " + s.getQueve());

}

System.*out*.println("idle: " + s.getIdle());

System.*out*.println("waitTimes: " + s.getWaitings());

**double** mx = s.averageWaiting();

**double** D = 0;

**for** (Integer x : s.getWaitings())

D += Math.*pow*(mx - x.doubleValue(), 2);

D /= s.getWaitings().size();

avgWaitings[j][i] = mx;

dispersions[j][i] = D;

idles[j][i] = 100\*((**double**)s.getIdle()/(**double**)scale);

}

lambda+=lambdaInc;

}

}

**public** JFreeChart createChart(CategoryDataset categoryDataset, String name) {

JFreeChart chart = ChartFactory.*createBarChart*(name, // chart title

"Lambda", // domain axis label

"Value", // range axis label

categoryDataset, // data

PlotOrientation.*VERTICAL*, // orientation

**true**, // include legend

**true**, // tooltips

**false** // urls

);

**return** chart;

}

**public** JFreeChart createChart(IntervalXYDataset dataset) {

JFreeChart chart = ChartFactory.*createHistogram*(**null**, **null**, **null**,

dataset, PlotOrientation.*VERTICAL*, **true**, **false**, **false**);

chart.getXYPlot().setForegroundAlpha(0.75f);

**return** chart;

}

**public** String[] getColumnKeys(){

String[] columnKeys = **new** String[m];

**double** lambda = lambdaInit;

**for** (**int** i = 0; i < m; i++) {

columnKeys[i] = **new** StringBuilder().append(Double.*toString*(lambda).substring(0, 3)).toString();

lambda+=lambdaInc;

}

**return** columnKeys;

}

**public** CategoryDataset createDispersionDataset() {

String[] columnKeys = getColumnKeys();

**return** DatasetUtilities.*createCategoryDataset*(

**new** String[] { "1","2","3"}, columnKeys, dispersions);

}

**public** CategoryDataset createAvgWaitingsDataset() {

String[] columnKeys = getColumnKeys();

**double**[][] data = **new** **double**[1][m];

data[0] = avgWaitings[0];

**return** DatasetUtilities.*createCategoryDataset*(

**new** String[] { "Values" }, columnKeys, data);

}

**public** CategoryDataset createIdlesDataset() {

String[] columnKeys = getColumnKeys();

**double**[][] data = **new** **double**[1][m];

data[0] = idles[0];

**return** DatasetUtilities.*createCategoryDataset*(

**new** String[] { "Values" }, columnKeys, data);

}

**public** **double**[] getNumbers() {

**double**[] numbers = **new** **double**[n];

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

numbers[i] = r.nextDouble();

}

**return** numbers;

}

}

**package** shortjobsirst;

**import** java.awt.BorderLayout;

**import** javax.swing.BorderFactory;

**import** javax.swing.JFrame;

**import** javax.swing.JTabbedPane;

**import** javax.swing.SwingUtilities;

**import** javax.swing.UIManager;

**import** javax.swing.border.EtchedBorder;

**import** org.dyno.visual.swing.layouts.GroupLayout;

**import** org.jfree.chart.ChartPanel;

//VS4E -- DO NOT REMOVE THIS LINE!

**public** **class** Interface **extends** JFrame {

**private** **static** **final** **long** *serialVersionUID* = 1L;

**private** ChartPanel chartPanel1;

**private** ChartPanel chartPanel2;

**private** ChartPanel chartPanel3;

**private** Controller controller;

**private** **int** width = 1000;

**private** **int** height = 300;

**private** JTabbedPane jTabbedPane0;

**private** **static** **final** String *PREFERRED\_LOOK\_AND\_FEEL* = "javax.swing.plaf.metal.MetalLookAndFeel";

**public** Interface() {

initComponents();

}

**private** **void** initComponents() {

controller = **new** Controller();

controller.test();

add(getJTabbedPane0(), BorderLayout.*CENTER*);

setSize(width, height);

}

**private** JTabbedPane getJTabbedPane0() {

**if** (jTabbedPane0 == **null**) {

jTabbedPane0 = **new** JTabbedPane();

jTabbedPane0.addTab("Avg Wait", getChartPanel1());

jTabbedPane0.addTab("Idle", getChartPanel2());

jTabbedPane0.addTab("Wait disp", getChartPanel3());

}

**return** jTabbedPane0;

}

**public** ChartPanel getChartPanel1() {

**if** (chartPanel1 == **null**) {

chartPanel1 = **new** ChartPanel(controller.createChart(controller

.createAvgWaitingsDataset(), "Average waitings"));

chartPanel1.setPreferredSize(**new** java.awt.Dimension(500, 370));

chartPanel1.setMouseZoomable(**true**, **false**);

chartPanel1.setBorder(BorderFactory.*createEtchedBorder*(

EtchedBorder.*LOWERED*, **null**, **null**));

chartPanel1.setLayout(**new** GroupLayout());

}

**return** chartPanel1;

}

**public** ChartPanel getChartPanel2() {

**if** (chartPanel2 == **null**) {

chartPanel2 = **new** ChartPanel(controller.createChart(controller

.createIdlesDataset(), "Idle"));

chartPanel2.setPreferredSize(**new** java.awt.Dimension(width, height));

chartPanel2.setMouseZoomable(**true**, **false**);

chartPanel2.setBorder(BorderFactory.*createEtchedBorder*(

EtchedBorder.*LOWERED*, **null**, **null**));

}

**return** chartPanel2;

}

**public** ChartPanel getChartPanel3() {

**if** (chartPanel3 == **null**) {

chartPanel3 = **new** ChartPanel(controller.createChart(controller

.createDispersionDataset(), "Waitings dispersion"));

chartPanel3.setPreferredSize(**new** java.awt.Dimension(width, height));

chartPanel3.setMouseZoomable(**true**, **false**);

chartPanel3.setBorder(BorderFactory.*createEtchedBorder*(

EtchedBorder.*LOWERED*, **null**, **null**));

}

**return** chartPanel3;

}

**private** **static** **void** installLnF() {

**try** {

String lnfClassname = *PREFERRED\_LOOK\_AND\_FEEL*;

**if** (lnfClassname == **null**)

lnfClassname = UIManager.*getCrossPlatformLookAndFeelClassName*();

UIManager.*setLookAndFeel*(lnfClassname);

} **catch** (Exception e) {

System.*err*.println("Cannot install " + *PREFERRED\_LOOK\_AND\_FEEL*

+ " on this platform:" + e.getMessage());

}

}

**public** **static** **void** main(String[] args) {

*installLnF*();

SwingUtilities.*invokeLater*(**new** Runnable() {

@Override

**public** **void** run() {

Interface frame = **new** Interface();

frame.setDefaultCloseOperation(Interface.*EXIT\_ON\_CLOSE*);

frame.setTitle("Interface");

frame.getContentPane().setPreferredSize(frame.getSize());

frame.pack();

frame.setLocationRelativeTo(**null**);

frame.setVisible(**true**);

}

});

}

}

# Приклад покрокової візуалізації

#1 CPU: 2 Q: [2, 5, 4]

#2 CPU: 1 Q: [2, 5, 5, 4]

#3 CPU: 0 Q: [2, 5, 5, 4]

#4 CPU: 2 Q: [1, 5, 5, 4]

#5 CPU: 1 Q: [1, 3, 5, 5, 4]

#6 CPU: 0 Q: [1, 2, 3, 5, 5, 4]

#7 CPU: 1 Q: [2, 3, 5, 5, 5, 4]

#8 CPU: 0 Q: [2, 3, 5, 5, 5, 5, 4]

#9 CPU: 2 Q: [1, 3, 5, 5, 5, 5, 4]

#10 CPU: 1 Q: [1, 3, 3, 5, 5, 5, 5, 4]

#11 CPU: 0 Q: [1, 2, 3, 3, 5, 5, 5, 5, 4]

#12 CPU: 1 Q: [1, 2, 3, 3, 5, 5, 5, 5, 4]

#13 CPU: 0 Q: [1, 2, 2, 3, 3, 5, 5, 5, 5, 4]

#14 CPU: 1 Q: [2, 2, 3, 3, 3, 5, 5, 5, 5, 4]

#15 CPU: 0 Q: [2, 2, 3, 3, 3, 5, 5, 5, 5, 5, 4]

#16 CPU: 2 Q: [2, 3, 3, 3, 4, 5, 5, 5, 5, 5, 4]

#17 CPU: 1 Q: [2, 3, 3, 3, 4, 5, 5, 5, 5, 5, 5, 4]

#18 CPU: 0 Q: [2, 2, 3, 3, 3, 4, 5, 5, 5, 5, 5, 5, 4]

#19 CPU: 2 Q: [1, 2, 3, 3, 3, 4, 5, 5, 5, 5, 5, 5, 4]

#20 CPU: 1 Q: [1, 1, 2, 3, 3, 3, 4, 5, 5, 5, 5, 5, 5, 4]

#21 CPU: 0 Q: [1, 1, 2, 2, 3, 3, 3, 4, 5, 5, 5, 5, 5, 5, 4]

#22 CPU: 1 Q: [1, 2, 2, 3, 3, 3, 4, 5, 5, 5, 5, 5, 5, 4]

#23 CPU: 0 Q: [1, 2, 2, 2, 3, 3, 3, 4, 5, 5, 5, 5, 5, 5, 4]

#24 CPU: 1 Q: [2, 2, 2, 3, 3, 3, 4, 5, 5, 5, 5, 5, 5, 5, 4]

#25 CPU: 0 Q: [1, 2, 2, 2, 3, 3, 3, 4, 5, 5, 5, 5, 5, 5, 5, 4]

#26 CPU: 1 Q: [1, 2, 2, 2, 3, 3, 3, 4, 5, 5, 5, 5, 5, 5, 5, 4]

#27 CPU: 0 Q: [1, 1, 2, 2, 2, 3, 3, 3, 4, 5, 5, 5, 5, 5, 5, 5, 4]

#28 CPU: 1 Q: [1, 1, 2, 2, 2, 3, 3, 3, 4, 5, 5, 5, 5, 5, 5, 5, 4]

#29 CPU: 0 Q: [1, 1, 2, 2, 2, 2, 3, 3, 3, 4, 5, 5, 5, 5, 5, 5, 5, 4]

#30 CPU: 1 Q: [1, 2, 2, 2, 2, 3, 3, 3, 4, 4, 5, 5, 5, 5, 5, 5, 5, 4]

idle: 0

waitTimes: [0, 2, 2, 2, 2, 1, 2, 0, 1, 4, 0, 0, 1, 0, 1, 1, 11, 18, 1, 0, 3]

# Висновки

В результаті виконання лабораторної роботи була реалізована система управління чергами за алгоритмом Short Job First. При реалізації нові заявки в чергу включалися за двійковим пошуком. Тому черга завжди відсортована за зростанням, і першими в ЦП на виконання заносяться найкоротші заявки.

Використання SJF знижує середній час очікування. Найбільша складність у практичній реалізації SJF полягає в неможливості наперед визначити величину часу наступного обслуговування.

Тому стратегія SJF часто застосовується в довгострокових (статичних) планувальниках, що обслуговують пакетний режим. У цьому випадку замість величини часу наступного обслуговування використовується допустимий максимальний час виконання завдання, котре програміст має специфікувати перед відправкою завдання в пакет.