**Вступ**

*Изм.*

*Лист*

*№ докум.*

*Підпис*

*Дата*

*Лист*

*2*

###### *ІАЛЦ.462637.003.ПЗ*

На сьогоднішній день підвищення продуктивності обчислювальних систем за рахунок підвищення тактової частоти обчислювальних елементів і підвищення зменшення розмірів елементів має технологічні обмеження, тому збільшується значення такого підходу як збільшення кількості елементів. При цьому важливу роль відіграють також алгоритми управління ресурсами в таких обчислювальних системах.

Вони можуть бути розроблені як для ідеального варіанту, при якому не враховуються взаємозв'язки між завданнями, а вузли обчислювальної системи мають однакову продуктивність, так і для більш реалістичних варіантів системи, при яких вузли системи можуть мати різну продуктивність, або завдання можуть бути взаємопов'язані між собою.

Оскільки кількість варіантів стратегій планування, типів завдань, їх зв'язності є дуже великою, стає зрозумілим, що не існує ідеального або точного оптимального алгоритму розподілу. Тому в планувальниках застосовуються різні логічно обґрунтовані стратегії планування, що дають результати, наближені до оптимальних. Вагомі переваги також може дати врахування особливостей топології системи.

В даній роботі розглядається багатопроцесорна обчислювальна система з організацією каналу даних на основі загальної шини.

Планувальник розроблений для даної системи повинен враховувати те, що в один момент часу може бути тільки одна передача між вузлами.

1. Огляд існуючих рішень

У даному розділі представлений короткий огляд трьох відомих алгоритмів статичного планування для неоднорідних систем - алгоритм планування динамічно рівнів (DLS) , алгоритм нормованого - Мінімального Часу, алгоритмBubble Scheduling and Allocation.

**1.1. Алгоритм планування динамічних рівнів – DLS.**

*Изм.*

*Лист*

*№ докум.*

*Підпис*

*Дата*

*Лист*

*3*

###### *ІАЛЦ.462637.003.ПЗ*

Алгоритм DLS використовує атрибут, який називається динамічний рівень (DL), що дорівнює різниці між статичним b-рівнем вершини і її найбільш раннім часом початку на процесорі. На кожному кроці планування, алгоритм обчислює динамічний рівень для кожної вершини в пулі готових вершин для всіх процесорів. Пара вершина-процесор, динамічний рівень якої найбільший вибирається для планування.

**1.2. Алгоритм найбільш раннього часу закінчення – HEFT.**

Даний алгоритм є представником класу алгоритмів планування за списками. Алгоритм виконується у два етапи. На першому етапі кожній задачі присвоюється пріоритет, на другому етапі кожна вершина зі списку присвоюється такому процесору. коли всі задачі з пріоритетами, ми плануємо кожну з них, починаючи з вершини з найвищим пріоритетом. Задача з найбільшим пріоритетом для якої всі залежні задачі виконалися плануємо на процесор, який призведе до найбільш раннього часу закінчення цієї задачі. Цей час закінчення залежить від часу спілкування щоб відправити всю необхідну інформацію для процесора, часу обчислення задачі в процесорі, і часу, коли цей процесор стає доступним (він може бути зайнятий іншою задачею). Алгоритм HEFT, по суті є жадним і не в змозі приймати короткострокові жертви для довгострокової вигоди.

**1.3** **Bubble Scheduling and Allocation**

Алгоритм розроблений Квоком та Ахмадом у 1995 році. Спочатку усі вершини направляються на один процесор, який має найбільшу кількість зв’язків i стає основним. На другій фазі починається переміщення вершин на сусідні процесори, якщо при цьому покращується час запуску. Після цього основним стає один із сусідніх процесорів i алгоритм повторюється .

2. Опис алгоритму

*Изм.*

*Лист*

*№ докум.*

*Підпис*

*Дата*

*Лист*

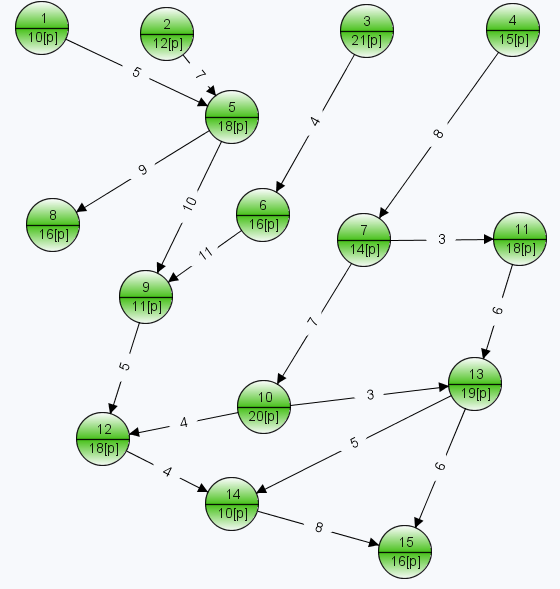
*4*

###### *ІАЛЦ.462637.003.ПЗ*

Для вирішення задачі було обрано алгоритм Bottom-Up . Даний алгоритм алгоритм планує виконання критичного шляху на одному процесорі, iншi вершини розподіляються так ,щоб збалансувати навантаження на процесори.

**3.Приклад роботи програми**

На малюнку нижче зображений граф(умова) задачі.



Введення даних в систему(задання графа) відбувається вручну .

Проаналізувавши введені дані , програма будує розподілення задач по процесорах , не використовуючи планувальник.

*Изм.*

*Лист*

*№ докум.*

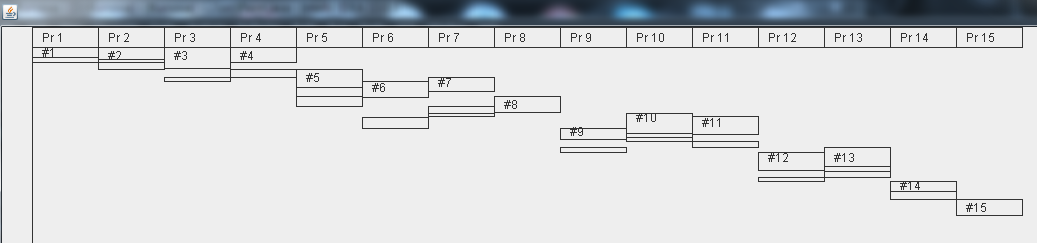
*Підпис*

*Дата*

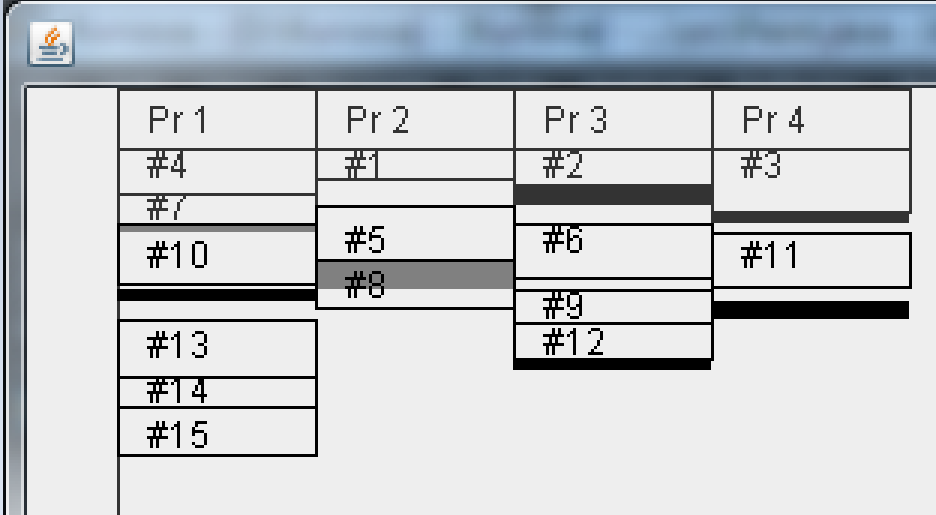
*Лист*

*5*

###### *ІАЛЦ.462637.003.ПЗ*



Далі відбувається розподілення задач на 4 процесорах , за обраним алгоритмом та з урахуванням топології.



4. Аналіз результатів

*Изм.*

*Лист*

*№ докум.*

*Підпис*

*Дата*

*Лист*

*6*

###### *ІАЛЦ.462637.003.ПЗ*

В попередньому розділі продемонстровано приклад роботи програми. . Оцінивши результати розподілу можна сказати, що однією з причин великої кількості тактів є час, який займає пересилка даних. Вагомою причиною є також те, що в один момент часу може виконуватись тільки одна пересилка, оскільки в топологія системи є загальна шина.

**5.Висновок**

При виконанні курсової роботи було реалізовано один з алгоритмів планування задач на ресурси неоднорідної обчислювальної системи з топологією загальна шина. Для більш наглядного зображення роботи алгоритму була розроблена програма, яка демонструє процес планування по процесорах. Основна ідея алгоритму – помістити вершини, що складають критичний шлях , на один процесор.

**6.Список використаної літератури**

1.Конспект лекцій з курсу: «Системне програмне забезпечення»

2.Y. Kwok and I. Ahmad, "Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multi¬processors," IEEE Trans. Parallel and Distributed Systems, vol. 7, no. 5, pp. 506-521, May 1996.  
3.[Таненбаум Э. С.](https://ru.wikipedia.org/wiki/%D0%A2%D0%B0%D0%BD%D0%B5%D0%BD%D0%B1%D0%B0%D1%83%D0%BC,_%D0%AD%D0%BD%D0%B4%D1%80%D1%8E), Вудхалл А. С. Операционные системы. Разработка и реализация = Operating Systems: Design and Implementation. — 3-е изд. — СПб.: [Питер](https://ru.wikipedia.org/wiki/%D0%9F%D0%B8%D1%82%D0%B5%D1%80_(%D0%B8%D0%B7%D0%B4%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D1%81%D1%82%D0%B2%D0%BE)), 2007. — 704 с. — [ISBN 978-5-469-01403-4](https://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D1%83%D0%B6%D0%B5%D0%B1%D0%BD%D0%B0%D1%8F:%D0%98%D1%81%D1%82%D0%BE%D1%87%D0%BD%D0%B8%D0%BA%D0%B8_%D0%BA%D0%BD%D0%B8%D0%B3/9785469014034).

**Додаток з лістингом програми**

**package** kursach;

**import** javax.swing.SwingUtilities;

**public** **class** Main {

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

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

@Override

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

**new** MainFrame();

}

});

}

}

**package** kursach;

**import** java.awt.Toolkit;

**import** javax.swing.JFrame;

**public** **class** MainFrame **extends** JFrame {

**public** MainFrame() {

setTitle("Planner");

setDefaultCloseOperation(JFrame.*EXIT\_ON\_CLOSE*);

Toolkit kit = Toolkit.*getDefaultToolkit*();

setLocation(kit.getScreenSize().width / 2 - 400,

kit.getScreenSize().height / 2 - 300);

setSize(800, 600);

setResizable(**false**);

add(**new** MainPanel());

setVisible(**true**);

}

}

**package** kursach;

**import** java.awt.BorderLayout;

**import** java.awt.event.ActionEvent;

**import** java.awt.event.ActionListener;

**import** java.util.ArrayList;

**import** java.util.Random;

**public** **class** MainPanel **extends** JComponent **implements** ActionListener {

**private** Object[][] data;

**private** Object[][] data2;

**private** JTable timeMatr;

**private** **int** nodeNumber;

**private** **int** procNumber;

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

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

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

ArrayList<Processor> proc;

Marker marker;

ArrayList<Vertex> tasks = **new** ArrayList<Vertex>();

ArrayList<Vertex> readyTasks = **new** ArrayList<Vertex>();

ArrayList<Connection> connections = **new** ArrayList<Connection>();

**private** **int** completeTime;

**private** **int** stepCounter;

Object[][] schedule;

**public** **void** prepare() {

**for** (**int** i = 0; i < lpf.get(0).size(); i++) {

**for** (**int** j = 0; j < tasks.size(); j++) {

**if** (tasks.get(j).getId() == (lpf.get(0).get(i) - 1)) {

tasks.get(j).setReady(**true**);

}

}

}

getReady();

**for** (**int** i = 0; i < readyTasks.size(); i++) {

proc.get(i).setT(readyTasks.get(i), readyTasks.get(i).getId());

}

completeTime = calcCompleteTime();

}

**public** **void** step() {

**if** (graphFinished() != **true**) {

System.*out*.println("--------------Step #" + stepCounter

+ "--------------");

updateTasks();

getReady();

Connection maxC = findMaxConnection();

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

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

ArrayList<Task> tmpTasks;

ArrayList<Integer> tmpID;

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

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

ArrayList<Vertex> V = **new** ArrayList<Vertex>();

**for** (**int** i = 0; i < proc.size(); i++) {

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

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

absentParents = findAbsentParents(

getTaskById(maxC.getEndNode()).getParents(), proc

.get(i).getExecutes());

System.*out*.println("proc#" + i + " need=" + absentParents);

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

**for** (**int** j = 0; j < absentParents.size(); j++) {

Connection con = findConnectionByStartNode(

absentParents.get(j), maxC.getEndNode());

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

tmpTasks.add(con);

tmpID.add(-1);

}

}

}

tmpTasks.add(getTaskById(maxC.getEndNode()));

V.add(getTaskById(maxC.getEndNode()));

tmpID.add(maxC.getEndNode());

T.add(tmpTasks);

ID.add(tmpID);

proc.get(i).setAbsPar(absentParents);

tempTime.add(proc.get(i).testLoad(tmpTasks));

}

System.*out*.println("tmpTime=" + tempTime);

**int** idx = findMinCompleteTime(tempTime);

while(true){

if(!marker.isBusy){

proc.get(idx).catchMarker(marker, T.get(idx));

break;

}else{

marker.setBusy(false);

}

}

proc.get(idx).load(T.get(idx), ID.get(idx));

**for** (**int** i = 0; i < V.size(); i++) {

**for** (**int** j = 0; j < T.get(idx).size(); j++) {

**if** (V.get(i).equals(T.get(idx).get(j))) {

System.out.println("Task to Finish = " + V.get(i));

finishTask(V.get(i));

**break**;

}

}

}

completeTime = calcCompleteTime();

**for** (**int** i = 0; i < proc.size(); i++) {

System.*out*.println(proc.get(i));

}

stepCounter++;

System.out.println("Tasks--->" + tasks);

} **else** {

showPlan.setEnabled(**true**);

System.*out*.println("Finished");

JOptionPane.*showMessageDialog*(**null**, "Scheduling Complete!!!", "Done",

JOptionPane.*INFORMATION\_MESSAGE*);

}

}

**public** **void** finishTask(Vertex v) {

**for** (**int** i = 0; i < tasks.size(); i++) {

**if** (v.equals(tasks.get(i))) {

tasks.get(i).setFinished(**true**);

tasks.get(i).setEndTime(endTime)

**break**;

}

}

}

**public** **int** findMinCompleteTime(ArrayList<Integer> time) {

**int** min = time.get(0), minId = 0;

**for** (**int** i = 0; i < time.size(); i++) {

**if** (time.get(i) < min) {

min = time.get(i);

minId = i;

}

}

System.*out*.println("MinTime=" + min + ", id=" + minId);

**return** minId;

}

**public** Connection findConnectionByStartNode(**int** start, **int** end) {

Connection c = **null**;

**for** (**int** i = 0; i < connections.size(); i++) {

**if** ((start == connections.get(i).getStartNode())

&& (end == connections.get(i).getEndNode())) {

c = connections.get(i);

**break**;

}

}

**return** c;

}

**public** ArrayList<Integer> findAbsentParents(ArrayList<Integer> par,

ArrayList<Integer> availablePar) {

ArrayList<Integer> tmp1 = **new** ArrayList<Integer>();

ArrayList<Integer> tmp2 = **new** ArrayList<Integer>();

**for** (**int** i = 0; i < par.size(); i++) {

tmp1.add(par.get(i));

}

**for** (**int** i = 0; i < availablePar.size(); i++) {

tmp2.add(availablePar.get(i));

}

System.*out*.println("tmp1=" + tmp1);

System.*out*.println("tmp2=" + tmp2);

**if** (tmp1.size() > tmp2.size()) {

**for** (**int** i = 0; i < tmp1.size(); i++) {

**for** (**int** j = 0; j < tmp2.size(); j++) {

**if** (tmp1.get(i) == tmp2.get(j)) {

tmp1.remove(i);

}

}

}

} **else** {

**for** (**int** i = 0; i < tmp2.size(); i++) {

**for** (**int** j = 0; j < tmp1.size(); j++) {

**if** (tmp2.get(i) == tmp1.get(j)) {

tmp1.remove(j);

}

}

}

}

**return** tmp1;

}

**public** **int** calcCompleteTime() {

**int** completeTime = 0;

**for** (**int** i = 0; i < proc.size(); i++) {

**if** (proc.get(i).completeTime() > completeTime)

completeTime = proc.get(i).completeTime();

}

System.*out*.println("CompleteTime=" + completeTime);

**return** completeTime;

}

**public** **boolean** graphFinished() {

**int** countFinish = 0;

**for** (**int** i = 0; i < tasks.size(); i++) {

**if** (tasks.get(i).isFinished() == **true**) {

System.out.println("Finished ->" + tasks.get(i));

countFinish++;

}

}

**if** (countFinish == tasks.size()) {

**return** **true**;

} **else** {

**return** **false**;

}

}

**public** **void** getReady() {

System.*out*.println("--------------GetReady--------------");

readyTasks = **new** ArrayList<Vertex>();

**for** (**int** i = 0; i < tasks.size(); i++) {

**if** (tasks.get(i).isReady() == **true**) {

**if** (tasks.get(i).isFinished() == **false**) {

readyTasks.add(tasks.get(i));

}

}

}

System.*out*.println("readyTasks" + readyTasks);

System.*out*.println("------------------------------------");

}

**public** **void** updateTasks() {

System.*out*.println("--------------Update--------------");

**for** (**int** i = 0; i < tasks.size(); i++) {

**if** (tasks.get(i).getParents() != **null**) {

**if** (isReady(tasks.get(i)) == **true**) {

tasks.get(i).setReady(**true**);

}

}

}

System.*out*.println("----------------------------------");

}

**public** Vertex getTaskById(**int** id) {

Vertex t = **null**;

**for** (**int** i = 0; i < tasks.size(); i++) {

**if** (tasks.get(i).getId() == id) {

t = tasks.get(i);

}

}

**return** t;

}

**public** **boolean** isReady(Vertex t) {

**int** countReady = 0;

**for** (**int** i = 0; i < t.getParents().size(); i++) {

**if** (tasks.get(t.getParents().get(i)).isFinished() == **true**) {

countReady++;

}

}

**if** (countReady == t.getParents().size()) {

**return** **true**;

} **else** {

**return** **false**;

}

}

**public** Connection findMaxConnection() {

Connection maxConn = connections.get(0);

**for** (**int** i = 0; i < connections.size(); i++) {

**if** (connections.get(i).getWeight() > maxConn.getWeight()) {

**if** (getTaskById(connections.get(i).getEndNode()).isReady() == **true**) {

**if** (getTaskById(connections.get(i).getEndNode())

.isFinished() == **false**) {

maxConn = connections.get(i);

}

}

}

}

System.*out*.println("maxConn=" + maxConn);

**return** maxConn;

}

**public** **void** findParents() {

**for** (**int** i = 0; i < tasks.size(); i++) {

**int** id = tasks.get(i).getId();

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

**for** (**int** j = 0; j < tasks.size(); j++) {

**if** (tasks.get(j).getChildren().contains(id)) {

parents.add(tasks.get(j).getId());

}

}

tasks.get(i).setParents(parents);

}

}

**public** **int** checkTables() {

**int** errorCode = 0;

**try** {

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

Integer.*parseInt*(data2[0][i].toString());

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

Integer.*parseInt*(data[i][j].toString());

}

}

} **catch** (NumberFormatException ex) {

errorCode = -1;

JOptionPane.*showMessageDialog*(**null**, "Invalid input", "Error",

JOptionPane.*ERROR\_MESSAGE*);

}

**return** errorCode;

}

**public** **void** tableToNodes() {

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

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

**int** weigth = Integer.*parseInt*(timeMatr.getValueAt(0, i).toString());

**int** id = i;

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

**if** (Integer

.*parseInt*(connectionMatr.getValueAt(i, j).toString()) != 0) {

children.add(j);

connections.add(**new** Connection(i, j, Integer

.*parseInt*(connectionMatr.getValueAt(i, j)

.toString())));

}

}

tasks.add(**new** Vertex(id, children, weigth));

}

}

**public** **void** getLPF() {

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

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

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

row.add(Integer.*parseInt*(connectionMatr.getValueAt(j, i)

.toString()));

}

idx.add(i + 1);

d.add(row);

}

System.out.println("-----Src matr-----");

sysout(d);

System.out.println("------------------");

**while** (d.size() > 0) {

ArrayList<Integer> zeros = findZeroRows();

removeZeros(zeros);

}

sysout(lpf);

}

**public** **void** sysout(ArrayList<ArrayList<Integer>> arr) {

**for** (**int** j = 0; j < arr.size(); j++) {

System.*out*.println(arr.get(j));

}

}

**public** ArrayList<Integer> findZeroRows() {

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

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

**for** (**int** i = 0; i < d.size(); i++) {

**if** (isZero(d.get(i)) == **true**) {

**int** index = idx.get(i);

layer.add(index);

zeros.add(i);

}

}

lpf.add(layer);

**return** zeros;

}

**public** **void** removeZeros(ArrayList<Integer> zeros) {

**for** (**int** i = zeros.size() - 1; i >= 0; i--) {

**int** ind = zeros.get(i);

d.remove(ind);

idx.remove(ind);

**for** (**int** j = 0; j < d.size(); j++) {

d.get(j).remove(ind);

}

}

}

**public** **boolean** isZero(ArrayList<Integer> row) {

**int** count = 0;

**for** (**int** i = 0; i < row.size(); i++) {

**if** (row.get(i) == 0) {

count++;

}

}

**if** (count == row.size()) {

**return** **true**;

} **else** {

**return** **false**;

}

}

**public** **void** findLPFWidth() {

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

**for** (**int** i = 0; i < lpf.size(); i++) {

w.add(lpf.get(i).size());

}

**int** max = 0;

**for** (**int** i = 0; i < w.size(); i++) {

**if** (w.get(i) > max) {

max = w.get(i);

}

}

procNumber = max;

}

}

**package** kursach;

**public** **abstract** **class** Task {

**private** **int** weight;

**private** String name;

**private** **boolean** finished;

**public** Task(**int** weight, String name) {

**this**.weight = weight;

**this**.name = name;

**this**.finished = **false**;

}

**public** **int** getWeight() {

**return** weight;

}

**public** **void** setWeight(**int** weight) {

**this**.weight = weight;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **boolean** isFinished() {

**return** finished;

}

**public** **void** setFinished(**boolean** finished) {

**this**.finished = finished;

}

}

**package** kursach;

**import** java.util.ArrayList;

**public** **class** Vertex **extends** Task {

**private** **int** id;

**private** ArrayList<Integer> parents;

**private** ArrayList<Integer> children;

**private** **boolean** ready;

**private** **int** endTime;

**public** Vertex(**int** id, ArrayList<Integer> children, **int** weight) {

**super**(weight, ""+id);

**this**.id = id;

**this**.children = children;

**this**.ready = **false**;

}

**public** ArrayList<Integer> getChildren() {

**return** children;

}

**public** ArrayList<Integer> getParents() {

**return** parents;

}

**public** **void** setParents(ArrayList<Integer> parents) {

**this**.parents = parents;

}

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** **void** setChildren(ArrayList<Integer> children) {

**this**.children = children;

}

**public** **boolean** isReady() {

**return** ready;

}

**public** **void** setReady(**boolean** ready) {

**this**.ready = ready;

}

**public** **int** getEndTime() {

**return** endTime;

}

**public** **void** setEndTime(**int** endTime) {

**this**.endTime = endTime;

}

@Override

**public** String toString() {

**return** "Vertex [id=" + id + ", parents=" + parents + ", children="

+ children + ", weigth=" + **super**.getWeight() + ", ready="

+ ready + ", finished=" + **super**.isFinished() + "]";

}

}

**package** kursach;

**public** **class** Connection **extends** Task {

**private** **int** startNode;

**private** **int** endNode;

**public** Connection(**int** startNode, **int** endNode, **int** weight) {

**super**(weight, startNode+"->"+endNode);

**this**.startNode = startNode;

**this**.endNode = endNode;

}

**public** **int** getStartNode() {

**return** startNode;

}

**public** **void** setStartNode(**int** startNode) {

**this**.startNode = startNode;

}

**public** **int** getEndNode() {

**return** endNode;

}

**public** **void** setEndNode(**int** endNode) {

**this**.endNode = endNode;

}

@Override

**public** String toString() {

**return** "Connection [" + startNode + "--" + **super**.getWeight() + "->"

+ endNode + "]";

}

}

**package** kursach;

**public** **class** Marker {

**private** **int** sender;

**private** **int** reciever;

**private** **int** businessTime;

**private** **boolean** busy;

**public** **int** getSender() {

**return** sender;

}

**public** **void** setSender(**int** sender) {

**this**.sender = sender;

}

**public** **int** getReciever() {

**return** reciever;

}

**public** **void** setReciever(**int** reciever) {

**this**.reciever = reciever;

}

**public** **boolean** isBusy() {

**return** busy;

}

**public** **void** setBusy(**boolean** busy) {

**this**.busy = busy;

}

**public** **int** getBusinessTime() {

**return** businessTime;

}

**public** **void** setBusinessTime(**int** businessTime) {

**this**.businessTime = businessTime;

}

@Override

**public** String toString() {

**return** "Marker [sender=" + sender + ", reciever=" + reciever

+ ", businessTime=" + businessTime + ", busy=" + busy + "]";

}

}

**package** kursach;

**import** java.util.ArrayList;

**public** **class** Processor {

**private** **int** id;

**private** **int** productCoef;

**private** **boolean** busy;

**private** ArrayList<Integer> executes = **new** ArrayList<Integer>();

**private** ArrayList<Task> tasks = **new** ArrayList<Task>();

**private** ArrayList<Integer> absentParents = **new** ArrayList<Integer>();

**private** Task t;

**public** Processor(**int** id, **int** productCoef) {

**this**.id = id;

**this**.productCoef = productCoef;

**this**.busy = **false**;

}

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** **int** getProductCoef() {

**return** productCoef;

}

**public** **void** setProductCoef(**int** productCoef) {

**this**.productCoef = productCoef;

}

**public** **boolean** isBusy() {

**return** busy;

}

**public** **void** setBusy(**boolean** busy) {

**this**.busy = busy;

}

**public** **void** catchMarker(Marker m, ArrayList<Task> t) {

m.setBusy(**true**);

for(int i=0;i<t.size();i++){

if(isNotConnection(t.get(i))==false){

m.setBusinessTime(t.getWeight());

}

}

tasks.add(t);

}

**public** Task getT() {

**return** t;

}

**public** ArrayList<Task> getTasks() {

**return** tasks;

}

**public** **void** setT(Task t, **int** id) {

**if** (isNotConnection(t)) {

**this**.executes.add(id);

}

**this**.t = t;

**this**.tasks.add(**this**.t);

**this**.t.setFinished(**true**);

}

**public** ArrayList<Integer> getExecutes() {

**return** executes;

}

**public** **void** setAbsPar(ArrayList<Integer> ap) {

this.absentParents = ap;

}

**public** **boolean** isNotConnection(Task t) {

**if** (t.getClass() == Vertex.**class**) {

**return** **true**;

} **else** {

**return** **false**;

}

}

**public** **void** load(ArrayList<Task> t, ArrayList<Integer> id) {

**for** (**int** i = 0; i < t.size(); i++) {

**if** (isNotConnection(t.get(i))) {

**this**.executes.add(id.get(i));

}

**this**.t = t.get(i);

**this**.tasks.add(t.get(i));

}

}

**public** **int** testLoad(ArrayList<Task> t) {

Task tmp1 = **this**.t;

ArrayList<Task> tmp2 = **new** ArrayList<Task>();

**for** (**int** i = 0; i < tasks.size(); i++) {

tmp2.add(tasks.get(i));

}

**for** (**int** i = 0; i < t.size(); i++) {

tasks.add(t.get(i));

}

**int** CT = completeTime();

**this**.t = tmp1;

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

**for** (**int** i = 0; i < tmp2.size(); i++) {

tasks.add(tmp2.get(i));

}

**return** CT;

}

**public** **int** completeTime() {

**int** time = 0;

**for** (**int** i = 0; i < tasks.size(); i++) {

**for** (**int** i = 0; i < tasks.size(); i++) {

**if**(absentParents.contains(tasks.get(i).getParents().get(j))){

time += tasks.get(i).getWeight() + tasks.get(i).getParents().get(j).getEndTime();

} **else** {

time += tasks.get(i).getWeight();

}

}

}

System.*out*.println("Process #" + id + ": CT=" + time);

**return** time;

}

@Override

**public** String toString() {

**return** "Process [id=" + id + ", productCoef=" + productCoef + ", busy="

+ busy + ",\nTasks:\n" + tasks

+ "]";

}

}