Форма № Н-6.01

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

___Кафедра обчислювальної техніки___
(повна назва кафедри, циклової комісії)

# КУРСОВИЙ ПРОЕКТ

## “ Системне програмне забезпечення 2. ”Операційні системи”
(назва дисципліни)

на тему: «Розробка просторового планувальника задач для багатопроцесорної системи з топологією «ланцюг процесорів - неоднорідний»»

Студента 4 курсу групи___ІО-93___

напряму підготовки **6.050102 Комп’ютерна інженерія**____
_____Свинарчук С. В._____
(прізвище та ініціали)
Керівник ___професор, доктор технічних наук_____
_____Сімоненко В.П._____
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Національна оцінка _____
Кількість балів: _____Оцінка: ECTS _____

Члени комісії _____ _____
(підпис) (вчене звання, науковий ступінь, прізвище та ініціали)
_____ _____
(підпис) (вчене звання, науковий ступінь, прізвище та ініціали)
_____ _____
(підпис) (вчене звання, науковий ступінь, прізвище та ініціали

Київ - 2013 рік

ЗМІСТ

## ВСТУП

Ефективне планування паралельної програми по процесорах є важливим для досягнення високої продуктивності паралельної обчислювальної системи. Коли структура паралельної програми: час виконання задач, залежність задач одна від одної, зв'язки завдань і синхронізація заздалегідь відомі, планування може бути виконано статично за час компіляції.

В даний час дуже часто доводиться обробляти великі об’єми інформації і ефективне використання багатопроцесорної системи є дуже важливим. А для цього потрібно розробити ефективний алгоритм планування , враховуючи при цьому особливості архітектури процесора і задач які будуть на ньому виконуватися.

У даній роботі представлений алгоритм для реалізації просторового планувальника для розподілу завдань у багатопроцесорній системі з топологією «Ланцюг процесорів - неоднорідний» та програма для покрокового моделювання виконання задач в такій системі.

## 1. Постановка задачі

Вихідні дані для побудови планувальника:

1) обчислювальна система, яка задана графом системи G={V, U, WV, WU},

де: V={V1,...,Vn}, Vi — вершина (підзадача),

i = 1,n; n — кількість вершин графу G;

U={Ul,...,Um}, m — кількість дуг графу G,

де: Ul={Vi, Vj};

$WV_R=g(V_l)$ - ваги вузлів Xi;

$WU_l=f(U_l)$ - ваги ребер.

2) потік завдань, які потрібно розподілити на заданій системі задається графом даних $G_J=(E_J,U_J,WE_J,WU_J)$, де

$E_J=\{Y_i \mid i=1,2,..,m\}$ - завдання;

$WE_J=h(Yi)$ - ваги завдань (тривалість виконання);

$U_J=\{(Yi,Yj) \mid Yi \in E_J, Yj \in E_J\}$ - дуги потоку (зв'язками між завданнями);

$WU_J=k(U_J)$ - ваги дуг потоку.

Під час розробки планувальника проаналізувати вхідні дані, визначити кількість процесорів, промоделювати роботу планувальника.

## 2. Модель топології багатопроцесорної системи

«Ланцюг процесорів» - топологія комп’ютерної системи в якій процесор має зв'язок з сусіднім (рис. 1). Топологія є доволі простою і дуже часто використовується в багатопроцесорних системах.

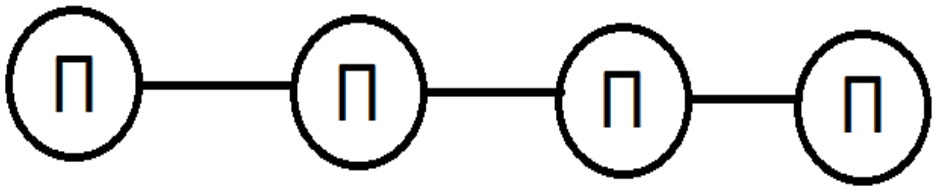


Рисунок 1 – Топологія «Ланцюг просесорів»

Топологія ланцюг процесорів має наступні властивості:

1) кожен процесор має зв'язок лише з сусіднім процесором;

2) процесор може передавати повідомлення відразу двом своїм сусідам;

3) процесор може одночасно виконувати обчислення і передавати інформацію;

4) перший та останній процесори не мають зв’язку між собою;

5) кожен процесор має свою швидкодію.

## 3. Огляд існуючих рішень

Розглянемо 4 існуючих алгоритму планування:

1) Розглянемо найпростіший («наївний») невитісняльний алгоритм, у якому потоки ставлять на виконання в порядку їхньої появи у системі й виконують до переходу в стан очікування, явної передачі керування або завершення. Чергу готових потоків при цьому організовують за принципом FIFO, тому алгоритм називають алгоритмом FIFO. Як тільки в системі створюється новий потік, його керуючий блок додається у хвіст черги. Коли процесор звільняється, його надають потоку з голови черги. У такого алгоритму багато недоліків:

• він за визначенням є невитісняльним;

• середній час відгуку для нього може бути доволі значним (наприклад, якщо

першим надійде потік із довгим інтервалом використання процесора, інші потоки чекатимуть, навіть якщо вони самі використовують тільки короткі інтервали);

• він підлягає ефекту конвою (convoy effect).

2) Планування за принципом кругового чергування припускає, що всі потоки однаково важливі. В іншому разі необхідно застосовувати планування із пріоритетами.

Основна ідея проста: кожному потокові надають пріоритет, при цьому на виконання ставитиметься потік із найвищим пріоритетом із черги готових потоків. Пріоритети можуть надаватися потокам статично або динамічно.

Одним із підходів до реалізації планування із пріоритетами є алгоритм багаторівневих черг (multilevel queues). У цьому разі організовують кілька черг для груп потоків із різними пріоритетами (потоки кожної групи звичайно мають різне призначення, можуть бути групи фонових потоків, інтерактивних тощо).

Рішення про вибір потоку для виконання приймають таким чином:

• якщо в черзі потоків із найвищим пріоритетом є потоки, для них слід використати якийсь простіший алгоритм планування (наприклад, кругового планування), не звертаючи уваги на потоки в інших чергах;

• якщо в черзі немає жодного потоку, переходять до черги потоків з нижчим пріоритетом і т. д.

Для різних черг можна використовувати різні алгоритми планування, крім того, кожній черзі може бути виділена певна частка процесорного часу.

Розподіл пріоритетів є складним завданням, невдале його розв'язання може призвести до того, що потоки процесів із низьким пріоритетом чекатимуть дуже довго. Наприклад, у 1973 році в Массачусетському технологічному інституті була зупинена машина, на якій знайшли процес із низьким пріоритетом - він був поставлений у чергу на виконання в 1967 році і з того часу так і не зміг запуститися. Таку ситуацію називають голодуванням (starvation).

Є різні способи розв'язання проблеми голодування. Наприклад, планувальник може поступово зменшувати пріоритет потоку, який виконують (такий процес називають старінням), і коли він стане нижче, ніж у наступного за пріоритетом потоку, перемкнути контекст на цей потік. Можна, навпаки, поступово підвищувати пріоритети потоків, які очікують.

3) метод оціночних функцій. Ціль методу - розбиття на зони. Проводиться оцінка, чи потрапило рішення в якусь зону. Спочатку на процесор заводиться перша задача і визначається, яку задачу треба виконати на цьому

процесорі далі. При цьому кожному рішенню присвоюється вага. Чим більша вага тим більшим буде ступінь претендування на виконання на даному ресурсі

4) генетичний алгоритм: на основі базового рішення, змінюючи параметри, знаходиться інше рішення. При цьому отримане рішення оцінюється чи краще воно за базове. Недоліком такого методу оптимізації є те, що оптимальний варіант отримується за велику кількість кроків. З цієї причини даний метод практично не використовується.

Більшість алгоритмів можна застосувати лише для вирішення окремої задачі. Тому не існує ефективного алгоритму для даної топології, але можна використовувати деякі їх частини для розв'язання поставленої задачі.

## 4. Опис роботи алгоритму

Алгоритм завантаження задач на ресурси системи:

1. Завантажуємо першу задачу на вільний процесор.
2. Продивляємося список процесорів і якщо є вільний, то завантажуємо на нього наступну задачу, котра не залежить від задачі, що ще не виконалася.
3. Якщо у нас з'явилася можливість виконання залежних завдань, то проводимо аналіз даної вершини і залежних від неї, якщо такі існують.
4. Якщо після виконання задачі можливе виконання наступної задачі, то ми передаємо данні в процесор на якому буде виконуватися задача-нащадок для того щоб ефективно завантажити всі процесори.
5. Якщо маємо вільний процесор з більшою швидкодією, то завантажуємо задачу на нього.
6. Після звільнення ресурсу на нього завантажується задача яка готова до виконання і значення ваги якої найбільше.
7. Якщо звільнилося декілька процесорів, то на процесор з більшою швидкодією завантажується задача з більшою вагою.
8. Якщо задача залежить від декількох задач, то потрібно дочекатися завершення всіх батьківських задач перед початком аналізу данної.

## 5. Опис програми

Лістинг програми наведений в Додатку А.

Програма виконана на мові програмування Java.

Опис класів програми

Клас Main

Базовий клас програми. Служить для запуску проекту.

Методи та поля класу:

main(String[] args) – точка запуску проекту.

Клас ATask

Моделює структуру завдань, які виконуються на ресурсах багатопроцесорної системи.

Методи та поля класу:

int id - ідентифікатор задачі;

int duration - тривалість виконання задачі (в тактах);

int remainder - залишок виконання (в тактах);

ArrayList<ATask> dependentATasks - залежні задачі(дочірні);

ArrayList<ATask> fatherTasks - задачі, від яких залежить дана задача(батьківські);

ArrayList<ATask> fatherTasksToDo - задачі, які треба виконати для початку виконання цієї задачі

boolean isPretended – прапор для визначення, чи якийсь процесор вже претендує на виконання даної задачі.

execute() – метод, який моделює виконання одного такту задачі.

Клас AProcessor

Моделює структуру та роботу процесора.

Методи та поля класу:

boolean isBusy – прапор для визначення, чи зайнятий даний процесор.

ATask executeATask – задача, яку виконує процесор

ArrayList<ATask> executedTasks - задачі, які були виконані на даному процесорі.

boolean pretends(ATask aTask, ArrayList<ATask> allExecutedTasks) - метод для визначення, чи може даний процесор претендувати на виконання задачі aTask.

Клас AConnection

Описує з'єднання між задачами.

Методи та поля класу:

ATask fromATask – задача, від якої йде з'єднання;

ATask toATask – задача, до якої йде зєднання;

int weight – вага з'єднання.

Клас ASystem

Моделює роботу багатопроцесорної системи та виконує роль просторового планувальника. Результат роботи (потактовий розподіл задач на ресурсах) зберігається в файл формату csv, який зручно переглядати за допомогою, наприклад, програми Microsoft Excel з програмного пакету MS Office або програми Calc програмного пакету Open Office.

Методи та поля класу:

int PROCESSOR_COUNT = 4 – кількість процесорів в системі;

AProcessor[] aProcessors - процесори

Map<Integer, ATask> tasks - відповідність індекс-задача(для спрощення пошуку необхідної задачі)

ArrayList<AConnection> aConnections – з'єднання між процесорами;

ArrayList<ATask> readyATasks = new ArrayList<ATask>() - готові до виконання задачі;

ArrayList<ATask> executedATasks - задачі, що виконуються в даний момент часу(на даному такті);

int numberOfTasks - кількість задач;

ArrayList<ATask> doneATasks - виконані задачі.

void run() – метод для запуску моделювання;

## 6. Моделювання роботи системи.

Занурюємо довільний граф у систему «ланцюг процесорів - неоднорідний». Виконуємо моделювання роботи алгоритму в даній системі.

Граф представлений на рис. 2.

Результати виконання записані в табл. 1. Колонка A->B показує з якого в якій процесори передаються дані в даний момент.

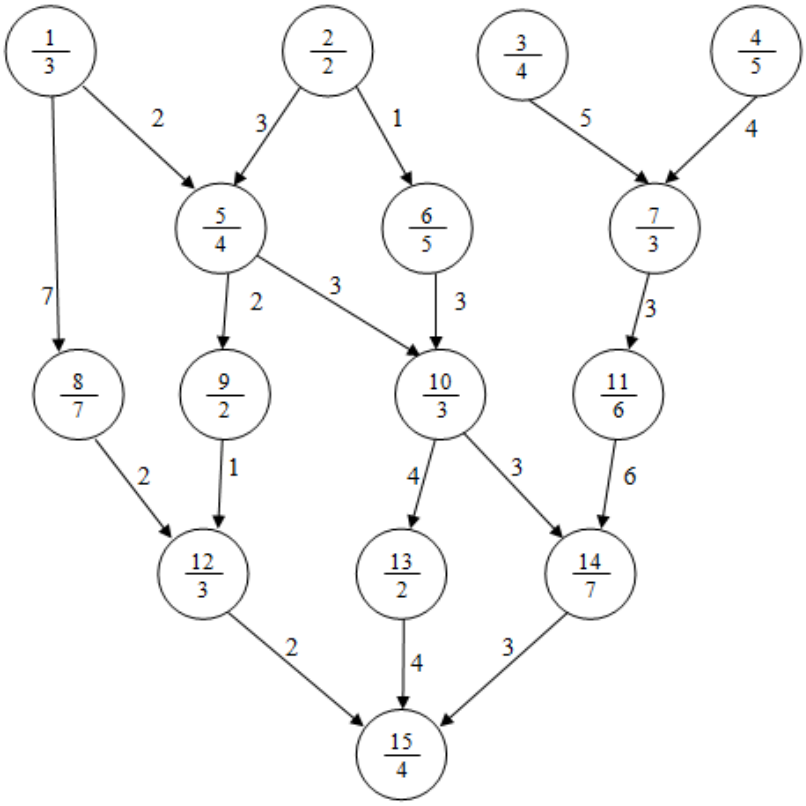


Рисунок 2 – Граф задач

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

    int[] freq = {1, 2, 2, 1};

    int[][] GRAPH_MATRIX = {
            {0, 0, 0, 2, 0, 0, 0, 7, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 3, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 2, 3, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 3, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6, 0},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
    };
    int[] TASKS_WEIGHTS = {
            3, 2, 4, 5, 4, 5, 3, 7, 2, 3, 6, 3, 2, 7, 4
    };
```

Рисунок 3 – Задання графа та швидкодії процесорів

| t | P1 | P2 | P3 | P4 | A->B |
|---|---|---|---|---|---|
| 1 | 1 | 2 | 3 | 4 | |
| 2 | 1 | 2 | 3 | 4 | |
| 3 | 1 | 2 | 3 | 4 | |
| 4 | | 2 | 3 | 4 | |
| 5 | | | 3 | 4 | 2->6;1->5 |
| 6 | 6 | | 3 | | 1->5 |
| 7 | 6 | 5 | 3 | | |
| 8 | 6 | 5 | 3 | | |
| 9 | 6 | 5 | | | 3->7 |
| 10 | 6 | 5 | | | 3->7 |
| 11 | 8 | 5 | | | 3->7;6->10 |
| 12 | 8 | 5 | | | 3->7;6->10 |
| 13 | 8 | 5 | | | 3->7;6->10 |
| 14 | 8 | 5 | | 7 | |
| 15 | 8 | 10 | | 7 | 5->9 |
| 16 | 8 | 10 | | 7 | 5->9 |
| 17 | 8 | 10 | 9 | 11 | |
| 18 | | 10 | 9 | 11 | |
| 19 | | 10 | 9 | 11 | |
| 20 | | 10 | 9 | 11 | |
| 21 | | 13 | | 11 | 9->12 |
| 22 | 12 | 13 | | 11 | |
| 23 | 12 | 13 | | 14 | |
| 24 | 12 | 13 | | 14 | |
| 25 | | | | 14 | 12->13;13->15 |
| 26 | | | | 14 | 12->13;13->15 |
| 27 | | | | 14 | 12->13;13->15 |
| 28 | | | | 14 | 12->13;13->15 |
| 29 | | | | 14 | 12->13;13->15 |
| 30 | | | | | 12->13;13->15 |
| 31 | | | | | 13->15 |
| 32 | | | | | 13->15 |
| 33 | | | | 15 | |
| 34 | | | | 15 | |
| 35 | | | | 15 | |
| 36 | | | | 15 | |

Таблиця 1 – Результат моделювання

**Висновок**

В даній роботі був реалізований алгоритм просторового планувальника для багатопроцесорної системи з топологією «ланцюг процесорів - неоднорідний». Система була реалізована на мові Java.

Під час моделювання системи найкращі результати були отримані на 4 процесорній системі при швидкодії 2 процесорів 1 операція за 1 такт і ще 2 процесорів 1 операція за 2 такти , але загрузка процесорів була не дуже великою, тому можна зробити висновок, що дана задача не підходить для цієї топології.

## СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ


1. Сімоненко В. П. “ Операційні системи ” Методичні вказівки до виконання курсової роботи
2. В. Шеховцов Операційні системи. Видавництво BHV, 2007 р.
3. Э. Таненбаум. Современные операционные системы. 2-е изд. СПб.: Питер, 2004.-1040с.
4. Конспект лекций по курсу ”Системное програмное обеспечение”
5. Вікіпедія. Вільна енциклопедія.. - http://uk.wikipedia.org/wiki/Список_алгоритмів

## Додаток А. Лістинг програми

```
/**
 * Created with IntelliJ IDEA.
 * User: vaifer
 * Date: 11.01.13
 * Time: 14:27
 * To change this template use File | Settings | File Templates.
 */
public class Test {
    public static void main(String[] args) {

       int[] freq = {1, 2, 2, 1};

        int[][] GRAPH_MATRIX = {
                {0, 0, 0, 2, 0, 0, 0, 7, 0, 0, 0, 0, 0, 0, 0},
                {0, 0, 0, 3, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                {0, 0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0, 0},
                {0, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0},
                {0, 0, 0, 0, 0, 0, 0, 0, 2, 3, 0, 0, 0, 0, 0},
                {0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6, 0, 0, 0, 0},
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0},
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0},
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 3, 0},
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6, 0},
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2},
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4},
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3},
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
        };
        int[] TASKS_WEIGHTS = {
                3, 2, 4, 5, 4, 5, 3, 7, 2, 3, 6, 3, 2, 7, 4
        };
        ASystem ASystem = new ASystem(GRAPH_MATRIX, TASKS_WEIGHTS);
        ASystem.run();
    }
}
import java.util.ArrayList;

/**
 * Created with IntelliJ IDEA.
 * User: vaifer
 * Date: 11.01.13
 * Time: 14:27
 * To change this template use File | Settings | File Templates.
 */
public class ATask {
    private int id; //ідентифікатор задачі
    private int duration;//тривалість (тактів)
    private int remainder;//залишок виконання(тактів)
    private ArrayList<ATask> dependentATasks;//залежні задачі(дочірні)
    private  ArrayList<ATask> fatherTasks;   // задачі, від яких залежить дана
задача
    private ArrayList<ATask> fatherTasksToDo;  // задачі, які треба виконати для
початку виконання цієї задачі
    private boolean isPretended = false; // для задачі є ресурс для виконання
(перетендує забрати дан задачу)

    public ATask(int id, int duration) {
        this.id = id;
        this.duration = duration;
        this.remainder = duration;
```

```
    }

    public int getId() {
        return id;
    }

    public int getDuration() {
        return duration;
    }

    public int getRemainder() {
        return remainder;
    }

    public void setDependentATasks(ArrayList<ATask> dependentATasks) {
        this.dependentATasks = dependentATasks;
    }

    public ArrayList<ATask> getDependentATasks() {
        return dependentATasks;
    }

    public ArrayList<ATask> getFatherTasksToDo() {
        return fatherTasksToDo;
    }

    public void setFatherTasks(ArrayList<ATask> fatherTasks) {
        this.fatherTasks = fatherTasks;
        this.fatherTasksToDo = (ArrayList<ATask>) fatherTasks.clone();
    }

    public boolean isPretended() {
        return isPretended;
    }

    public void setPretended(boolean pretended) {
        isPretended = pretended;
    }

    public void execute() {
        remainder--;
    }
}
import java.util.ArrayList;

/**
 * Created with IntelliJ IDEA.
 * User: vaifer
 * Date: 11.01.13
 * Time: 14:28
 * To change this template use File | Settings | File Templates.
 */
public class AProcessor {
    private boolean isBusy;
    private ATask executeATask;
    private boolean isMarkered;
    private ArrayList<ATask> executedTasks = new ArrayList<ATask>();

    public AProcessor(boolean busy, boolean markered) {
        isBusy = busy;
        isMarkered = markered;
    }
```

```
    public boolean isBusy() {
        return isBusy;
    }

    public void setBusy(boolean busy) {
        isBusy = busy;
    }

    public boolean isMarkered() {
        return isMarkered;
    }

    public void setMarkered(boolean markered) {
        isMarkered = markered;
    }

    public ATask getExecuteATask() {
        return executeATask;
    }

    public void setExecuteATask(ATask executeATask) {
        this.executeATask = executeATask;
    }

    public ArrayList<ATask> getExecutedTasks() {
        return executedTasks;
    }

    public void addExecutedTasks() {
        this.executedTasks.add(this.executeATask);
    }

    // визначає, чи претендує даний процесор на виконання чергової задачі.
    public boolean pretends(ATask aTask, ArrayList<ATask> allExecutedTasks) {
        for (int i = 0; i < executedTasks.size(); i++) {
            ArrayList<ATask>                    taskArrayList                    =
executedTasks.get(i).getDependentATasks();
            for (int j = 0; j < allExecutedTasks.size(); j++) {
                if (taskArrayList.contains(allExecutedTasks.get(j))) {
                    taskArrayList.remove(allExecutedTasks.get(j));
                }
            }
            if (taskArrayList.contains(aTask)) {
                return true;
            }
        }
        return false;
    }
}
/**
 * Created with IntelliJ IDEA.
 * User: vaifer
 * Date: 11.01.13
 * Time: 14:28
 * To change this template use File | Settings | File Templates.
 */
public class AConnection {
    private ATask fromATask;  //від задачі
    private ATask toATask;    //до задачі
    private int weight;     //вага

    public AConnection(ATask fromATask, ATask toATask, int weight) {
        this.fromATask = fromATask;
```

```
        this.toATask = toATask;
        this.weight = weight;
    }

    public ATask getFromATask() {
        return fromATask;
    }

    public ATask getToATask() {
        return toATask;
    }

    public int getWeight() {
        return weight;
    }
}
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;

/**
 * Created with IntelliJ IDEA.
 * User: vaifer
 * Date: 11.01.13
 * Time: 14:30
 * To change this template use File | Settings | File Templates.
 */
public class ASystem {
    private final int PROCESSOR_COUNT = 1;
    private AProcessor[] aProcessors;                          //процесори
    private Map<Integer, ATask> tasks;                         //індекс<->задача
    private boolean isBusFree = true;
    private ArrayList<AConnection> aConnections;
    private ArrayList<ATask> readyATasks = new ArrayList<ATask>();  //готові до
виконання задачі
    private ArrayList<ATask> executedATasks = new ArrayList<ATask>();// задачі,
що виконуються
    private int numberOfTasks;                                 //кількість
задач
    private ArrayList<ATask> doneATasks = new ArrayList<ATask>();    //виконані
задачі

    public ASystem(int[][] graphMatrix, int[] tasksWeights) {
        numberOfTasks = graphMatrix.length;
        setSystemParametres(graphMatrix, tasksWeights);
        //створення процесорів
        aProcessors = new AProcessor[PROCESSOR_COUNT];
        for (int i = 0; i < PROCESSOR_COUNT; i++) {
            AProcessor AProcessor = new AProcessor(false, false);
            aProcessors[i] = AProcessor;
        }
    }

    //запуск моделюванння
    public void run() {
        marker = 0;//маркер у процесора №0
        int takt = 0;
        marker = 0;
        aProcessors[0].setMarkered(true);
        ArrayList<String> modelingResult = new ArrayList<String>();
```

```
        String taktLine = "";
        String split = ";";
        taktLine += "t" + split + "         ";
        for (int i = 1; i <= PROCESSOR_COUNT; i++) {
            taktLine += "P" + i + split + "     ";
        }
        taktLine += "      Bus";
        System.out.println(taktLine);
        modelingResult.add(taktLine);
        while (doneATasks.size() != numberOfTasks) {
            takt++;
            //перевіряємо чи виконали процесори свої задачі і чи не з'явились
вільні задачі
            for (int i = 0; i < aProcessors.length; i++) {
                if                 ((aProcessors[i].isBusy())                 &&
(aProcessors[i].getExecuteATask().getRemainder() == 0)) {
                    aProcessors[i].setBusy(false);
                    aProcessors[i].addExecutedTasks();
                    doneATasks.add(aProcessors[i].getExecuteATask());
                    executedATasks.remove(aProcessors[i].getExecuteATask());
                    //чи зявились вільні задачі
                    for (int j = 0; j < numberOfTasks; j++) {
                        if        ((!doneATasks.contains(tasks.get(j)))        &&
(!executedATasks.contains(tasks.get(j)))) {
                            if
(tasks.get(j).getFatherTasksToDo().contains(aProcessors[i].getExecuteATask())) {

tasks.get(j).getFatherTasksToDo().remove(aProcessors[i].getExecuteATask());
                            }
                            if  ((tasks.get(j).getFatherTasksToDo().size()  ==  0)
&& (!readyATasks.contains(tasks.get(j)))) {
                                readyATasks.add(tasks.get(j));
                            }
                        }
                    }
                }
            }
            //призначаємо вільним процесорам готові задачі (циклічно проходимо
всі процесори, поки не буде конфліктів по призначенню задач)
            for (int i = 0; i < PROCESSOR_COUNT; i++) {
                if (!aProcessors[i].isBusy()) {
                    if (readyATasks.size() != 0) {
                        //сортуємо готові задачі за тривлістю по спаданню
                        for (int j = 0; j < readyATasks.size() - 1; j++) {
                            for (int k = +1; k < j; k++) {
                                if      (readyATasks.get(k).getDuration()      >
readyATasks.get(j).getDuration()) {
                                    readyATasks.add(j, readyATasks.get(k));
                                    readyATasks.remove(k + i);
                                }
                            }
                        }
                        //вибираємо задачу для даного процесора
                        ATask taskForExecute = null;
                        for (int j = 0; j < readyATasks.size(); j++) {
                            if      (aProcessors[i].pretends(readyATasks.get(j),
executedATasks)) {
                                if (taskForExecute != null) {
                                    //вибираємо   задачу   з   більшим   ваговим
коефіцієнтом передачі
                                    if    (aProcessors[i].getExecuteATask()    !=
null) {
```

```
                                        if
(getConnectionWeight(readyATasks.get(j),   aProcessors[i].getExecuteATask())   >
getConnectionWeight(taskForExecute, aProcessors[i].getExecuteATask())) {
                                            taskForExecute = readyATasks.get(j);
                                        }
                                    }
                                } else {
                                    taskForExecute = readyATasks.get(j);
                                }
                            }
                        }
                        //при можливості передаємо дані
                        if (aProcessors[i].isMarkered() & (!isBusFree)) {
                            isBusFree = false;
                        }
                        if (taskForExecute != null) {
                            aProcessors[i].setExecuteATask(taskForExecute);

                            executedATasks.add(taskForExecute);
                            readyATasks.remove(taskForExecute);
                        }  else  {//вибираємо  задач  з  найбільшою  тривалістю
виконання
                            aProcessors[i].setExecuteATask(readyATasks.get(0));
                            executedATasks.add(readyATasks.get(0));
                            readyATasks.remove(0);
                        }
                        aProcessors[i].setBusy(true);
                    }
                }
            }
            //виконуємо такт всіх процесорів
            for (int i = 0; i < aProcessors.length; i++) {
                if (aProcessors[i].isBusy()) {
                    aProcessors[i].getExecuteATask().execute();
                }
            }
            //виводимо результат роботи такту в колнсоль
            taktLine = takt + split + "         ";
            for (int i = 0; i < PROCESSOR_COUNT; i++) {
                ATask aTask = null;
                if (aProcessors[i].isBusy()) {
                    aTask = aProcessors[i].getExecuteATask();
                }
                if (aTask != null) {
                    taktLine += (aProcessors[i].getExecuteATask().getId() + 1);
                } else {
                    taktLine += "-";
                }
                //з маркером?
                if (aProcessors[i].isMarkered()) {
                    taktLine += "*" + split + "      ";
                } else {
                    taktLine += split + "      ";
                }
            }

            System.out.println(taktLine);
            modelingResult.add(taktLine);
        }
        //запис в csv файл
        writeToFile(modelingResult);
    }
```

```
    //встановлює параметри системи на основі вхідної матриці переходів та вагів
задач
    private void setSystemParametres(int[][] graphMatrix, int[] tasksWeights) {
        //створюємо задачі
        tasks = new HashMap<Integer, ATask>();
        for (int i = 0; i < numberOfTasks; i++) {
            ATask ATask = new ATask(i, tasksWeights[i]);
            tasks.put(i, ATask);
        }
        //встановлення дочірніх задач
        for (int i = 0; i < numberOfTasks; i++) {
            ArrayList<ATask> dependentTasks = new ArrayList<ATask>();
            for (int j = 0; j < numberOfTasks; j++) {
                if (graphMatrix[i][j] != 0) {
                    dependentTasks.add(tasks.get(j));
                }
            }
            tasks.get(i).setDependentATasks(dependentTasks);
        }
        //встановлення батьківських задач
        for (int i = 0; i < numberOfTasks; i++) {
            ArrayList<ATask> fatherTasks = new ArrayList<ATask>();
            for (int j = 0; j < numberOfTasks; j++) {
                if (graphMatrix[j][i] != 0) {
                    fatherTasks.add(tasks.get(j));
                }
            }
            tasks.get(i).setFatherTasks(fatherTasks);
        }
        //створюємо зв`язки
        aConnections = new ArrayList<AConnection>();
        for (int i = 0; i < numberOfTasks; i++) {
            for (int j = 0; j < numberOfTasks; j++) {
                if (graphMatrix[i][j] != 0) {
                    AConnection AConnection = new AConnection(tasks.get(i),
tasks.get(j), graphMatrix[i][j]);
                    aConnections.add(AConnection);
                }
            }

        }
        //визначення задач, готових до виконання( задачі верхнього рівня)
        for (int i = 0; i < numberOfTasks; i++) {
            if (isTopTask(graphMatrix, tasks.get(i))) {
                readyATasks.add(tasks.get(i));
            }
        }
    }

    // визначає, чи на верхньому рівні знаходиться задача
    private boolean isTopTask(int[][] graphMatrix, ATask task) {
        for (int i = 0; i < graphMatrix.length; i++) {
            if (graphMatrix[i][task.getId()] != 0) {
                return false;
            }
        }
        return true;
    }

    // повертає значення ваги переходу між задачами
    private int getConnectionWeight(ATask task1, ATask task2) {
        for (int i = 0; i < aConnections.size(); i++) {
```

```
            if      ((aConnections.get(i).getFromATask().equals(task1))      &&
(aConnections.get(i).getToATask().equals(task2))) {
                return aConnections.get(i).getWeight();
            }
            if      ((aConnections.get(i).getFromATask().equals(task2))      &&
(aConnections.get(i).getToATask().equals(task1))) {
                return aConnections.get(i).getWeight();
            }
        }
        return -1;
    }

    //записує результат моделювання в файл формату csv
    private void writeToFile(ArrayList<String> modelingResult) {
        String fileName = "result.csv";
        try {
            BufferedWriter     newCSVFile     =     new     BufferedWriter(new
FileWriter(fileName));
            for (String s : modelingResult) {
                newCSVFile.write(s + "\n");
            }
            newCSVFile.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```