Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут”
Кафедра АСОІУ

ЗВІТ

про виконання комп’ютерного практикуму № 2
з дисципліни
“Операційні системи”

Прийняв:
Проф. Сімоненко В. П.

Виконав:
студент 3-го курсу
гр. ІП-52 ФІОТ
Набоков Едуард
Максимович

Київ 2017

ЗМІСТ:

1. ПОСТАНОВКА ЗАДАЧІ

Завдання: Скласти програму реалізації угорського методу для неоднорідної обчислювальної системи з обмеженнями реального часу.

2. ОПИС АЛГОРИТМУ

2.1. Загальний алгортим

1. ПОКИ задач більше, ніж кількість процесорів:
   a. Знайти дві задачі з найменшим необхідним часом виконання
   b. Об’єднати їх в одну задачу
2. Обрахувати матрицю штрафів, які будуть начисленні, якщо задача *i* буде виконуватися на процесорі *j*.
3. В отриманій матриці замінити всі додатні елементи на 0.
4. В матриці з пунтку 4 замінити замінити всі значення на відповідні їм абсолютні (abs).
5. На отриманій матриці застосувати алгоритм Куна.

2.2. **Алгоритм Куна**:

```
1: function hungarian (matrix)
2:   for i form 0 to height-1
3:     do create in matrix new zero element
4:     while new zero element creates collision
5:           Find greatest match chain
6:           Change marks along greatest match chain
7:   return marked ellements
```

3. ОЦІНКА СКЛАДНОСТІ РОБОТИ

В даній програмі ми маємо два вкладених цикла розміру N (N – кількість процесорів в системі), причому операція "створити новий нульовий елемент" потребує оновлення допоміжних масивів і вимагає часу N. Разом час роботи алгоритму - O (N ^ 3).

Основна обчислювальна складність реалізації лежить в рядках 3 і 4: створювати нові нульові елементи потрібно так, щоб не "зіпсувати" вже отриманий незалежний набір, і так, щоб в результаті отримати чередуючийся ланцюжок.

4. ЛІСТИНГ ПРОГРАМИ

“ Hungarian.h”

```
/* Realization from: http://acm.mipt.ru/twiki/bin/view/Algorithms/
HungarianAlgorithmCPP
 */

#pragma once

#include <vector>
#include <limits>
using namespace std;

typedef pair<int, int> PInt;
typedef vector<float> VFloat;
typedef vector<VFloat> VVFloat;
typedef vector<PInt> VPFloat;

const int inf = numeric_limits<int>::max();


VPFloat hungarian(const VVFloat &matrix);
```

“ Hungarian.cpp”

```
#include "Hungarian.h"


VPFloat hungarian(const VVFloat &matrix) {

	// Размеры матрицы
	int height = matrix.size(), width = matrix[0].size();

	// Значения, вычитаемые из строк (u) и столбцов (v)
	VFloat u(height, 0), v(width, 0);

	// Индекс помеченной клетки в каждом столбце
	VFloat markIndices(width, -1);

	// Будем добавлять строки матрицы одну за другой
	for (int i = 0; i < height; i++) {
		VFloat links(width, -1);
		VFloat mins(width, inf);
		VFloat visited(width, 0);

		int markedI = i, markedJ = -1, j;
		while (markedI != -1) {
			// Обновим информацию о минимумах в посещенных строках
непосещенных столбцов
			// Заодно поместим в j индекс непосещенного столбца с самым
маленьким из них
			j = -1;
			for (int j1 = 0; j1 < width; j1++)
				if (!visited[j1]) {
					if (matrix[markedI][j1] - u[markedI] - v[j1] <
mins[j1]) {
						mins[j1] = matrix[markedI][j1] - u[markedI]
- v[j1];
```

```
                                    links[j1] = markedJ;
                              }
                              if (j == -1 || mins[j1] < mins[j])
                                    j = j1;
                        }

                                    int delta = mins[j];
                  for (int j1 = 0; j1 < width; j1++)
                        if (visited[j1]) {
                              u[markIndices[j1]] += delta;
                              v[j1] -= delta;
                        }
                        else {
                              mins[j1] -= delta;
                        }
                        u[i] += delta;

                        // Если коллизия не разрешена - перейдем к следующей
итерации
                        visited[j] = 1;
                        markedJ = j;
                        markedI = markIndices[j];
            }

            // Пройдем по найденной чередующейся цепочке клеток, снимем отметки с
            // отмеченных клеток и поставим отметки на неотмеченные
            for (; links[j] != -1; j = links[j])
                  markIndices[j] = markIndices[links[j]];
            markIndices[j] = i;
      }

      // Вернем результат в естественной форме
      VPFloat result;
      for (int j = 0; j < width; j++)
            if (markIndices[j] != -1)
                  result.push_back(PInt(markIndices[j], j));
      return result;
}
```

“ Utils.h”

```
#pragma once
#include <random>
#include <algorithm>
#include <vector>

using namespace std;

struct tasks {
      // Constraints:
      // {timeOfExec} > 0
      // {taskEndTime} > 0

      float taskEndTime; // in terms of most powerful CPU
      float timeOfExec; // in terms of most powerful CPU
};

vector<unsigned short> generateCPUs(unsigned int numberOfCPUs);
vector<tasks> generateTasks(unsigned int numberOfTasks);
vector<vector<pair<unsigned int, tasks> > > mergeTasks(vector<tasks>
tasksToMerge, unsigned int outputNumberOfTasks);
```

```
vector<vector<float>> generateScheduleMatrix(vector<vector<pair<unsigned int,
tasks> > > taskList, vector<float> CPUs);
float enumulateSchedulerWork(unsigned int numberOfTasks, unsigned int
numberOfCPUs);

// ----
vector<pair<unsigned int, tasks> >  sortTasks(vector<pair<unsigned int, tasks> >
taskList);
vector<unsigned short> sortCPUs(vector<unsigned short> CPUs);
```

“ Utils.cpp”

```
#include "Utils.h"
#include "Hungarian.h"
#include <stdio.h>

bool tasks_list_predicate(pair<unsigned int, tasks>& x, pair<unsigned int,
tasks>& y) {
	if (x.second.timeOfExec < y.second.timeOfExec) {
		return true;
	}
	else {
		return false;
	}
}

float enumulateSchedulerWork(unsigned int numberOfTasks, unsigned int
numberOfCPUs) {
	vector<unsigned short> CPUs =  generateCPUs(numberOfCPUs);
	CPUs = sortCPUs(CPUs);

	vector<float> relevantCPUs = vector<float>(CPUs.size());
	for (int i = 0, size = CPUs.size(); i < size; i++) {
		relevantCPUs[i] = CPUs[i] * 1. / CPUs[size - 1];
	}

	vector<tasks> task = generateTasks(numberOfTasks);
	vector<vector<pair<unsigned int, tasks> > > taskList = mergeTasks(task,
numberOfCPUs);

	printf("Generated CPUs:\n");
	for (int i = 0; i < numberOfCPUs; i++) {
		printf("Id: %d - Speed: %d MHz\n", i, CPUs[i]);
	}

	printf("\nGenerated tasks:\n");
	float sum = 0;
	for (int i = 0; i < numberOfTasks; i++) {
		printf("Id: %d - Exec resources: %.3f MHz - Critical time: %.3f
MHz\n", i,
			task[i].timeOfExec * CPUs[CPUs.size() - 1], task[i].taskEndTime
* CPUs[CPUs.size() - 1]);
		sum += task[i].timeOfExec * CPUs[CPUs.size() - 1];
	}
	printf("Total tasks on %.3f MHz time\n", sum);

	vector<vector<float>> matrix = generateScheduleMatrix(taskList,
relevantCPUs);
	for (int i = 0, size_1 = matrix.size(); i < size_1; i++) {
		for (int z = i, size_2 = matrix[i].size(); z < size_2; z++) {
			if (matrix[i][z] >= 0) {
				matrix[i][z] = 0;
			} else {
```

```
                        matrix[i][z] = -matrix[i][z];
                }

                matrix[z][i] = matrix[i][z];
        }
    }

    printf("\nExecution matrix:\n");
    for (int i = 0; i < numberOfCPUs; i++) {
        for (int z = 0; z < numberOfCPUs; z++) {
            printf("%10.3f ", -matrix[i][z]);
        }
        printf("\n");
    }

    vector<PInt> VPFloat = hungarian(matrix);
    float penalty = 0;
    matrix = generateScheduleMatrix(taskList, relevantCPUs);

    printf("\nResults matrix:\n");
    for (int i = 0; i < VPFloat.size(); i++) {
        for (int z = 0, size = taskList[VPFloat[i].first].size(); z < size;
z++) {
            printf("Task %d to CPU %d\n", taskList[VPFloat[i].first]
[z].first, VPFloat[i].second);
            penalty += matrix[VPFloat[i].first][VPFloat[i].second];
        }
    }

    return -penalty;
}


vector<unsigned short> generateCPUs(unsigned int numberOfCPUs) {
    vector<unsigned short> CPUs = vector<unsigned short>(numberOfCPUs);

    for (int i = 0; i < numberOfCPUs; i++) {
        CPUs[i] = rand() % 30 + 5;
    }

    return CPUs;
}


vector<tasks> generateTasks(unsigned int numberOfTasks) {
    vector<tasks> task = vector<tasks>(numberOfTasks);

    for (int i = 0; i < numberOfTasks; i++) {
        task[i].timeOfExec = rand() * 1.0f / RAND_MAX * 30 + 0.1;
        task[i].taskEndTime = rand() * 1.0f / RAND_MAX * 30 +
task[i].timeOfExec;
    }

    return task;
}


vector<vector<pair<unsigned int, tasks> > > mergeTasks(vector<tasks>
tasksToMerge, unsigned int outputNumberOfTasks) {
    vector<vector<pair<unsigned int, tasks> > > merged_tasks =
vector<vector<pair<unsigned int, tasks> > >(outputNumberOfTasks);
    float* cumsum = new float[outputNumberOfTasks];
    for (int i = 0; i < outputNumberOfTasks; i++) {
        cumsum[i] = 0.;
    }
```

```
	for (int i = 0, size = tasksToMerge.size(); i < size; i++) {
		int index_of_min = 0;
		for (int z = 1; z < outputNumberOfTasks; z++) {
			if (cumsum[z] < cumsum[index_of_min]) index_of_min = z;
		}

		cumsum[index_of_min] += tasksToMerge[i].timeOfExec;
		merged_tasks[index_of_min].push_back(std::make_pair(i,
tasksToMerge[i]));
	}


	for (int i = 0; i < outputNumberOfTasks; i++) {
		merged_tasks[i] = sortTasks(merged_tasks[i]);
	}

	delete[] cumsum;
	return merged_tasks;
}


vector<vector<float>> generateScheduleMatrix(vector<vector<pair<unsigned int,
tasks> > > taskList, vector<float> CPUs) {
	unsigned int n = CPUs.size();
	vector<vector<float>> matrix = vector< vector<float> >(n);
	for (int i = 0; i < n; i++) {
		matrix[i] = vector<float>(n);
	}

	for (int i = 0; i < n; i++) {
		for (int z = i; z < n; z++) {
			matrix[i][z] = 0;
			float curr_time = 0;
			for (int j = 0; j < taskList[i].size(); j++) {
				curr_time += taskList[i][j].second.timeOfExec / CPUs[z];
				if (curr_time > taskList[i][j].second.taskEndTime) {
					matrix[i][z] -= curr_time - taskList[i]
[j].second.taskEndTime;
				}
			}

			matrix[z][i] = matrix[i][z];
		}
	}

	return matrix;
}


vector<pair<unsigned int, tasks> >  sortTasks(vector<pair<unsigned int, tasks> >
taskList) {
	std::sort(taskList.begin(), taskList.end(), tasks_list_predicate);
	return taskList;
}


vector<unsigned short> sortCPUs(vector<unsigned short> CPUs) {
	std::sort(CPUs.begin(), CPUs.end());
	return CPUs;
}
```

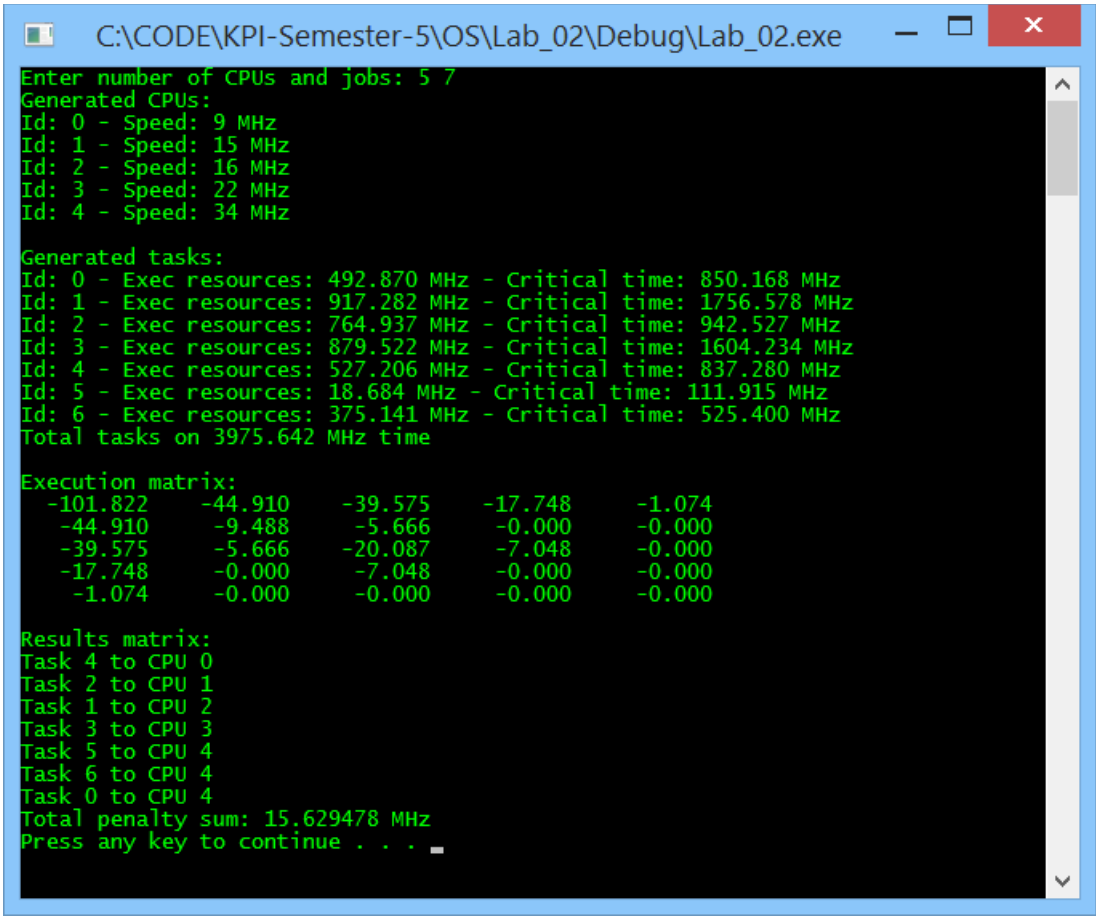
“main.cpp”

```
#include "Utils.h"
#include <stdlib.h>

int main(void) {
	unsigned int CPUs, jobs;
	printf("Enter number of CPUs and jobs: ");
	scanf("%d%d", &CPUs, &jobs);

	printf("Total penalty sum: %f MHz\n", enumulateSchedulerWork(jobs, CPUs));
	system("pause");
}
```

5. ПРИКЛАД ВИКОНАННЯ ПРОГРАМИ



Робота програми