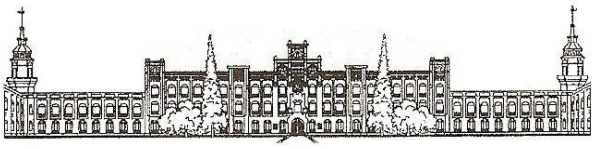

Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут»

Лабораторна робота №1

з дисципліни «Комп'ютерні системи»

«ПЛАНУВАННЯ ЗАДАЧ У БАГАТОПРОЦЕСОРНИХ КОМП'ЮТЕРНИХ СИСТЕМАХ»

Виконав студент групи: КВ-11

ПІБ: Гультяєв Дмитро Антонович

Перевірив: _____

Київ 2024

Завдання

До системи кожну мілісекунду (мс) поступають з певною імовірністю задачі. Кожна задача задається наступним чином: Zi (перелік процесорів; складність (кількість) операцій). Перелік процесорів – це номери процесорних елементів, які можуть реалізувати цю задачу. Складність задач обирається випадково, виходячи з того, що задача повинна бути виконана на «найслабшому» (з точки зору швидкодії) процесорі не менше, ніж за 10 мс і не більше, ніж за 200 мс. Задачі встановлюються у чергу. У цьому випадку враховуємо, що час управління чергою складає 4 мс. Усі задачі є незалежними та мають однаковий пріоритет. Кожний процесор задається за допомогою параметру «потужність» або «швидкість обробки» - n операцій на секунду. Незважаючи на те, що процесори відрізняються один від одного (мають різні набори команд та потужність), будемо вважати, що одиниця вимірювання однакова і відповідає нашому завданню.

Розподіл ресурсів в системі може відбуватись за трьома схемами:

1. FIFO;
2. З окремим процесором-планувальником;
3. З накладенням функцій планування на самий потужний процесор, що періодично переривається на керування чергою.

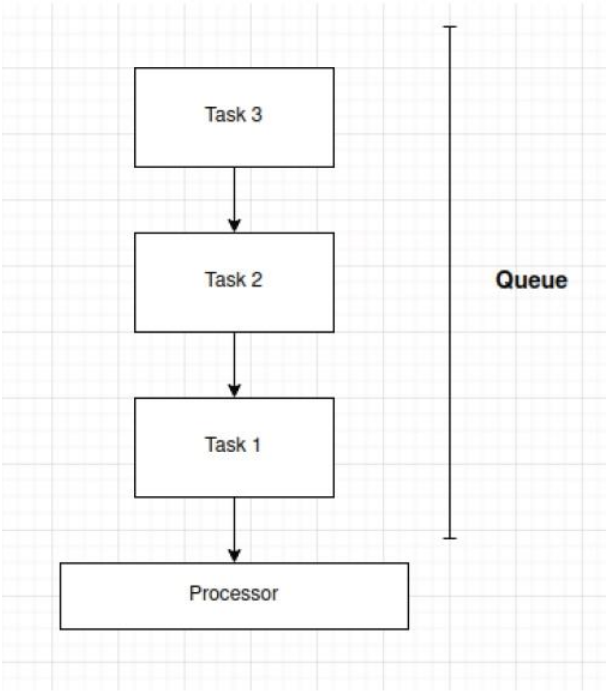

Рис. 1 Блок-схема алгоритму FIFO

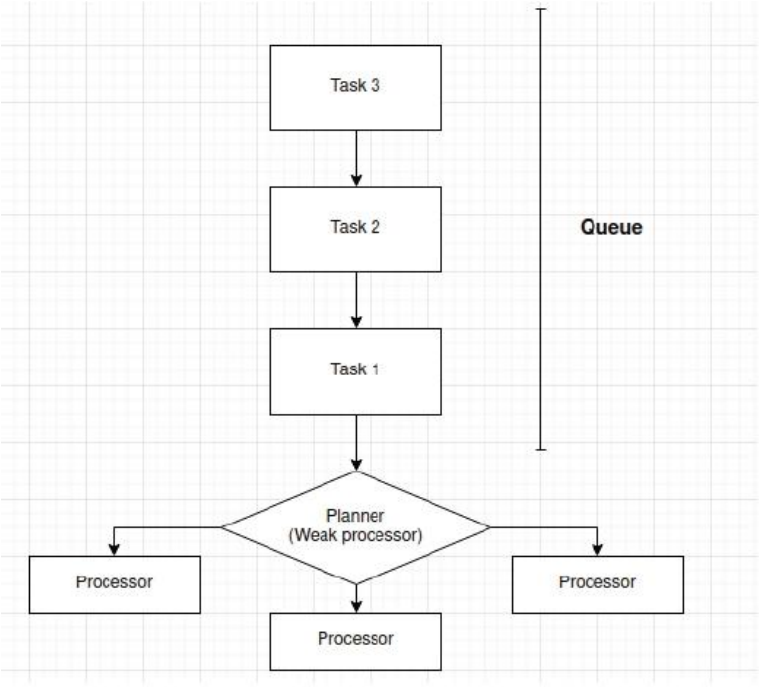

Рис. 2 Блок-схема алгоритму з окремим процесором-планувальником (найслабшим)

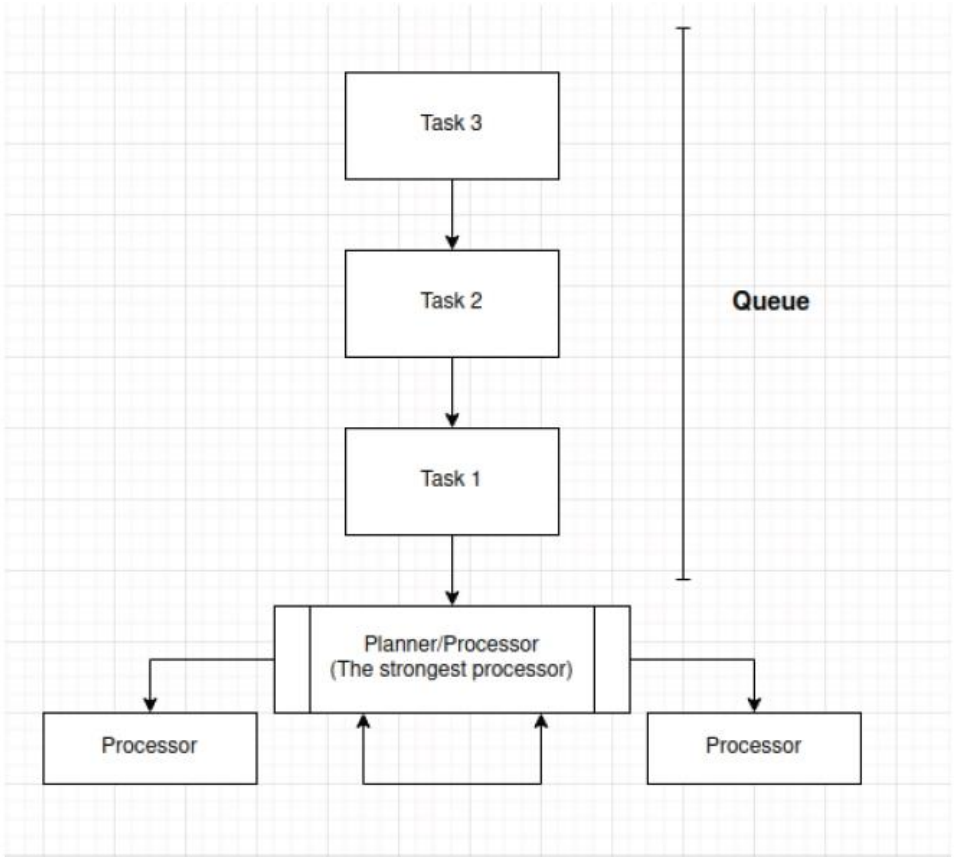

Рис. 3 Блок-схема алгоритму з окремим процесором-планувальником (найпотужнішим)

Код програми:

Processor.java

```
public class Processor {

    private int index;
    private int power;
    private boolean isFree;
    private int timeWorkTask;
    private int timeWork;
    private int timeSleep;

    public Processor(int index, int power) {
        this.index = index;
        this.power = power;
        this.isFree = true;
        this.timeWorkTask = 0;
        this.timeWork = 0;
        this.timeSleep = 0;
    }

    public int getIndex() {
        return index;
    }

    public int getPower() {
```

```
        return power;
    }

    public boolean isFree() {
        return isFree;
    }

    public int getTimeWorkTask() {
        return timeWorkTask;
    }

    public int getTimeWork() {
        return timeWork;
    }

    public int getTimeSleep() {
        return timeSleep;
    }

    public void setPower(int power) {
        this.power = power;
    }

    public void setFree(boolean free) {
        isFree = free;
    }

    public void setTimeWorkTask(int timeWorkTask) {
        this.timeWorkTask = timeWorkTask;
    }

    public void resetProcessor(){
        timeSleep = 0;
        timeWork = 0;
        timeWorkTask = 0;
    }

    public void run(){
        if(isFree){
            timeSleep++;
        }
        else {
            timeWork++;
            timeWorkTask--;
            if(timeWorkTask == 0){
                isFree = true;
            }
        }
    }
}
```

ListOfProcessors.java

```
import java.util.Arrays;

import java.util.List;

public class ListOfProcessors {
    private List<List<Integer>> list;

    public ListOfProcessors(){
        list = Arrays.asList(
```

```
                    Arrays.asList(1, 2, 3, 4, 0),
                    Arrays.asList(1, 2, 3, 4),
                    Arrays.asList(2, 3, 4, 0),
                    Arrays.asList(1, 3, 4, 0),
                    Arrays.asList(1, 2, 4, 0),
                    Arrays.asList(1, 2, 3, 0),
                    Arrays.asList(1, 2, 3),
                    Arrays.asList(1, 2, 4),
                    Arrays.asList(1, 2, 0),
                    Arrays.asList(1, 3, 4),
                    Arrays.asList(1, 3, 0),
                    Arrays.asList(2, 3, 0),
                    Arrays.asList(2, 3, 0),
                    Arrays.asList(1, 4, 0),
                    Arrays.asList(3, 4, 0),
                    Arrays.asList(1, 2),
                    Arrays.asList(1, 3),
                    Arrays.asList(1, 4),
                    Arrays.asList(1, 0),
                    Arrays.asList(2, 3),
                    Arrays.asList(2, 4),
                    Arrays.asList(2, 0),
                    Arrays.asList(3, 4),
                    Arrays.asList(3, 0),
                    Arrays.asList(4, 0),
                    Arrays.asList(1),
                    Arrays.asList(2),
                    Arrays.asList(3),
                    Arrays.asList(4)
            );
    }

    public List<List<Integer>> getList() {
        return list;
    }
}
```

Task.java

```
import java.util.List;

public class Task {
    private int countOfOperations;
    private List<Integer> processors;

    public Task(int countOfOperations, List<Integer> CPUs) {
        this.countOfOperations = countOfOperations;
        this.processors = CPUs;
    }

    public int getCountOfOperations() {
        return countOfOperations;
    }

    public List<Integer> getProcessors() {
        return processors;
    }
}
```

Controller.java

```
import java.util.*;

public class Controller {
    private int amountOfProcessors = 5;
    private List<Processor> processors;
    private int minTaskTime;
    private int maxTaskTime;
    private int taskProbability;
    private int taskAmount = 10000;

    public void createProcessors(){
        processors = new ArrayList<>();
        for(int i = 0; i < amountOfProcessors; i++){
            Scanner scanner = new Scanner(System.in);
            System.out.println("Enter power amount for processor with id " +
i);
            int powerOfProcessor = scanner.nextInt();
            processors.add(new Processor(i,powerOfProcessor));
        }
    }

    public void setTimeLimitForTask(){
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter min time for task: ");
        minTaskTime = scanner.nextInt();
        System.out.println("Enter max time for task: ");
        maxTaskTime = scanner.nextInt();
    }

    public void setProbability(){
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter new probability (from 1 to 99): ");
        taskProbability = scanner.nextInt();
    }

    public List<Task> generateTasks(){
        List<Task> tasks = new ArrayList<>();
        int minProcessorPower = processors.stream()
                .map(Processor::getPower)
                .min(Comparator.naturalOrder())
                .orElseThrow(() -> new RuntimeException("ERROR to get min
processor power"));
        int minTaskComplexity = minTaskTime * minProcessorPower;
        int maxTaskComplexity = maxTaskTime * minProcessorPower;
        ListOfProcessors listOfProcessors = new ListOfProcessors();
        int sizeOfListOfProcessor = listOfProcessors.getList().size();

        for(int i = 0; i < taskAmount; i++){
            int countOfOperations = new Random().nextInt(maxTaskComplexity -
minTaskComplexity + 1) + minTaskComplexity;
            List<Integer> processors = listOfProcessors.getList().get(new
Random().nextInt(sizeOfListOfProcessor));
            tasks.add(new Task(countOfOperations, processors));
        }

        return tasks;
    }

    public int searchProcessorForTask(Task task){
        for(int i = 0; i < amountOfProcessors; i++){
```

```
            if(processors.get(i).isFree() &&
task.getProcessors().contains(i)){
                return i;
            }
        }
        return -1;
    }

    public boolean isProcessorFree() {
        for (int i = 0; i < amountOfProcessors; i++) {
            Processor processor = processors.get(i);
                if (!processor.isFree()) {
                    return false;
                }
            }
        return true;
    }

    public Task searchTaskForProcessor(Processor processor, List<Task>
tasks){
        if(!tasks.isEmpty()){
            for(int i = 0; i < tasks.size(); i++){

if(tasks.get(i).getProcessors().contains(processor.getIndex())){
                    return tasks.get(i);
                }
            }
        }
        return null;
    }

    public void resultAnalyzer(int runTime, int n){
        for(int i = n; i < amountOfProcessors; i++){
            Processor processor = processors.get(i);
            System.out.println("Processor with ID " + i + " worked " +
processor.getTimeWork() + " and slept " + processor.getTimeSleep());
        }

        int countOfOperations = 0;
        for(int i = n; i < amountOfProcessors; i++){
            long countTime = (10000L * processors.get(i).getTimeWork()) /
runTime;
            countOfOperations += (int) (processors.get(i).getPower() *
countTime);
        }

        int theoreticalOperationsValue = processors.stream()
                .filter(p -> p.getIndex() >= n)
                .map(Processor::getPower)
                .mapToInt(power -> power * 10000)
                .sum();

        System.out.println("Count of operations in 10 seconds: " +
countOfOperations);
        System.out.println("Theoretical operations value in 10 seconds: " +
theoreticalOperationsValue);
        System.out.println("Efficiency: " + ((float)countOfOperations /
(float)theoreticalOperationsValue * 100) + " %");
    }

    public void FIFO(){
        List<Task> tasks = generateTasks();
        int runTime = 0;
```

```
        for(Processor processor : processors){
            processor.resetProcessor();
        }

        int taskIndex = 0;

        while(true){
            if(new Random().nextInt(100) < taskProbability){
                if(taskIndex < tasks.size()){
                    Task task = tasks.get(taskIndex);
                    int processorID = searchProcessorForTask(task);
                    if (processorID > -1) {
                        processors.get(processorID).setFree(false);

processors.get(processorID).setTimeWorkTask(task.getCountOfOperations() /
processors.get(processorID).getPower());
                        taskIndex++;
                    }
                }
            }

            for (Processor processor : processors) {
                processor.run();
            }
            runTime++;

            if(taskIndex >= tasks.size()){
                boolean checkIsProcessorFree = isProcessorFree();
                if(checkIsProcessorFree){
                    break;
                }
            }
        }
        resultAnalyzer(runTime, 0);
    }

    public void weakProcessorScheduler(){
        List<Task> tasks = generateTasks();
        int runTime = 0;

        for(Processor processor : processors){
            processor.resetProcessor();
        }

        while(true){
            for(int i = 1; i < amountOfProcessors; i++){
                Processor processor = processors.get(i);
                if(processor.isFree()){
                    Task task = searchTaskForProcessor(processor, tasks);
                    if(task != null){
                        processor.setFree(false);
                        processor.setTimeWorkTask(task.getCountOfOperations()
/ processor.getPower());
                        tasks.remove(task);
                    }
                }
            }

            for (Processor processor : processors) {
                processor.run();
            }
            runTime++;

            if(tasks.isEmpty()){
```

```
                boolean checkIsProcessorFree = isProcessorFree();
                if(checkIsProcessorFree){
                    break;
                }
            }
        }

        resultAnalyzer(runTime, 1);
    }

    public void strongProcessorScheduler(int workTime, int planTime){
        List<Task> tasks = generateTasks();
        int runTime = 0;
        int timeWorkAndSleepProcPlan = 0;

        for(Processor processor : processors){
            processor.resetProcessor();
        }

        while(true){
            for(int i = 0; i < amountOfProcessors; i++){
                Processor processor = processors.get(i);
                if(processor.isFree()){
                    Task task = searchTaskForProcessor(processor, tasks);
                    if(task != null){
                        processor.setFree(false);
                        processor.setTimeWorkTask(task.getCountOfOperations()
/ processor.getPower());
                        tasks.remove(task);
                    }
                }
            }

            for (Processor processor : processors) {
                processor.run();
            }
            runTime++;

            if (timeWorkAndSleepProcPlan <= workTime) {
                timeWorkAndSleepProcPlan++;
            }
            else if (timeWorkAndSleepProcPlan <= workTime + planTime) {
                processors.get(4).setFree(false);
                timeWorkAndSleepProcPlan++;
            }
            else {
                processors.get(4).setFree(true);
                timeWorkAndSleepProcPlan = 0;
            }

            if(tasks.isEmpty()){
                boolean checkIsProcessorFree = isProcessorFree();
                if(checkIsProcessorFree){
                    break;
                }
            }
        }

        resultAnalyzer(runTime, 0);
    }
}
```

Main.java

```
public class Main {
    public static void main(String[] args) {
        Controller controller = new Controller();

        controller.createProcessors();
        controller.setTimeLimitForTask();
        controller.setProbability();

        System.out.println("FIFO:");
        controller.FIFO();

        System.out.println("\nWeak processor:");
        controller.weakProcessorScheduler();

        System.out.println("\nStrong processor:");
        controller.strongProcessorScheduler(20,4);

        System.out.println("\nStrong processor custom:");
        controller.strongProcessorScheduler(30,4);
    }
}
```

Результати виконання:

Для отримання результатів програми задані такі вхідні параметри в програмному інтерфейсі:

```
Enter power amount for processor with id 0
50
Enter power amount for processor with id 1
60
Enter power amount for processor with id 2
70
Enter power amount for processor with id 3
80
Enter power amount for processor with id 4
90
Enter min time for task:
10
Enter max time for task:
200
Enter new probability (from 1 to 99):
99
```

Алгоритм FIFO:

```
FIFO:
Processor with ID 0 worked 165919 and slept 36857
Processor with ID 1 worked 166916 and slept 35860
Processor with ID 2 worked 153362 and slept 49414
Processor with ID 3 worked 146820 and slept 55956
Processor with ID 4 worked 128181 and slept 74595
Count of operations in 10 seconds: 2580460
Theoretical operations value in 10 seconds: 3500000
Efficiency: 73.72743 %
```

FIFO є найпростішим в реалізації алгоритмом, але має суттєві недоліки, основний з яких це те що процесори часто є незавантаженими, наприклад, якщо дві або більше задачі в черзі можуть бути виконані тільки на одному процесорі послідовно, то інші процесори в цей час просто стоять без задачі.
Це негативно впливає на продуктивність всієї системи, і це ми прекрасно бачимо в результатах. КПД системи складає всього 73.7%, а також вище видно, що процесори велику кількість часу просто стоять без задачі. Отже, FIFO є прости в реалізації алгоритмом, але не ефективним.

Алгоритм з використанням найслабшого процесора для планування задач:

```
Weak processor:
Processor with ID 1 worked 171865 and slept 47
Processor with ID 2 worked 171892 and slept 20
Processor with ID 3 worked 171846 and slept 66
Processor with ID 4 worked 171912 and slept 0
Count of operations in 10 seconds: 2999360
Theoretical operations value in 10 seconds: 3000000
Efficiency: 99.97867 %
```

В цьому алгоритмі ми відводимо один процесор для вирішення питань планування, тобто визначення того, яка задача буде виконуватися на конкретному процесорі. Після завершення роботи певного процесора, процесор-планувальник вибирає наступне завдання, яке підходить для цього процесора і направляє йому вказівник на це завдання. Оскільки алгоритм

планування є нескладним, розумно відділити найслабший процесор для цих цілей. Головна перевага цього підходу полягає в низькому часі простою процесорів. Однак головний недолік полягає в тому, що система втрачає один процесор, який міг би використовуватися для виконання задач.

Згідно результатів, цей алгоритм показав високу продуктивність 99.98%. Такий високий рівень КПД пов'язаний з головною перевагою цього підходу, а саме в низькому часі простою процесорів.

Алгоритм з використанням найпотужнішого процесора для планування задач:

```
Strong processor:
Processor with ID 0 worked 106379 and slept 60
Processor with ID 1 worked 106439 and slept 0
Processor with ID 2 worked 106321 and slept 118
Processor with ID 3 worked 106384 and slept 55
Processor with ID 4 worked 106330 and slept 109
Count of operations in 10 seconds: 3497390
Theoretical operations value in 10 seconds: 3500000
Efficiency: 99.92543 %

Strong processor custom:
Processor with ID 0 worked 117688 and slept 49
Processor with ID 1 worked 117671 and slept 66
Processor with ID 2 worked 117534 and slept 203
Processor with ID 3 worked 117737 and slept 0
Processor with ID 4 worked 117528 and slept 209
Count of operations in 10 seconds: 3496510
Theoretical operations value in 10 seconds: 3500000
Efficiency: 99.90028 %
```

Цей алгоритм включає в себе виділення окремого процесора-планувальника, який виконує подвійну роль. Протягом певного періоду він працює як звичайний процесор, а під час іншого періоду - як планувальник. Користувач системи може налаштовувати час роботи та час планування (за замовчуванням, 20 мс - звичайний режим, 4 мс - режим планування розподілу завдань). Цю роль відіграє найпотужніший процесор системи. Головна

перевага цього підходу полягає в низькому часі простою процесорів. Однак головний недолік включає в себе складність реалізації алгоритму.. Крім того, цей алгоритм вимагає обережного використання, оскільки неправильні або недоцільні параметри для процесорів можуть призвести до значного уповільнення процесу. Наприклад, якщо інші процесори заслабкі, то процес буде уповільненим.

Висновки:

1) Алгоритм FIFO має найнижчу продуктивність, КПД системи складає всього 73.7%, а також процесори велику кількість часу просто стоять без задачі. Також цей алгоритм здійснив найменшу кількість операцій за 10 секунд (2 580 460) Отже, FIFO є простим в реалізації алгоритмом, але не ефективним.
2) Алгоритм “weak processor” показав високий рівень КПД - 99.98%. Але при цьому виконав менше операцій за 10 секунд ніж “strong processor”. Значення операцій за 10 секунд склало 2 999 360
3) Алгоритм “strong processor” показав рівень КПД трохи нижче за “weak processor”, значення склало 99.92%, але при цьому цей алгоритм серед всіх виконав найбільше операцій за 10 секунд, а саме 3 497 390. Також виявилося, що для моєї конфігурації потужностей процесорів співвідношення 20 мс / 4 мс є найоптимальнішим.