Національний технічний університет України

«Київський політехнічний інститут»

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Розрахунково-графічна робота

з дисципліни

«Інтелектуальні вбудовані системи»

Виконав:

студент 3-го курсу

групи ІП-83, ФІОТ

Розгон Олександр Олександрович

Залікова - ІП-8317

Київ 2021

# **Основні теоретичні відомості**

**Планування виконання завдань** (англ. Scheduling) є однією з ключових концепцій в багатозадачності і багатопроцесорних систем, як в операційних системах загального призначення, так і в операційних системах реального часу. Планування полягає в призначенні пріоритетів процесам в черзі з пріоритетами.

Найважливішою метою планування завдань є якнайповніше завантаження доступних ресурсів. Для забезпечення загальної продуктивності системи планувальник має опиратися на:

* Використання процесора(-ів) — дати завдання процесору, якщо

це можливо.

* Пропускна здатність — кількість процесів, що виконуються за

одиницю часу.

* Час на завдання — кількість часу, для повного виконання

певного процесу.

* Очікування — кількість часу, який процес очікує в черзі готових.
* Час відповіді — час, який проходить від подання запиту до

першої відповіді на запит.

* Справедливість — Рівність процесорного часу для кожної ниті

У середовищах обчислень реального часу, наприклад, на пристроях, призначених для автоматичного управління в промисловості (наприклад, робототехніка), планувальник завдань повинен забезпечити виконання процесів в перебігу заданих часових проміжків (час відгуку); це критично для підтримки коректної роботи системи реального часу.

Система масового обслуговування (СМО) — система, яка виконує

обслуговування вимог (заявок), що надходять до неї. Обслуговування вимог у СМО проводиться обслуговуючими приладами. Класична СМО містить від одного до нескінченного числа приладів. В залежності від наявності можливості очікування вхідними вимогами початку обслуговування СМО (наявності черг) поділяються на:

1) системи з втратами, в яких вимоги, що не знайшли в момент надходження жодного вільного приладу, втрачаються;

2) системи з очікуванням, в яких є накопичувач нескінченної ємності для буферизації надійшли вимог, при цьому очікують вимоги утворюють чергу;

3) системи з накопичувачем кінцевої ємності (чеканням і обмеженнями), в яких довжина черги не може перевищувати ємності накопичувача; при цьому вимога, що надходить в переповнену СМО (відсутні вільні місця для очікування), втрачається.

**Основні поняття СМО:**

* Вимога (заявка) — запит на обслуговування.
* Вхідний потік вимог — сукупність вимог, що надходять у СМО.
* Час обслуговування - період часу, протягом якого обслуговується вимогу.

## **Вимоги до системи**

**Вхідні задачі**

Вхідними заявками є обчислення, які проводилися в лабораторних роботах 1-3, а саме обчислення математичного очікування, дисперсії, автокореляції, перетворення Фур’є.

**Вхідні заявки характеризуються наступними параметрами:**

1) час приходу в систему – Tp – потік заявок є потоком Пуассона або потоком Ерланга k-го порядку (інтенсивність потоків та їх порядок задаються варіантом);

2) час виконання (обробки) – То; математичним очікуванням часу виконання є середнє значення часу виконання відповідних обчислень в попередніх лабораторних роботах;

3) крайній строк завершення (дедлайн) – Тd –задається (випадково?);

якщо заявка залишається необробленою в момент часу t = Td, то її

обробка припиняється і вона покидає систему.

**Потік вхідних задач**

Потоком Пуассона є послідовність випадкових подій, середнє значення

інтервалів між настанням яких є сталою величиною, що дорівнює 1/λ , де λ – інтенсивність потоку.

Потоком Ерланга k-го порядку називається потік, який отримується потоку Пуассона шляхом збереження кожної (k+1)-ї події (решта відкидаються). Наприклад, якщо зобразити на часовій осі потік Пуассона, поставивши у відповідність кожній події деяку точку, і відкинути з потоку кожну другу подію (точку на осі), то отримаємо потік Ерланга 2-го порядку.

Залишивши лише кожну третю точку і відкинувши дві проміжні, отримаємо потік Ерланга 3-го порядку і т.д. Очевидно, що потоком Ерланга 0-го порядку є потік Пуассона.

# **Пристрій обслуговування**

Пристрій обслуговування складається з P незалежних рівноправних

обслуговуючих приладів - обчислювальних ресурсів (процесорів). Кожен

ресурс обробляє заявки, які йому надає планувальник та може перебувати у двох станах – вільний та зайнятий. Обробка заявок може виконуватися

повністю (заявка перебуває на обчислювальному ресурсі доти, доки не

обробиться повністю) або поквантово (ресурс обробляє заявку лише

протягом певного часу – кванту обробки – і переходить до обробки наступної

заявки).

## **Дисципліна обслуговування**

Вибір заявки з черги на обслуговування здійснюється за допомогою так

званої дисципліни обслуговування. Їх прикладами є FIFO (прийшов першим - обслуговується першим), LIFO (прийшов останнім – обслуговується першим), RANDOM (випадковий вибір). У системах з очікуванням накопичувач в загальному випадку може мати складну структуру.

**Дисципліна RR**

Алгоритм Round-Robin (від англ. round-robin — циклічний) – алгоритм

розподілу навантаження на розподілену (або паралельну) обчислювальну

систему методом перебору і впорядкування її заявок по круговому циклу.

Даний алгоритм не враховує пріоритети вхідних заявок. Нехай є Р ресурсів (з порядковими номерами р) та X заявок (з порядковими номерами x), які необхідно виконати. Тоді перша заявка (x = 1) назначається для виконання на першому ресурсі (р = 1), друга (х = 2) – другому і т.д., до досягнення зайнятості останнього ресурсу (р = Р, х = Р) або до вичерпування необроблюваних заявок (х = Х). Усі наступні заявки будуть розподілені по ресурсах аналогічно до попередніх, починаючи з першого ресурсу (х = Р + 1 → р = 1, х = Р + 2 → р = 2 і т.д.). Іншими словами відбувається перебір ресурсів по циклу (по колу – round).

Обчислення задач розділене на кванти часу, причому по закінченню кванту завершені та прострочені задачі виходять з системи, незавершені – здвигаються по колу на 1 ресурс (тобто задача першого об’єкта передається другому, другого – третьому і т.д., останнього – першому).

**Дисципліна EDF**

Алгоритм планування Earliest Deadline First (по найближчому строку

завершення) використовується для встановлення черги заявок в операційних системах реального часу.

При наставанні події планування (завершився квант часу, прибула нова

заявка, завершилася обробка заявки, заявка прострочена) відбувається пошук найближчої до крайнього часу виконання (дедлайну) заявки і призначення її виконання на перший вільний ресурс або на той, який звільниться найшвидше.

**Завдання**

1. Змоделювати планувальник роботи системи реального часу. Дві

дисципліни планування: перша – RR, друга задається викладачем

або обирається самостійно.

2. Знайти наступні значення:

1) середній розмір вхідної черги заявок, та додаткових черг (за їх

наявності);

2) середній час очікування заявки в черзі;

3) кількість прострочених заявок та її відношення до загальної

кількості заявок

3. Побудувати наступні графіки:

1) Графік залежності кількості заявок від часу очікування при

фіксованій інтенсивності вхідного потоку заявок.

2) Графік залежності середнього часу очікування від інтенсивності

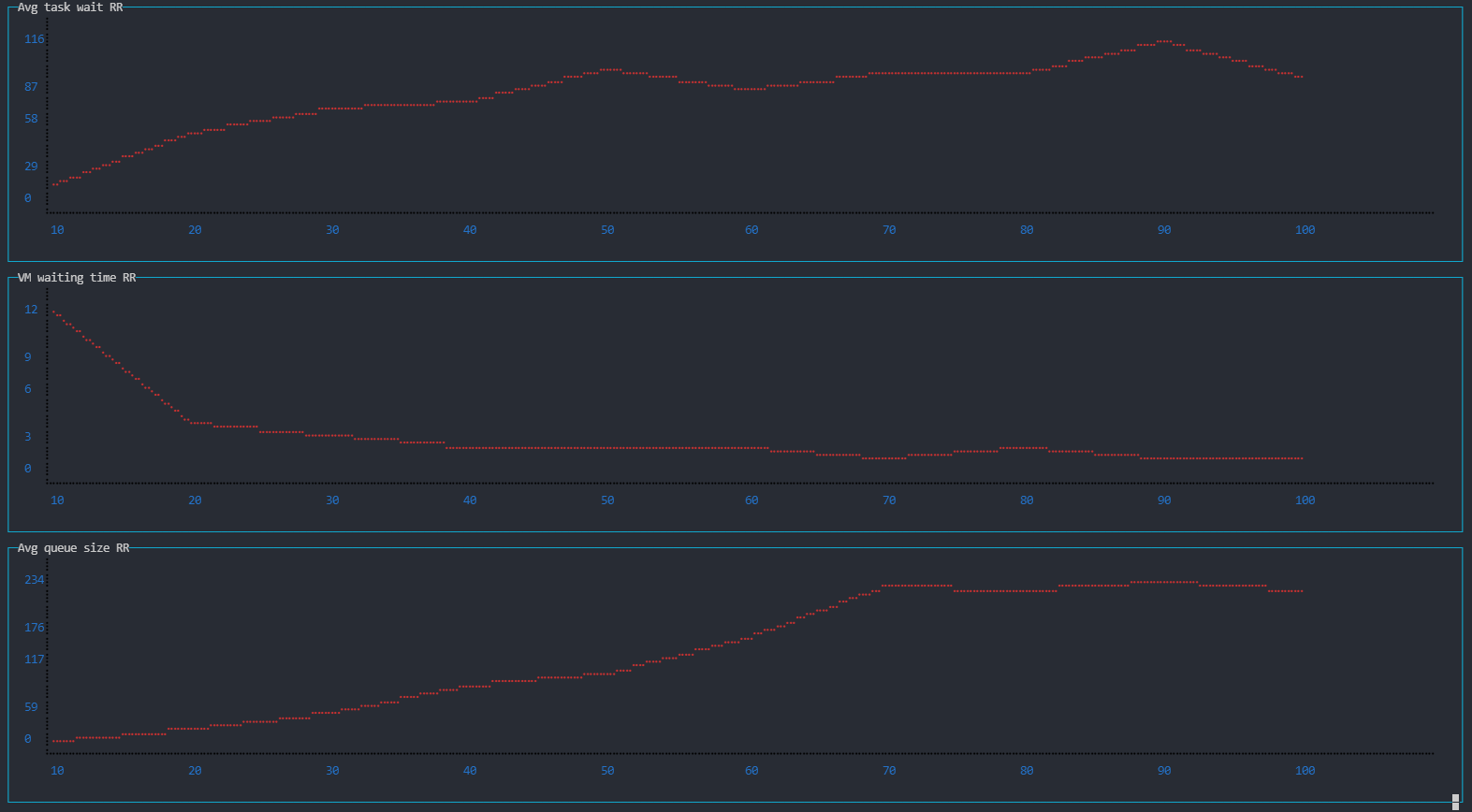
вхідного потоку заявок.

3) Графік залежності проценту простою ресурсу від інтенсивності

вхідного потоку заявок.

**Виконання:**

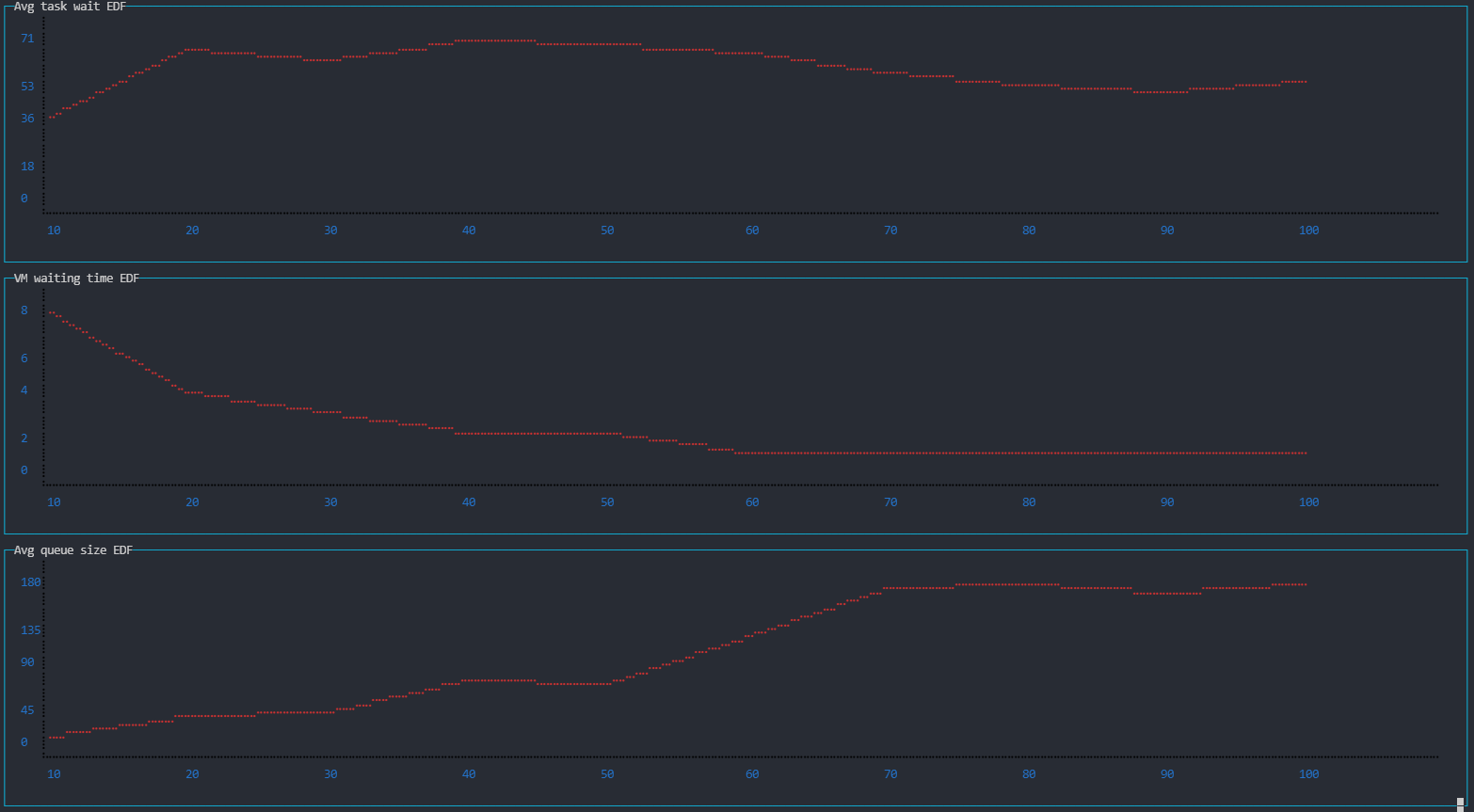
**RR:**

****

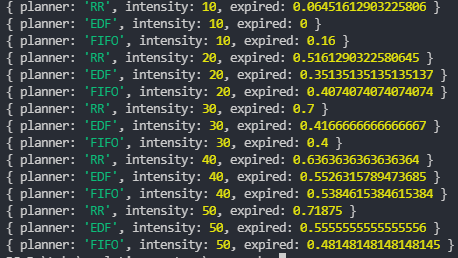
**FIFO:**

****

**EDF:**

****

Відсоток просрочених заявок:



**Лістинг коду:**

'use strict'

class BasePlanner {

    constructor() {

        this.heap = [];

    };

    add(task) {

        throw new Error();

    };

    get() {

        throw new Error();

    };

    wait() {

        this.heap.forEach(task => task.wait());

        return this;

    };

    return(task) {

        throw new Error();

    };

};

module.exports = BasePlanner;

const BasePlanner = require("./BasePlanner");

class EDF extends BasePlanner {

    constructor() {

        super();

    };

    add(task) {

        this.heap.push(task);

        return this;

    };

    get() {

        const task = this.heap

            .sort((a, b) => b.getTimeToDeadline() - a.getTimeToDeadline())

            .pop();

        return task;

    };

    return(task) {

        this.heap.push(task);

        return this;

    }

};

module.exports = EDF;

'use strict'

const BasePlanner = require("./BasePlanner");

class FIFO extends BasePlanner {

    constructor() {

        super();

    };

    add(task) {

        this.heap.unshift(task);

        return this;

    };

    get() {

        return this.heap.pop();

    };

    return(task) {

        this.heap.push(task);

        return this;

    };

};

module.exports = FIFO;

'use strict'

const BasePlanner = require('./BasePlanner');

class RR extends BasePlanner{

    constructor() {

        super();

    };

    add(task) {

        this.heap.unshift(task);

        return this;

    };

    get() {

        return this.heap.pop();

    };

    return(task) {

        this.heap.unshift(task);

        return this;

    };

};

module.exports = RR;

'use strict'

module.exports = {

    RR: require('./RR'),

    EDF: require('./EDF'),

    FIFO: require('./FIFO'),

};

'use strict'

class Task {

    constructor(complexity, deadline, priority) {

        this.complexity = complexity;

        this.process = complexity;

        this.waitingTime = 0;

        this.priority = priority ?? 0;

        this.deadline = deadline ?? Infinity;

        this.executed = false;

    };

    execute() {

        if(this.process <= 0) throw new Error();

        this.process--

        if(this.process === 0) this.executed = true;

        return this.executed || this.isExpired();

    };

    wait() {

        this.waitingTime++;

        return;

    };

    isExpired() {

        const {deadline, process, waitingTime} = this;

        const time = process + waitingTime;

        return deadline <= time;

    };

    getTimeToDeadline() {

        const {deadline, process, waitingTime} = this;

        const remnant = deadline - process - waitingTime;

        return remnant;

    };

};

module.exports = Task;

'use strict'

class VM {

    constructor(planner, quantum) {

        this.planner = planner;

        this.quantum = quantum;

        this.stats = {};

        this.tick = 0;

        this.waitingTime = 0;

        this.timer;

        this.currentTask;

        this.currentQuantum = 0;

    };

    \_tick() {

        this.currentTask = this.currentTask ?? this.planner?.get();

        const task = this.currentTask;

        if(!task) {

            this.tick++;

            this.waitingTime++;

            return

        }

        const executedOrExpired = task.execute();

        if(executedOrExpired) {

            this.stats[this.tick] = task;

            this.currentTask = null;

            this.currentQuantum = 0;

            return

        };

        this.currentQuantum++;

        this.tick++;

        this.planner.wait();

        if(this.currentQuantum >= this.quantum) {

            this.planner.return(task);

            this.currentTask = null;

            this.currentQuantum = 0;

        };

    };

    addTask(task) {

        this.planner.add(task);

        return this;

    };

    start() {

        this.timer = setInterval(() => this.\_tick(), 0);

        return this;

    };

    stop() {

        clearInterval(this.timer);

        this.timer = null;

        return this;

    };

};

module.exports = VM;

'use strict'

const {RR, EDF, FIFO} = require('./planners');

const Task = require('./Task');

const VM = require('./VM');

const Screen = require('../lab-1/chart-renderer');

const chartStyle = {line: 'red', text: 'blue', baseline: 'black'};

const random = (min, max) => Math.floor(Math.random() \* (max - min + 1)) + min;

const run = async (Planner, intensity, quantum, maxComplexity) => {

    const planner = new Planner();

    const vm = new VM(planner, quantum);

    vm.start();

    const interval = setInterval(() => {

        const complexity = random(1, maxComplexity);

        const deadline = random(complexity, maxComplexity \* 10);

        vm.addTask(new Task(complexity, deadline))

    }, 1000 / intensity);

    await new Promise(res => {

        setTimeout(() => {

            clearInterval(interval),

            vm.stop();

            res();

        }, 5000);

    });

    return vm;

};

const intensityTest = async Planner => {

    const avgTaskWait = {};

    const VMWait = {};

    const avgQueueSize = {};

    for(let i = 10; i <= 50; i += 10) {

        const vm = await run(Planner, i, 5, 20);

        const { stats, waitingTime } = vm;

        const tasks = Object.values(stats);

        const avgWaitingTime = tasks

            .reduce((acc, task) => acc += task.waitingTime, 0) / tasks.length;

        avgTaskWait[i] = avgWaitingTime;

        VMWait[i] = waitingTime;

        avgQueueSize[i] = vm.planner?.heap?.length;

        console.dir({

            planner: Planner.name,

            intensity: i,

            expired: tasks.filter(task => !task.executed).length / tasks.length

        }

        );

    };

    return {

        planner: Planner.name,

        avgTaskWait,

        VMWait,

        avgQueueSize,

    };

};

(async () => {

    const reports = await Promise

        .all([RR, EDF, FIFO].map(planner => intensityTest(planner)))

        .then(reports => reports.map(({ planner, avgTaskWait, VMWait, avgQueueSize }) => ({

            planner,

            avgTaskWait: {

                x: Object.keys(avgTaskWait),

                y: Object.values(avgTaskWait),

            },

            VMWait: {

                x: Object.keys(VMWait),

                y: Object.values(VMWait),

            },

            avgQueueSize: {

                x: Object.keys(avgQueueSize),

                y: Object.values(avgQueueSize),

            },

        })));

    const screen = new Screen(reports.length \* 3);

    reports.map(({ planner, avgTaskWait, VMWait, avgQueueSize }) => {

        screen

            .addChart(chartStyle, avgTaskWait, `Avg task wait ${planner}`)

            .addChart(chartStyle, VMWait, `VM waiting time ${planner}`)

            .addChart(chartStyle, avgQueueSize, `Avg queue size ${planner}`)

    });

    screen.render();

})();