НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Кафедра обчислювальної техніки
(повна назва кафедри, циклової комісії)

РОЗРАХУНКОВО-ГРАФІЧНА РОБОТА

з дисципліни «Інтелектуальні вбудовані системи»
(назва дисципліни)

на тему: «Дослідження роботи планувальників роботи систем реального часу»

Студента 3 курсу групи ІП-84
спеціальності
121 «Інженерія програмного забезпечення»

Шмалько Б.І.
(прізвище та ініціали)

Керівник доцент Волокіта А.Н.

Київ – 2021 рік

Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет (інститут) Інформатики та обчислювальної техніки
(повна назва)

Кафедра Обчислювальної техніки
(повна назва)

Освітньо-кваліфікаційний рівень Бакалавр

Напрям підготовки 121 «Інженерія програмного забезпечення»
(шифр і назва)

З А В Д А Н Н Я
НА РОЗРАХУНКОВО-ГРАФІЧНУ РОБОТУ

Шмальку Богдану Ігоровичу
(прізвище, ім’я, по батькові)

Тема роботи «Дослідження роботи планувальників роботи систем реального часу»

керівник роботи Волокіта Артем Миколайович к.т.н., доцент
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

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

ЗМІСТ

З *А В Д А Н Н Я* 2
Основні теоретичні відомості 4
Вимоги до системи 5
Дисципліна обслуговування 6
Розробка програми 7
Результати 7
Додатки 10

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 нитi

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) – To; математичним очікуванням часу виконання є середнє значення часу виконання відповідних обчислень в попередніх лабораторних роботах;

3) крайній строк завершення (дедлайн) – Td –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 $1/\lambda$, де λ – інтенсивність потоку.

Потоком Ерланга k-го порядку називається потік, який отримується з 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).

Пріоритети заявок

Заявки можуть мати пріоритети – явно задані, або обчислені системою (в залежності від алгоритму обслуговування або реалізації це може бути час обслуговування (обчислення), час до дедлайну і т.д.). Заявки в чергах сортуються за пріоритетом. Є два види обробки пріоритетів заявок:

1) без витіснення – якщо в чергу до ресурсу потрапляє заявка з більшим пріоритетом, ніж та, що в даний момент часу обробляється ним, то вона чекає завершення обробки ресурсом його задачі.

2) з витісненням – якщо в чергу до ресурсу потрапляє заявка з більшим пріоритетом, ніж та, що в даний момент часу обробляється ним, то вона витісняє її з обробки; витіснена задача стає в чергу.

Дисципліна обслуговування

Вибір заявки з черги на обслуговування здійснюється за допомогою так званої дисципліни обслуговування. Їх прикладами є FIFO (прийшов першим - обслуговується першим), RM (Пріоритет в цьому алгоритмі пропорційний частоті), EDF (планування по найближчому терміну завершення). У системах з очікуванням накопичувач в загальному випадку може мати складну структуру.

Дисципліна FIFO

FIFO — перший прийшов перший вийшов — є загальний принцип накопичення та обробки завдань (об'єктів). Принцип пов'язаний з поняттям черги: хто перший прийшов — той перший отримав обслуговування. Чергу можна представити у вигляді труби — з однієї сторони щось входить (стає в чергу) з іншої сторони виходить (оброблюється або обслуговується).

Дисципліна RM

В інформатиці монотонне планування (RM) - це алгоритм присвоєння пріоритетів, що використовується в операційних системах реального часу (RTOS) із класом планування статичного пріоритету. Статичні пріоритети призначаються відповідно до тривалості циклу завдання, тому менша тривалість циклу призводить до вищого пріоритету роботи.

Ці операційні системи, як правило, є переважними та мають детерміновані гарантії щодо часу відгуку. Монотонний аналіз ставок використовується разом із цими системами для забезпечення гарантій планування для конкретного застосування.

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.

Розробка програми

Мова програмування: Java Script

Для більш зручного використання була створена програма (mainPage.html), яку можна запустити у браузері.

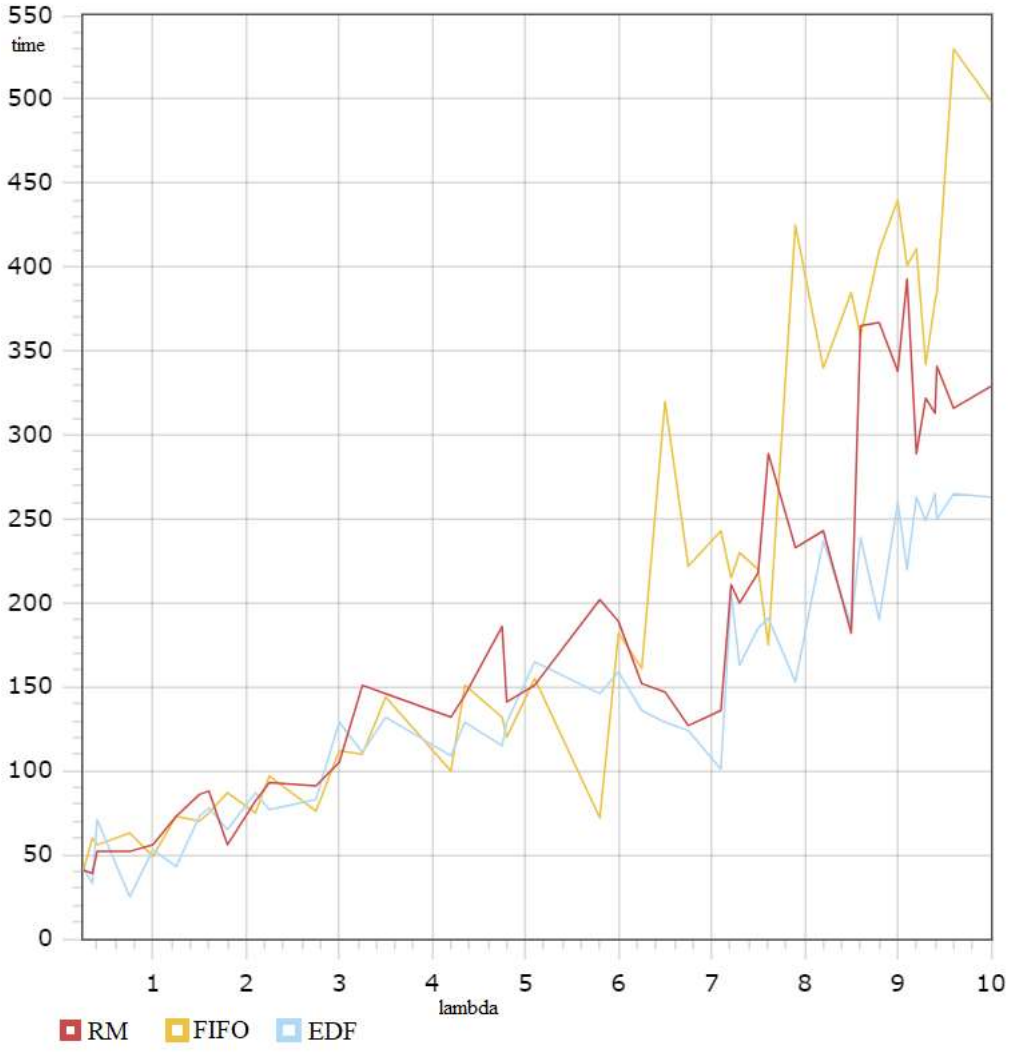
Для обчислення і розподілу ресурсів була створена функція basic(allTasks, algorithm), яка першим параметром приймає всі таски, а другим – алгоритм планування і повертає результат, статистику і затрачений час.

Для побудови графіка планування була створена функція scheduling(canvas, context, result, statistic, time), яка приймає в якості параметрів полотно, його контекст, результат виконання планування, статистику і затрачений час.

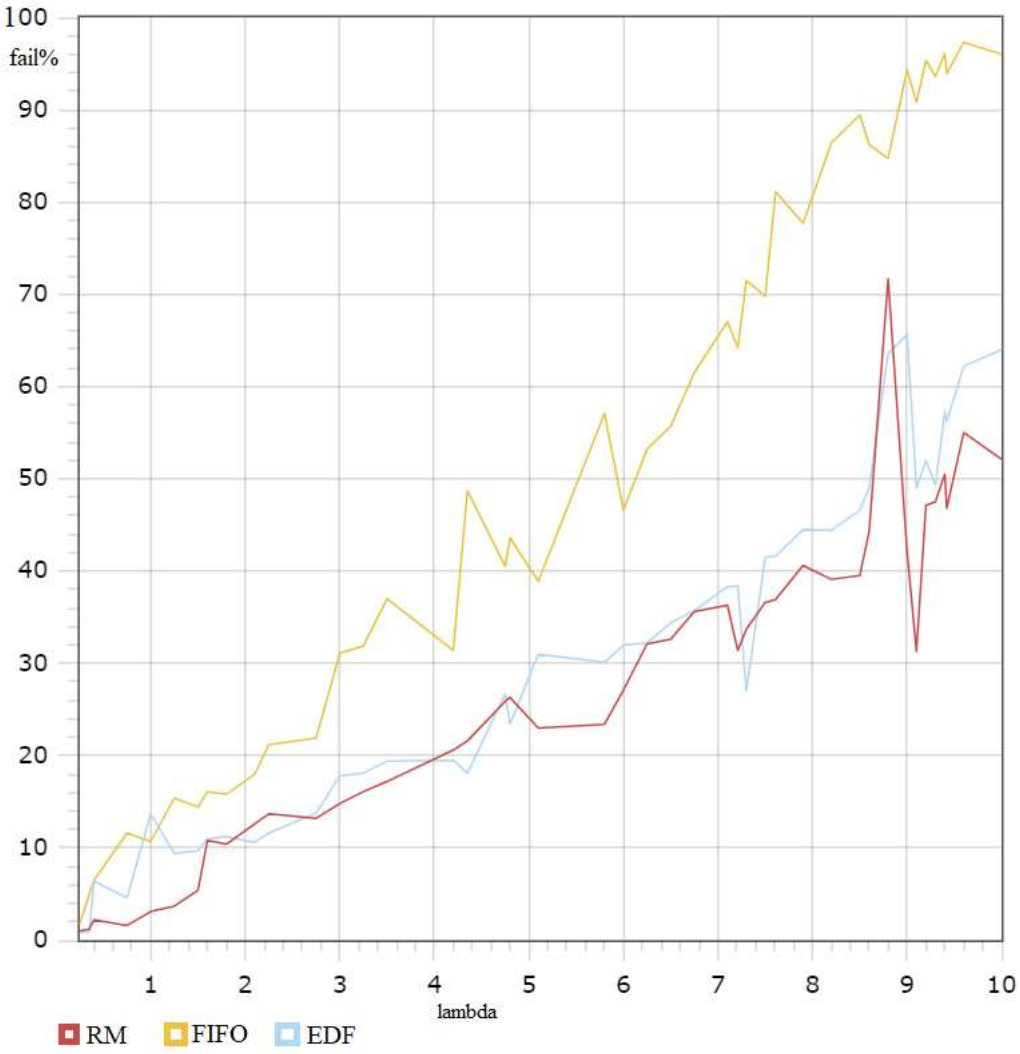
(повний код можна переглянути у додатках)

Результати

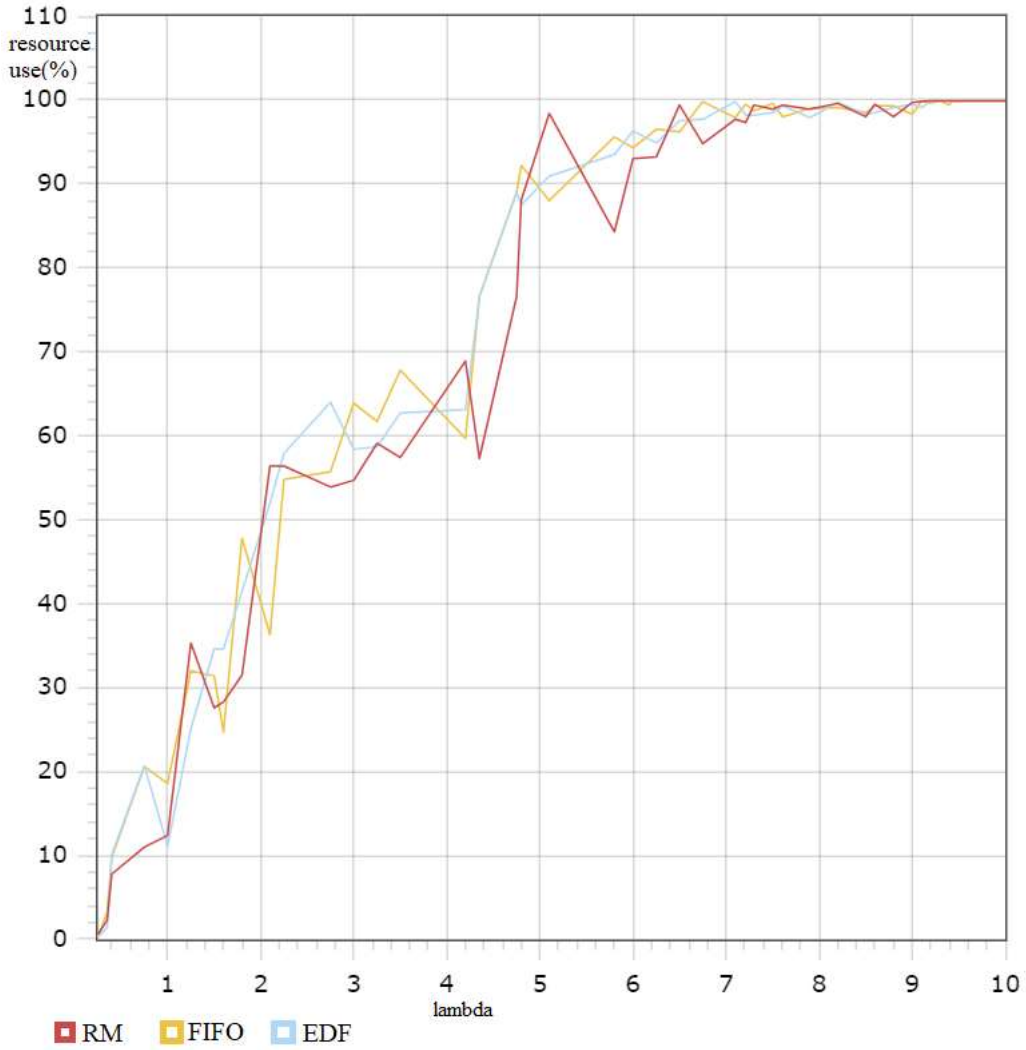
Графік залежності середнього часу очікування від інтенсивності вхідного потоку заявок



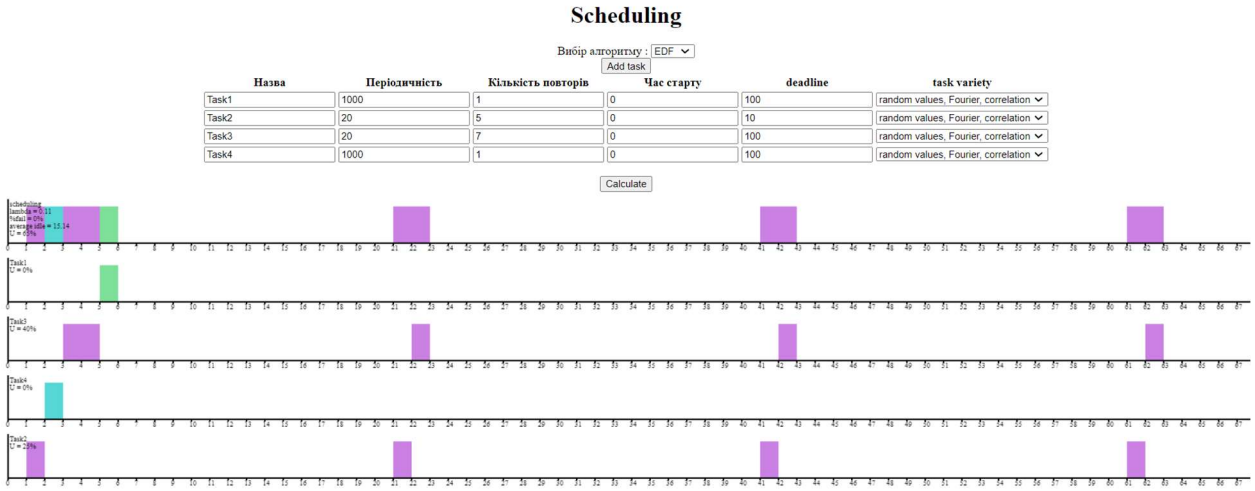
Графік відмов вхідних заявок у відсотках залежно від інтенсивності вхідного потоку заявок

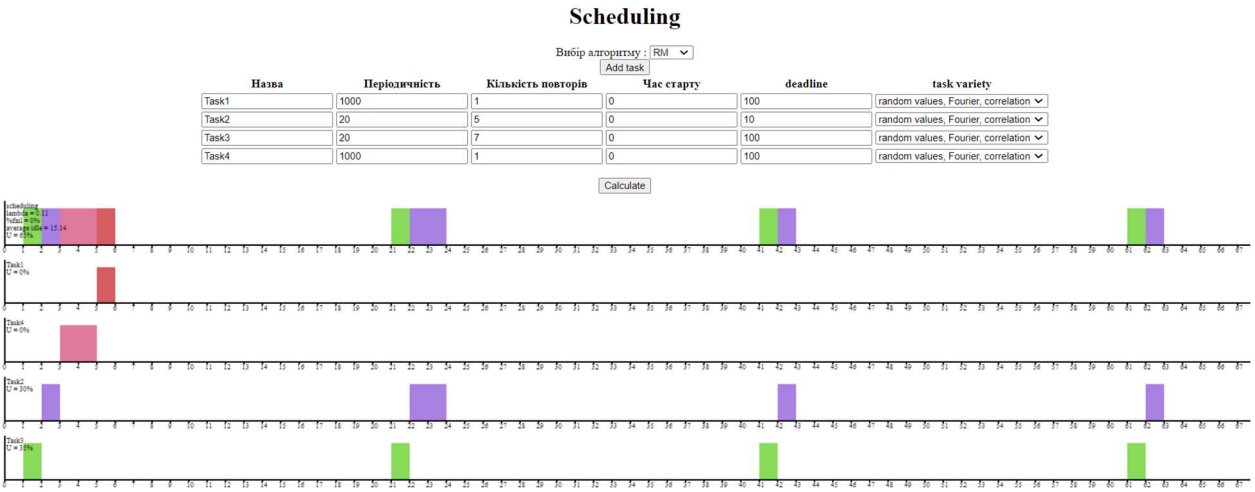


Графік залежності проценту простою ресурсу від інтенсивності вхідного потоку заявок



Результати виконання планувальників для різних алгоритмів (mainPage.html)





Додатки

drawScheduling.js

```
const sumTime = (arr, isStart) => {
    let sum = 0;
    arr.forEach(timeObj => {
        if(isStart) sum += timeObj.start
        else sum += timeObj.end
    })
    return sum
}

const drawEmptyScheduling = (canvas, ctx, index, tasks, tacksCount, Us, name,
statistic, time) => {
    const cnvWidth = canvas.width
    const cnvHeight = canvas.height / tacksCount - 30
    const iH = num => canvas.height - (num + index * 80)
    const wLine = cnvWidth / 25;

    ctx.beginPath()
    ctx.lineWidth = 2;
    ctx.font = '10px serif'

    ctx.moveTo(3, iH(10))
    ctx.lineTo(3, iH(cnvHeight))
    ctx.moveTo(3, iH(10))
    ctx.lineTo(cnvWidth, iH(10))
```

```
    for (let i = 0; i < wLine; i++) {
        ctx.moveTo(i * 25 + 3, iH(10))
        ctx.lineTo(i * 25 + 3, iH(7))
        ctx.fillText(i, i * 25, iH(0))
    }


    tasks.forEach(task => {
        const color = task.color
        ctx.fillStyle = color
        task.times.forEach(({start, end}) => {
            (end - start) % 2 !== 0 ?
                ctx.fillRect((start) * 25 + 3, iH(60), (end - start) * 25,
50) :
                ctx.fillRect((start) * 25 + 3, iH(60), (end - start) * 25,
50);
        })

    })
    ctx.fillStyle = "black"
    if (name) {
        ctx.fillText(name, 5, iH(cnvHeight - 10))
        ctx.fillText(`lambda = ${Math.floor(statistic.taskCount / time * 100)
/ 100}`, 5, iH(cnvHeight - 20))
        ctx.fillText(`%fail = ${Math.floor(statistic.failCount /
statistic.taskCount * 10000) / 100}%`, 5, iH(cnvHeight - 30))
        ctx.fillText(`average idle = ${Math.floor(statistic.idle.reduce((a,
c) => a + c) / statistic.idle.length * 100) / 100}`, 5, iH(cnvHeight - 40))
        ctx.fillText(`U = ${Us.reduce((a, c) => a + c)}%`, 5, iH(cnvHeight -
50))
    } else{
        ctx.fillText(tasks[0].name, 5, iH(cnvHeight - 10))
        const endSum = sumTime(tasks[0].times, false)
        const startSum = sumTime(tasks[0].times, true)
        const  U = Math.floor((endSum - startSum) / tasks[0].period * 100)
        Us.push(U)
        ctx.fillText(`U = ${U}%`, 5, iH(cnvHeight - 20))
    }


    ctx.stroke()
}

const scheduling = (canvas, ctx, tasks, statistic, time) => {
    ctx.strokeStyle = 'black';
    const Us = []
    tasks.forEach((task, index) => {
        drawEmptyScheduling(canvas, ctx, index, [task], tasks.length, Us);
    })
    drawEmptyScheduling(canvas, ctx, tasks.length, tasks, tasks.length, Us,
"scheduling", statistic, time);
}
```

basic.js

```
const findName = (all, name, period, data) => {
    let isFind = false

    for (let i = 0; i < all.length; i++){
        if(all[i].name === name){
            if(data)
                all[i].times.push(data)
```

```
                all[i].period = period
            isFind = true
            break
        }
    }
    if(!isFind){
        let times = []
        if(data) times.push(data)
        all.push({
            name,
            times,
            period,
            color: getColor()
        })
    }
}

const basic = (tasks, algorithm) => {
    const result = []
    const statistic = {taskCount: 0, failCount: 0, idle: []}
    const tasksTurn = []
    const tasksCopy = [...tasks]
    const start = new Date().getTime()
    let end = new Date().getTime()
    let deleteCount = 0;
    let idleStart = 0;
    while (tasksCopy.length !== deleteCount || tasksTurn.length !== 0) {
        //algorithm
        switch (algorithm){
            case "FIFO" : {
                for (let i = 0; i < tasksCopy.length; i++) {
                    if (tasksCopy[i] && end - start > tasksCopy[i].start) {
                        const {task, start, deadline, name, period} =
tasksCopy[i]
                        tasksTurn.unshift({task, deadline: start + deadline,
name, period})
                        tasksCopy[i] = undefined
                        deleteCount++
                    }
                }
                break
            }
            case "EDF" : {
                for (let i = 0; i < tasksCopy.length; i++) {
                    if (tasksCopy[i] && end - start > tasksCopy[i].start) {
                        const {task, start, deadline, name, period} =
tasksCopy[i]
                        for (let j = 0; j <= tasksTurn.length; j++){
                            if(!tasksTurn[j] || tasksTurn[j].deadline >=
start + deadline){
                                tasksTurn.splice(j, 0, {task, deadline: start
+ deadline, name, period});
                                break
                            }
                        }
                        tasksCopy[i] = undefined
                        deleteCount++
                    }
                }
                break;
            }
            case "RM" : {
                for (let i = 0; i < tasksCopy.length; i++) {
                    if (tasksCopy[i] && end - start > tasksCopy[i].start) {
```

```
                        const {task, start, deadline, name, period} =
tasksCopy[i]
                        for (let j = 0; j <= tasksTurn.length; j++){
                            if(!tasksTurn[j] || tasksTurn[j].period >=
period){
                                tasksTurn.splice(j, 0, {task, deadline: start
+ deadline, name, period});
                                break
                            }
                        }
                        tasksCopy[i] = undefined
                        deleteCount++
                    }
                }
                break;
            }
        }
        //idle
        if (tasksTurn.length === 0 && idleStart === 0) {
            idleStart = new Date().getTime()
        } else if (tasksTurn.length !== 0 && idleStart !== 0) {
            statistic.idle.push(new Date().getTime() - idleStart)
            idleStart = 0
        }
        //if deadline is end
        if (tasksTurn[0] && tasksTurn[0].deadline < end - start) {
            findName(result, tasksTurn[0].name, tasksTurn[0].period)
            tasksTurn.shift()
            statistic.failCount++
            statistic.taskCount++
        //calculate result
        } else if (tasksTurn[0]) {
            switch (tasksTurn[0].task) {
                case "correlation" : {
                    const allX = geterateValues(n, N, omega, topA, topFi)
                    const finalDataX = averageOfArrays(allX)
                    const Mx = mathematicalExpectation(finalDataX)
                    dispersion(finalDataX, Mx)
                    correlation(finalDataX, Mx)
                    FastFourier(finalDataX)
                    break
                }
                case "factorization" : {
                    ferma(191835338283)
                    break
                }
                case "neuron" : {
                    neuron([[1, 2], [3, 4], [5, 6], [7, 8]], 4, 0.001, 50,
1000)
                    break
                }
            }
            findName(result, tasksTurn[0].name, tasksTurn[0].period, {
                start: end - start,
                end: new Date().getTime() - start
            })
            tasksTurn.shift()
            statistic.taskCount++
        }

        end = new Date().getTime()
    }
    result.unshift()
```

```
    return {result, statistic, time: end - start}
}
```