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

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Розрахунково графічна робота
з дисципліни
«Інтелектуальні вбудовані системи»
на тему
«Дослідження роботи планувальників роботи систем
реального часу»

Виконала: студентка групи ІП-83
Расюк Альона
номер залікової книжки: 8519

Перевірив:
викладач
Регіда Павло Геннадійович

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 нитіУ 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.
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 P ресурсів (з порядковими номерами p) та X заявок (з порядковими номерами x), які необхідно виконати. Тоді перша заявка (x = 1) назначається для виконання на першому ресурсі (p = 1), друга (x = 2) –другому і т.д., до досягнення зайнятості останнього ресурсу (p = P, x = P) або до вичерпування необроблюваних заявок (x = X). Усі наступні заявки будуть розподілені по ресурсах аналогічно до попередніх, починаючи з першого ресурсу (x = P + 1 → p = 1, x = P + 2 → p = 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 на лабораторну роботу

1. Змоделювати планувальник роботи системи реального часу. Дві дисципліни планування: перша – RR (у нас RM), друга задається викладачем або обирається самостійно (у нас EDF).
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.

Лістинг програми

algorithms.js

```
let finishedTasks = [];
let failedTasks = [];
let idleTime = 0;
let allTime = 0;
const task1 = (t, arrT, d, aET) => {
return {
task: t,
deadline: d,
arrivalTime: arrT,
startTime: 0,
endtime: 0,
avarageExecutionTime: aET
```

```
}
}
const fifo = (queue) => {
finishedTasks = []
failedTasks = []
idleTime = 0
allTime = 0
let timeWhileExecuting = 0
const timeStart = Date.now()
while (queue.length != 0) {
const timeRan = Date.now() - timeStart
if(timeRan >= queue[0].arrivalTime){
const timeNow = Date.now()
queue[0].startTime = timeRan
const execTime = executeTask(queue[0].task)
queue[0].endtime = queue[0].startTime + (execTime == 0 ? 1: execTime)
timeWhileExecuting += (Date.now() - timeNow)
if(queue[0].deadline < queue[0].endtime) {
console.log("fifo deadline exceeded")
failedTasks.push(queue.shift())
continue
}
} else {
continue
}
finishedTasks.push(queue.shift())
}
allTime = Date.now() - timeStart
idleTime = allTime - timeWhileExecuting
}
const edf = (queue) => {
finishedTasks = []
failedTasks = []
idleTime = 0
allTime = 0
let timeWhileExecuting = 0
queue.sort((a, b) => {a.deadline - b.deadline})
const timeStart = Date.now()
while (queue.length != 0) {
allTime++;
const timeRan = Date.now() - timeStart
let currentIndex = 0
const currentTask = queue.find(element => {
currentIndex++
return element.arrivalTime <= timeRan
```

```
})
if(currentTask != undefined){
const timeNow = Date.now()
currentTask.startTime = timeRan
const execTime = executeTask(currentTask.task)
currentTask.endtime = currentTask.startTime + (execTime == 0 ? 1:
execTime)
timeWhileExecuting += (Date.now() - timeNow)
if(currentTask.deadline < currentTask.endtime) {
console.log("edf deadline exceeded")
failedTasks.push(currentTask)
queue.splice(currentIndex - 1, 1)
continue
}
finishedTasks.push(currentTask)
queue.splice(currentIndex - 1, 1)
} else {
continue
}
}
allTime = Date.now() - timeStart
idleTime = allTime - timeWhileExecuting
}
const rm = (queue) => {
finishedTasks = []
failedTasks = []
idleTime = 0
allTime = 0
let timeWhileExecuting = 0
queue.sort((a, b) => a.avarageExecutionTime - b.avarageExecutionTime)
const timeStart = Date.now()
while (queue.length != 0) {
allTime++;
const timeRan = Date.now() - timeStart
let currentIndex = 0
const currentTask = queue.find(element => {
currentIndex++
return element.arrivalTime <= timeRan
})
if(currentTask != undefined){
const timeNow = Date.now()
currentTask.startTime = timeRan
const execTime = executeTask(currentTask.task)
currentTask.endtime = currentTask.startTime + (execTime == 0 ? 1:
execTime)
```

```
timeWhileExecuting += (Date.now() - timeNow)
if(currentTask.deadline < currentTask.endtime) {
console.log("rm deadline exceeded")
failedTasks.push(currentTask)
queue.splice(currentIndex - 1, 1)
continue
}
finishedTasks.push(currentTask)
queue.splice(currentIndex - 1, 1)
} else {
idleTime++
continue
}
}
allTime = Date.now() - timeStart
idleTime = allTime - timeWhileExecuting
}
const calculateAvarageWaitingTime = () => {
let sum = 0;
for (task of finishedTasks) {
sum += (task.startTime - task.arrivalTime)
}
for(task of failedTasks) {
sum += (task.startTime - task.arrivalTime)
        }
        return sum/(finishedTasks.length + failedTasks.length)
        }
        const executeTask = (task) => {
        const signal = getSignal()
        let executionTime;
        switch (task) {
        case "discreteFourier":
        executionTime = discreteFourier(signal)
        break
        case "fastFourier":
        executionTime = fastFourier(signal)
        break
        case "getCorrelation":
        executionTime = getCorrelation(signal)
        break
        case "getMean":
        executionTime = getMean(signal)
        break
        case "getVariance":
        executionTime = getVariance(signal)
```

```
        break
        }
        return executionTime
        }
```

drawChart.html

```
<!DOCTYPE HTML>
<html>
<head>
<script>
window.onload = function () {
function testFIFO(queue, intensity) {
fifo(queue)
fifoResults.push({x: intensity, y: calculateAvarageWaitingTime()})
fifoIdleTime.push({x: intensity, y: 1 - (idleTime/allTime)})
fifoFailed.push({x: intensity , y: (failedTasks.length/100)})
}
function testEDF(queue, intensity) {
edf(queue)
edfResults.push({x: intensity, y: calculateAvarageWaitingTime()})
edfIdleTime.push({x: intensity, y: 1 - (idleTime/allTime)})
edfFailed.push({x: intensity , y: (failedTasks.length/100)})
}
function testRM(queue , intensity) {
rm(queue)
rmResults.push({x: intensity, y: calculateAvarageWaitingTime()})
rmIdleTime.push({x: intensity, y: 1 - (idleTime/allTime)})
rmFailed.push({x: intensity , y: (failedTasks.length/100)})
}
const fifoResults = []
const edfResults = []
const rmResults = []
const fifoIdleTime = []
const edfIdleTime = []
const rmIdleTime = []
const fifoFailed = []
const edfFailed = []
const rmFailed = []
for (let i = 1; i <= 10; i++) {
const queue = generateQueue(100, i)
const queue2 = [...queue]
const queue3 = [...queue]
testFIFO(queue, i)
```

```
testEDF(queue2, i)
testRM(queue3, i)
}
var chart = new CanvasJS.Chart("chartContainer", {
animationEnabled: true,
theme: "light1",
data: [{
type: "line",
indexLabelFontSize: 16,
showInLegend: true,
name: "fifo",
dataPoints: fifoResults
},
{
type: "line",
indexLabelFontSize: 16,
showInLegend: true,
name: "edf",
dataPoints: edfResults
},
{
type: "line",
indexLabelFontSize: 16,
showInLegend: true,
name: "rm",
dataPoints: rmResults
}]
});
var chart2 = new CanvasJS.Chart("chartContainer2", {
animationEnabled: true,
theme: "light1",
data: [{
type: "line",
indexLabelFontSize: 16,
showInLegend: true,
name: "fifo",
dataPoints: fifoIdleTime
}, {
type: "line",
indexLabelFontSize: 16,
showInLegend: true,
name: "edf",
dataPoints: edfIdleTime
}, {
type: "line",
```

```
indexLabelFontSize: 16,
showInLegend: true,
name: "rm",
dataPoints: rmIdleTime
}]
});
var chart3 = new CanvasJS.Chart("chartContainer3", {
animationEnabled: true,
theme: "light1",
data: [{
type: "line",
indexLabelFontSize: 16,
showInLegend: true,
name: "fifo",
dataPoints: fifoFailed
}, {
type: "line",
indexLabelFontSize: 16,
showInLegend: true,
name: "edf",
dataPoints: edfFailed
}, {
type: "line",
indexLabelFontSize: 16,
showInLegend: true,
name: "rm",
dataPoints: rmFailed
}]
});
chart.render();
chart2.render();
chart3.render();
}
</script>
</head>
<body>
<div id="chartContainer" style="height: 500px; width: 500px;"></div>
<div id="chartContainer2" style="height: 500px; width: 500px;"></div>
<div id="chartContainer3" style="height: 500px; width: 500px;"></div>
<script src="https://canvasjs.com/assets/script/canvasjs.min.js"></script>
<script src="signalGenerator.js"></script>
<script src="algorithms.js"></script>
<script src="generateQueue.js"></script>
</body>
</html>
```

generateQueue.js

```
const generateQueue = (numberOfTasks, intensity) => {
const queue = [];
let x = Math.ceil(numberOfTasks/intensity);
for (let j = 0; j < x; j++) {
const currentDeadline = 100 * j
for (let i = 0; i < intensity; i++) {
const rand = Math.floor(Math.random() * 4)
const rand2 = Math.floor(Math.random() * 2) + 1
let task;
switch (rand) {
case 0:
task = task1("discreteFourier", 100 * j, currentDeadline + 100 * rand2 * intensity, 66)
break
case 1:
task = task1("fastFourier", 100 * j, currentDeadline + 80 * rand2 * intensity, 35)
break
case 2:
task = task1("getCorrelation", 100 * j, currentDeadline + 15 * rand2 * intensity, 8)
break
case 3:
task = task1("getMean", 100 * j, currentDeadline + 1 * rand2 * intensity , 1)
break
}
queue.push(task)
if (queue.length === numberOfTasks) {
break
}
}
}
return queue
}
```

signalGenerator.js

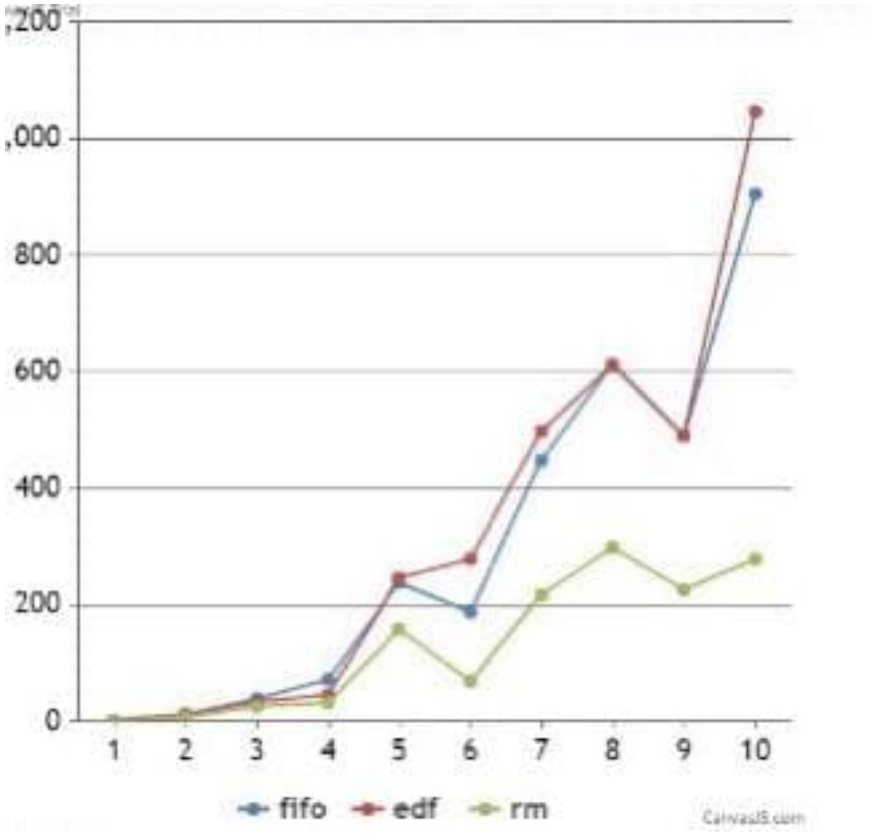
```
const n = 6
const w = 2100
const N = 1024
const getSignal = () => {
const x = []
while(x.length != N) {
x.push({y: 0})
}
```

```
for(let i = 0; i < n; i++) {
const omega = w/n * (i + 1);
const A = Math.random();
const Fi = Math.random();
for(let t = 0; t < N; t++) {
x[t].y += A * Math.sin(omega * t + Fi)
}
}
return x
}
const complexNumber = () => {
return {real:0, im: 0}
}
const discreteFourier = (signals) => {
const result = []
const start = Date.now()
for (let p = 0; p < N; p++) {
let num = complexNumber()
for(let k = 0; k < N; k++) {
num.real += signals[k].y * Math.cos(2 * Math.PI * p * k / N)
num.im -= signals[k].y * Math.sin(2 * Math.PI * p * k / N)
}
result.push({y: Math.sqrt(Math.pow(num.im, 2) + Math.pow(num.real, 2))})
}
const end = Date.now()
return end - start
}
const fastFourier = (signal) => {
const result = [];
result.length = N;
const start = Date.now()
for(let p = 0; p < N/2; p++) {
let even = complexNumber()
let odd = complexNumber()
for(let k = 0; k < N/2; k++) {
even.real += signal[2*k].y * Math.cos(2 * Math.PI * p * 2*k / N);
even.im -= signal[2*k].y * Math.sin(2 * Math.PI * p * 2*k / N);
odd.real += signal[2*k + 1].y * Math.cos(2 * Math.PI * p * (2*k + 1) / N);
odd.im -= signal[2*k + 1].y * Math.sin(2 * Math.PI * p * (2*k + 1) / N);
}
result[p] = {y: Math.sqrt((even.real + odd.real) 2 + (even.im + odd.im) 2)};
result[N/2 + p] = {y: Math.sqrt((even.real - odd.real) 2 + (even.im - odd.im) 2)};
}
const end = Date.now()
return end - start
```

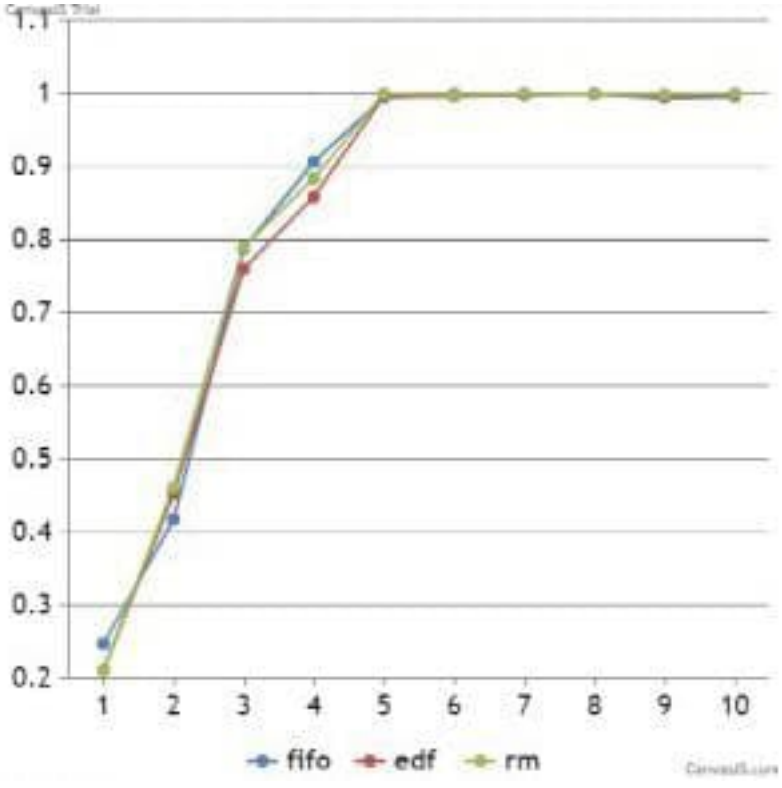
```
}
const getMean = (arrOfValues) => {
const start = Date.now()
let sum = 0;
for(let i = 0; i < arrOfValues.length; i++) {
sum += arrOfValues[i].y;
}
const end = Date.now()
return end - start
}
const getVariance = (arrOfValues, mean) => {
const start = Date.now()
let sum = 0;
for(let i = 0; i < arrOfValues.length; i++) {
sum += Math.pow(arrOfValues[i].y - mean, 2);
}
const end = Date.now()
return end - start
}
const getCorrelation = (x, y = null) => {
if(y === null) {
y = x
} else {
}
let startTime = Date.now()
let meanX = getMean(x)
let meanY = getMean(y)
const result = []
for(let i = 0; i < N; i++) {
let sum = 0
for(let j = 0;j < N-i; j++) {
sum += (x[j].y - meanX)*(y[j+i].y - meanY)
}
result.push({y: sum/(N - 1)})
}
let endTime = Date.now()
return endTime - startTime
}
```

Результати виконання

Графік залежності середнього часу очікування від інтенсивності.



Графік залежності проценту простою від інтенсивності.



Графік залежності кількості відмов від інтенсивності.