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 теми
«Дослідження роботи планувальників роботи систем

реального часу»

Виконав:

студент групи ІП-84
Сімонов Павло
номер залікової книжки: 8421

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 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.

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

MAIN.PY

```
import matplotlib.pyplot as plt
import random
import Erlang as Erlang
import Task
import SMO
import numpy

a = []
lam = 1
k = 2
E = Erlang.ErlangDistribution(k, lam)

# Generate a task queue with Erlang creation time and normal completion time
def GenerateQ():
    Q = []
    i = 0
    time = GenerateTime()
    while i < 10000:
        i += E.GenerateNextInternal() * 8
        t = Task.Task(i, time)
        Q.append(t)
    return Q

# Normal distribution, 10% Rxx, 10% Rxy, 30% Dx, 50% Mx
def GenerateTime():
    rnd = random.random()
    if rnd < 0.3:
        ans = random.randrange(7)  # Rxy
    elif 0.3 <= rnd < 0.6:
        ans = random.randrange(5)  # Rxx
    elif 0.6 <= rnd < 0.8:
        ans = random.randrange(3)  # Dx
    else:
        ans = random.randrange(2)  # Mx
    return ans + 40


if __name__ == "__main__":
```

```
QuFIFO = []
QuEDF = []
QuRM = []

FIFOs = []
EDFs = []
RMs = []

Tw = []
Tww = []
Tn = []
t = [x for x in range(10000)]

faults = []
faultschange = []
Tnchange = []
Twchange = []
FullWaitTime = []

for lam in numpy.arange(1, 20, 0.5):
    E.ChangeLambda(lam)
    temp = GenerateQ()
    QuFIFO.append(temp)

QuRM = QuFIFO[:]
QuEDF = QuFIFO[:]

for i in range(len(QuFIFO)):
    FIFOs.append(SMO.FIFO(QuFIFO[i]))
    RMs.append(SMO.RM(QuRM[i]))
    EDFs.append(SMO.EDF(QuEDF[i]))

buf1 = []
buf2 = []
buf3 = []

def calculate(arr, name):
    for elem in arr:
        elem.work()
        buf1 = elem.GetFaults()
        buf2 = elem.GetWaitTimes()
        buf3 = elem.GetProcessorFreeTime()
        faults.append(buf1[:])
        Tw.append(buf2[:])
        Tn.append(buf3[:])
```

```
        for elem in range(len(arr)):
            faultschange.append(faults[elem][9999])
            Tnchange.append(Tn[elem][9999])

        for o in range(len(arr)):
            time = 0
            for elem in range(10000):
                time += Tw[o][elem]
            FullWaitTime.append(time)

        for elem in range(len(arr)):
            Tww.append(FullWaitTime[elem])

        plt.plot(Tnchange, 'r')
        plt.title(name)
        plt.show()
        plt.plot(faultschange, 'g')
        plt.title(name)
        plt.show()
        plt.plot(Tww, 'k')
        plt.title(name)
        plt.show()

        buf1.clear()
        buf2.clear()
        buf3.clear()
        Tw.clear()
        Tn.clear()
        faults.clear()
        faultschange.clear()
        Tnchange.clear()
        Twchange.clear()
        Tww.clear()

    calculate(FIFOs, "FIFO")
    calculate(EDFs, "EDF")
    calculate(RMs, "RM")
```

SMO.PY

```
class FIFO:
    currentTime = 0
    Q = []  # Task queue
```

```
Qready = []
Tw = [0 for x in range(10000)]  # Wait times in queue
Tn = [0 for y in range(10000)]  # Processor free
faults = [0 for z in range(10000)]  # Deadlines missed
currentTask = None

def __init__(self, Q):
    self.Q = Q

def GetEDTask(self, removeFromQ):
    timeBuf = 9999999  # never going to happen
    taskwED = None
    for i in range(len(self.Qready)):
        newTime = self.Qready[i].getDeadline()
        if timeBuf > newTime:
            timeBuf = newTime
            taskwED = self.Qready[i]
    if removeFromQ:
        self.Qready.remove(taskwED)
    return taskwED

def ToReadyQueue(self):
    for i in range(len(self.Q)):
        if self.Q[i].getCreationTime() == self.currentTime:
            self.Qready.append(self.Q[i])

def CheckForDeadlines(self):
    flt = 0
    flti = []
    for i in range(len(self.Qready)):
        if self.Qready[i].getDeadline() < self.currentTime:
            flt += 1
            flti.append(i)
    if self.currentTime == 0:
        self.faults[self.currentTime] = flt
    else:
        self.faults[self.currentTime] = self.faults[self.currentTime - 1] + flt
    for i in range(len(flti)):
        del self.Qready[flti[i]]
        for j in range(i, len(flti)):
            flti[j] -= 1

# Modelling
def work(self):
    self.currentTime = 0
    for self.currentTime in range(10000):
```

```
            if self.currentTime != 0:
                self.Tn[self.currentTime] = self.Tn[self.currentTime - 1]
            timewait = 0
            self.CheckForDeadlines()
            self.ToReadyQueue()
            if self.currentTask is not None and self.currentTask.getExecutionTime() == 0:
                self.currentTask = None
            elif self.currentTask is not None:
                self.currentTask.workedOn()
            if self.GetEDTask(False) is None and self.currentTask is None:  # Check for tasks in
queue
                self.Tn[self.currentTime] += 1
                continue
            elif self.currentTask is None:
                self.currentTask = self.GetEDTask(True)  # Take an ED task
            for task in self.Qready:
                task.wait()
                timewait += 1  # Adds time for each task that is in the queue
            self.Tw[self.currentTime] = timewait

    def GetWaitTimes(self):
        return self.Tw

    def GetFaults(self):
        return self.faults

    def GetProcessorFreeTime(self):
        return self.Tn


class RM:
    currentTime = 0
    Q = []  # Task queue
    Qready = []
    Tw = [0 for x in range(10000)]  # Wait times in queue
    Tn = [0 for y in range(10000)]  # Processor free
    faults = [0 for z in range(10000)]  # Deadlines missed
    currentTask = None

    def __init__(self, Q):
        self.Q = Q

    def getEDTask(self, removeFromQ):
        timeBuf = 9999999  # never going to happen
        taskwED = None
        for i in range(len(self.Qready)):
```

```
            newTime = self.Qready[i].getDeadline()
            if timeBuf > newTime:
                timeBuf = newTime
                taskwED = self.Qready[i]
        if removeFromQ:
            self.Qready.remove(taskwED)
        return taskwED

    def ToReadyQueue(self):
        for i in range(len(self.Q)):
            if self.Q[i].getCreationTime() == self.currentTime:
                self.Qready.append(self.Q[i])

    def CheckForDeadlines(self):
        flt = 0
        flti = []
        for i in range(len(self.Qready)):
            if self.Qready[i].getDeadline() < self.currentTime:
                flt += 1
                flti.append(i)
        if self.currentTime == 0:
            self.faults[self.currentTime] = flt
        else:
            self.faults[self.currentTime] = self.faults[self.currentTime - 1] + flt
        for i in range(len(flti)):
            del self.Qready[flti[i]]
            for j in range(i, len(flti)):
                flti[j] -= 1

    # Modelling
    def work(self):
        self.currentTime = 0
        for self.currentTime in range(10000):
            if self.currentTime != 0:
                self.Tn[self.currentTime] = self.Tn[self.currentTime - 1]
            timewait = 0
            self.CheckForDeadlines()
            self.ToReadyQueue()
            if self.currentTask is not None and self.currentTask.getExecutionTime() == 0:
                self.currentTask = None
            elif self.currentTask is not None and self.getEDTask(False) is not None:
                if self.getEDTask(False).getExecutionTime() < self.currentTask.getExecutionTime():
                    self.Qready.append(self.currentTask)
                    self.currentTask = self.getEDTask(True)
            elif self.currentTask is not None:
                self.currentTask.workedOn()
```

```
            if self.getEDTask(False) is None and self.currentTask is None:  # Check for tasks in
queue
                self.Tn[self.currentTime] += 1
                continue
            elif self.currentTask is None:
                self.currentTask = self.getEDTask(True)  # Take an ED task
            for task in self.Qready:
                task.wait()
                timewait += 1  # Adds time for each task that is in the queue
            self.Tw[self.currentTime] = timewait

    def GetWaitTimes(self):
        return self.Tw

    def GetFaults(self):
        return self.faults

    def GetProcessorFreeTime(self):
        return self.Tn

class EDF:
    currentTime = 0
    Q = []  # Task queue
    Qready = []
    Tw = [0 for x in range(10000)]  # Wait times in queue
    Tn = [0 for y in range(10000)]  # Processor free
    faults = [0 for z in range(10000)]  # Deadlines missed
    currentTask = None

    def __init__(self, Q):
        self.Q = Q

    def GetEDTask(self, removeFromQ):
        timeBuf = 9999999  # never going to happen
        taskwED = None
        for i in range(len(self.Qready)):
            newTime = self.Qready[i].getDeadline()
            if timeBuf > newTime:
                timeBuf = newTime
                taskwED = self.Qready[i]
        if removeFromQ:
            self.Qready.remove(taskwED)
        return taskwED

    def ToReadyQueue(self):
        for i in range(len(self.Q)):
```

```
            if self.Q[i].getCreationTime() == self.currentTime:
                self.Qready.append(self.Q[i])

    def CheckForDeadlines(self):
        flt = 0
        flti = []
        for i in range(len(self.Qready)):
            if self.Qready[i].getDeadline() < self.currentTime:
                flt += 1
                flti.append(i)
        if self.currentTime == 0:
            self.faults[self.currentTime] = flt
        else:
            self.faults[self.currentTime] = self.faults[self.currentTime - 1] + flt
        for i in range(len(flti)):
            del self.Qready[flti[i]]
            for j in range(i, len(flti)):
                flti[j] -= 1

    # Modelling
    def work(self):
        self.currentTime = 0
        for self.currentTime in range(10000):
            if self.currentTime != 0:
                self.Tn[self.currentTime] = self.Tn[self.currentTime - 1]
            timewait = 0
            self.CheckForDeadlines()
            self.ToReadyQueue()
            if self.currentTask is not None and self.currentTask.getExecutionTime() == 0:
                self.currentTask = None
            elif self.currentTask is not None and self.GetEDTask(False) is not None:
                if self.GetEDTask(False).getDeadline() < self.currentTask.getDeadline():
                    self.Qready.append(self.currentTask)
                    self.currentTask = self.GetEDTask(True)
            elif self.currentTask is not None:
                self.currentTask.workedOn()
            if self.GetEDTask(False) is None and self.currentTask is None:  # Check for tasks in
queue
                self.Tn[self.currentTime] += 1
                continue
            elif self.currentTask is None:
                self.currentTask = self.GetEDTask(True)  # Take an ED task
            for task in self.Qready:
                task.wait()
                timewait += 1  # Adds time for each task that is in the queue
            self.Tw[self.currentTime] = timewait
```

```
    def GetWaitTimes(self):
        return self.Tw

    def GetFaults(self):
        return self.faults

    def GetProcessorFreeTime(self):
        return self.Tn
```

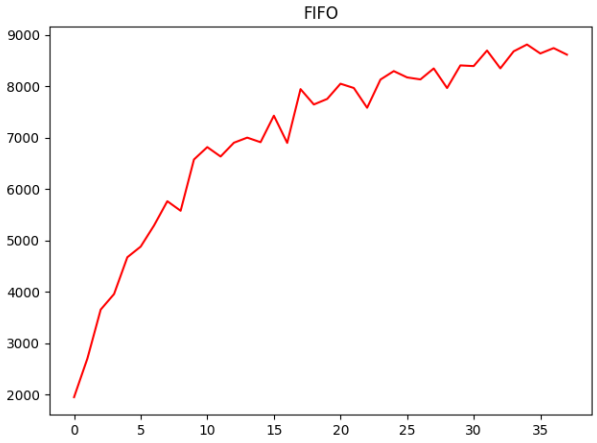
ERLANG.PY

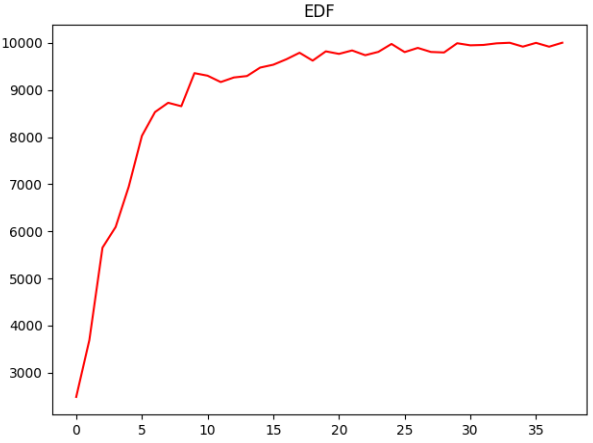
```
import random
import math

class ErlangDistribution:

    k = 0           # order
    lam = 0         # rate

    def __init__(self, k, lam):
        self.lam = lam
        self.k = k
        if k == 0:
            raise Exception("Order parameter can't be less than 1!")
        if lam <= 0:
            raise Exception("Streaming rate can't be less or equal 0!")

    # Uses random (normal distribution) values to generate Erlang
    # random value
    def GenerateNext(self):
        res = 0
        for n in range(self.k - 1):
            if res != 0:
                res = random.random() * res
            else:
                res = random.random()
        res = 0 - (math.log(res) / self.lam)
        return res

    def GenerateNextInternal(self):
        return random.gammavariate(self.k, self.lam)

    def ChangeLambda(self, lam):
        self.lam = lam
```

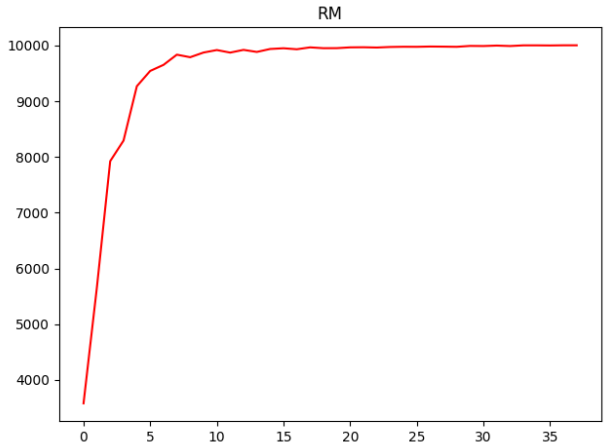
TASK.PY

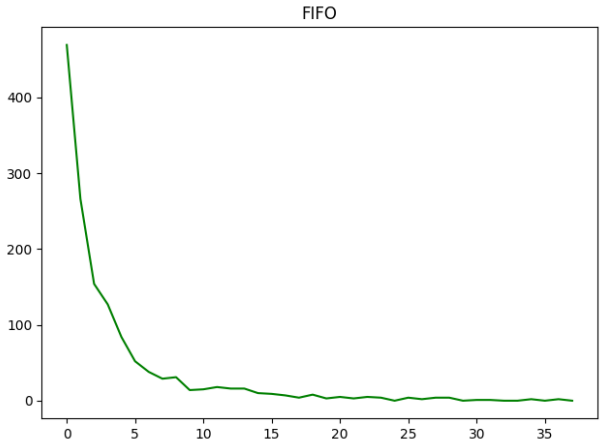
```
class Task:

    deadline = 0
    creationTime = 0
    executionTime = 0
    deadlineMultiplier = 0  # less than 1 will result in faults
    waitTime = 0

    def __init__(self, creationTime, executionTime):
        self.executionTime = int(executionTime)
        self.creationTime = int(creationTime)
        self.deadline = int(creationTime + executionTime * self.deadlineMultiplier)

    def getTimeLimit(self):
        return self.deadline + self.creationTime

    def getDeadline(self):
        return self.deadline

    def workedOn(self):
        self.executionTime = self.executionTime - 1

    def wait(self):
        self.waitTime = self.waitTime + 1

    def getExecutionTime(self):
        return self.executionTime

    def getCreationTime(self):
        return self.creationTime
```

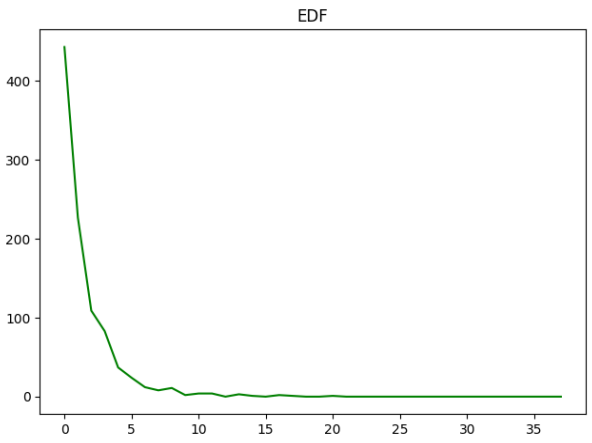
Результати роботи програми

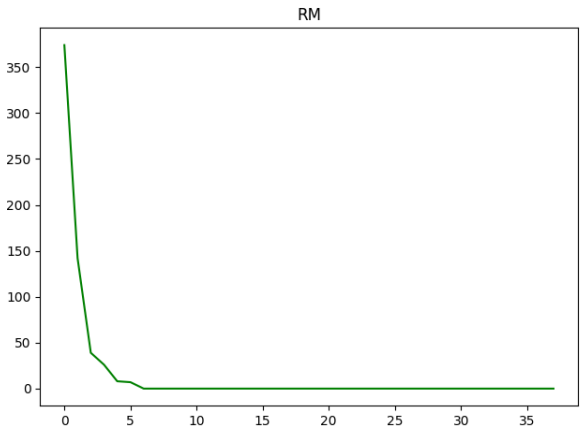

FIFO
9000
8000
7000
6000
5000
4000
3000
2000
0
5
10
15
20
25
30
35

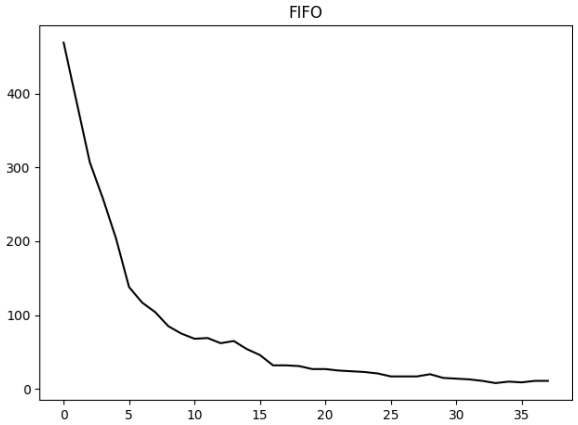

EDF
10000
9000
8000
7000
6000
5000
4000
3000
0
5
10
15
20
25
30
35

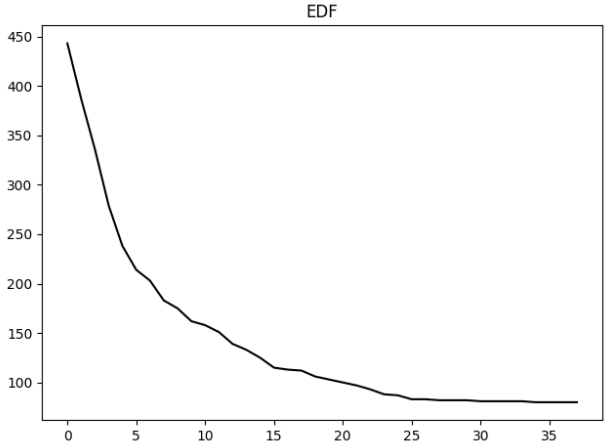

RM
10000
9000
8000
7000
6000
5000
4000
0
5
10
15
20
25
30
35

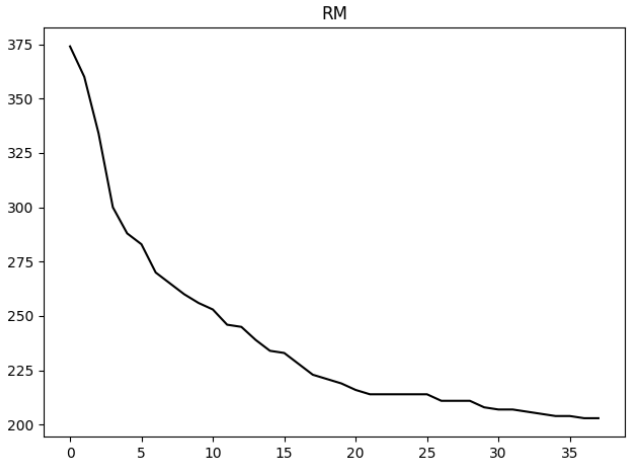

FIFO
400
300
200
100
0
0
5
10
15
20
25
30
35


EDF
400
300
200
100
0
0
5
10
15
20
25
30
35


RM
350
300
250
200
150
100
50
0
0
5
10
15
20
25
30
35







Висновки

У ході виконання лабораторної роботи було розроблено три СМО: FIFO, RM та EDF. Було досліджено використання СМО в системах реального часу та змодельовано їх роботу