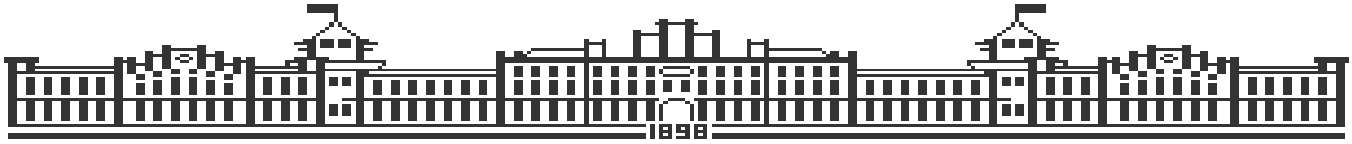

Міністерство освіти і науки України

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»

Перевірив:
доцент, кандидат технічних наук
Волокита А. М.

ВИКОНАВ:
студент 3 курсу
групи ІП-83, ФІОТ
Мінченко В.Ю.
Залікова книжка №8315
Варіант – 18

Київ 2021

Зміст

Основні теоретичні відомості

1.1 Планування виконання завдань

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.

1.2 Система масового обслуговування

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.

1.3 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.

1.4 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).

1.5 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.

1.5.1 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.

1.5.2 Дисципліна RM

У комп'ютерній науці швидкісно-монотонне планування (RM) є алгоритмом призначення пріоритетів, який використовується в операційних системах реального часу (RTOS) з класом планування статичного пріоритету. Статичні пріоритети призначаються відповідно до тривалості циклу завдання, тому коротша тривалість циклу призводить до більш високого пріоритету роботи.

Ці операційні системи, як правило, є переважними і мають детерміновані гарантії щодо часу реагування. Швидкість монотонного аналізу використовується

в поєднанні з цими системами для забезпечення гарантій планування для конкретного застосування.

Завдання

1. Змоделювати планувальник роботи системи реального часу. Дві дисципліни планування: перша – RM, друга 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.

Виконання

Для перевірки правильності виконання програми було взято 2 потоки Ерланга 10-го порядку, в одному із яких знаходяться заявки із часом виконання 1-2 ЛР, а у другому відповідно із 3 та 4 ЛР.

Було протестовано 3 планувальники : EDF, RM та FIFO.

При виконанні програми були отримані наступні графіки:

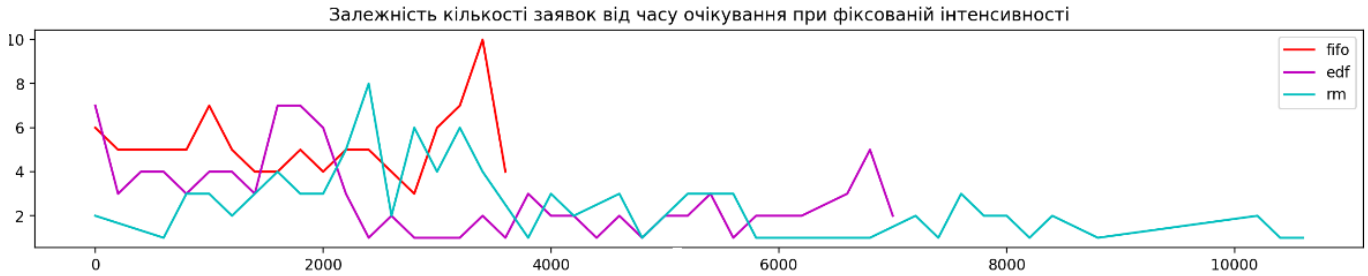

Рисунок 3.1 – Графік залежності кількості заявок від часу очікування при фіксованій інтенсивності

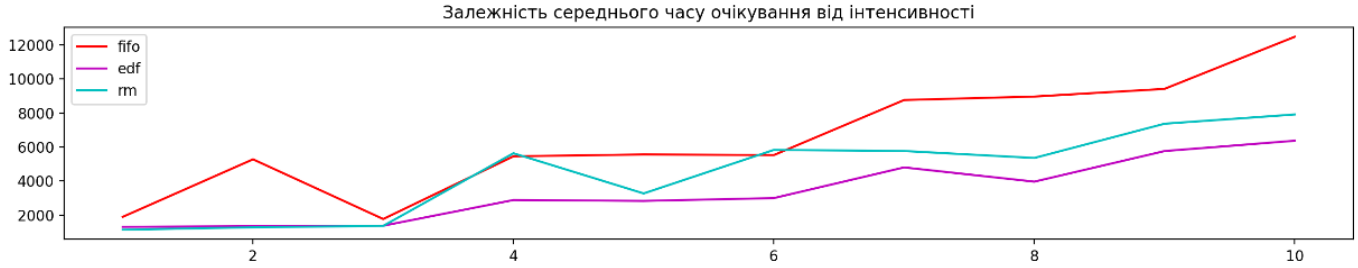

Рисунок 3.2 – Графік залежності середнього часу очікування від інтенсивності

Рисунок 3.3 – Графік залежності проценту простою від інтенсивності

Основні результати та висновки до роботи

У результаті даної розрахунково-графічної роботи можна зробити наступні висновки:

а) У першому розділі даної роботи було розглянуто основні теоретичні відомості по темі розробки планувальників роботи систем реального часу, у тому числі були розглянуті принципи таких дисциплін планування, як EDF та RM.

б) У третьому розділі відбулося виконання програми для трьох планувальників – за дисциплінами FIFO, EDF та RM, в результаті чого були отримані 3 графіки:

- графік залежності кількості заявок від часу очікування при фіксованій інтенсивності
- графік залежності середнього часу очікування від інтенсивності
- графік залежності проценту простою від інтенсивності

в) Як видно з графіка №1 (Рисунок 3.1), залежність кількості заявок від часу очікування при фіксованій інтенсивності є доволі різною для усіх трьох дисциплін планування, що спричинене специфікою самих дисциплін. Для FIFO пік кількості заявок припадає приблизно на 3400 тактів очікування, а найбільшим значенням часу очікування є 3600 тактів, тобто довше ніж стільки заявки у цього планувальника не очікують своєї черги. Це спричинене тим, що у FIFO заявки формують черги за принципом «перший прийшов-перший пішов», і не виконуються тільки якщо пройшов дедлайн. Пік кількості заявок встановлений у майже максимальній точці можливого часу очікування каже про те, що дана дисципліна є не дуже добре оптимізованою. Із кривої EDF бачимо, що пік кількості заявок припадає приблизно на 1600 тактів, що є гарним результатом, максимальним часом очікування є приблизно 7000 тактів. Це спричинене тим, що даний планувальник в пріоритеті бере ті задачі, дедлайн яких ще не пройшов та є найближчим. У підсумку, цей алгоритм є досить оптимізованим, адже пік кількості заявок знаходиться у першій третині його графіку. Як видно із кривої RM, пік кількості заявок у даного планувальника припадає на 2200 тактів. Максимальний час очікування приблизно 10000 тактів. Це спричинене тим, що спочатку шукається заявка із мінімальним часом виконання, а потім із мінімальною лямбдою. Даний

алгоритм є добре оптимізованим, адже пік кількості заявок знаходиться у першій чверті графіка.

г) Графік №2 (Рисунок 3.2) відображає залежність середнього часу очікування від інтенсивності. Як бачимо, зі збільшенням інтенсивності все помітнішою стає різниця між середнім часом очікування для трьох дисциплін. Для FIFO вона має найбільше значення, що є цілком логічним, адже планування за цим принципом є найменш оптимальним із усіх трьох. Для RM при найбільшій інтенсивності середній час очікування є в 1,5 рази менше ніж для FIFO, тож планування за цією дисципліною є набагато вигіднішим для потоку з великою кількістю заявок. Але найбільш вигідним виявилося планування EDF, адже середній час очікування його заявок при максимальній інтенсивності є меншим майже у 3 рази, ніж у FIFO і у 1,5 рази меншим, ніж у RM.

д) Як можемо побачити із графіка №3 (Рисунок 3.3), графіки для усіх трьох дисциплін майже на всьому діапазоні накладаються, тож залежність проценту простою від інтенсивності є майже однаковою для всіх планувальників. Мінімальну різницю можна помітити до інтенсивності 3 включно, але в деяких планувальників тут процент простою виявився трохи більшим через фактор рандому а також через замалу інтенсивність для заданих кількостей часу виконання заявок.

Лістинг коду

Main.py

```
import numpy as np
from dataclasses import dataclass
from random import randint
from typing import List
import matplotlib.pyplot as plt
from times import get_times
from concurrent.futures import ProcessPoolExecutor, wait, ALL_COMPLETED
from collections import defaultdict


@dataclass
class Task:
    time_start: int
    time_count: int
    k: int
    l: int

    @property
    def time_deadline(self) -> int: return self.time_start + self.k *
self.time_count


def fifo(queue: List[Task]) -> int:
    return 0


def defalt(): return 0


def edf(queue: List[Task]) -> int:
    return min(enumerate(queue), key=lambda t: t[1].time_deadline)[0]


def rm(queue: List[Task]) -> int:
    smallest_time_index = min(enumerate(queue), key=lambda t: t[1].time_count)[0]

    smallest_tasks_by_time = [(i, t) for i, t in enumerate(queue) if
                              t.time_count ==
queue[smallest_time_index].time_count]

    return min(smallest_tasks_by_time, key=lambda t: t[1].l)[0]


class Scheduler:
    number_of_ticks = 0
    queue: List[Task] = list()
    rejected = 0
    work_ticks = 0
    left_to_exec = None
    total_wt = 0
    total_processed_tasks = 0

    def __init__(self, erlang1, erlang2, n=256, simulation_length=1000,
selection_strategy=fifo, k=2, l1=3, l2=5):
        self.simulation_length = simulation_length
        self.selection_strategy = selection_strategy
        self.l1 = l1
        self.l2 = l2
        self.k = k
        self.erlang1 = erlang1
```

```
        self.erlang2 = erlang2
        self.times = get_times(n=n)
        self.dict = defaultdict(defalt)

    def __calculate_wt(self, t: Task):
        fake_queue = list(self.queue)
        wait_time = 0
        while True:
            i = self.selection_strategy(fake_queue)
            if fake_queue[i] == t:
                break
            wait_time += fake_queue[i].time_count
            fake_queue.pop(i)
        return wait_time

    def __select(self):
        return self.selection_strategy(self.queue)

    def __new_task(self):
        was_added = False
        if self.erlang1[self.number_of_ticks] > self.erlang1[self.number_of_ticks
- 1]:
            time_count = self.times[randint(0, 1)]
            self.queue.append(Task(self.number_of_ticks, time_count, self.k,
self.l1))
            was_added = True
        if self.erlang2[self.number_of_ticks] > self.erlang2[self.number_of_ticks
- 1]:
            time_count = self.times[2 + randint(0, 1)]
            self.queue.append(Task(self.number_of_ticks, time_count, self.k,
self.l2))
            was_added = True
        if was_added is True:
            wt = self.__calculate_wt(self.queue[-1])
            self.dict[wt // 200] += 1
            self.total_wt += wt
            self.total_processed_tasks += 1

    def one_tick(self):
        if self.number_of_ticks != 0:
            self.__new_task()
        for i, t in enumerate(self.queue):
            if self.number_of_ticks > t.time_deadline - t.time_count:
                self.rejected += 1
                del self.queue[i]
        if self.left_to_exec is None and len(self.queue) != 0:
            i = self.__select()
            self.left_to_exec = self.queue.pop(i).time_count
        if self.left_to_exec is not None:

            self.work_ticks += 1

            self.left_to_exec -= 1

            if self.left_to_exec == 0:
                self.left_to_exec = None
        self.number_of_ticks += 1

    def simulate(self):
        for i in range(self.simulation_length):
            self.one_tick()
        return self

    @property
    def awt(self):
        return self.total_wt / self.total_processed_tasks
```

```
    @property
    def work_time(self):
        return self.work_ticks / self.number_of_ticks


def main():
    simulation_length = 100
    awt = [[], [], []]
    work_time = [[], [], []]
    xs = list(range(1, 11))
    erlang_k = 10
    k = 5
    with ProcessPoolExecutor() as pool:
        for l in xs:
            print(f'l = {l}')

            erlang1 = np.cumsum(np.random.gamma(1 / erlang_k, 1 / l,
simulation_length))
            erlang2 = np.cumsum(np.random.gamma(1 / erlang_k, 1 / l,
simulation_length))
            filo_scheduler = Scheduler(erlang1=erlang1, erlang2=erlang2,
selection_strategy=fifo,
                                       simulation_length=simulation_length, l1=l,
l2=l, k=k)
            edf_scheduler = Scheduler(erlang1=erlang1, erlang2=erlang2,
selection_strategy=edf,
                                      simulation_length=simulation_length, l1=l,
l2=l, k=k)
            rm_scheduler = Scheduler(erlang1=erlang1, erlang2=erlang2,
selection_strategy=rm,
                                     simulation_length=simulation_length, l1=l,
l2=l, k=k)
            schedulers = [filo_scheduler, edf_scheduler, rm_scheduler]
            futures = []
            for s in schedulers:
                futures.append(pool.submit(s.simulate))
            wait(futures, return_when=ALL_COMPLETED)
            schedulers = [f.result() for f in futures]
            for i, s in enumerate(schedulers):
                awt[i].append(s.awt)
                work_time[i].append(s.work_time)

    simulation_length = 100
    l = 4
    erlang1 = np.cumsum(np.random.gamma(1 / erlang_k, 1 / l, simulation_length))
    erlang2 = np.cumsum(np.random.gamma(1 / erlang_k, 1 / l, simulation_length))
    filo_scheduler = Scheduler(erlang1=erlang1, erlang2=erlang2,
selection_strategy=fifo,
                               simulation_length=simulation_length, l1=l, l2=l,
k=k)
    edf_scheduler = Scheduler(erlang1=erlang1, erlang2=erlang2,
selection_strategy=edf,
                              simulation_length=simulation_length, l1=l, l2=l,
k=k)
    rm_scheduler = Scheduler(erlang1=erlang1, erlang2=erlang2,
selection_strategy=rm,
                             simulation_length=simulation_length, l1=l, l2=l, k=k)
    filo_scheduler.simulate()
    edf_scheduler.simulate()
    rm_scheduler.simulate()
    wtd = [None, None, None]
    wtd[0] = sorted(filo_scheduler.dict.items())
    wtd[1] = sorted(edf_scheduler.dict.items())
    wtd[2] = sorted(rm_scheduler.dict.items())
    plt.figure(dpi=200, figsize=(16, 9))
```

```
    plt.subplot(311)
    plt.title('Залежність кількості заявок від часу очікування при фіксованій
інтенсивності')
    plt.plot(list(map(lambda v: v[0] * 200, wtd[0])), list(map(lambda v: v[1],
wtd[0])), label='fifo', color="r")
    plt.plot(list(map(lambda v: v[0] * 200, wtd[1])), list(map(lambda v: v[1],
wtd[1])), label='edf', color="m")
    plt.plot(list(map(lambda v: v[0] * 200, wtd[2])), list(map(lambda v: v[1],
wtd[2])), label='rm', color="c")
    plt.legend()
    plt.subplot(312)
    plt.title('Залежність середнього часу очікування від інтенсивності')
    plt.plot(xs, awt[0], label='fifo', color="r")
    plt.plot(xs, awt[1], label='edf', color="m")
    plt.plot(xs, awt[2], label='rm', color="c")
    plt.legend()
    plt.subplot(313)
    plt.title('Залежність проценту простою від інтенсивності')
    plt.step(xs, work_time[0], label='fifo', color="r")
    plt.step(xs, work_time[1], label='edf', color="m")
    plt.step(xs, work_time[2], label='rm', color="c")
    plt.legend()
    plt.savefig('out/out.png', pad_inches=0.1, bbox_inches='tight')
    plt.show()


if __name__ == '__main__':
    main()
```

times.py

```
from os import path
from time import time
import signal_generator
import statistics_utils
import lab2_1
import lab2_2
from pathlib import Path
import numpy as np
from math import ceil

FILE_PATH = 'out/times.txt'
HARMONICS = 10
FREQUENCY = 1500
DISCRETE_CALLS = 256

def get_times(n: int = 256, repeat_times=100, is_main=False):
    Path((path.dirname(FILE_PATH))).mkdir(exist_ok=True)

    if Path(FILE_PATH).is_file() and is_main is False:
        with open(FILE_PATH, 'r') as f:
            lines = [line.strip() for line in f]
            if n == int(lines[0]):
                times = [int(line) for line in lines[1:]]
                return times

    times = np.empty((4, repeat_times))
    for i in range(repeat_times):
        signal = signal_generator.generate_signal(
            HARMONICS,
            FREQUENCY,
            DISCRETE_CALLS
```

```
        )
        start = time()
        statistics_utils.math_expectation(signal)
        statistics_utils.math_variance(signal)
        end = time()
        times[0, i] = (end - start)
        start = time()
        statistics_utils.auto_correlation(signal, signal)
        end = time()

        times[1, i] = (end - start)
        start = time()
        lab2_1.dft(signal)
        end = time()
        times[2, i] = (end - start)
        start = time()
        lab2_2.fft(signal)
        end = time()
        times[3, i] = (end - start)
    times = [ceil(t * 10_000) for t in np.mean(times, axis=1)]
    with open(FILE_PATH, 'w') as f:
        print(n, file=f)
        for t in times:
            print(t, file=f)
    return times

if __name__ == ' main ':
    get_times(is_main=True)
```

statistics_utils.py

```
import numpy as np


def math_expectation(signal_data):
    return np.mean(signal_data)


def math_variance(signal_data):
    return np.var(signal_data)


def cross_correlation(signal1_data, signal2_data):
    result = np.correlate(signal1_data, signal2_data, mode='same')
    return result


def auto_correlation(signal_data):
    result = np.correlate(signal_data, signal_data, mode='full')
    return result[result.size // 2:]
```

signal_generator.py

```
import random
import numpy as np


def generate_signal(signal_harmonics, frequency, discrete_calls):
    signals = np.zeros(discrete_calls)
```

```
    for i in range(signal_harmonics):
        frequency_step = frequency / signal_harmonics * (i + 1)
        amplitude = random.random()
        phase = random.random()
        for t in range(discrete_calls):
            signals[t] += amplitude * np.sin(frequency_step * t + phase)
    return signals
```

lab2_1.py

```
import numpy as np
from math import sin, cos, pi


import math

def calc_w(p, k, N):
    argument = 2.0 * math.pi * p * k / N
    return complex(math.cos(argument), -math.sin(argument))

def dft(signal):
    N = len(signal)
    spectre = [0] * N
    for p in range(N):
      for k in range(N):
        x = signal[k]
        w = calc_w(p, k, N)
        spectre[p] += w * x
    return [*map(lambda el: abs(el), spectre)]
```

lab2_2.py

```
import math

from lab2_1 import calc_w


def fft(signal):
    N = len(signal)
    spectre = [0] * N
    if N == 1: return signal
    even_signal, odd_signal = signal[::2], signal[1::2]
    even_transformed = fft(even_signal)
    odd_transformed = fft(odd_signal)
    for k in range(0, int(N / 2)):
        w = calc_w(1, k, N)
        spectre[k] = even_transformed[k] + w * odd_transformed[k]
        spectre[k + int( N / 2)] = even_transformed[k] - w * odd_transformed[k]
    return [*map(lambda el: abs(el), spectre)]
```