

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 планвальників роботи системи реального часу»

Виконав:
студент групи ІП-84
Гудь В.В.
№ залікової книжки: ІП-8405

Перевірів:
Волокита А.М.

Київ 2021

ЗМІСТ

ОСНОВНІ ТЕОРЕТИЧНІ ВІДОМОСТІ.....	3
ВИМОГИ ДО СИСТЕМИ.....	4
Вхідні задачі.....	4
Потік вхідних задач.....	4
Пріоритети заявок.....	4
Дисципліни обслуговування.....	5
Дисципліна FIFO.....	5
Дисципліна EDF.....	5
Дисципліна RM.....	5
РОЗРОБКА ПРОГРАМИ.....	5
РЕЗУЛЬТАТИ ВИКОНАННЯ.....	6
ВИКОРИСТАНІ ДЖЕРЕЛА.....	7
ДОДАТКИ.....	8

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.
- Пропускну 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[2]

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 вимог, при цьому очікувані 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[4].

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 – T_r – потік заявок є потоком Пуассона або потоком Ерланга k -го порядку;
2. час виконання (обробки) – T_o ; математичним очікуванням часу виконання є середнє значення часу виконання відповідних обчислень в попередніх лабораторних роботах;
3. крайній строк завершення (дедлайн) – T_d , якщо заявка залишається необробленою в момент часу $t = T_d$, 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 $(k + i)$ -ї події (решта відкидаються). Наприклад, якщо зобразити на часовій осі потік Пуассона, поставивши у відповідність кожній події деяку точку, і відкинути з потоку кожен другу подію (точку на осі), то отримаємо потік Ерланга 2-го порядку. Залишивши лише кожен 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.

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.

В даній роботі алгоритми реалізовані без витіснення за явно заданими пріоритетами. При цьому деякі алгоритми можуть витіснити задачі базуючись на внутрішніх обчислюваних пріоритетах

Дисципліни 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.

В даній роботі використовувалися дисципліни обслуговування FIFO(First In First Out), EDF(Earliest Deadline First) та RM(Rate Monotonic)

Дисципліна FIFO

Алгоритм FIFO(Перший прийшов - перший пішов) є статичним алгоритмом планування, що означає пріоритет задачі визначається лише зовні. В даному алгоритму задача, що прийшла перша обслуговується першою.

Дисципліна EDF

Алгоритм EDF(спочатку найперший термін) є динамічним алгоритмом планування з динамічним пріоритетом. Планувальник надає перевагу задачі, що знаходиться найблище до свого терміну виконання. Цей алгоритм гарантує високу вірогідність виконання задачі в заданий термін, але через це може відбуватись затриманні заявок з довгим терміном виконання в черзі.

Дисципліна RM

Алгоритм RM це динамічний алгоритм з статичними пріоритетами заявок. Алгоритм в першу чергу обробляє ті заявки що надходять частіше. Через це черга заявок при виконанні планування менша ніж в інших алгоритмів, але збільшується ризик порушення дедлайну задач, що приходять рідше.

РОЗРОБКА ПРОГРАМИ

Мова програмування: Python

Клас Scheduler при ініціалізації приймає 4 аргументи - чергу, кількість тактів симуляції, алгоритм планування(RM, EDF чи FIFO) та розмір такту. Для початку планування викликається функція schedule.

Клас Generator генерує чергу заявок. При ініціалізації приймає 2 аргументи - інтенсивність та кількість тактів. Для генерації викликається функція generate_queue, що приймає середній час виконання задач(з значення). Генератор генерує завдання для кожного такту згідно інтенсивності.

Клас Task симулює задачу. При ініціалізації приймає 3 аргументи - час потрапляння в систему, час виконання заявки та термін виконання.

Для збору статистики використовується спеціально виділений об'єкт. В ньому збирається статистика про виконані задачі, відкинуті, довжину черги, час очікування, тощо.

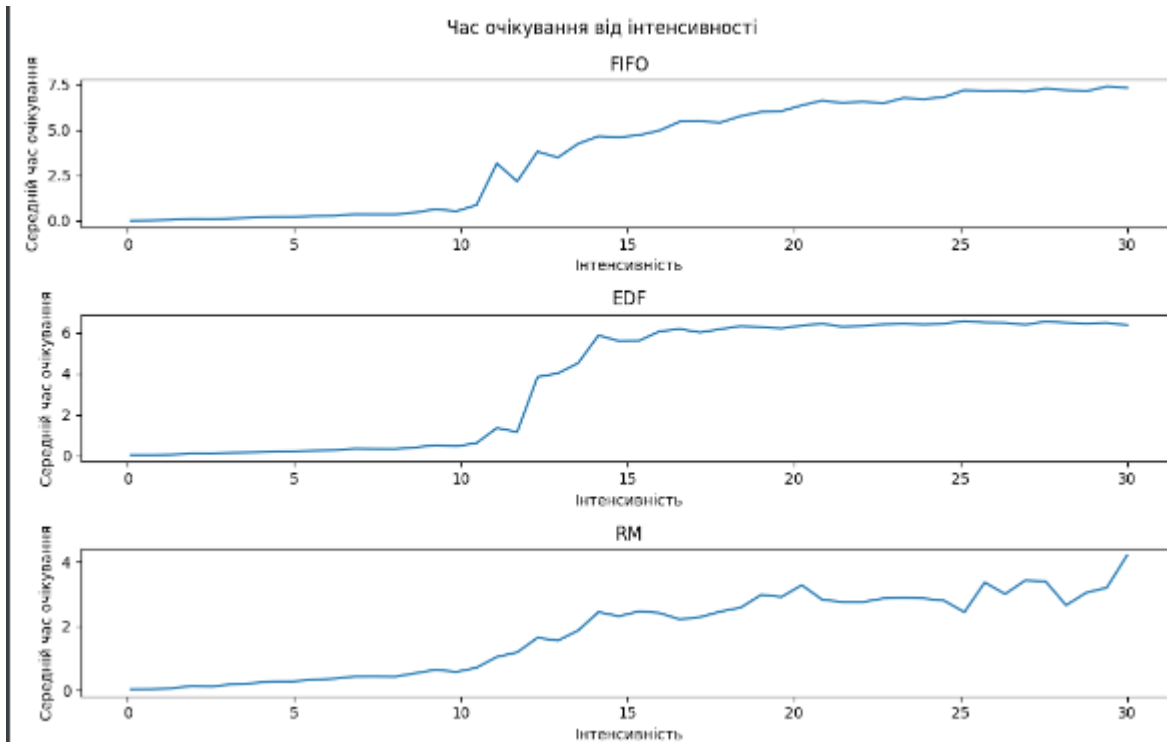
Для малювання графіків було розроблено функцію plot_stats, що приймає алгоритм малювання та відображає зібрану статистику.

РЕЗУЛЬТАТИ

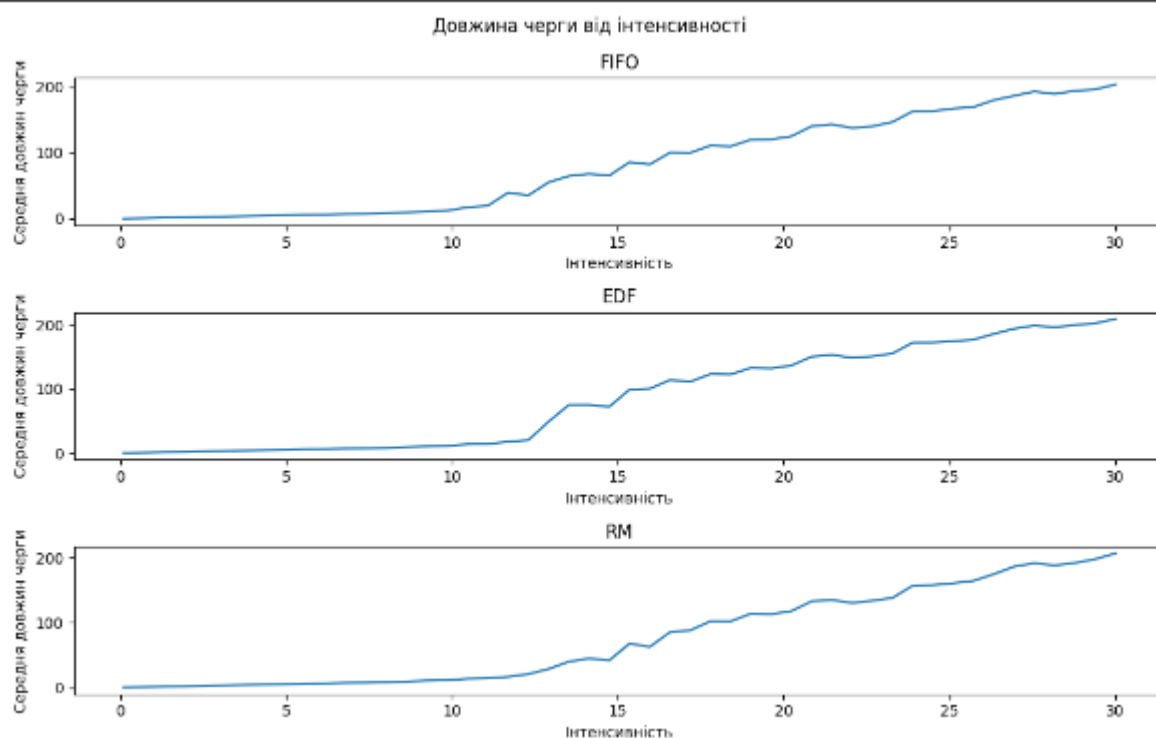
Початкові параметри

- Кількість тактів - 100
- Середній час виконання заявок - 0.045, 0.09 та 0.2
- Розмір такту - 1
- Інтенсивність - від 0.1 до 30, з кроком в 0.99

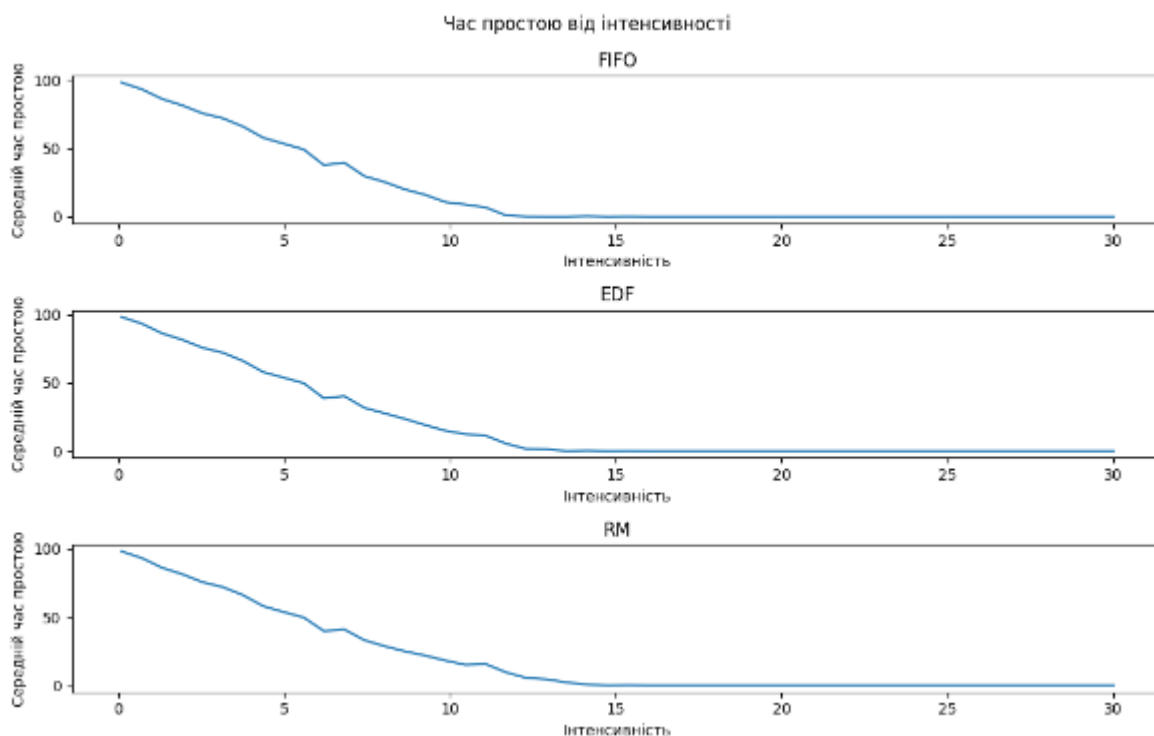
Графіки



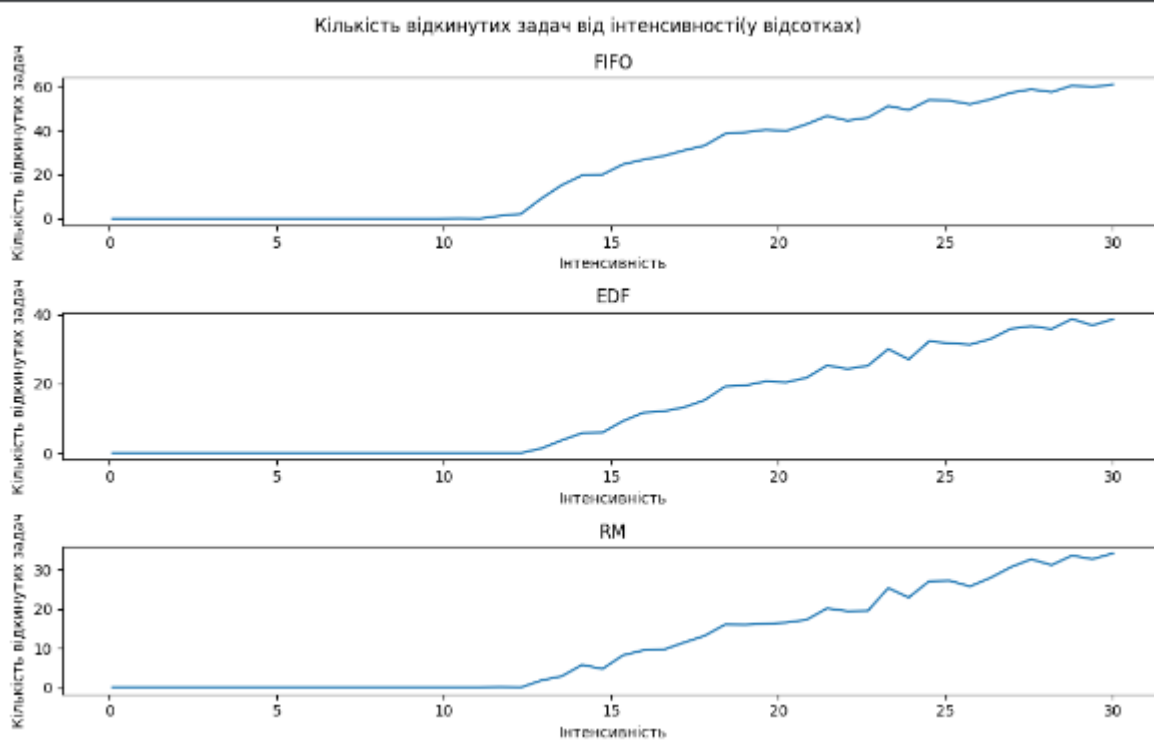
Час очікування алгоритму RM є найменшим, EDF показує себе трохи кращим за FIFO



Довжина черги у всіх трьох алгоритмів росте приблизно однаково, хоч RM має перевагу приблизно до інтенсивності 15, а черга FIFO починає швидко рости раніше інших - при інтенсивності 10.

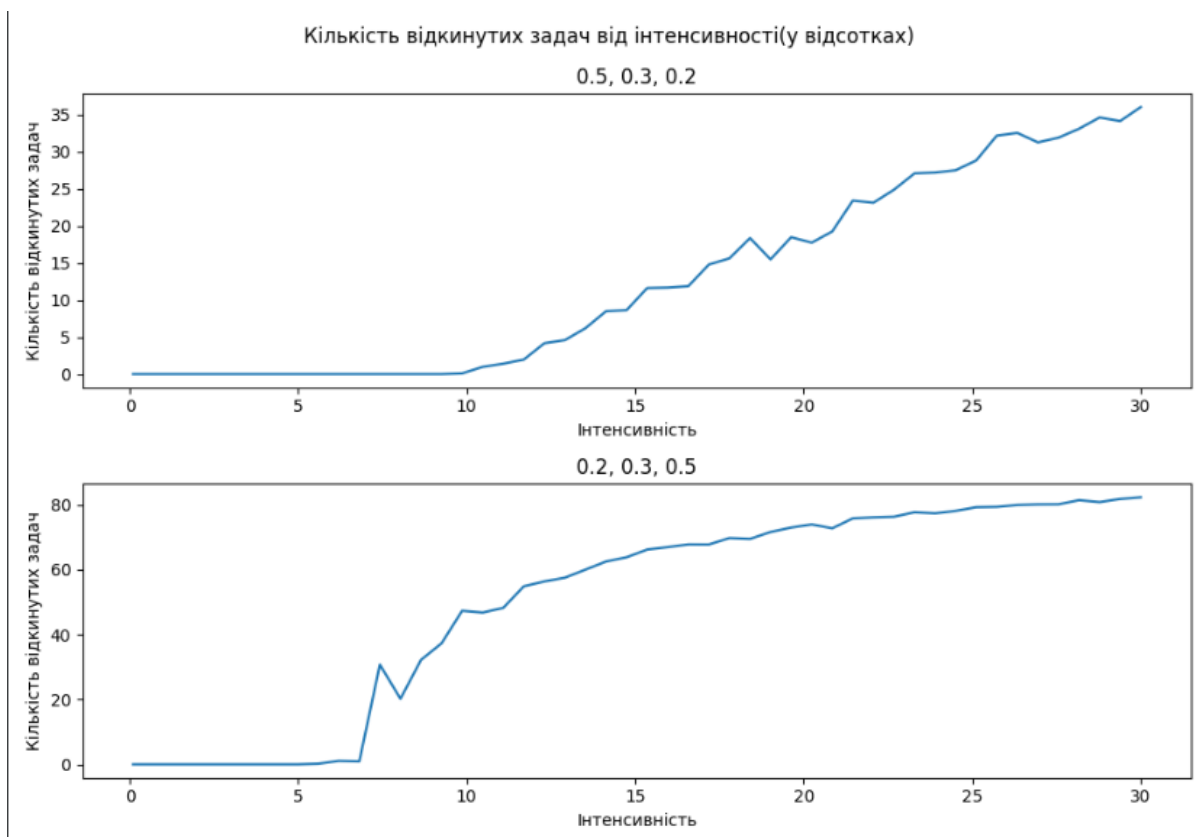


Час простою всіх алгоритмів є практично ідентичним



Можна помітити, що у алгоритмі RM кількість відкинутих задач росте найповільніше, тоді як у FIFO - найшвидше

Додаткове завдання



Змінивши інтенсивність відповідних типів задач, можна помітити, що на графіку де найбільше створювалось задач з найдовшим часом виконання відсоток відкинутих задач алгоритмом RM є значно вищим, ніж у графіка, де найкоротші задачі є найчастішими.

ВИКОРИСТАНА ЛІТЕРАТУРА

1. Scheduling [Електронний ресурс] – Режим доступу до ресурсу: [https://en.wikipedia.org/wiki/Scheduling_\(computing\)](https://en.wikipedia.org/wiki/Scheduling_(computing))
2. Poisson distribution [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Poisson_distribution
3. EDF scheduling algorithm [Електронний ресурс] – Режим доступу до ресурсу: <https://www.geeksforgeeks.org/earliest-deadline-first-edf-cpu-scheduling-algorithm/>
4. Rate monotonic scheduling [Електронний ресурс] – Режим доступу до ресурсу: <https://www.geeksforgeeks.org/rate-monotonic-scheduling/>
5. First in first out [Електронний ресурс] – Режим доступу до ресурсу: <https://www.geeksforgeeks.org/program-for-fcfs-cpu-scheduling-set-1/>

ДОДАТОК

main.py

```

from EarliestDeadlineFirst import EDF
from RateMonotonic import RM
from Plotter import plot_stats,\
    plot_wait_time,\
    plot_free_time,\
    plot_avg_queue_len,\
    plot_discarded,\
    plot_rm_reverse_intensity

# plot_stats('FIFO')

# plot_stats(EDF)
# plot_stats(RM)

# plot_wait_time()
# plot_free_time()
# plot_avg_queue_len()
# plot_discarded()

plot_rm_reverse_intensity()

```

Plotter.py

```

import numpy as np
from PoissonGenerator import Generator
from SMO import Scheduler
import matplotlib.pyplot as plt
from EarliestDeadlineFirst import EDF
from RateMonotonic import RM

```

```

AVG_SOLUTION_1 = 0.045
AVG_SOLUTION_2 = 0.09
AVG_SOLUTION_3 = 0.2
QUEUE_SIZE = 100
intensities = np.linspace(0.1, 30)
tact_size = 1

```

```

def plot_stats(algorithm):
    stats_wait_time = []
    stats_free_time = []
    stats_avg_queue_length = []
    for i in intensities:

```

```

generator = Generator(i, QUEUE_SIZE, 0.5, 0.3, 0.2)
queue = generator.generate_queue(AVG_SOLUTION_1, AVG_SOLUTION_2,
AVG_SOLUTION_3)
scheduler = Scheduler(
    queue, QUEUE_SIZE, algorithm, tact_size
)
stats = scheduler.schedule
stats_wait_time.append(stats['avg_wait_time'])
stats_free_time.append(stats['free_time'])
stats_avg_queue_length.append(stats['avg_queue_length'])

```

```

fig = plt.figure(figsize=(11, 7))
if algorithm != 'FIFO':
    fig.suptitle(algorithm.__name__)
else:
    fig.suptitle('FIFO')
plot = fig.add_subplot(311)
plot.set_title('Час очікування від інтенсивності')
plot.set_xlabel('Інтенсивність')
plot.set_ylabel('Середній час очікування')
plot.plot(intensities, stats_wait_time)
plot = fig.add_subplot(312)
plot.set_title('Довжина черги від інтенсивності')
plot.set_xlabel('Інтенсивність')
plot.set_ylabel('Середня довжин черги')
plot.plot(intensities, stats_avg_queue_length)
plot = fig.add_subplot(313)
plot.set_title('Час простою від інтенсивності')
plot.set_xlabel('Інтенсивність')
plot.set_ylabel('Середній час простою')
plot.plot(intensities, stats_free_time)
plt.subplots_adjust(wspace=0.1, hspace=1, bottom=0.1, top=0.9)
plt.tight_layout()
plt.show()

```

```

def plot_wait_time():
    stats_wait_time_1 = []
    stats_wait_time_2 = []
    stats_wait_time_3 = []

    for i in intensities:
        generator = Generator(i, QUEUE_SIZE, 0.5, 0.3, 0.2)

```

```

    queue_1 = generator.generate_queue(AVG_SOLUTION_1, AVG_SOLUTION_2,
    AVG_SOLUTION_3)
    queue_2 = queue_1.copy()
    queue_3 = queue_1.copy()
    scheduler_1 = Scheduler(
        queue_1, QUEUE_SIZE, 'FIFO', tact_size
    )
    scheduler_2 = Scheduler(
        queue_2, QUEUE_SIZE, EDF, tact_size
    )
    scheduler_3 = Scheduler(
        queue_3, QUEUE_SIZE, RM, tact_size
    )
    stats_1 = scheduler_1.schedule
    stats_2 = scheduler_2.schedule
    stats_3 = scheduler_3.schedule
    stats_wait_time_1.append(stats_1['avg_wait_time'])
    stats_wait_time_2.append(stats_2['avg_wait_time'])
    stats_wait_time_3.append(stats_3['avg_wait_time'])

```

```

fig = plt.figure(figsize=(11, 7))
fig.suptitle('Час очікування від інтенсивності')
plot = fig.add_subplot(311)
plot.set_title('FIFO')
plot.set_xlabel('Інтенсивність')
plot.set_ylabel('Середній час очікування')
plot.plot(intensities, stats_wait_time_1)
plot = fig.add_subplot(312)
plot.set_title('EDF')
plot.set_xlabel('Інтенсивність')
plot.set_ylabel('Середній час очікування')
plot.plot(intensities, stats_wait_time_2)
plot = fig.add_subplot(313)
plot.set_title('RM')
plot.set_xlabel('Інтенсивність')
plot.set_ylabel('Середній час очікування')
plot.plot(intensities, stats_wait_time_3)
plt.subplots_adjust(wspace=0.1, hspace=1, bottom=0.1, top=0.9)
plt.tight_layout()
plt.show()

```

```

def plot_free_time():
    stats_free_time_1 = []

```

```

stats_free_time_2 = []
stats_free_time_3 = []

for i in intensities:
    generator = Generator(i, QUEUE_SIZE, 0.5, 0.3, 0.2)
    queue_1 = generator.generate_queue(AVG_SOLUTION_1, AVG_SOLUTION_2,
AVG_SOLUTION_3)
    queue_2 = queue_1.copy()
    queue_3 = queue_1.copy()
    scheduler_1 = Scheduler(
        queue_1, QUEUE_SIZE, 'FIFO', tact_size
    )
    scheduler_2 = Scheduler(
        queue_2, QUEUE_SIZE, EDF, tact_size
    )
    scheduler_3 = Scheduler(
        queue_3, QUEUE_SIZE, RM, tact_size
    )
    stats_1 = scheduler_1.schedule
    stats_2 = scheduler_2.schedule
    stats_3 = scheduler_3.schedule
    stats_free_time_1.append(stats_1['free_time'])
    stats_free_time_2.append(stats_2['free_time'])
    stats_free_time_3.append(stats_3['free_time'])

fig = plt.figure(figsize=(11, 7))
fig.suptitle('Час простою від інтенсивності')
plot = fig.add_subplot(311)
plot.set_title('FIFO')
plot.set_xlabel('Інтенсивність')
plot.set_ylabel('Середній час простою')
plot.plot(intensities, stats_free_time_1)
plot = fig.add_subplot(312)
plot.set_title('EDF')
plot.set_xlabel('Інтенсивність')
plot.set_ylabel('Середній час простою')
plot.plot(intensities, stats_free_time_2)
plot = fig.add_subplot(313)
plot.set_title('RM')
plot.set_xlabel('Інтенсивність')
plot.set_ylabel('Середній час простою')
plot.plot(intensities, stats_free_time_3)
plt.subplots_adjust(wspace=0.1, hspace=1, bottom=0.1, top=0.9)
plt.tight_layout()

```

```
plt.show()
```

```
def plot_avg_queue_len():
    stats_avg_queue_length_1 = []
    stats_avg_queue_length_2 = []
    stats_avg_queue_length_3 = []

    for i in intensities:
        generator = Generator(i, QUEUE_SIZE, 0.5, 0.3, 0.2)
        queue_1 = generator.generate_queue(AVG_SOLUTION_1, AVG_SOLUTION_2,
        AVG_SOLUTION_3)
        queue_2 = queue_1.copy()
        queue_3 = queue_1.copy()
        scheduler_1 = Scheduler(
            queue_1, QUEUE_SIZE, 'FIFO', tact_size
        )
        scheduler_2 = Scheduler(
            queue_2, QUEUE_SIZE, EDF, tact_size
        )
        scheduler_3 = Scheduler(
            queue_3, QUEUE_SIZE, RM, tact_size
        )
        stats_1 = scheduler_1.schedule
        stats_2 = scheduler_2.schedule
        stats_3 = scheduler_3.schedule
        stats_avg_queue_length_1.append(stats_1['avg_queue_length'])
        stats_avg_queue_length_2.append(stats_2['avg_queue_length'])
        stats_avg_queue_length_3.append(stats_3['avg_queue_length'])

    fig = plt.figure(figsize=(11, 7))
    fig.suptitle('Довжина черги від інтенсивності')
    plot = fig.add_subplot(311)
    plot.set_title('FIFO')
    plot.set_xlabel('Інтенсивність')
    plot.set_ylabel('Середня довжин черги')
    plot.plot(intensities, stats_avg_queue_length_1)
    plot = fig.add_subplot(312)
    plot.set_title('EDF')
    plot.set_xlabel('Інтенсивність')
    plot.set_ylabel('Середня довжин черги')
    plot.plot(intensities, stats_avg_queue_length_2)
    plot = fig.add_subplot(313)
    plot.set_title('RM')
```

```

plot.set_xlabel('Інтенсивність')
plot.set_ylabel('Середня довжин черги')
plot.plot(intensities, stats_avg_queue_length_3)
plt.subplots_adjust(wspace=0.1, hspace=1, bottom=0.1, top=0.9)
plt.tight_layout()
plt.show()

```

```

def plot_discarded():
    stats_discarded_1 = []
    stats_discarded_2 = []
    stats_discarded_3 = []

    for i in intensities:
        generator = Generator(i, QUEUE_SIZE, 0.5, 0.3, 0.2)
        queue_1 = generator.generate_queue(AVG_SOLUTION_1, AVG_SOLUTION_2,
        AVG_SOLUTION_3)
        queue_2 = queue_1.copy()
        queue_3 = queue_1.copy()
        scheduler_1 = Scheduler(
            queue_1, QUEUE_SIZE, 'FIFO', tact_size
        )
        scheduler_2 = Scheduler(
            queue_2, QUEUE_SIZE, EDF, tact_size
        )
        scheduler_3 = Scheduler(
            queue_3, QUEUE_SIZE, RM, tact_size
        )
        stats_1 = scheduler_1.schedule
        stats_2 = scheduler_2.schedule
        stats_3 = scheduler_3.schedule
        stats_discarded_1.append(100 * stats_1['discarded'] / (stats_1['complete'] +
        stats_1['discarded']))
        stats_discarded_2.append(100 * stats_2['discarded'] / (stats_2['complete'] +
        stats_2['discarded']))
        stats_discarded_3.append(100 * stats_3['discarded'] / (stats_3['complete'] +
        stats_3['discarded']))

    fig = plt.figure(figsize=(11, 7))
    fig.suptitle('Кількість відкинутих задач від інтенсивності(у відсотках)')
    plot = fig.add_subplot(311)
    plot.set_title('FIFO')
    plot.set_xlabel('Інтенсивність')
    plot.set_ylabel('Кількість відкинутих задач')

```

```

plot.plot(intensities, stats_discarded_1)
plot = fig.add_subplot(312)
plot.set_title('EDF')
plot.set_xlabel('Інтенсивність')
plot.set_ylabel('Кількість відкинутих задач')
plot.plot(intensities, stats_discarded_2)
plot = fig.add_subplot(313)
plot.set_title('RM')
plot.set_xlabel('Інтенсивність')
plot.set_ylabel('Кількість відкинутих задач')
plot.plot(intensities, stats_discarded_3)
plt.subplots_adjust(wspace=0.1, hspace=1, bottom=0.1, top=0.9)
plt.tight_layout()
plt.show()

```

```

def plot_rm_reverse_intensity():
    stats_discarded_1 = []
    stats_discarded_2 = []

    for i in intensities:
        generator_1 = Generator(i, QUEUE_SIZE, 0.5, 0.3, 0.2)
        generator_2 = Generator(i, QUEUE_SIZE, 0.2, 0.3, 0.5)
        queue_1 = generator_1.generate_queue(AVG_SOLUTION_1, AVG_SOLUTION_2,
        AVG_SOLUTION_3)
        queue_2 = generator_2.generate_queue(AVG_SOLUTION_1, AVG_SOLUTION_2,
        AVG_SOLUTION_3)
        scheduler_1 = Scheduler(
            queue_1, QUEUE_SIZE, RM, tact_size
        )
        scheduler_2 = Scheduler(
            queue_2, QUEUE_SIZE, RM, tact_size
        )
        stats_1 = scheduler_1.schedule
        stats_2 = scheduler_2.schedule
        stats_discarded_1.append(100 * stats_1['discarded'] / (stats_1['complete'] +
        stats_1['discarded']))
        stats_discarded_2.append(100 * stats_2['discarded'] / (stats_2['complete'] +
        stats_2['discarded']))

    fig = plt.figure(figsize=(10, 7))
    fig.suptitle('Кількість відкинутих задач від інтенсивності(у відсотках)')
    plot = fig.add_subplot(211)
    plot.set_title('0.5, 0.3, 0.2')

```



```

plot.set_xlabel('Інтенсивність')
plot.set_ylabel('Кількість відкинутих задач')
plot.plot(intensities, stats_discarded_1)
plot = fig.add_subplot(212)
plot.set_title('0.2, 0.3, 0.5')
plot.set_xlabel('Інтенсивність')
plot.set_ylabel('Кількість відкинутих задач')
plot.plot(intensities, stats_discarded_2)
plt.subplots_adjust(wspace=0.1, hspace=1, bottom=0.1, top=0.9)
plt.tight_layout()
plt.show()

```

SMO.py

```
class Scheduler:
```

```
    def __init__(self, queue, queue_size, algorithm, tact_size):
```

```
        self.queue = queue
```

```
        self.queue_size = queue_size
```

```
        if algorithm != 'FIFO':
```

```
            self.algorithm = algorithm
```

```
        else:
```

```
            self.algorithm = 'FIFO'
```

```
        self.tact_size = tact_size
```

```
        self.stats = {
```

```
            'avg_wait_time': 0,
```

```
            'complete': 0,
```

```
            'free_time': 0,
```

```
            'avg_queue_length': 0,
```

```
            'queue_length': [],
```

```
            'discarded': 0
```

```
        }
```

```
        self.rt_queue = []
```

```
        self.sys_clock = 0
```

```
        self.remainder_tact_time = 0
```

```
        self.processing = 0
```

```
@property
```

```
def schedule(self):
```

```
    while True:
```

```
        if self.processing == 0:
```

```
            all_tasks = len(self.rt_queue)
```

```
            # discard tasks with expired deadline
```

```
            self.rt_queue = [task for task in self.rt_queue if task.deadline > self.sys_clock]
```

```

        filtered = len(self.rt_queue)
        self.stats['discarded'] += all_tasks - filtered
    if len(self.rt_queue) == 0 and len(self.queue) == 0:
        # if all tasks complete break the loop
        break
    if len(self.queue) != 0 and self.remainer_tact_time == 0:
        # a new task comes if there are any in the queue and its a start of the tact!
        self.rt_queue += self.queue.pop(0)
        # recording queue length
        self.stats['queue_length'].append(len(self.rt_queue))
    if self.processing == 0:
        # if no task is processed sort tasks in queue according to algorithm
        if self.algorithm != 'FIFO':
            self.rt_queue = self.algorithm.sort_tasks(self.rt_queue)
    if len(self.rt_queue) != 0:
        if self.remainer_tact_time != 0:
            if self.rt_queue[0].wcet > self.remainer_tact_time:
                # if we can't do a task in one tact
                self.sys_clock += self.remainer_tact_time
                self.rt_queue[0].wcet -= self.remainer_tact_time
                self.processing = 1
                self.remainer_tact_time = 0
            else:
                # if we can do a task in one tact
                self.remainer_tact_time -= self.rt_queue[0].wcet
                self.sys_clock += self.rt_queue[0].wcet
                self.stats['avg_wait_time'] += self.sys_clock - self.rt_queue[0].start -
self.rt_queue[0].wcet
                # print('SysClock {} Task start time {} Task solution time
{}'.format(self.sys_clock,
                #
                self.rt_queue[0].start,
                #
                self.rt_queue[0].wcet))
                self.stats['complete'] += 1
                self.processing = 0
                del self.rt_queue[0]
        else:
            # if we have no time in this tact left, proceed to the next tact
            if self.rt_queue[0].wcet > self.tact_size:
                # if solution time is too long for next tact
                self.rt_queue[0].wcet -= self.tact_size
                self.processing = 1
                self.remainer_tact_time = 0
                self.sys_clock += self.tact_size
            else:

```

```

        # if task can be complete
        self.sys_clock += self.rt_queue[0].wcet
        self.stats['avg_wait_time'] += self.sys_clock - self.rt_queue[0].start -
self.rt_queue[0].wcet
        self.stats['complete'] += 1
        self.remainder_tact_time = self.tact_size - self.rt_queue[0].wcet
        self.processing = 0
        del self.rt_queue[0]
    else:
        if self.remainder_tact_time != 0:
            self.stats['free_time'] += self.remainder_tact_time
            self.sys_clock += self.remainder_tact_time
            self.remainder_tact_time = 0
        else:
            self.sys_clock += self.tact_size
            self.stats['free_time'] += self.tact_size

    print(self.stats)

    self.stats['avg_wait_time'] = (self.stats['avg_wait_time'] / self.stats['complete'])
    self.stats['avg_queue_length'] = sum(self.stats['queue_length']) / self.queue_size

    return self.stats

```

Task.py

```

class Task:
    def __init__(self, start, wcet, deadline, intensity):
        self.start = start
        self.wcet = wcet
        self.deadline = deadline
        self.intensity = intensity

```

EarliestDeadlineFirst.py

```

class EDF:
    @staticmethod
    def sort_tasks(queue):

```

```

queue.sort(key=lambda el: el.deadline)
return queue

```

RateMonotonic.py

```

class RM:
    @staticmethod
    def sort_tasks(queue):
        queue.sort(key=lambda el: el.intensity, reverse=True)
        return queue

```

PoissonGenerator.py

```

import numpy as np
import random as r
from Task import Task

```

```

class Generator:
    def __init__(self, intensity, tact_number, coef_1, coef_2, coef_3):
        self.poisson_1 = np.random.poisson(intensity * coef_1, tact_number)
        self.poisson_2 = np.random.poisson(intensity * coef_2, tact_number)
        self.poisson_3 = np.random.poisson(intensity * coef_3, tact_number)
        self.coef_1 = coef_1
        self.coef_2 = coef_2
        self.coef_3 = coef_3
        self.intensity = intensity
        self.queue_size = tact_number

    def generate_queue(self, solution_time_1, solution_time_2, solution_time_3):
        queue = []
        queue_1 = self.iterate([], solution_time_1, self.poisson_1, self.intensity * self.coef_1)
        queue_2 = self.iterate([], solution_time_2, self.poisson_2, self.intensity * self.coef_2)
        queue_3 = self.iterate([], solution_time_3, self.poisson_3, self.intensity * self.coef_3)
        for i in range(len(queue_1)):
            queue.append(queue_1[i] + queue_2[i] + queue_3[i])
            r.shuffle(queue[i])
        return queue

    def iterate(self, queue, solution_time, poisson, intensity):
        for i in range(len(poisson)):
            tact = []
            for j in range(poisson[i]):
                margin = r.uniform(solution_time * 0.2, solution_time * 0.4) / 2
                if r.random() < 0.5:

```

```
        tact.append(  
            Task(i, solution_time + margin, i + (solution_time + margin) + r.randint(4, 10),  
intensity)  
        )  
    else:  
        tact.append(  
            Task(i, solution_time - margin, i + (solution_time - margin) + r.randint(4, 10),  
intensity)  
        )  
    queue.append(tact)  
    return queue
```