Національний технічний університет України
«Київський політехнічний інститут ім. Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

# РОЗРАХУНКОВО-ГРАФІЧНА РОБОТА

з дисципліни «Системи реального часу»

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

Студентки 3 курсу
групи ІП-83
Гомілко Д.В

Перевірив: Волокита А.М.

Київ – 2021

**ЗМІСТ**

# РОЗДІЛ 1. ТЕОРЕТИЧНА ЧАСТИНА

## 1.1 Планування виконання завдань

Протягом існування процесу виконання його потоків може бути багаторазово перерване і продовжене. Перехід від виконаннання одного потоку до іншого здійснюється в результаті планування і диспетчеризації. Робота по визначенню того, в який момент необхідно перервати виконання поточного активного потоку і якому потоку дати можливість виконуватися, називається плануванням[1].

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 нитi[2]

## 1.2 Системи масового обслуговування

Система масового обслуговування (СМО) - це система, яка обслуговує вимоги, що надходять до неї (заявки). Основними елементами системи є вхідний потік вимог, канали обслуговування, черга вимог та вихідний потік вимог.

Вимоги (заявки) на обслуговування надходять через дискретні (постійні або випадкові) інтервали часу. Важливо знати закон розподілу вхідного потоку. Обслуговування триває деякий час, постійний або випадковий. Випадковий характер потоку заявок та часу обслуговування призводить до того, що в деякі моменти часу на вході СМО може виникнути черга, в інші моменти – канали можуть бути недозавантаженими або взагалі простоювати.

Задача теорії масового обслуговування полягає в побудові моделей, які пов‘язують задані умови роботи СМО з показниками ефективності системи, що описують її спроможність впоратися з потоком вимог. Під ефективністю обслуговуючої системи розуміють характеристику рівня виконання цією системою функцій, для яких вона призначена[3].

В залежності від наявності можливості очікування вступниками вимогами початку обслуговування СМО поділяються на:

- системи з втратами, в яких вимоги, що не знайшли в момент надходження жодного вільного приладу, втрачаються;
- системи з очікуванням, в яких є накопичувач нескінченної ємності для буферизації надійшли вимог, при цьому очікують вимоги утворюють чергу;
- 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].

### 1.3 Вхідний потік заявок

Потоком подій називається послідовність однорідних подій, наступних одне за іншим і що відбуваються у випадкові моменти часу. Для опису потоку заявок в загальному випадку необхідно задати інтервали часу між сусідніми моментами надходження заявок.

Основною характеристикою потоку заявок є його інтенсивність - середнє число заявок, що надходять на вхід СМО за одиницю часу. Величина 1/X визначає середній інтервал часу між двома послідовними заявками.

Потік називається детермінованим, якщо інтервали часу між сусідніми заявками приймають певні заздалегідь відомі значення. Якщо при цьому інтервали однакові, то потік називається регулярним.

Потік, в якому інтервали часу між сусідніми заявками є випадковими величинами, називається випадковим. Для повного опису випадкового потоку заявок в загальному випадку необхідно задати закони розподілів кожного з інтервалів часу[5].

Пуассонівський процес — це поняття теорії випадкових процесів, що моделює кількість випадкових подій, що стались, якщо тільки вони відбуваються зі сталим середнім значенням інтервалів між їхніми настаннями.

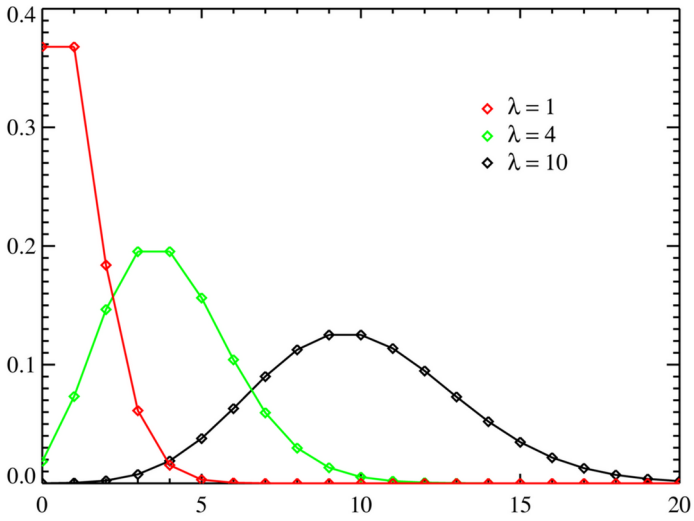


*Рисунок 1: Розподіл Пуассона для різних значень λ*

У випадку вибраних одиниць вимірювання, це середнє значення дорівнює 1/λ кількостей подій за одиницю часу, де λ — параметр процесу. Цей параметр часто називають інтенсивністю пуассонівського процесу[6].

## 1.4 Алгоритми планування процесів

Найчастіше зустрічаються такі дві групи алгоритмів планування: побудовані на принципі квантування або на принципі пріоритетів.

В першому випадку зміна активного процесу відбувається, якщо:

- процес закінчився і покинув систему;
- процес перейшов в стан Очікування;
- закінчився квант процесорного часу, відведений даному процесові.

Процес, для якого закінчився його квант, переводиться в стан готовності і очікує, коли йому буде надано новий квант процесорного часу, а на виконання у відповідності з певним правилом вибирається новий процес з черги готових. Жодний процес не захоплює процесор надовго, тому квантування широко використовується в системах розподілу часу. По-різному може бути організована черга готових процесів:

- циклічно;
- FIFO (перший прийшов — перший обслуговується);
- LIFO (останній прийшов — перший обслуговується ).

В другому випадку використовується поняття ”пріоритет”. Пріоритет — це число, яке характеризує ступінь привілейованості процесу при використанні ресурсів комп’ютеру, зокрема, процесорного часу. Чим вище пріоритет, тим вище привілеї, тим менше часу він буде проводити в чергах.

Пріоритетами можуть призначатьсь адміністратором системи в залежності від важливості роботи, або внесеної плати, або обчислюватись самою ОС за певними правилами. Він може залишатись фіксованим на протязі всього життя процесу або мінятись в часі у відповідності з деяким законом. В останньому випадку пріоритети називають динамічними.

Є алгоритми, які використовують:

- відносні пріоритети;
- абсолютні пріоритети[7].

### 1.5 First-Come, First-Served

Простим алгоритмом планування є алгоритм, який прийнято позначати абревіатурою FCFS по перших буквах його англійської назви — First Come, First Served (першим прийшов, першим обслужений). Уявимо собі, що процеси,що знаходяться в стані готовність, організовані в чергу. Коли процес переходить в стан готовність, він, а точніше посилання на його PCB, поміщається в кінець цієї черги. Вибір нового процесу для виконання здійснюється з початку черги з видаленням звідти посилання на його PCB. Черга подібного типу має в програмуванні спеціальне найменування FIFO — скорочення від First In, First Out (першим ввійшов, першим вийшов).

Треба відзначити, що абревіатура FCFS використовується для цього алгоритму планування замість стандартної абревіатури FIFO для механізмів подібного типу для того, щоб підкреслити, що організація готових процесів в чергу FIFO можлива і при інших алгоритмах планування (наприклад, для Round Robin).

Такий алгоритм вибору процесу здійснює планування що не витісняє. Процес, що одержав в своє розпорядження процесор, займає його до закінчення свого поточного CPU burst. Після цього для виконання вибирається новий процес з початку черги.

Перевагою алгоритму FCFS є легкість його реалізації, в той же час він має і багато недоліків. Зміст алгоритму полягає у тому, що середній час очікування і середній повний час виконання для цього алгоритму істотно залежать від порядку розташування процесів в черзі. Якщо у нас є процес з тривалим CPU burst, то короткі процеси, що перейшли в стан готовність після тривалого процесу, дуже довго чекатимуть початку свого виконання. Тому алгоритм FCFS практично непридатний для систем розподілення часу. Дуже великим виходить середній час відгуку в інтерактивних процесах[8].

## 1.6 Earliest Deadline First

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.

Алгоритму планування EDF не потрібно, щоб завдання або процеси були періодичними, а також завдання або процеси вимагають фіксованого часу виконання обробки. У EDF будь-яке виконуване завдання може бути витіснене, якщо інша задача із меншим терміном виконання стане активною[9].

## 1.7 Rate Monotonic

Rate Monotonic - це алгоритм призначення пріоритетів, який використовується в операційних системах реального часу (RTOS) із 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, мають детерміновані гарантії щодо часу реагування.

Підтримувати аперіодичні та спорадичні завдання в рамках RMA дуже важко. Крім того, RMA не є оптимальним, коли дедлайн і термін виконання завдань різняться[10].

# РОЗДІЛ 2. РОЗРОБКА ПЛАНУВАЛЬНИКА

## 2.1 Вимоги до системи

Вхідними заявками є обчислення, які проводилися в лабораторних роботах 1-7. Вхідні заявки характеризуються наступними параметрами:

1) час приходу в систему – потік заявок є потоком Пуассона або потоком Ерланга k-го порядку;

2) час виконання (обробки) – математичним очікуванням часу виконання є середнє значення часу виконання відповідних обчислень в попередніх лабораторних роботах;

3) крайній строк завершення (дедлайн) – задається за правилом d = a + k*wcet, де a — час приходу в систему, wcet — час виконання, k — випадкове число від 10 до 20; якщо заявка залишається необробленою в момент часу t = Td, то її обробка припиняється і вона покидає систему.

Вибір заявки з черги на обслуговування здійснюється за допомогою однієї з трьох дисциплін: FIFO, RM та EDF.

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.

## 2.2 Ключові особливості реалізації

Вхідними параметрами, необхідними для роботи планувальника, є границя роботи планувальника в часі, інтенсивність, розмір такту та розмір проміжку, в якому виникатимуть задачі. Для вхідного потоку заявок було обрано потік Пуасона. Допоміжна функція для генерації випадкових чисел за пуасонівським розподілом приймає середнє значення інтенсивності, на основі якого розраховуються конкретні значення для кожного з 7 типів задач. Таким чином, під час генерації заявок кожного типу для планувальника за допомогою вищеописаної функції отримується випадкове число, яке позначає кількість заявок, що надходять у поточному інтервалі часу, звідки обраховується період. Таким чином, завявки, для яких задана більша інтенсивність, виникатимуть з меншим періодом і навпаки.

Для представлення задачі, що приходить на вхід планувальника, було створено клас, що містить 4 поля: час надходження задачі, час її виконання, період та дедлайн, який розраховується на основі перших 2 значень. З вірогідністю 5% задача може бути спорадичною, тобто мати малий відрізок часу до дедлайну (коефіцієнт k від 1 до 3).

```
class Task:
arrivalTime: float
wcet: float
period: float
deadline: float = field(init=False)
```

Для моделювання обробленої задачі було розроблено клас, який містить поля з часом прибуття заявки в систему, інтервалами часу, в який відбувалася обробка задачі, часом початку обробки, часом завершення обробки та часом очікування. Час очікування обчислюється на основі різниці між часами прибуття задачі та початку її обробки, а також проміжками часу, коли задача перебувала в стані очікування.

```
class ProcessedTask:
arrival: float
timestamps: list[float]
```

```
deadlineMissed: bool
start: float = field(init=False)
end: float = field(init=False)

waitingTime: float = field(init=False)
```

Результати роботи кожного алгоритму планування зберігаються в класі SchedulingResult, який містить список оброблених задач, час простою системи та його відсоток, повний час роботи, кількість та відсоток просрочених задач та дані для діаграми Ганта на основі інтервалів часу виконання кожної заявки.

```
class SchedulingResult:

result: list[ProcessedTask]
idleTime: float
totalTime: float
idlePercent: float = field(init=False)
avgWait: float = field(init=False)
missedTasksCount: int = field(init=False)
missedTasksPercent: float = field(init=False)

ganttChartData: list[list[tuple()]] = field(init=False)
```

Реалізація FIFO полягає в наступному: задачі, що приходять на вхід системи, сортуються за часом прибуття, і виконуються строго в цьому порядку без витіснення. EDF та RM базуються на одній загальній реалізації динамічного планувальника з можливістю витіснення BaseDynamicScheduler. Якщо під час обробки певної задачі з’явиться задача з вищим пріоритетом, то поточна задача відкидається й замінюється наступною. Робота планувальника є потактовою, при цьому розмір такта задається у вхідних даних планувальника. Конкретні функції RM та EDF отримуються за допомогою прив’язки до аргументів функції відповідного значення функції сортування. Для RM відбувається сортування за періодом (якщо період однаковий — за дедлайном), для EDF — за дедлайном. Повний код програми наведений у Додатку.

```
def estimatePriorityEDF(tasks):
return tasks.sort(key=lambda x: (x.deadline, x.wcet))

def estimatePriorityRM(tasks):
return tasks.sort(key=lambda x: (x.period, x.wcet, x.deadline))

EDF = partial(BaseDynamicScheduler, estimatePriorityEDF)
RM = partial(BaseDynamicScheduler, estimatePriorityRM)
```

# РОЗДІЛ 3. РЕЗУЛЬТАТИ РОБОТИ

## 3.1 Аналіз отриманих результатів

Нище неведені результати роботи для одного і того ж набору задач із інтенсивністю 7. Результат для FIFO:

```
Scheduling using FIFO:
        Average waiting time: 1.2935
        Idle time percent: 56.0879%(total: 11.2552, idle: 6.3128)
        Missed tasks count: 1 (10.0% out of total 10)

Task 1: arrived: 5.5556, waited: 0.0, started processing: 5.5556, finish: 5.619;
Task 2: arrived: 5.5556, waited: 0.1268, started processing: 5.619, finish: 6.7842;
Task 3: arrived: 6.25, waited: 1.0684, started processing: 6.7842, finish: 7.0872;
Task 4: arrived: 6.25, waited: 1.6744, started processing: 7.0872, finish: 7.5127;
Task 5: arrived: 7.1428, waited: 0.7398, started processing: 7.5127, finish: 7.5762;
Task 6: arrived: 8.3334, waited: 0.0, started processing: 8.3334, finish: 9.3615;
Task 7: arrived: 8.3334, waited: 2.0562, started processing: 9.3615, finish: 9.3615; DEADLINE MISSED
Task 8: arrived: 8.3334, waited: 2.0562, started processing: 9.3615, finish: 10.5267;
Task 9: arrived: 9.375, waited: 2.3034, started processing: 10.5267, finish: 10.8297;
Task 10: arrived: 9.375, waited: 2.9094, started processing: 10.8297, finish: 11.2552;
```

Результат для RM:

```
Scheduling using RM:
        Average waiting time: 0.9004
        Idle time percent: 55.7739%(total: 11.3186, idle: 6.3128)
        Missed tasks count: 0 (0.0% out of total 10)

Task 1: arrived: 5.5556, waited: 0.0, started processing: 5.5556, finish: 5.619;
Task 2: arrived: 5.5556, waited: 0.1268, started processing: 5.619, finish: 6.7842;
Task 3: arrived: 6.25, waited: 1.0684, started processing: 6.7842, finish: 7.0872;
Task 4: arrived: 6.25, waited: 1.6744, started processing: 7.0872, finish: 7.5127;
Task 5: arrived: 7.1428, waited: 0.7398, started processing: 7.5127, finish: 7.5762;
Task 6: arrived: 8.3334, waited: 0.0, started processing: 8.3334, finish: 8.3968;
Task 7: arrived: 8.3334, waited: 0.1268, started processing: 8.3968, finish: 9.562;
Task 8: arrived: 9.375, waited: 0.374, started processing: 9.562, finish: 9.865;
Task 9: arrived: 9.375, waited: 0.98, started processing: 9.865, finish: 10.2905;
Task 10: arrived: 8.3334, waited: 3.9142, started processing: 10.2905, finish: 11.3186;
```

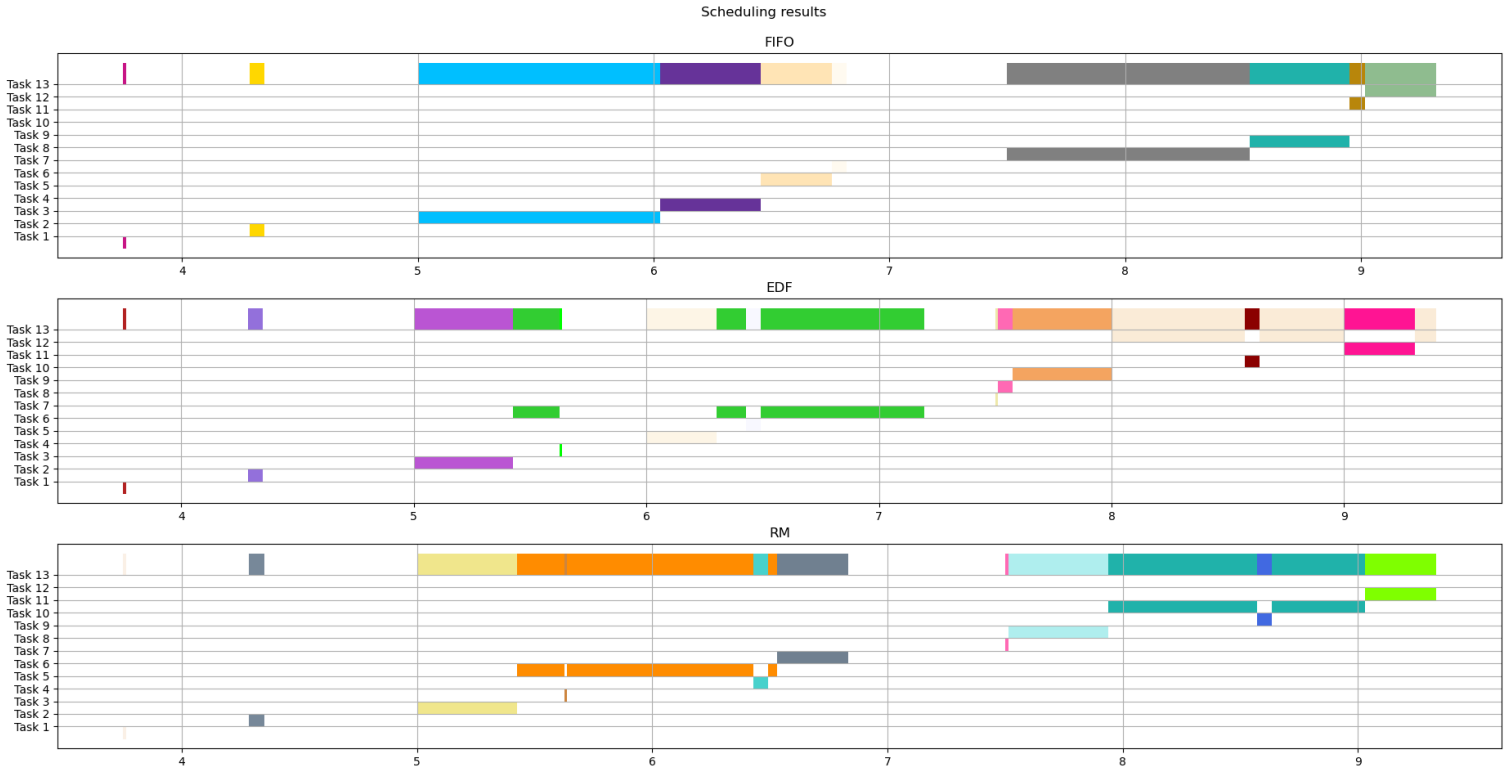
Результат для EDF:

```
Scheduling using EDF:
        Average waiting time: 0.6871
        Idle time percent: 55.7739%(total: 11.3186, idle: 6.3128)
        Missed tasks count: 0 (0.0% out of total 10)

Task 1: arrived: 5.5556, waited: 0.0, started processing: 5.5556, finish: 5.619;
Task 2: arrived: 6.25, waited: 0.0, started processing: 6.25, finish: 6.553;
Task 3: arrived: 6.25, waited: 0.606, started processing: 6.553, finish: 6.9785;
Task 4: arrived: 7.1428, waited: 0.0, started processing: 7.1428, finish: 7.2063;
Task 5: arrived: 5.5556, waited: 0.9188, started processing: 5.619, finish: 7.5762;
Task 6: arrived: 8.3334, waited: 0.0, started processing: 8.3334, finish: 8.3968;
Task 7: arrived: 9.375, waited: 0.0, started processing: 9.375, finish: 9.8005;
Task 8: arrived: 9.375, waited: 0.851, started processing: 9.8005, finish: 10.1035;
Task 9: arrived: 8.3334, waited: 0.8553, started processing: 8.3968, finish: 10.1534;
Task 10: arrived: 8.3334, waited: 3.6399, started processing: 10.1534, finish: 11.3186;
```

Як бачимо, навіть для невеликої кількості задач з незначною інтенсивністю використання FIFO призводить до прострочення дедлайнів. У той же час, EDF та RM виконали всі задачі вчасно з однаковим відсотком простою системи. Варто також зазначити, що середній час очікування для Rate Monotonic виявився вищим, ніж у Earliest Deadline First.

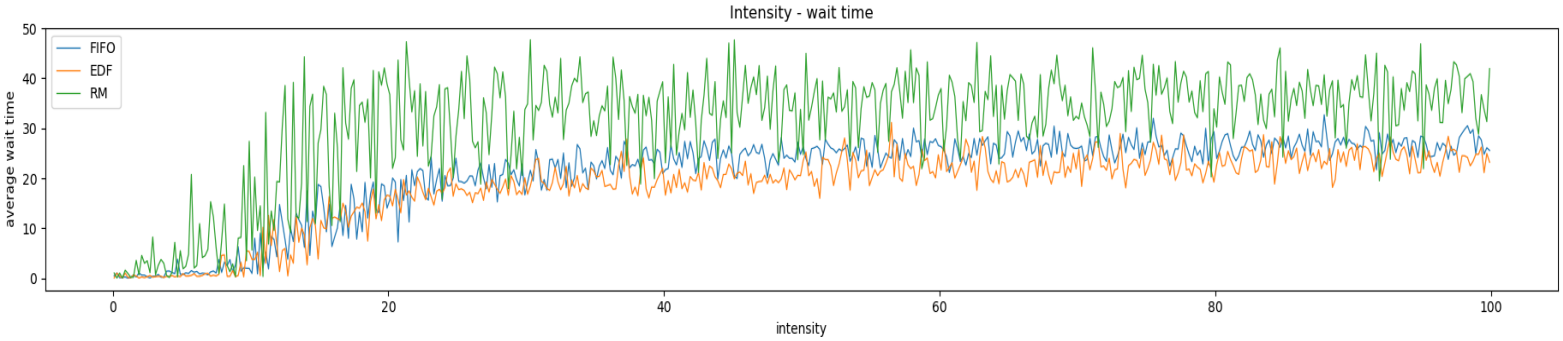
Приклад діаграм Ганта для результатів роботи планувальника:




Як видно з результатів, через відсутність пріоритетів та витіснення FIFO допускає прострочення дедлайнів, коли певні задачі не встигають навіть почати виконання. Це демонструє перевагу динамічних алгоритмів планування.

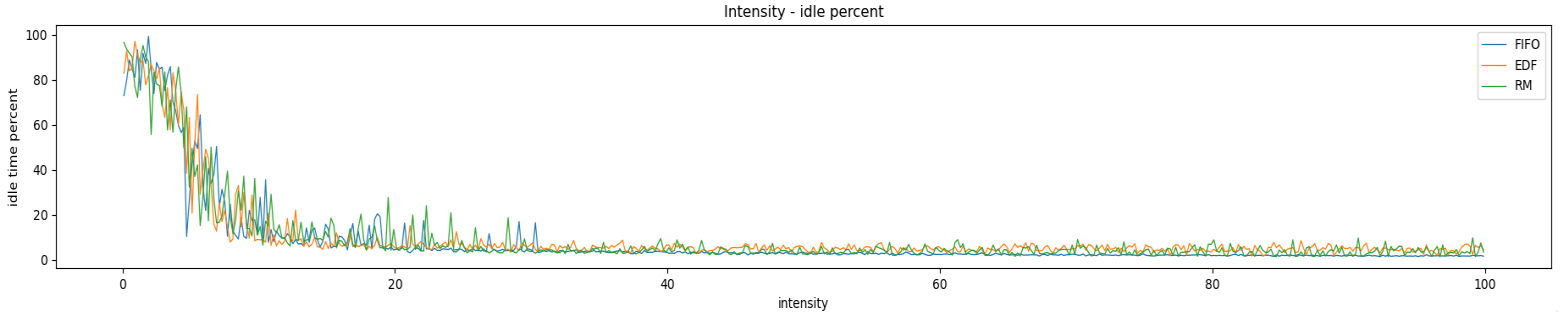
### 3.2 Аналіз отриманих графіків

Залежність середнього часу очікування заявок від інтенсивності:



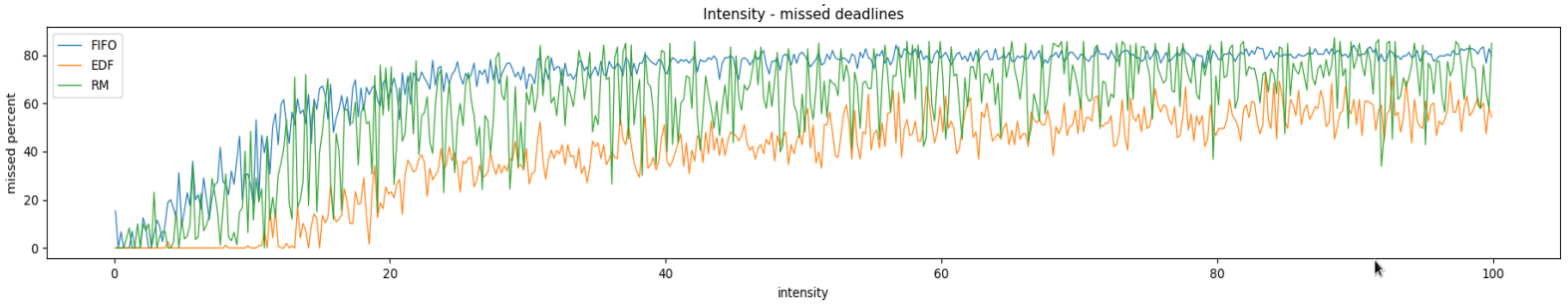

Як бачимо, при збільшенні інтенсивності середній час очікування спершу різко зростає, а згодом лишається на відносно сталому рівні, коли система є завантаженою і певна частина заявок постійно відкидається. Найменший середній час очікування заявок виявився у EDF. У той же час середній час очікування RM є навіть більшим, ніж у FIFO. Це пояснюється тим, що дана дисципліна не є оптимальною, коли для системи характерні спорадичні задачі, а також коли період задачі та її дедлайн відрізняються.

Залежність відсотку простою системи від інтенсивності:



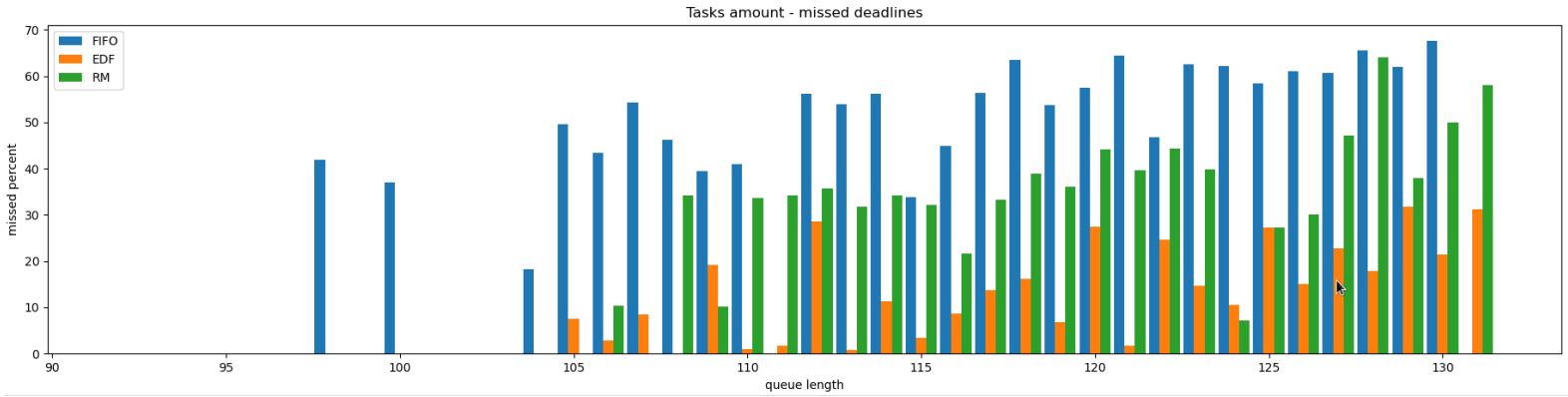

З наведеного графіка видно, що відсоток простою системи стрімко спадає і наближається до 0 зі зростанням інтенсивності незалежно від обраного алгоритму планування. Для обраних в конкретному випадку параметрів системи такою точкою насиченості є інтенсивність 10. Але навіть при значних показниках інтенсивності EDF та RM лишаються менш завантаженими, ніж FIFO.

Залежність відсотку пропущених дедлайнів від інтенсивності:




Як видно з графіка, найбільший відсоток пропущених дедлайнів має алгоритм FIFO. Алгоритм RM має певний відсоток пропущених заявок навять при незначній інтенсивності. Це пояснюється тим, що у вхідному потоці є спорадичні заявки, з якими даний алгоритм не може справитися вчасно. При збільшенні інтенсивності частота відкидання заявок та їх кількість збільшуються, але лишаються меншими, ніж у FIFO. Нарешті, дисципліна планування EDF має найменший відсоток пропущених дедлайнів, так як при появі спорадичної задачі з малим значенням дедлайну вона отримає вищий пріоритет і витіснить поточну задачу.

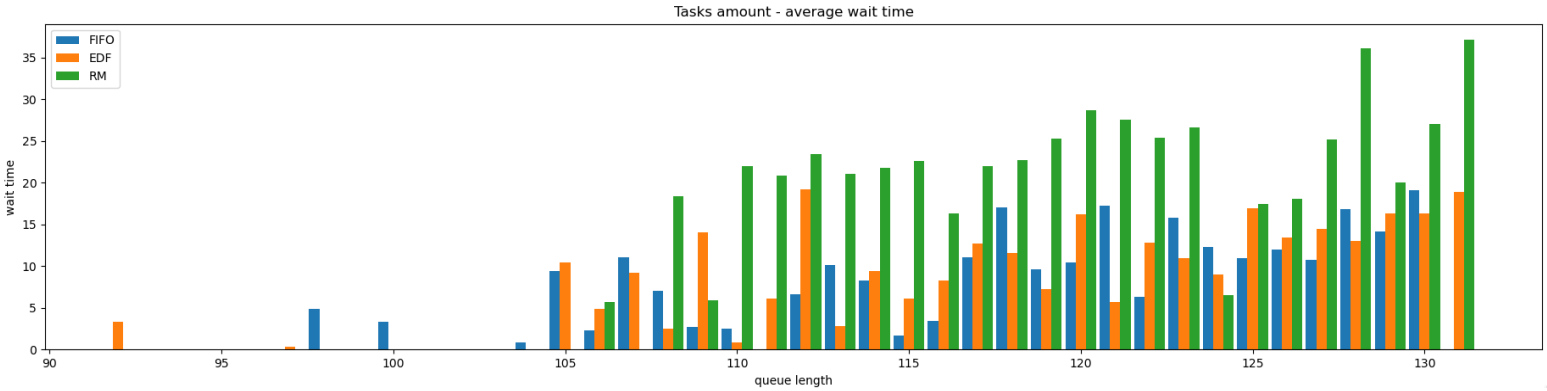
Залежність відсотку пропущених дедлайнів від кількості заявок:




При збільшенні кількості заявок, що надходять до системи, відсоток пропущених дедлайнів зростає. При цьому найбільш стрімко це відбувається для  FIFO. Відмови при використанні RM починаються раніше, ніж з EDF, і їх частота зростає швидше. У той же час, відсоток відмов для EDF з великою

кількістю заявок менший, і постійні відмови починаються лише з довжини вхідного потоку, більшого за 110.

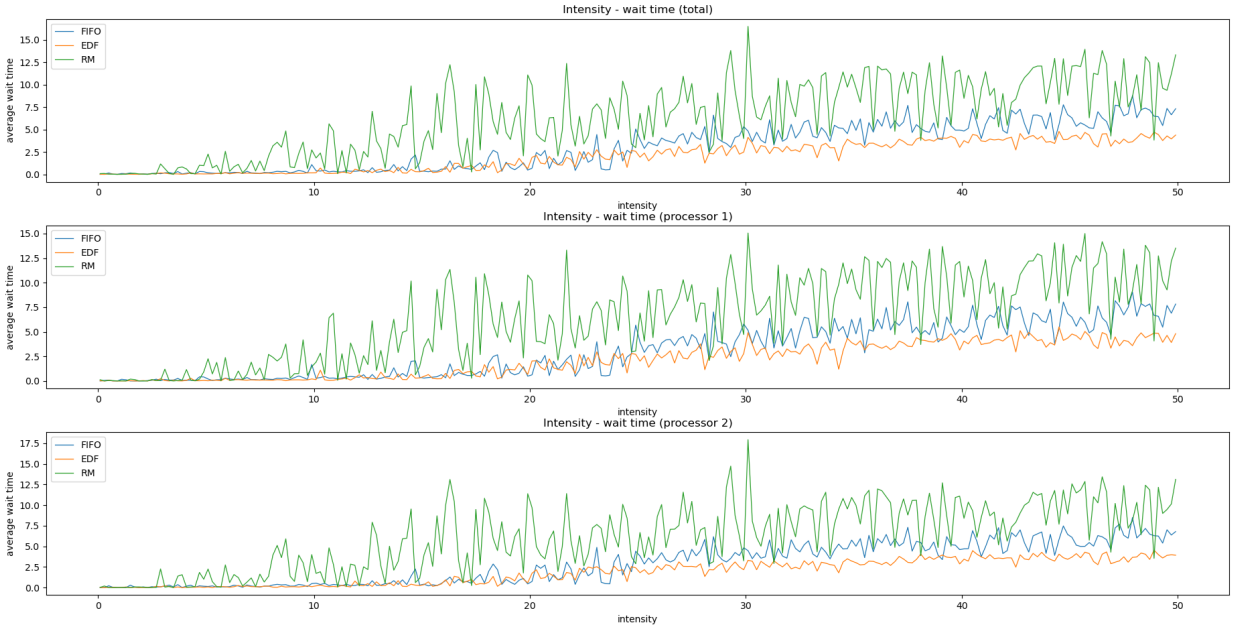
Залежність середнього часу очікування від кількості заявок:

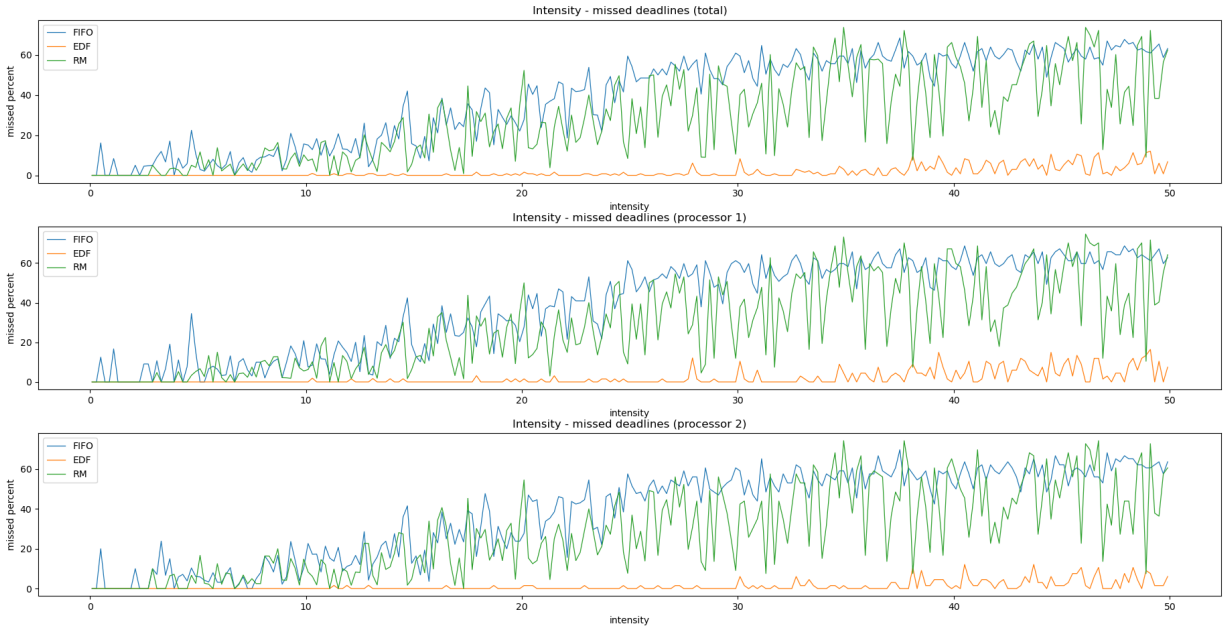



Даний графік підтверджує наведене раніше твердження про те, що середній найбільший час очікування має RM, а найменший – EDF. Як видно, це справджується не лише для окремо взятой кількості задач, а й для проміжку від 90 до 130.

## Додаткове завдання

Графіки роботи при наявності 2 процесорів:






## 3.3 Висновки

Під час виконання розрахунково-графічної роботи ми ознайомилися з теорією планування виконання завдань та систем масового обслуговування. У результаті роботи було створено планувальник з 3 доступними дисциплінами: First-In First-Out, Rate Monotonic та Earliest Deadline First.  Вхідні заявки генеруються за допомогою потоку Пуасона, засновуючись на даних 7 попередніх лабораторних робіт. Результатами роботи програми є консольний вивід з данимим про роботу певного алгоритму планування, графіки, що дозволяють порівняти особливості роботи дисциплін планування, та діаграми Ганта для результатів планування.

На основі отриманих результатів видно, що дисципліна планування FIFO має найбільшу кількість задач з пропущеними дедлайнами. Перевагою EDF над RM є краща стійкість до спорадичних задач. Перевагою RM над EDF є краща пристосованість до періодичних задач. Таким чином, в системах реального часу доцільніше імплементувати дисципліни динамічного планування EDF чи RM за FIFO. Для розробленої системи найкращим варіантом як за середнім часом очікування, так і за кількістю пропущених дедлайнів виявився EDF.

## ВИКОРИСТАНІ ДЖЕРЕЛА


1. Планувальник операційної системи [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%9F%D0%BB%D0%B0%D0%BD%D1%83%D0%B2%D0%B0%D0%BB%D1%8C%D0%BD%D0%B8%D0%BA_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D1%96%D0%B9%D0%BD%D0%BE%D1%97_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B8.

2. Зайцев В. Г. Комп'ютерні системи реального часу [Електронний ресурс] / В. Г. Зайцев, Є. І. Цибаєв. – 2019. – Режим доступу до ресурсу: https://ela.kpi.ua/bitstream/123456789/29604/1/Kompyuterni_systemy_realnoho_chasu.pdf.

3. Основні поняття теорії масового обслуговування [Електронний ресурс]. – 2014. – Режим доступу до ресурсу: https://er.nau.edu.ua/bitstream/NAU/21000/23/%D0%9B%D0%B5%D0%BA%D1%86%D0%B8%D1%8F_CA_12.pdf.

4. Система масового обслуговування [Електронний ресурс]. – 2020. – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D0%BC%D0%B0%D1%81%D0%BE%D0%B2%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D1%81%D0%BB%D1%83%D0%B3%D0%BE%D0%B2%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F.

5. Остовні поняття систем масового обслуговування [Електронний ресурс] – Режим доступу до ресурсу: https://stud.com.ua/163945/informatika/osnovni_ponyattya_sistem_masovogo_obslugovuvannya.

6. Пуассонівський процес [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%9F%D1%83%D0%B0%D1%81%D1%81%D0%BE%D0%BD%D1%96%D0%B2%D1%81%D1%8C%D0%BA%D0%B8%D0%B9_%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81.

7. Алгоритми планування процесів [Електронний ресурс] – Режим доступу до ресурсу: https://studopedia.su/10_69185_algoritmi-planuvannya-protsesiv.html.

8. First-Come, First-Served (FCFS) [Електронний ресурс] – Режим доступу до ресурсу: https://wiki.cuspu.edu.ua/index.php/1._First-Come,_First-Served_(FCFS).

9. Earliest Deadline First (EDF) CPU scheduling algorithm [Електронний ресурс] – Режим доступу до ресурсу: https://www.geeksforgeeks.org/earliest-deadline-first-edf-cpu-scheduling-algorithm/.
10. Rate-monotonic scheduling [Електронний ресурс] – Режим доступу до ресурсу: https://www.geeksforgeeks.org/rate-monotonic-scheduling/.

# ДОДАТКИ

## consts.py

```
SIGNAL_GENERATOR = 0.303

CORRELATION = 0.0635
DFT = 1.02808
FFT = 0.42549
FERMAT_FACTORIZE = 0.01252
PERCEPTRON = 0.0634
GENETIC_ALGORITHM = 1.1652

possibleTasks = [
SIGNAL_GENERATOR, CORRELATION,
DFT, FFT, FERMAT_FACTORIZE,
PERCEPTRON, GENETIC_ALGORITHM
]

TACT_SIZE = 0.5
```

## queueGenerator.py

```
from math import exp

from random import uniform, triangular
from consts import possibleTasks
from task import Task

def poisson(lambdaVal):
exponent = exp(-lambdaVal)
randEl = uniform(0.0, 1.0)
temp = exponent
factorial = 1
pow = 1
generated = 0
while randEl > temp:
generated += 1
factorial = generated * factorial
pow = pow * lambdaVal
temp = temp + pow * exponent / factorial
return generated * 0.1

def getPeriods(meanLambda, interval):
upperBound = meanLambda * 1.5
lowerBound = meanLambda / 1.5
mean = 3 * meanLambda - lowerBound - upperBound
lams = [triangular(lowerBound, upperBound, mean) for _ in possibleTasks]
periods = [round(interval / (poisson(l) or uniform(0.01, 0.5)), 4) for l in lams]
return periods

def generateTasks(intensity, interval, limit = 20):
curTime = 0
timeLimit = limit * interval
queue = list()
tasks = dict()
periods = getPeriods(intensity, interval)
for i, task in enumerate(possibleTasks):
period = periods[i]
tasks[task] = [period, period]
while curTime < timeLimit:
```

```
for wcet, val in tasks.items():
period, lastArrival = val
arrival = lastArrival + period
if arrival > curTime: continue
task = Task(round(arrival, 4), wcet, period)
queue.append(task)
tasks[wcet][1] = arrival
curTime += interval
return queue
```

task.py

```
from dataclasses import dataclass, field
from random import uniform, randint

def getIntervals(time):
return [(time[i], time[i + 1] - time[i]) for i in range(0, len(time), 2)]

def getWaitTime(task):
time = task.timestamps
diff = time[0] - task.arrival
if len(time) > 2:
return diff + sum([time[i] - time[i - 1] for i in range(2, len(time), 2)])
else: return diff

@dataclass
class Task:
arrivalTime: float
wcet: float
period: float
deadline: float = field(init=False)

def __post_init__(self):
chance = randint(0, 100)
k = uniform(1.0, 3.0) if chance < 5 else uniform(10.0, 20.0)
self.deadline = round(self.arrivalTime + k * self.wcet, 4)

@dataclass
class ProcessedTask:
arrival: float
deadline: float
timestamps: list[float]
deadlineMissed: bool
start: float = field(init=False)
end: float = field(init=False)
waitingTime: float = field(init=False)
def __post_init__(self):
self.start = self.timestamps[0]
self.end = self.timestamps[-1]
self.waitingTime = round(getWaitTime(self), 4)

@dataclass
class SchedulingResult:
result: list[ProcessedTask]
idleTime: float
totalTime: float
idlePercent: float = field(init=False)
avgWait: float = field(init=False)
missedTasksCount: int = field(init=False)
```

```
missedTasksPercent: float = field(init=False)
ganttChartData: list[list[tuple()]] = field(init=False)

def __post_init__(self):
waitTimesSum = sum(map(lambda x: x.waitingTime, self.result))
missedTasks = list(filter(lambda x: x.deadlineMissed, self.result))
tasksNum = len(self.result)
self.avgWait = round(waitTimesSum / tasksNum, 4)
self.missedTasksCount = len(missedTasks)
self.missedTasksPercent = round(self.missedTasksCount / tasksNum * 100, 4)
self.idlePercent = round(self.idleTime / self.totalTime * 100, 4)
self.ganttChartData = [getIntervals(t.timestamps) for t in self.result]
```

## fifo.py

```
from task import ProcessedTask, SchedulingResult
from copy import deepcopy

def FIFO(allTasks):
tasks = deepcopy(allTasks)
tasks.sort(key=lambda x: x.arrivalTime)
processedTasks = list()
idleTime = 0
curTime = 0
for task in tasks:
arrival = task.arrivalTime
if curTime < arrival:
idleTime += arrival - curTime
curTime = arrival
startTime = curTime
missed = curTime + task.wcet > task.deadline
if not missed: curTime = curTime + task.wcet
task = ProcessedTask(arrival, task.deadline, [startTime, curTime], missed)
processedTasks.append(task)
return SchedulingResult(processedTasks, round(idleTime, 4), round(curTime, 4))
```

## dynamicSchedulers.py

```
from functools import partial
from copy import deepcopy
from task import ProcessedTask, SchedulingResult
from consts import TACT_SIZE

tasks, activeTasks = list(), list()

def taskID(task):
return f'{task.arrivalTime}{task.period}{task.deadline}'

def areTasksEqual(taskA, taskB):
return taskID(taskA) == taskID(taskB)

def checkForNewTasks(time):
global tasks, activeTasks
futureTasks = lambda x: x.arrivalTime > time
arrivedTasks = lambda x: x.arrivalTime <= time
arrived = list(filter(arrivedTasks, tasks))
if not arrived: return False
```

```
activeTasks += arrived
tasks = list(filter(futureTasks, tasks))
return True

def waitForTasks(curTime, interval, clb):
global activeTasks
maxWaitTime = curTime + interval
if not checkForNewTasks(maxWaitTime): return None
clb(activeTasks)
return activeTasks[0]

def checkMissedTasks(tasks, resTasks):
diff = list()
for task in tasks:
processed = False
for taskRes in resTasks:
if task.arrivalTime == taskRes.arrival \
and task.deadline == taskRes.deadline:
processed = True
if not processed: diff.append(task)
return diff

def BaseDynamicScheduler(estimatePriority, allTasks):
global tasks, activeTasks
tasks = deepcopy(allTasks)
tasks.sort(key=lambda x: x.arrivalTime)
timestamps = dict()
curTime, tactCount, idle = 0, 0, 0
for task in tasks: timestamps[taskID(task)] = list()
resTasks = list()
while activeTasks or tasks:
if tasks: checkForNewTasks(curTime)
activeTasks = list(filter(lambda x: x.deadline > curTime, activeTasks))
missed = list(filter(lambda x: x.deadline <= curTime, activeTasks))
estimatePriority(activeTasks)
for missedTask in missed:
task = ProcessedTask(
missedTask.arrivalTime,
missedTask.deadline,
[curTime, curTime],
True
)
resTasks.append(task)
remainedTime = tactCount or TACT_SIZE
task = None
if activeTasks:
task = activeTasks[0]
else:
task = waitForTasks(curTime, remainedTime, estimatePriority)
if not task:
idle += remainedTime
curTime += remainedTime
if tactCount: tactCount = 0
continue

if task.arrivalTime > curTime:
diff = task.arrivalTime - curTime
idle += diff
curTime = task.arrivalTime
tactCount -= diff
```

```
timestamps[taskID(task)].append(curTime)
fitsIntoTact = task.wcet < remainedTime
elapsedInTact = task.wcet if fitsIntoTact else remainedTime
newTask = waitForTasks(curTime, elapsedInTact, estimatePriority)
if newTask and not areTasksEqual(task, newTask):
task.wcet -= newTask.arrivalTime - curTime
curTime = newTask.arrivalTime
timestamps[taskID(task)].append(curTime)
task = newTask
timestamps[taskID(task)].append(curTime)
fitsIntoTact = task.wcet < remainedTime
elapsedInTact = task.wcet if fitsIntoTact else remainedTime
curTime += elapsedInTact
timestamps[taskID(task)].append(curTime)
if not fitsIntoTact:
task.wcet -= remainedTime
tactCount = 0
else:
missed = task.deadline < curTime
task = ProcessedTask(
task.arrivalTime,
task.deadline,
timestamps[taskID(task)],
missed
)
resTasks.append(task)
activeTasks.pop(0)
tactCount = remainedTime - elapsedInTact

missed = checkMissedTasks(allTasks, resTasks)
if missed:
for task in missed:
task = ProcessedTask(
task.arrivalTime,
task.deadline,
[curTime, curTime],
True
)
resTasks.append(task)
return SchedulingResult(resTasks, idle, curTime)

def estimatePriorityEDF(tasks):
return tasks.sort(key=lambda x: (x.deadline, x.wcet))

def estimatePriorityRM(tasks):
return tasks.sort(key=lambda x: (x.period, x.wcet, x.deadline))

EDF = partial(BaseDynamicScheduler, estimatePriorityEDF)
RM = partial(BaseDynamicScheduler, estimatePriorityRM)
```

utils.py

```
import matplotlib

import matplotlib.pyplot as plt
from queueGenerator import generateTasks
from fifo import FIFO
from dynamicSchedulers import RM, EDF
from random import choice
```

```
def printResult(schedRes, type):
tasks = schedRes.result
out = (
f'Scheduling using {type}:\n'
f'\tAverage waiting time: {schedRes.avgWait}\n'
f'\tIdle time percent: {schedRes.idlePercent}%'
f'(total: {round(schedRes.totalTime, 4)}, '
f'idle: {round(schedRes.idleTime, 4)})\n'
f'\tMissed tasks count: {schedRes.missedTasksCount} '
f'({schedRes.missedTasksPercent}% out of total {len(tasks)})\n'
)
for i, task in enumerate(tasks):
out += (
f'\nTask {i + 1}: arrived: {task.arrival}, waited: {task.waitingTime}, '
f'started processing: {round(task.start, 4)}, '
f'finish: {round(task.end, 4)}; '
)
if (task.deadlineMissed): out += f'DEADLINE MISSED'
print(out)

def runSchedulers():
hex_colors_dic, rgb_colors_dic = dict(), dict()
hex_colors_only = list()
for name, hex in matplotlib.colors.cnames.items():
hex_colors_only.append(hex)
hex_colors_dic[name] = hex
rgb_colors_dic[name] = matplotlib.colors.to_rgb(hex)

tasks = generateTasks(7, 1.5, 7)
size = len(tasks)
schedulers = {
'FIFO': FIFO,
'EDF': EDF,
'RM': RM
}
resStats = {t: sched(tasks).ganttChartData for t, sched in schedulers.items()}
fig, axs = plt.subplots(3)
plt.subplots_adjust(left=0.05, bottom=0.03, right=0.97, top=0.92, wspace=0.1)
fig.suptitle('Scheduling results')
for i, title in enumerate(resStats):
gnt = axs[i]
gnt.grid(True)
gnt.set_title(title)
gnt.set_yticks([x * 3 + 3 for x in range(size)])
gnt.set_yticklabels(f'Task {x + 1}' for x in range(size))

for task in range(size):
for i, key in enumerate(resStats):
color = choice(hex_colors_only)
result = resStats[key]
data = result[task]
axs[i].broken_barh(data, (task * 3, 3), facecolors=color)
axs[i].broken_barh(data, (size * 3, 5), facecolors=color)

def intensityData(scheduler):
intensities = [i * 0.1 for i in range(1, 1000, 2)]
avgWait, idlePercent, missedPercent = list(), list(), list()
for intensity in intensities:
tasks = generateTasks(intensity, 2.5)
results = scheduler(tasks)
```

```
avgWait.append(results.avgWait)
idlePercent.append(results.idlePercent)
missedPercent.append(results.missedTasksPercent)
return intensities, avgWait, idlePercent, missedPercent

def sizeData(scheduler):
intensity = 15
results = dict()
lens, avgWait, missedPercent = list(), list(), list()
for _ in range(1000):
tasks = generateTasks(intensity, 2.5)
result = scheduler(tasks)
results[len(tasks)] = [result.avgWait, result.missedTasksPercent]
for key in sorted(results):
lens.append(key)
avgWait.append(results[key][0])
missedPercent.append(results[key][1])
return lens, avgWait, missedPercent

def plot(titles, stats, bar = False):
size = len(titles)
fig, axs = plt.subplots(size)
plt.subplots_adjust(left=0.05, bottom=0.05, right=0.97, top=0.95, hspace=0.27)

width = 0.3
offset = - width
for title, data in stats.items():
if bar:
x = [x + offset for x in data[0]]
axs[0].bar(x, data[1], width, label=title)
axs[1].bar(x, data[2], width, label=title)
offset += width
else:
axs[0].plot(data[0], data[1], label=title, linewidth=0.9)
axs[1].plot(data[0], data[2], label=title, linewidth=0.9)
axs[2].plot(data[0], data[3], label=title, linewidth=0.9)

for i, title in enumerate(titles):
axes = titles[title]
axs[i].set_title(title)
axs[i].set(xlabel=axes[0], ylabel=axes[1])
axs[i].legend()

def showGraphs():
titlesInt = {
'Intensity - wait time': ['intensity', 'average wait time'],
'Intensity - idle percent': ['intensity', 'idle time percent'],
'Intensity - missed deadlines': ['intensity', 'missed percent']
}
titlesLen = {
'Tasks amount - average wait time': ['queue length', 'wait time'],
'Tasks amount - missed deadlines': ['queue length', 'missed percent']
}
schedulers = {
'FIFO': FIFO,
'EDF': EDF,
'RM': RM
}
resStats = {t: [intensityData(s), sizeData(s)] for t, s in schedulers.items()}
plot(titlesInt, {k: v[0] for k, v in resStats.items()})
```

```
plot(titlesLen, {k: v[1] for k, v in resStats.items()}, True)
```

example.py

```
from dynamicSchedulers import EDF, RM
from fifo import FIFO
from utils import printResult, showGraphs, runSchedulers
from queueGenerator import generateTasks
import matplotlib.pyplot as plt

INTERVAL = 2.5
INTENSITY = 7
LIMIT = 5
schedulers = {
'FIFO': FIFO,
'EDF': EDF,
'RM': RM
}
tasks = generateTasks(INTENSITY, INTERVAL, LIMIT)
for name, sched in schedulers.items():
printResult(sched(tasks), name)
runSchedulers()
showGraphs()
plt.show()
```