Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

Розрахунково-графічна робота

з дисципліни
«Інтелектуальні вбудовані системи»

на тему
«Дослідження роботи планувальників роботи систем реального часу»

Виконала:

студентка групи ІП-84

Скрипник Єлена Сергіївна

номер залікової книжки: 8422

Перевірив:

доц. Волокита А. М.

Київ 2021

Зміст

Розділ 1. 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.

Операційні системи можуть включати до трьох різних типів планувальників: довготривалий планувальник (або планувальник дозволу виконання), середньостроковий планувальник і короткостроковий планувальник (також відомий як диспетчер). Самі назви вже описують відносну частоту, з якою планувальник виконує свої функції.

Довготривалий планувальник вирішує, які завдання або процеси будуть додані в чергу процесів, готових до виконання; тобто, коли проводиться спроба запуску процесу, довготривалий планувальник або додає новий процес в чергу готових процесів (допускає до виконання), або відкладає цю дію. Таким чином, довготривалий планувальник вирішує, які процеси виконуватимуться одночасно, тим самим контролюючи ступінь паралелізму і пропорцію між процесами, що інтенсивно виконують введення-виведення, і процесами, що інтенсивно використовують процесор.

Зазвичай в настільних комп'ютерах не застосовується довготривалий планувальник і нові процеси допускаються до виконання автоматично. Але цей планувальник дуже важливий для систем реального часу, оскільки при надмірному навантаженні системи процесами, що паралельно виконуються, час відгуку системи може стати більше потрібного, що неприпустимо.

У всіх системах з віртуальною пам'яттю середньостроковий планувальник тимчасово переміщає (вивантажує) процеси з основної пам'яті у вторинну (наприклад, на жорсткий диск), і навпаки. Ці дії називаються підкачуванням або свопінгом (англ. swapping). Середньостроковий планувальник може ухвалити рішення вивантажити процес з основної пам'яті якщо:

- процес був неактивним якийсь час;
- процес має низький пріоритет;
- процес часто викликає помилки сторінок (page fault);
- процес займає велику частку основної пам'яті, а системі потрібна вільна пам'ять для інших цілей (наприклад, щоб задовольнити запит виділення пам'яті для іншого процесу).

Короткостроковий планувальник (також відомий як диспетчер, або шедулер) вирішує, які з готових процесів у пам'яті мають бути виконані (відданні на виконання ЦП) за наступним перериванням годинника, перериванням введення-виведення, системним викликом або від іншої форми сигналу. Таким чином, короткостроковий планувальник робить планування рішень набагато частіше, ніж довгострокові і середньострокові планувальники — щонайменше одне рішення має бути зроблене після кожного часу квантування процесу, а це дуже короткий проміжок. Це планувальник може упереджувальний або витісняючий (мається на увазі, що він здатний примусово видалити процеси з процесора, якщо він вирішить передати процесор іншому процесу), або не упереджувальний (також відоме як «добровільний» або «кооперативний»), в якому планувальник не в «силі» вилучити процес від процесора.

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:

- 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.

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

Найважливішою частиною операційної системи, безпосередньо впливає на функціонування обчислювальної машини, є підсистема керування процесами. Процес (або по-іншому, завдання) -абстракція, що описує виконувану програму. Для операційної системи процес являє собою одиницю роботи, заявку на споживання системних ресурсів.

Підсистема управління процесами планує виконання процесів, тобто розподіляє процесорний час між декількома одночасно існуючими в системі процесами, а також займається створенням і знищенням процесів, забезпечує процеси необхідними системними ресурсами, підтримує взаємодію між процесами. Створення процесів і потоків

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).

Приклади описателів процесу:

- Блок управління завданням (TCB) в OS/360;
- Керуючий блок процесу (PCB) в OS/2;
- Дескриптор процесу в UNIX;
- Об'єкт-процес Windows;

1.5 Пріоритети заявок

Пріоритет - це число, характеризує ступінь привілейованості процесу при використанні ресурсів обчислювальної машини, зокрема, процесорного часу: чим вище пріоритет, тим вище привілеї.

Існує безліч різних алгоритмів планування процесів, по-різному що вирішують перераховані вище завдання, переслідують різні цілі і забезпечують різне якість мультипрограмування. Серед цієї безлічі алгоритмів розглянемо докладніше дві групи найбільш часто зустрічаються алгоритмів: алгоритми, засновані на квантуванні, і алгоритми, засновані на пріоритетах. Відповідно з алгоритмами, засновані на квантуванні, зміна активного процесу відбувається, якщо: процес завершився і покинув систему; помилка; процес перейшов у стан ОЧІКУВАННЯ; вичерпано квант процесорного часу, відведений даному процесу.

Інша група алгоритмів використовує поняття "пріоритет" процесу. Пріоритет - це число, характеризує ступінь привілейованості процесу при використанні ресурсів обчислювальної машини, зокрема, процесорного часу: чим вище пріоритет, тим вище привілеї.

Пріоритет може виражатися цілими чи дробовими, позитивним чи негативним значенням. Ніж вище привілеї процесу, тим менше часу він буде проводити в чергах. Пріоритет може призначатися директивно адміністратором системи в залежності від важливість роботи або внесеної плати, або обчислюватися самої ОС за певним правилами, він може залишатися фіксованим протягом усього життя процесу або змінюватися в часі відповідно з деяким законом. В останньому разі пріоритети називаються динамічними.

Існує дві різновиди пріоритетних алгоритмів: алгоритми, що використовують відносні пріоритети, і алгоритми, що використовують абсолютні пріоритети. Алгоритми планування процесів, засновані на квантуванні.

Відповідно з алгоритмами, засновані на квантуванні, зміна активного процесу відбувається, якщо:

- процес завершився і залишив систему,
- сталася помилка,
- процес перейшов у стан ОЧІКУВАННЯ,
- вичерпано квант процесорного часу, відведений даного процесу.

Потік, який вичерпав свій квант, переводиться в стан готовності і очікує, коли йому буде наданий новий квант процесорного часу, а на виконання відповідно з певним правилом обирається новий потік з черги готових. Кванти, виділені потоків, можуть бути однаковими для всіх потоків або різними.

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:

- 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.
- 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.

1.6 Дисципліна RR

У програмуванні планування "Round-robin" (переклад з англ. циклічне планування) є одним із алгоритмів планування процесів або комутації пакетів даних у мережі.

При роботі планувальника операційної системи інтервали часу, які часто називають квантами часу присвоюються кожному процесові або потокові однаковим чином у циклічному порядку, опрацьовуючи всі процесі без пріоритету (також відоме як циклічне виконання). Таке циклічне планування є простим, легким у виконанні, і без ресурсного голоду.

Планування Round-robin також можна застосувати і до інших задач, таких як диспетчеризація пакетів даних у комп'ютерних мережах.

Для рівноправного чесного планування процесів, циклічний планувальник в основному виконує розподіл часу, даючи кожній задачі часовий проміжок або квант (його доступ до процесорного часу), і переривання задачі, якщо вона не завершила роботу за цей квант. Задача продовжує роботу наступного разу, коли їй буде виділено наступний квант процесорного часу. Якщо процес закінчує свою роботу, або змінює свій стан на стан очікування, у момент коли йому було виділено час, планувальник вибере наступний перший процес із черги тих, що готові до виконання. Якби такого планування часу не було, або якщо кванти часу були б великими по відношенню до розмірів задач, процес який би виконував великі тривалі задачі мав би більший пріоритет над іншими задачами.

1.7 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.

1.8 Дисципліна RM

RM-алгоритм є найбільш часто використовуваним в додатках реального часу, тому що його досить просто реалізувати для ядер сучасних комерційних ОС. Дійсно, RM-планувальник може бути реалізований просто шляхом призначення кожному завданню фіксованого пріоритету, обернено пропорційного його періоду. З іншого боку, реалізуючи схему динамічного планування, (наприклад, EDF-алгоритм) потрібно, щоб ядро ОС відстежувало всі абсолютні граничні терміни і виконувало динамічне зіставлення між абсолютними термінами і пріоритетами для задач з наявного набору. Таке зіставлення має виконуватися кожного разу, коли нове ініційоване завдання отримує свій пріоритет.

Така додаткова складність реалізації через динамічне управління пріоритетами не сприяє впровадженню EDF-планування в ОСРВ, хоча це збільшило б загальну завантаженість процесора.

1.9 Постановка задачі

1. Змоделювати планувальник роботи системи реального часу. Дві дисципліни планування: перша – RR, друга задається викладачем або обирається самостійно.
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.

Розділ 2. Розробка програми для дисципліни EDF

2.1 Розробка програми

Алгоритм планування по найближчому терміну завершення (EDF) є одним з динамічних алгоритмів планування і використовується в операційних системах реального часу для установки черзі процесів. При настанні події планування (завдання завершена, встановлена нова задача і т. Д.) Чергу шукає найближчу до крайнього часу її виконання завдання, і цей процес призначається для виконання.

Нам необхідно досягти аби спочатку виконувалися заявки, які на виконання потребують менше часу ніж інші доступні заявки, незважаючи на будь-які інші параметри. Тому створимо початкову заявку в момент часу 0, отримуватимемо значення часу очікування у черзі, а також перевірятимемо чи було досягнуто дедлайну. Будемо зберігати значення і змінюватимемо інтенсивність появи вхідних заявок.

Лістинг даного алгоритму представлений у додатку.

2.2 Графіки

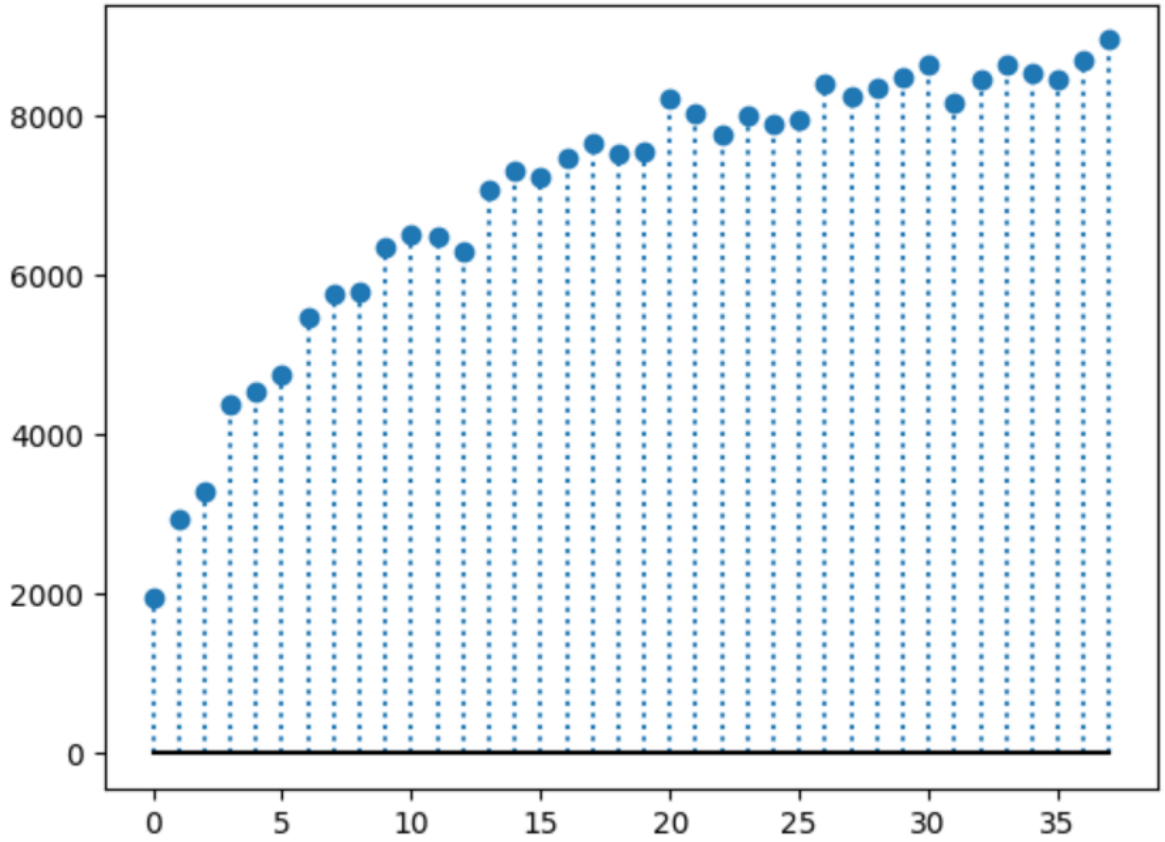

Рис. 2.2.1 – Графік залежності кількості заявок від часу очікування при фіксованій інтенсивності вхідного потоку заявок

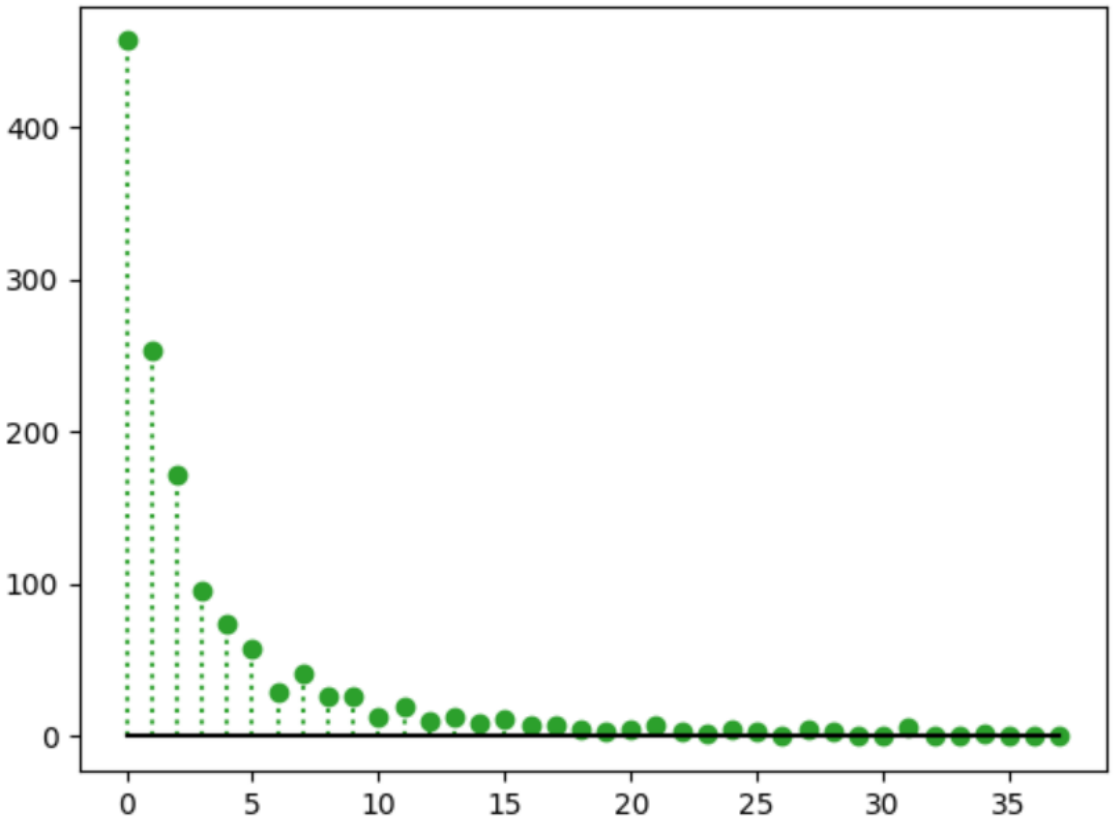

Рис.2.2.2 – Графік залежності середнього часу очікування від інтенсивності вхідного потоку заявок

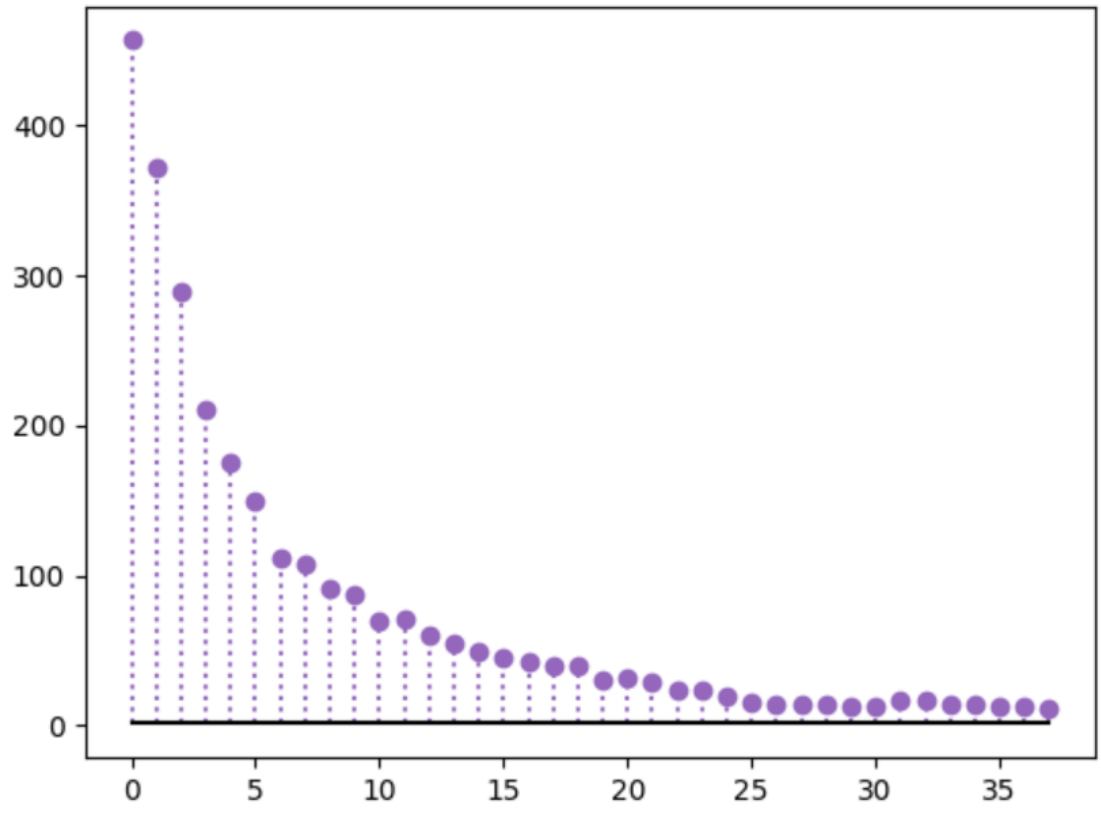

Рис.2.2.3 – Графік залежності проценту простою ресурсу від інтенсивності вхідного потоку заявок

2.3 Висновки

Було створено програму моделювання роботи системи масового обслуговування з дисципліни EDF та проведено її тестування. Графіки та отримані дані підтверджують правильність роботи програми.

Розділ 3. Розробка програми для дисципліни RM

3.1 Розробка програми

Перевага RM по відношенню до EDF полягає в тому, що якщо кількість рівнів пріоритетів для набору завдань не є великою, алгоритм RM може бути реалізований більш ефективно, розбивши при цьому чергу готових завдань на кілька черг FIFO, по одній для кожного рівня пріоритету. На жаль, таке ж рішення не може бути прийнято для EDF, оскільки кількість черг в такому випадку буде занадто великим. Іншим недоліком EDF з точки зору складності реалізації є те, що абсолютні крайні терміни змінюються і їх необхідно обчислювати за будь-якої активації нового запиту.

Цікаво спостерігати різну поведінку RM і EDF при збільшенні завантаження процесора. У RM кількість витіснення постійно збільшується зі збільшенням коефіцієнта утилізації U, тому що завдання з більш довгим терміном виконання мають більше шансів бути витісненими новими завданнями з високим пріоритетом. Тим не менш, в рамках EDF збільшення часу виконання завдань далеко не завжди означає більшу кількість витіснення, тому що завдання з довгим періодом може мати дедлайн коротше, ніж завдання з меншим періодом.

Ще одна важлива відмінність між RM і EDF стосується кількості витіснення, що відбуваються в розкладі. Як нам відомо, в RM варіанті загальна кількість витіснення дорівнює п'яти. У EDF варіанті одна задача витісняється тільки один раз.

Менше кількість витіснення в EDF є прямим наслідком призначення динамічних пріоритетів, яке в будь-який момент робить більш пріоритетне завдання з найменшим крайнім терміном di, незалежно від періодів інших завдань.

Аби реалізувати даний алгоритм, створимо початкову заявку в момент часу 0, отримуватимемо значення часу очікування у черзі, а також перевірятимемо чи було досягнуто дедлайну. Будемо перевіряти чи немає в черзі заявки із меншим часом дедлайну: у разі якщо така заявка буде, то блокуватимемо виконання поточної задачі, додаватимемо її в чергу та оберемо нову задачу на виконання.

Лістинг даного алгоритму представлений у додатку.

3.2 Графіки

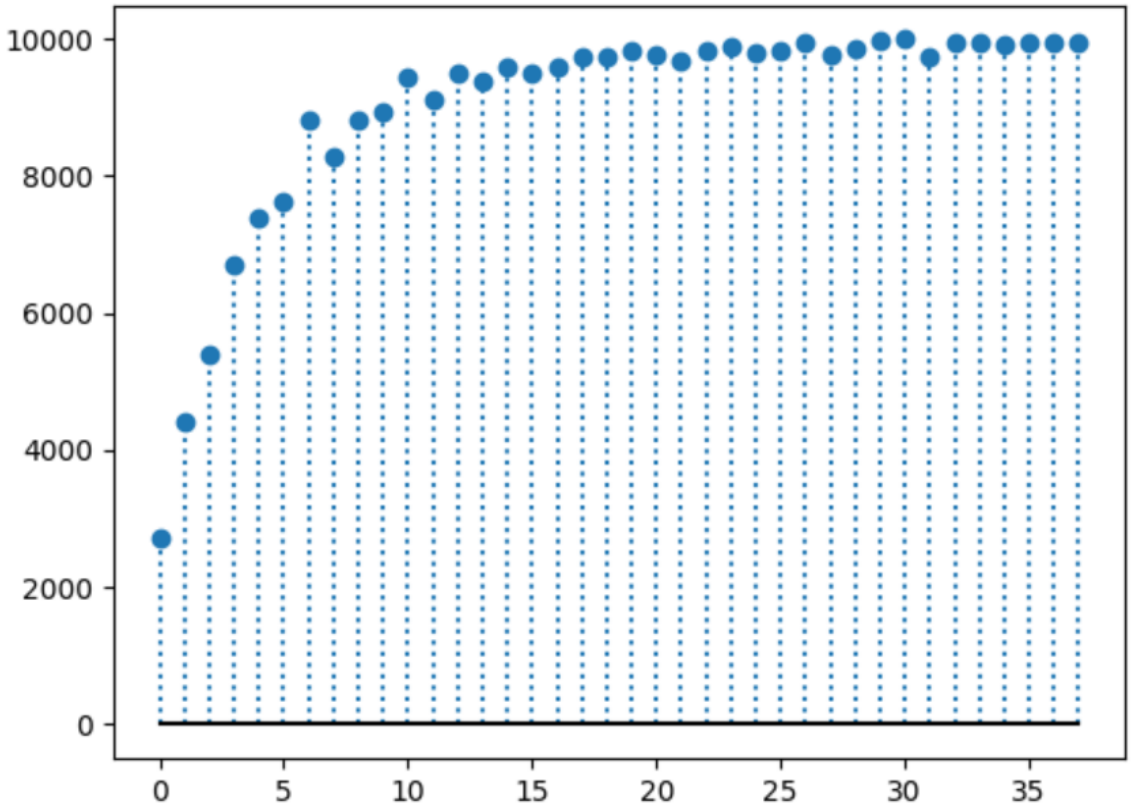

Рис. 3.2.1 – Графік залежності кількості заявок від часу очікування при фіксованій інтенсивності вхідного потоку заявок

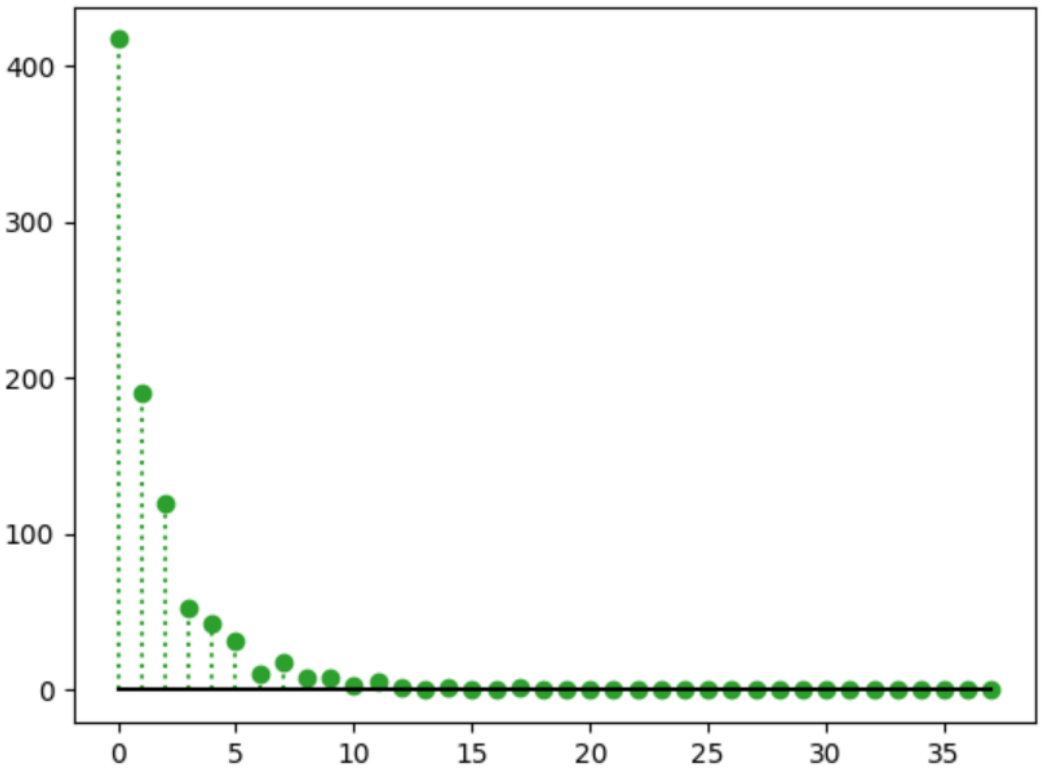

Рис.3.2.2 – Графік залежності середнього часу очікування від інтенсивності вхідного потоку заявок

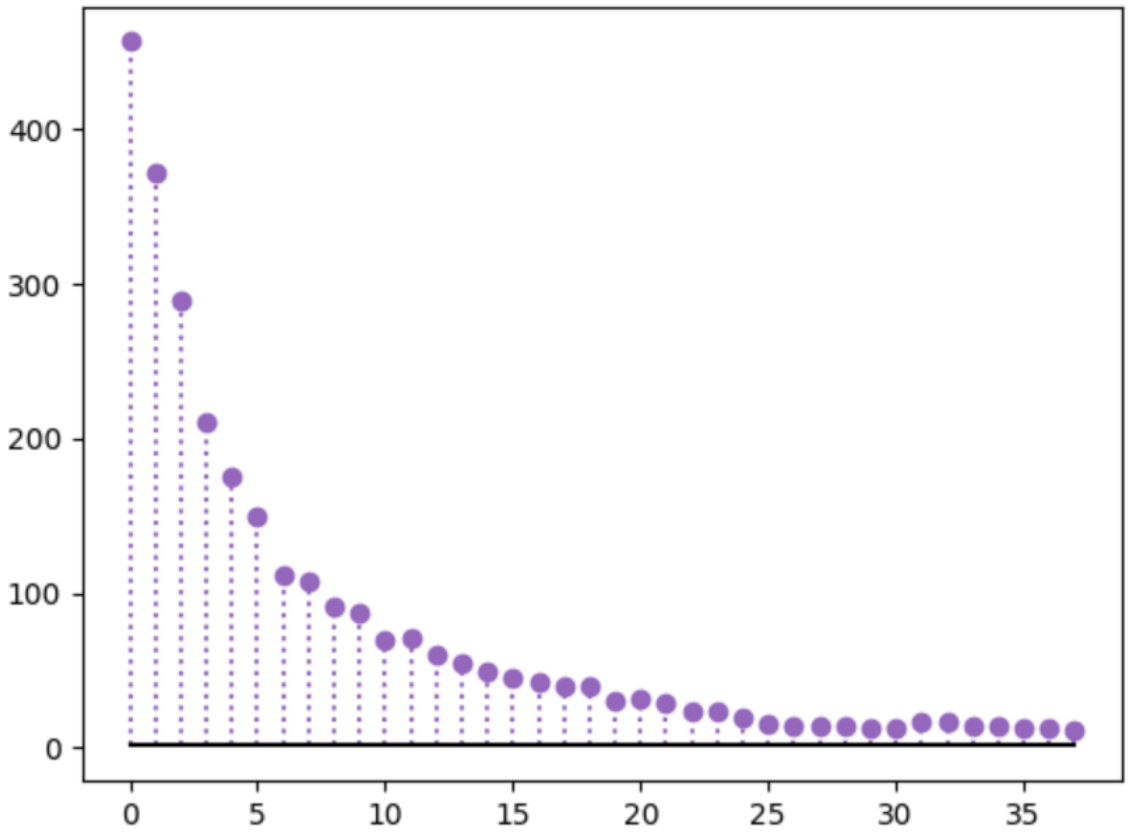

Рис.3.2.3 – Графік залежності проценту простою ресурсу від інтенсивності вхідного потоку заявок

3.3 Висновки

Було створено програму моделювання роботи системи масового обслуговування з дисципліни RM та проведено її тестування. Графіки та отримані дані підтверджують правильність роботи програми.

Висновки

На даній розрахунково-графічній роботі перед нами була задача ознайомитися та дослідити роботу планувальників систем реального часу, а також продемонструвати роботу основних алгоритмів планування. Основною задачею було змоделювати роботу планувальника задач у системі реального часу. Для досягнення поставленої задачі було запропоновано декілька алгоритмів планування. Серед них: найвідоміший та певно найбільш розповсюджений – алгоритм Round Robin, а також будь який інший на вибір – EDF, FB, RM і так далі. Для саме цієї розрахунково-графічної роботи було обрано алгоритми EDF (Earliest Deadline First) та RM (Rate Monotonic), адже вони не настільки розповсюджені як RR. Було розроблено програми для моделювання, а

також створено графіки, які підтверджують теоретичні дані та правильність роботи.

Джерела

1. Планування - Scheduling (computing) - Вікіпедія [Електронний ресурс] – Режим доступу до ресурсу: Посилання.
2. Алгоритми планування у сучасних ОС [Електронний ресурс] – Режим доступу до ресурсу: Посилання.
3. Алгоритм Round Robin – Вікіпедія [Електронний ресурс] – Режим доступу до ресурсу: Посилання.
4. Алгоритм EDF – Вікіпедія [Електронний ресурс] – Режим доступу до ресурсу: Посилання.
5. Алгоритм RM – Вікіпедія [Електронний ресурс] – Режим доступу до ресурсу: Посилання.

Додаток

RGR.py

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

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

def GenerateQ():
    Q = []
    next = 0
    i = 0
    time = GenerateTime()
    while (i < 10000):
        i += E.GenerateNextInternal() * 8
        t = Task.Task(i, time)
        Q.append(t)
    return Q

def GenerateTime():
    rnd = random.random()
    if(rnd < 0.3):
```

```
        return random.randrange(7) + 40
    elif (rnd >= 0.3 and rnd < 0.6):
        return random.randrange(5) + 40
    elif (rnd >= 0.6 and rnd < 0.8):
        return random.randrange(3) + 40
    else:
        return random.randrange(2) + 40


if __name__ == "__main__":
    QuEDF = []
    QuRM = []
    SMOs = []
    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()
        QuEDF.append(temp)

    QuRM = QuEDF[:]

    for i in range(len(QuEDF)):
        SMOs.append(SMO.EDF(QuEDF[i]))
        RMs.append(SMO.RM(QuRM[i]))

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

    for i in SMOs:
        i.Work()
        buf1 = i.GetFaults()
        buf2 = i.GetWaitTimes()
        buf3 = i.GetProcessorFreeTime()
        faults.append(buf1[:])
        Tw.append(buf2[:])
        Tn.append(buf3[:])
```

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

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

for i in range(len(SMOs)):
    Tww.append(FullWaitTime[i])

plt.stem(Tnchange, linefmt='C0:', markerfmt='C0o', bottom=1.1, basefmt='black')
plt.show()
plt.stem(faultschange, linefmt='C2:', markerfmt='C2o', bottom=1.1, basefmt='black')
plt.show()
plt.stem(Tww, linefmt='C4:', markerfmt='C4o', bottom=1.1, basefmt='black')
plt.show()

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

for i in RMs:
    i.Work()
    buf1 = i.GetFaults()
    buf2 = i.GetWaitTimes()
    buf3 = i.GetProcessorFreeTime()
    faults.append(buf1[:])
    Tw.append(buf2[:])
    Tn.append(buf3[:])

for i in range(len(SMOs)):
    faultschange.append(faults[i][9999])
    Tnchange.append(Tn[i][9999])

time = 0
```

```
    for o in range(len(SMOs)):
        time = 0
        for i in range(10000):
            time += Tw[o][i]
        FullWaitTime.append(time)

    for i in range(len(SMOs)):
        Tww.append(FullWaitTime[i])

    plt.stem(Tnchange, linefmt='C0:', markerfmt='C0o', bottom=1.1, basefmt='black')
    plt.show()
    plt.stem(faultschange, linefmt='C2:', markerfmt='C2o', bottom=1.1, basefmt='black')
    plt.show()
    plt.stem(Tww, linefmt='C4:', markerfmt='C4o', bottom=1.1, basefmt='black')
    plt.show()
```

Task.py

```
class Task:

    deadline = 0
    creationTime = 0
    executionTime = 0
    deadlineMultiplier = 0
    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
```

Erlang.py

```
import random
import math

class ErlangDistribution:

    k = 0
    lam = 0

    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!")

    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
```

SMO.py

```
import math
import Task

class EDF:

    currentTime = 0
    Q = []
    Qready = []
    Tw = [0 for x in range(10000)]
    Tn = [0 for x in range(10000)]
    faults = [0 for x in range(10000)]
    currentTask = None
```

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

def GetEDTask(self, removeFromQ):
    timeBuf = 9999999
    newTime = 0
    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


def Work(self):
    self.currentTime = 0
    timewait = 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 != None and self.currentTask.GetExecutionTime() == 0):
                self.currentTask = None
            elif(self.currentTask != None and self.GetEDTask(False) != None):
                if(self.GetEDTask(False).GetExecutionTime() < self.currentTask.GetExecutionTime()):
                    self.Qready.append(self.currentTask)
                    self.currentTask = self.GetEDTask(True)
            elif(self.currentTask != None):
                self.currentTask.WorkedOn()
            if(self.GetEDTask(False) == None and self.currentTask == None):
                self.Tn[self.currentTime] += 1
                continue
            elif(self.currentTask == None):
                self.currentTask = self.GetEDTask(True)
            for task in self.Qready:
                task.Wait()
                timewait += 1
            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 = []
    Qready = []
    Tw = [0 for x in range(10000)]
    Tn = [0 for x in range(10000)]
    faults = [0 for x in range(10000)]
    currentTask = None

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

    def GetEDTask(self, removeFromQ):
        timeBuf = 9999999
        newTime = 0
        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


    def Work(self):
        self.currentTime = 0
        timewait = 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 != None and self.currentTask.GetExecutionTime() == 0):
                self.currentTask = None
            elif (self.currentTask != None):
                self.currentTask.WorkedOn()
            if (self.GetEDTask(False) == None and self.currentTask == None):
                self.Tn[self.currentTime] += 1
                continue
            elif (self.currentTask == None):
                self.currentTask = self.GetEDTask(True)
```

```
        for task in self.Qready:
            task.Wait()
            timewait += 1
        self.Tw[self.currentTime] = timewait

def GetWaitTimes(self):
    return self.Tw

def GetFaults(self):
    return self.faults

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