A picture containing text

Description automatically generated

**Національний Технічний Університет України КПІ**

Факультет Інформатики та Обчислювальної Техніки  
Кафедра інформатики та програмної інженерії

**Практична робота №2**

З дисципліни «Моделювання систем»

ОБ’ЄКТНО-ОРІЄНТОВАНИЙ ПІДХІД ДО ПОБУДОВИ ІМІТАЦІЙНИХ

МОДЕЛЕЙ ДИСКРЕТНО-ПОДІЙНИХ СИСТЕМ

**Перевірив:**

Асистент

Бернатович Анатолій Олександрович

Оцінка:\_\_\_\_\_\_\_\_\_\_\_**Виконав:**

Студент групи ІТ-92

Яцевський О. І..

Дата: \_\_\_\_\_\_\_\_\_\_\_\_\_

Завдання до практичної роботи

1. Реалізувати алгоритм імітації простої моделі обслуговування одним пристроєм з використанням об’єктно-орієнтованого підходу. **5 балів.**
2. Модифікувати алгоритм, додавши обчислення середнього завантаження пристрою. **5 балів.**
3. Створити модель за схемою, представленою на рисунку нище. **30 балів.**

Diagram

Description automatically generated

1. Виконати верифікацію моделі, змінюючи значення вхідних змінних та параметрів моделі. Навести результати верифікації у таблиці. **10 балів.**
2. Модифікувати клас PROCESS, щоб можна було його використовувати для моделювання процесу обслуговування кількома ідентичними пристроями. **20 балів.**
3. Модифікувати клас PROCESS, щоб можна було організовувати вихід в два і більше наступних блоків, в тому числі з поверненням у попередні блоки. **30 балів.**

Теоретичні відомості

Для данної лабораторної роботи була обрана універсальна мова програмування Python. Такий вибір був здійсненний через широку сферу застосування мови, обширний асортимент математичних фунцій у бібліотеках як numpy, а також наявність зручних інструментів візуалізаціїї і вводу / виводу.

Приблизну структуру об’єктно-орієнтованої програми представимо діаграмою класів:

Timeline

Description automatically generated

Модель складається з елементів, які беруть за основу клас Element. Таким чином, всі наступні класи матимуть всі наступні властивості :

|  |  |
| --- | --- |
| Назва | Коментар |
| tcurr | поточний момент часу |
| tnext | момент часу наступної події |
| delayMean | середеє значення часової затримки |
| delayDev | середнє квадратичне відхилення часової затримки |
| getDelay() | функція для розрахуноку часової затримками |
| inAct() | розрахунки, які відбуваються на вході в компонент |
| outAct() | виході з компоненту |
| nextElement | Наступний елемент моделі |

Важливим для з’єднання елементів є полу nextElement, яку встановлює зв’язки між елементами, об’єднуючи їх в одну модель.

Модельі відповідає клас Model, що містить в собі список елементів і здійсюнє імітацію на інтервалі часу time. Імітація здійснюється за наступним принципом: визначається момент найближчої події, просувається час в момент найближчої події та здійснюється відповідна подія.

Для оптимізації розрахунків ми можемо вести розрахунки для процесів, які відбуваються в поточний момент часу, паралельно.

Виконання лабораторної роботи

Побудуємо алгоритм імітації простої моделі обслуговування одним пристроєм з використанням об’єктно-орієнтованого підходу, використовуючи приклад, який було наведено у лекції, та розрахуємо середнє завантаження пристрою. Наш алгоритм буде виглядати наступним чином:

Shape

Description automatically generated with medium confidence

Емуляцію будемо виконувати на проміжку часу рівним у 1000 одиниць. Затримка (delay) для CREATE та PROCESS буде рівним 5. Для розрахунку коли відбудиться наступна подія ми будемо використовувати експоненційний розподіл.

Не вдаючись у подробиці внутрішньої реалізації коду, (детальний опис коду можна знайти у мене на сторінці у GitHub (https://github.com/Oleks-Y/system-modeling) побудова даної моделі буде виглядати так:

c = Create(5)

p1 = Process(5)

p2 = Process(5)

p3 = Process(5)

c.next\_element = [p1]

p1.next\_element = [p2, p3]

p1.probability = ([0.5, 0.5])

p1.max\_queue = 5

p2.max\_queue = 5

p3.max\_queue = 5

c.distribution = 'exp'

p1.distribution = 'exp'

p2.distribution = 'exp'

p3.distribution = 'exp'

c.name = 'Creator'

p1.name = 'Process 1'

p2.name = 'Process 2'

p3.name = 'Process 3'

elements = [c, p1, p2, p3]

model = Model(elements)

model.simulate(1000)

Виконаємо програму та отримаємо наступні результати:

Text

Description automatically generated

* “Creator quantity” – кількість опрацьованих елементів процесом CREATE;
* “Process 1 quantity” – кількість опрацьованих елементів процесом PROCESS;
* “Average queue length” – середня довжина черги для PROCESS;
* “Failure probability” – ймовірність не опрацювання завдання для PROCESS;
* “Average load” – середнє завантаження PROCESS.

Побудуємо модель за схемою наведеною у завданні до практичної роботи разом із верифікацією моделі.

p1.max\_queue = 5

c.distribution = 'exp'

p1.distribution = 'exp'

c.name = 'Creator'

p1.name = 'Process 1'

c.next\_element = [p1]

elements = [c, p1]

model = Model(elements)

model.simulate(1000)

def test\_model():

n\_param = 15

delay\_create\_list = [4, 10, 4, 4, 4, 4, 4, 4, 0.5, 4, 4, 4, 4, 4, 4]

delay\_process1\_list = [4, 4, 10, 4, 4, 4, 4, 4, 4, 0.5, 4, 4, 4, 4, 4]

delay\_process2\_list = [4, 4, 4, 10, 4, 4, 4, 4, 4, 4, 0.5, 4, 4, 4, 4]

delay\_process3\_list = [4, 4, 4, 4, 10, 4, 4, 4, 4, 4, 4, 0.5, 4, 4, 4]

maxQ\_list1 = [5, 5, 5, 5, 5, 10, 5, 5, 5, 5, 5, 5, 1, 5, 5]

maxQ\_list2 = [5, 5, 5, 5, 5, 5, 10, 5, 5, 5, 5, 5, 5, 1, 5]

maxQ\_list3 = [5, 5, 5, 5, 5, 5, 5, 10, 5, 5, 5, 5, 5, 5, 1]

distribution = ['exp', 'exp', 'exp', 'exp', 'exp', 'exp', 'exp', 'exp', 'exp', 'exp', 'exp', 'exp', 'exp', 'exp',

'exp', 'exp']

df = pd.DataFrame()

rows = []

for i in range(n\_param):

c = Create(delay\_create\_list[i])

p1 = Process(delay\_process1\_list[i])

p2 = Process(delay\_process2\_list[i])

p3 = Process(delay\_process3\_list[i])

p1.max\_queue = maxQ\_list1[i]

p2.max\_queue = maxQ\_list2[i]

p3.max\_queue = maxQ\_list3[i]

c.distribution = distribution[i]

p1.distribution = distribution[i]

p2.distribution = distribution[i]

p3.distribution = distribution[i]

c.name = 'Creator'

p1.name = 'Process 1'

p2.name = 'Process 2'

p3.name = 'Process 3'

c.next\_element = [p1]

p1.next\_element = [p2]

p2.next\_element = [p3]

elements = [c, p1, p2, p3]

model = Model(elements)

res = model.simulate(1000)

param = {'delay\_create': delay\_create\_list[i],

'delay\_process1': delay\_process1\_list[i],

'delay\_process2': delay\_process2\_list[i],

'delay\_process3': delay\_process3\_list[i],

'max\_queue1': maxQ\_list1[i],

'max\_queue2': maxQ\_list2[i],

'max\_queue3': maxQ\_list3[i],

'process1\_processed': p1.quantity,

'process1\_failed': p1.failure,

'process2\_processed': p2.quantity,

'process2\_failed': p2.failure,

'process3\_processed': p3.quantity,

'process3\_failed': p3.failure,

'distribution': distribution[i]}

rows.append({\*\*param, \*\*res})

# назва файлу xlsx

file\_name = 'ModelData.xlsx'

# імпорт в Excel

df = df.append(rows)

df.to\_excel(file\_name)

print(tabulate(df, headers='keys', tablefmt='fancy\_grid', numalign="center"))

Diagram

Description automatically generated

Отримаємо наступні результати запуску програми:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **delay\_create** | **delay\_process1** | **delay\_process2** | **delay\_process3** | **max\_queue1** | **max\_queue2** | **max\_queue3** | **process1\_processed** | **process1\_failed** | **process2\_processed** | **process2\_failed** | **process3\_processed** | **process3\_failed** | **distribution** | **global\_max\_observed\_queue\_length** | **global\_mean\_queue\_length** | **global\_failure\_probability** | **global\_mean\_load** |
| **0** | 4 | 4 | 4 | 4 | 5 | 5 | 5 | 239 | 61 | 199 | 35 | 187 | 11 | exp | 5 | 1,96329665883134 | 0,136153846153846 | 0,817246870830475 |
| **1** | 10 | 4 | 4 | 4 | 5 | 5 | 5 | 103 | 0 | 103 | 0 | 102 | 0 | exp | 5 | 0,234065311464701 | 0 | 0,392058257149655 |
| **2** | 4 | 10 | 4 | 4 | 5 | 5 | 5 | 91 | 158 | 91 | 0 | 91 | 0 | exp | 5 | 1,54153022070771 | 0,211512717536814 | 0,548288112083285 |
| **3** | 4 | 4 | 10 | 4 | 5 | 5 | 5 | 214 | 25 | 86 | 122 | 86 | 0 | exp | 5 | 2,06332714380488 | 0,230380323999571 | 0,738033586945565 |
| **4** | 4 | 4 | 4 | 10 | 5 | 5 | 5 | 197 | 36 | 188 | 7 | 80 | 103 | exp | 5 | 2,66015449650992 | 0,251081801240107 | 0,857521512904235 |
| **5** | 4 | 4 | 4 | 4 | 10 | 5 | 5 | 248 | 4 | 228 | 17 | 198 | 24 | exp | 10 | 2,12768148474995 | 0,0644562930277216 | 0,843621199719974 |
| **6** | 4 | 4 | 4 | 4 | 5 | 10 | 5 | 230 | 15 | 219 | 4 | 195 | 22 | exp | 10 | 2,21864711244413 | 0,0601813993353743 | 0,827468627231342 |
| **7** | 4 | 4 | 4 | 4 | 5 | 5 | 10 | 218 | 30 | 196 | 20 | 181 | 4 | exp | 10 | 1,91120880157469 | 0,0783939853832327 | 0,824921081794771 |
| **8** | 0,5 | 4 | 4 | 4 | 5 | 5 | 5 | 253 | 1803 | 227 | 24 | 197 | 25 | exp | 5 | 3,13232050670273 | 0,361725222594973 | 0,9261935255634 |
| **9** | 4 | 0,5 | 4 | 4 | 5 | 5 | 5 | 268 | 0 | 232 | 30 | 209 | 19 | exp | 5 | 1,34825710210104 | 0,0659457167090755 | 0,621434122841721 |
| **10** | 4 | 4 | 0,5 | 4 | 5 | 5 | 5 | 207 | 68 | 207 | 0 | 198 | 6 | exp | 5 | 1,34575052513332 | 0,0922281639928699 | 0,596724495376134 |
| **11** | 4 | 4 | 4 | 0,5 | 5 | 5 | 5 | 206 | 46 | 199 | 2 | 198 | 0 | exp | 5 | 1,08401389836523 | 0,0641633104319671 | 0,564292674510968 |
| **12** | 4 | 4 | 4 | 4 | 1 | 5 | 5 | 177 | 82 | 173 | 2 | 172 | 1 | exp | 5 | 0,583041732344169 | 0,111270411617232 | 0,630336170395235 |
| **13** | 4 | 4 | 4 | 4 | 5 | 1 | 5 | 204 | 41 | 161 | 42 | 148 | 9 | exp | 5 | 1,1071545000635 | 0,143856110421326 | 0,706268091985473 |
| **14** | 4 | 4 | 4 | 4 | 5 | 5 | 1 | 230 | 35 | 206 | 23 | 155 | 51 | exp | 5 | 1,43617767200842 | 0,160028322818267 | 0,777969815176557 |

p1.max\_queue = 5

c.distribution = 'exp'

p1.distribution = 'exp'

c.name = 'Creator'

p1.name = 'Process 1'

c.next\_element = [p1]

elements = [c, p1]

model = Model(elements)

model.simulate(1000)

def test\_model():

n\_param = 15

delay\_create\_list = [4, 10, 4, 4, 4, 4, 4, 4, 0.5, 4, 4, 4, 4, 4, 4]

delay\_process1\_list = [4, 4, 10, 4, 4, 4, 4, 4, 4, 0.5, 4, 4, 4, 4, 4]

delay\_process2\_list = [4, 4, 4, 10, 4, 4, 4, 4, 4, 4, 0.5, 4, 4, 4, 4]

delay\_process3\_list = [4, 4, 4, 4, 10, 4, 4, 4, 4, 4, 4, 0.5, 4, 4, 4]

maxQ\_list1 = [5, 5, 5, 5, 5, 10, 5, 5, 5, 5, 5, 5, 1, 5, 5]

maxQ\_list2 = [5, 5, 5, 5, 5, 5, 10, 5, 5, 5, 5, 5, 5, 1, 5]

maxQ\_list3 = [5, 5, 5, 5, 5, 5, 5, 10, 5, 5, 5, 5, 5, 5, 1]

distribution = ['exp', 'exp', 'exp', 'exp', 'exp', 'exp', 'exp', 'exp', 'exp', 'exp', 'exp', 'exp', 'exp', 'exp',

'exp', 'exp']

df = pd.DataFrame()

rows = []

for i in range(n\_param):

c = Create(delay\_create\_list[i])

p1 = Process(delay\_process1\_list[i])

p2 = Process(delay\_process2\_list[i])

p3 = Process(delay\_process3\_list[i])

p1.max\_queue = maxQ\_list1[i]

p2.max\_queue = maxQ\_list2[i]

p3.max\_queue = maxQ\_list3[i]

c.distribution = distribution[i]

p1.distribution = distribution[i]

p2.distribution = distribution[i]

p3.distribution = distribution[i]

c.name = 'Creator'

p1.name = 'Process 1'

p2.name = 'Process 2'

p3.name = 'Process 3'

c.next\_element = [p1]

p1.next\_element = [p2]

p2.next\_element = [p3]

elements = [c, p1, p2, p3]

model = Model(elements)

res = model.simulate(1000)

param = {'delay\_create': delay\_create\_list[i],

'delay\_process1': delay\_process1\_list[i],

'delay\_process2': delay\_process2\_list[i],

'delay\_process3': delay\_process3\_list[i],

'max\_queue1': maxQ\_list1[i],

'max\_queue2': maxQ\_list2[i],

'max\_queue3': maxQ\_list3[i],

'process1\_processed': p1.quantity,

'process1\_failed': p1.failure,

'process2\_processed': p2.quantity,

'process2\_failed': p2.failure,

'process3\_processed': p3.quantity,

'process3\_failed': p3.failure,

'distribution': distribution[i]}

rows.append({\*\*param, \*\*res})

# назва файлу xlsx

file\_name = 'ModelData.xlsx'

# імпорт в Excel

df = df.append(rows)

df.to\_excel(file\_name)

print(tabulate(df, headers='keys', tablefmt='fancy\_grid', numalign="center"))

Таблиця демонструє, що модель дає очікувані результати при зміні її параметрів. Наприклад, збільшення значення черги призводить до зменшення кількості необроблених елементів, тоді як зменшення цього значення призводить до збільшення кількості необроблених елементів. Так само більші затримки призводять до більшої кількості необроблених елементів на поточному пристрої та меншої кількості на наступних пристроях. І навпаки, зменшення затримок на пристрої зменшує кількість необроблених елементів на ньому. Загалом ці спостереження підтверджують точність моделі.

Таким чином, клас PROCESS був змінений для моделювання процесу обслуговування кількох ідентичних пристроїв за допомогою системи каналів. Це дозволяє одному процесу мати кілька каналів, які можуть виконувати роботу, зменшуючи кількість елементів у черзі та кількість необроблених елементів. Клас також було змінено, щоб дозволити організовувати вивід у кількох наступних блоках, включаючи опцію повернення до попередніх блоків. Це досягається використанням масиву для властивості next\_element і вибором наступної дії на основі ймовірності. Загалом, ці зміни в класі PROCESS представляють значний відхід від стандартної реалізації, яка обговорюється в лекційних матеріалах.:

Зробимо модифікацію алгоритму імітації простої моделі обслуговування одним пристроєм. Тепер пристрій буде мати два потоки:

class Process(e.Element):

def \_\_init\_\_(self, delay, channels=1):

super().\_\_init\_\_(delay)

self.queue = 0

self.max\_observed\_queue = 0

self.max\_queue = float('inf')

self.mean\_queue = 0.0

self.failure = 0

self.mean\_load = 0

self.channel = channels

self.t\_next = [np.inf]\*self.channel

self.state = [0]\*self.channel

self.probability = [1]

def in\_act(self):

free\_route = self.get\_free\_channels()

if len(free\_route) > 0:

for i in free\_route:

self.state[i] = 1

self.t\_next[i] = self.t\_curr + super().get\_delay()

break

else:

if self.queue < self.max\_queue:

self.queue += 1

else:

self.failure += 1

def out\_act(self):

current\_channel = self.get\_current\_channel()

for i in current\_channel:

super().out\_act()

self.t\_next[i] = np.inf

self.state[i] = 0

if self.queue > 0:

self.queue -= 1

self.state[i] = 1

self.t\_next[i] = self.t\_curr + self.get\_delay()

if self.next\_element is not None:

next\_el = np.random.choice(a=self.next\_element, p=self.probability)

next\_el.in\_act()

def get\_free\_channels(self):

free\_channels = []

for i in range(self.channel):

if self.state[i] == 0:

free\_channels.append(i)

return free\_channels

def get\_current\_channel(self):

current\_channels = []

for i in range(self.channel):

if self.t\_next[i] == self.t\_curr:

current\_channels.append(i)

return current\_channels

def print\_info(self):

super().print\_info()

print(f'failure = {str(self.failure)}, queue\_length = {str(self.queue)}')

def calculate(self, delta):

self.mean\_queue += self.queue \* delta

if self.queue > self.max\_observed\_queue:

self.max\_observed\_queue = self.queue

for i in range(self.channel):

self.mean\_load += self.state[i] \* delta

self.mean\_load = self.mean\_load / self.channel

p1.max\_queue = 5

c.distribution = 'exp'

p1.distribution = 'exp'

c.name = 'Creator'

p1.name = 'Process 1'

c.next\_element = [p1]

elements = [c, p1]

model = Model(elements)

model.simulate(1000)

Отримаємо наступні результати:

Graphical user interface, text

Description automatically generated

Використовуючи канали, ми змогли досягти майже нульового навантаження на пристрої, оскільки завдання розподілялися між ними. Ми можемо вдосконалити систему, додавши ще два пристрої. Тепер Process1 виводитиме на Process2 і Process3 одночасно, з однаковою ймовірністю переходу на будь-який пристрій :

c = Create(5)

p1 = Process(5)

p2 = Process(5)

p3 = Process(5)

c.next\_element = [p1]

p1.next\_element = [p2, p3]

p1.probability = ([0.5, 0.5])

p1.max\_queue = 5

p2.max\_queue = 5

p3.max\_queue = 5

c.distribution = 'exp'

p1.distribution = 'exp'

p2.distribution = 'exp'

p3.distribution = 'exp'

c.name = 'Creator'

p1.name = 'Process 1'

p2.name = 'Process 2'

p3.name = 'Process 3'

elements = [c, p1, p2, p3]

model = Model(elements)

model.simulate(1000)

Shape

Description automatically generated with medium confidence

Як можна побачити, дійсно, тепер задачі розподілені 50/50 між другим та третім пристроєм.

Висновок

У цій лабораторії ми успішно розробили об’єктно-орієнтований алгоритм для моделювання простої моделі сервісу з одним пристроєм. Ми модифікували його, щоб дозволити створювати більш складні моделі, і перевірили точність моделі, змінивши вхідні змінні та параметри моделі. Результати показали, що модель правильно побудована і функціонує як задумано.