

Міністерство освіти і науки України

Національний технічний університет України „КПІ імені Ігоря Сікорського ”

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

|  |  |
| --- | --- |
| Перевірила:  Стеценко Інна Вячеславівна | Виконав:  Група ІТ-94  Котяй Данило |

Київ 2022

**Завдання**

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 2.1. **30 балів**.

4. Виконати верифікацію моделі, змінюючи значення вхідних змінних та

параметрів моделі. Навести результати верифікації у таблиці. **10 балів**.

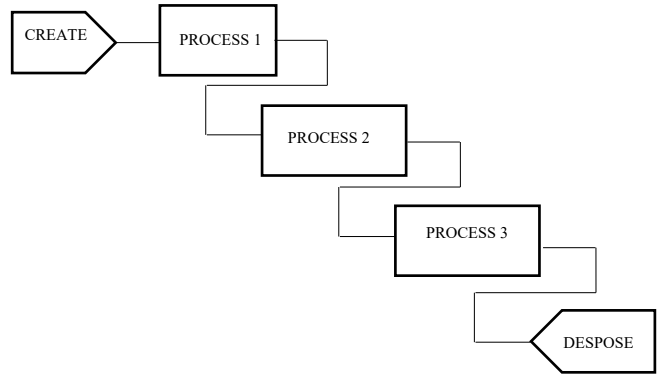


Рисунок 2.1 – Схема моделі.

5. Модифікувати клас PROCESS, щоб можна було його використовувати

для моделювання процесу обслуговування кількома ідентичними

пристроями. **20 балів.**

6. Модифікувати клас PROCESS, щоб можна було організовувати вихід в

два і більше наступних блоків, в тому числі з поверненням у попередні

блоки. **30 балів.**

**Хід роботи**

1. Реалізувати алгоритм імітації простої моделі обслуговування одним пристроєм з використанням об’єктно-орієнтованого підходу.

Використаємо мову програмування Python для лекційного прикладу. Лістинг коду за посиланням на GitHub:

def task1(self):

creator = Create(5)

p1 = Process(5)

p1.maxqueue = 5

creator.distribution = 'exp'

p1.distribution = 'exp'

creator.name = 'Creator'

p1.name = 'Process 1'

creator.nextElement = [p1]

elements = [creator, p1]

model = Model(elements)

res = model.simulate(1000)

Тут ми збираємося запустити симуляцію на проміжку 1000, а також робимо затримку 5 для процесів і застосовуємо експоненційний розподіл. Ліміт для черги буде 5 елементів, також задаємо кожному елементу своє ім’я.

Запускаємо функцію:

from simModel import Sim

sim = Sim()

sim.task1()

Отриманий результат:

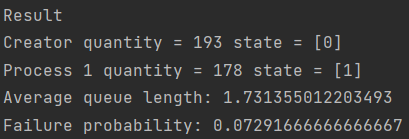


Рисунок 1 – результат виконання функції

В результаті можемо бачити:

* **Creator quantity** - кількість опрацьованих елементів процесом CREATE
* **Process 1 quantity** - кількість опрацьованих процесом PROCESS
* **Average queue length** - середня довжина черги для PROCESS
* **Failure probability** - ймовірність помилки завдання для PROCESS

2. Модифікувати алгоритм, додавши обчислення середнього завантаження пристрою

Робимо обчислення середнього навантаження і виводимо його:

def doStatistics(self, delta):

self.meanqueue += self.queue \* delta

for i in range(self.threads):

self.meanload += self.state[i] \* delta

self.meanload /= self.threads

print(f"Average load: {self.get\_meanload(e)}")

3. Створити модель за схемою, представленою на рисунку 2.1

Знову запускаємо симуляцію на проміжку 1000, а також робимо затримку 5 для процесів і застосовуємо експоненційний розподіл. Ліміт для черги буде 5 елементів, також задаємо кожному елементу своє ім’я.

def task3(self):

creator = Create(5)

p1 = Process(5)

p2 = Process(5)

p3 = Process(5)

creator.nextElement = [p1]

p1.nextElement = [p2]

p2.nextElement = [p3]

p1.maxqueue = 5

p2.maxqueue = 5

p3.maxqueue = 5

creator.distribution = 'exp'

p1.distribution = 'exp'

p2.distribution = 'exp'

p3.distribution = 'exp'

creator.name = 'Creator'

p1.name = 'Process 1'

p2.name = 'Process 2'

p3.name = 'Process 3'

elements = [creator, p1, p2, p3]

model = Model(elements)

res = model.simulate(1000)

В цій реалізації ми тепер маємо два нових процеси.

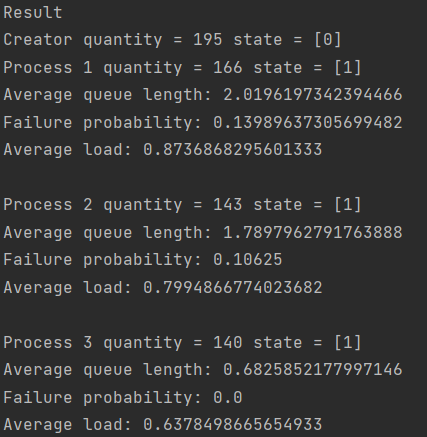


Рисунок 2 – результат виконання функції

4. Виконати верифікацію моделі, змінюючи значення вхідних змінних та

параметрів моделі. Навести результати верифікації у таблиці

def task4(self):

creator\_delays = [12, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 1, 6]

process\_delays = [

[6, 12, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 1, 6, 6],

[6, 6, 12, 6, 6, 6, 6, 6, 6, 6, 6, 1, 6, 6, 6],

[6, 6, 6, 12, 6, 6, 6, 6, 6, 6, 1, 6, 6, 6, 6],

]

maxqueue\_values = [

[4, 4, 4, 4, 12, 4, 4, 4, 4, 1, 4, 4, 4, 4, 4],

[4, 4, 4, 4, 4, 12, 4, 4, 1, 4, 4, 4, 4, 4, 4],

[4, 4, 4, 4, 4, 4, 12, 1, 4, 4, 4, 4, 4, 4, 4],

]

distribution = 'exp'

result\_array = []

for i in range(len(creator\_delays)):

creator = Create(creator\_delays[i])

p1 = Process(process\_delays[0][i])

p2 = Process(process\_delays[1][i])

p3 = Process(process\_delays[2][i])

p1.maxqueue = maxqueue\_values[0][i]

p2.maxqueue = maxqueue\_values[1][i]

p3.maxqueue = maxqueue\_values[2][i]

creator.distribution = distribution

p1.distribution = distribution

p2.distribution = distribution

p3.distribution = distribution

creator.name = 'Creator'

p1.name = 'Process 1'

p2.name = 'Process 2'

p3.name = 'Process 3'

creator.nextElement = [p1]

p1.nextElement = [p2]

p2.nextElement = [p3]

elements = [creator, p1, p2, p3]

model = Model(elements)

res = model.simulate(1000)

i\_result = [

creator\_delays[i],

process\_delays[0][i],

process\_delays[1][i],

process\_delays[2][i],

maxqueue\_values[0][i],

maxqueue\_values[1][i],

maxqueue\_values[2][i],

p1.quantity,

p1.failure,

p2.quantity,

p2.failure,

p3.quantity,

p3.failure,

]

result\_array.append(i\_result)

columns = [

'Creator Delay',

'Process Delay 1',

'Process Delay 2',

'Process Delay 3',

'Max Queue 1',

'Max Queue 2',

'Max Queue 3',

'Process 1 - success',

'Process 1 - fail',

'Process 2 - success',

'Process 2 - fail',

'Process 3 - success',

'Process 3 - fail',

]

df = pd.DataFrame(result\_array, columns=columns)

pd.set\_option('display.max\_columns', None)

print(f"Distribution: {distribution}")

print(df)

Змінюємо значення для того щоб верифікувати модель, для цього використовуємо масиви delays, щоб виконувати затримки, також існує масив який відповідає за ліміт черги.

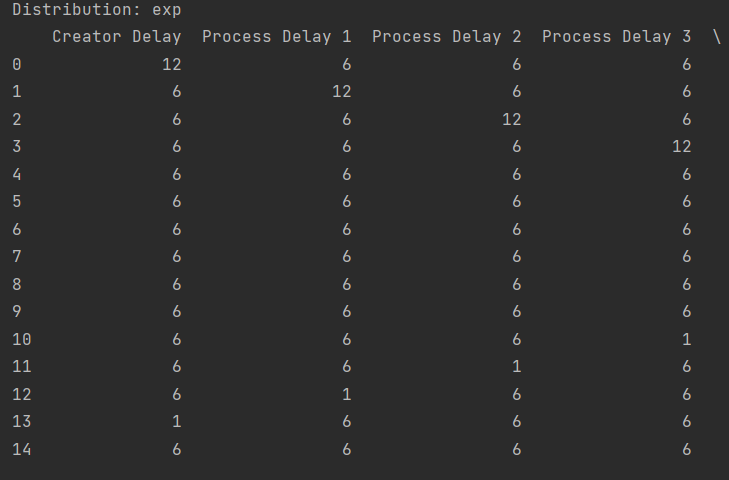


Рисунок 3 – результат верифікації моделі, частина перша

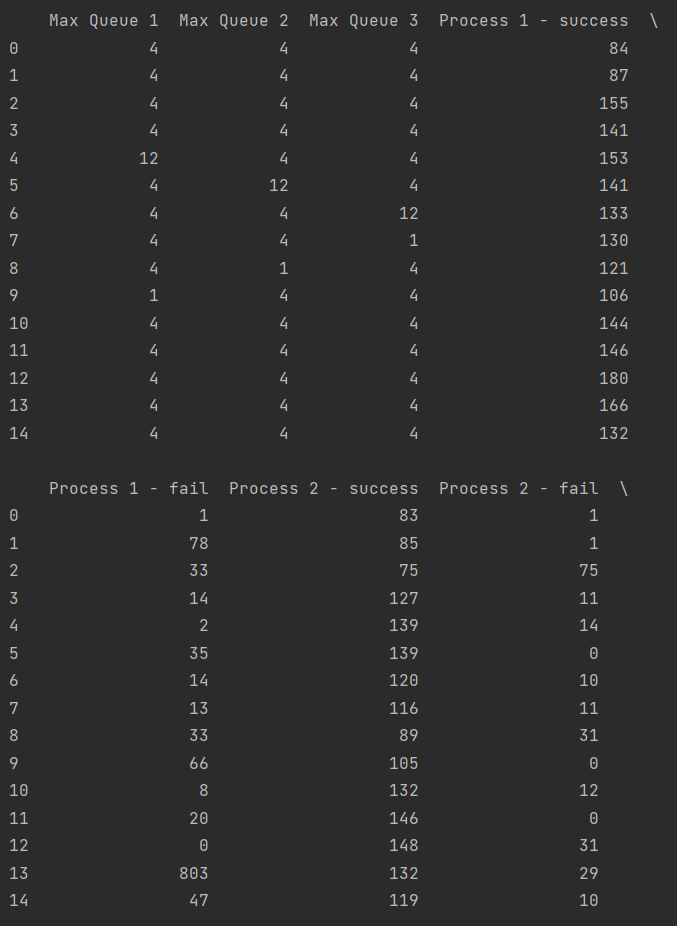


Рисунок 4 – результат верифікації моделі, частина друга

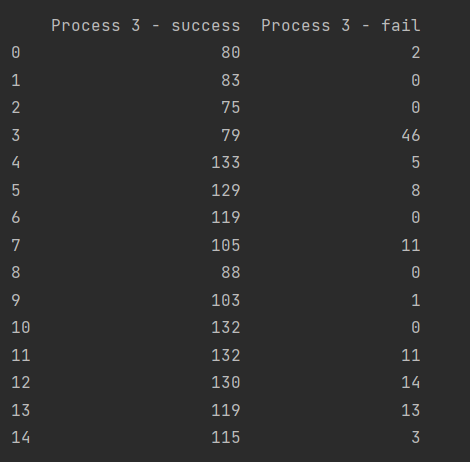


Рисунок 5 – результат верифікації моделі, частина третя

Дивлячись на результат можемо зробити наступні висновки:

* підвищуючи кількість в черзі, отримуємо покращення відповідності успішних процесів до провалених
* при кількості 12 в черзі 1 маємо значне збільшення відсотку успішних
* зменшуючи кількість до 1, маємо протилежний результат
* збільшуючи час затримки, збільшиться кількість завалених процесів
* зменшуючи час затримки, зменшиться кількість завалених процесів

Отже, модель працює правильно.

5-6. Модифікувати клас PROCESS, щоб можна було його використовувати для моделювання процесу обслуговування кількома ідентичними пристроями. Модифікувати клас PROCESS, щоб можна було організовувати вихід в два і більше наступних блоків, в тому числі з поверненням у попередні блоки.

Потрібно створити багатопоточну систему, таким чином, якщо один потік уже зайнятий, то роботу може виконувати інший потік. Це покращує кількість елементів в черзі. Створюємо масив наступних елементів та систему переходу між процесами:

import numpy as np

from element import Element

class Process(Element):

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

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

self.queue = 0

self.maxqueue = float('inf')

self.meanqueue = 0.0

self.failure = 0

self.threads = threads

self.tnext = [np.inf] \* self.threads

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

self.probability = [1]

self.meanload = 0

def inAct(self):

freeThreads = self.getAvailable()

if len(freeThreads) > 0:

for i in freeThreads:

self.state[i] = 1

self.tnext[i] = self.tcurr + super().getDelay()

break

else:

if self.queue < self.maxqueue:

self.queue += 1

else:

self.failure += 1

def outAct(self):

threads = self.getCurrent()

for i in threads:

super().outAct()

self.tnext[i] = np.inf

self.state[i] = 0

if self.queue > 0:

self.queue -= 1

self.state[i] = 1

self.tnext[i] = self.tcurr + self.getDelay()

if self.nextElement is not None:

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

next\_el.inAct()

def getAvailable(self):

threads = []

for i in range(self.threads):

if self.state[i] == 0:

threads.append(i)

return threads

def getCurrent(self):

threads = []

for i in range(self.threads):

if self.tnext[i] == self.tcurr:

threads.append(i)

return threads

def printInfo(self):

super().printInfo()

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

def doStatistics(self, delta):

self.meanqueue += self.queue \* delta

for i in range(self.threads):

self.meanload += self.state[i] \* delta

self.meanload /= self.threads

def task5\_6(self):

creator = Create(5)

p1 = Process(5, 2)

p1.maxqueue = 5

creator.distribution = 'exp'

p1.distribution = 'exp'

creator.name = 'Creator'

p1.name = 'Process 1'

creator.nextElement = [p1]

elements = [creator, p1]

model = Model(elements)

res = model.simulate(1000)

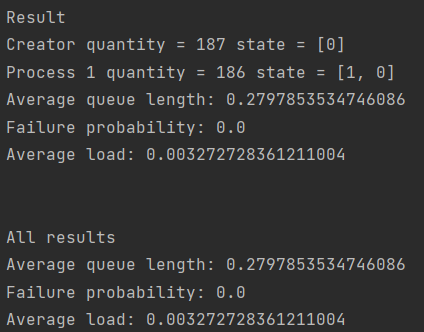


Рисунок 6 – результати виконання модифікацій

Можемо зробити висновок, що навантаження майже дорівнює нулю.

**Висновок**

В цій лабораторній роботі ми реалізували алгоритм імітації простої моделі обслуговування одним пристроєм з використанням об’єктно-орієнтованого підходу, а також перевірили працездатність за допомогою різних значень. Потім модифікували алгоритм для створення розгалужень в більш складних моделях, перевірили та впевнилися, що наші моделі працюють та виконують свої задачі вірно.