

**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

**Перевірила:**

Асистент

Бернатович Анатолій Олександрович

Оцінка:\_\_\_\_\_\_\_\_\_\_\_

**Виконав:**

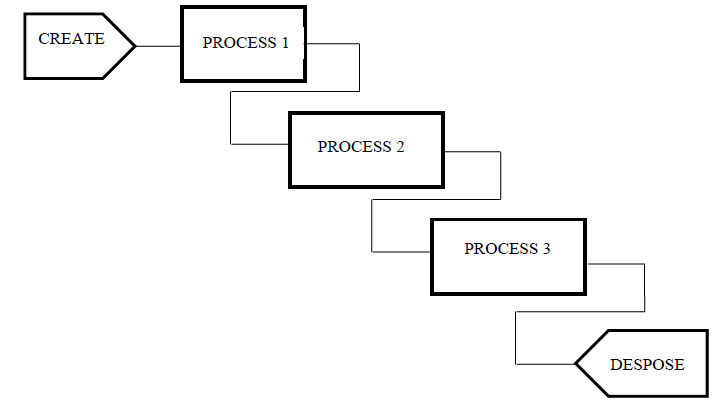
Студент групи ІТ-92

Рафалюк Р.Б.

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.**



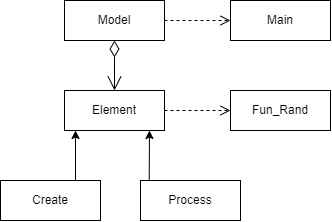
1. Виконати верифікацію моделі, змінюючи значення вхідних змінних та параметрів моделі. Навести результати верифікації у таблиці. **10 балів.**
2. Модифікувати клас PROCESS, щоб можна було його використовувати для моделювання процесу обслуговування кількома ідентичними пристроями. **20 балів.**
3. Модифікувати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

**Виконання Лабораторної роботи**

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



Приблизна структура об’єктно-орієнтованої програми:



Модель складається з елементів, які є нащадками одного універсального типу Element. Цей клас містить основні поля та методи елементу моделі такі, як tcurr (поточний момент часу), tnext (момент часу наступної події), delayMean (середеє значення часової затримки), delayDev(середнє квадратичне відхилення часової затримки), getDelay() (розрахунок часової затримками), inAct() (вхід в елемент), outAct() (вихід з елементу). Використання універсального класу надає можливість уніфікувати використання різних елементів в імітаційній програмі.

Важливим для з’єднання елементів в єдину модель є поле nextElement, що вказує на наступний (в маршруті слідування вимоги) елемент моделі.

Клас Model містить метод simulate(double time), що здійснює імітацію на інтервалі часу time. Імітація здійснюється за відомим з попередньої теми принципом: визначається момент найближчої події, просувається час в момент найближчої події та здійснюється відповідна подія. Щоб зменшити обсяг обчислень, введемо також здійснення відповідної події для всіх елементів, час наступної події яких співпадає з поточним моментом часу. Оскільки ми не розглядаємо можливість присвоєння пріоритету елементам моделі, то ця дія цілком допустима.

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.

Побудова данної моделі мовою Python:

c = Create(5)

proc = Process(5)

proc.max\_queue = 5

c.distribution = 'exp'

proc.distribution = 'exp'

c.name = 'Creator'

proc.name = 'Process 1'

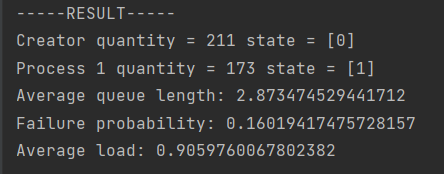
c.next\_element = [proc]

elements = [c, proc]

model = Model(elements)

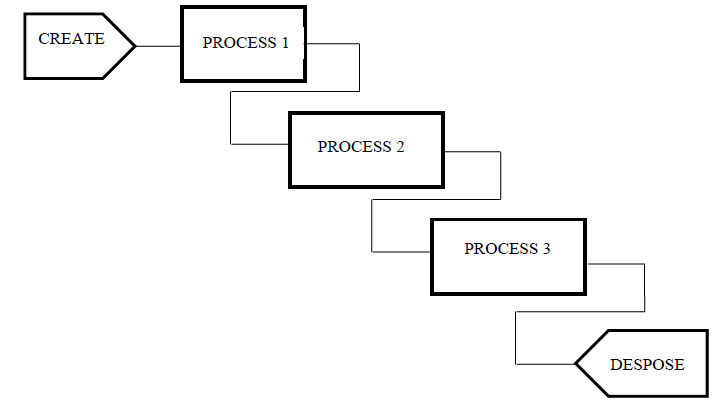
res = model.simulate(1000)

Виконуємо програму та отримаємо наступні результати:



* “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:



n\_param = 15

delay\_create\_list = [2, 10, 1, 6, 7, 8, 1, 3, 0.5, 5, 5, 4, 4, 4, 4]

delay\_process1\_list = [2, 1, 5, 4, 4, 5, 0.7, 0.4, 4, 1, 4, 4, 4, 0.3, 4]

delay\_process2\_list = [2, 1, 4, 4, 4, 4, 4, 4, 6, 4, 0.5, 4, 4, 4, 4]

delay\_process3\_list = [2, 1, 4, 4, 10, 4, 4, 4, 4, 4, 4, 0.7, 4, 4, 4]

maxQ\_list1 = [5, 4, 5, 5, 5, 15, 5, 9, 6, 6, 5, 7, 1, 5, 5]

maxQ\_list2 = [5, 3, 7, 4, 5, 8, 1, 5, 5, 6, 7, 5, 8, 1, 5]

maxQ\_list3 = [5, 5, 5, 8, 5, 6, 5, 10, 5, 6, 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])

proc1 = Process(delay\_process1\_list[i])

proc2 = Process(delay\_process2\_list[i])

proc3 = Process(delay\_process3\_list[i])

proc1.max\_queue = maxQ\_list1[i]

proc2.max\_queue = maxQ\_list2[i]

proc3.max\_queue = maxQ\_list3[i]

c.distribution = distribution[i]

proc1.distribution = distribution[i]

proc2.distribution = distribution[i]

proc3.distribution = distribution[i]

c.name = 'Creator'

proc1.name = 'Process 1'

proc2.name = 'Process 2'

proc3.name = 'Process 3'

c.next\_element = [proc1]

proc1.next\_element = [proc2]

proc2.next\_element = [proc3]

elements = [c, proc1, proc2, proc3]

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': proc1.quantity,

'process1\_failed': proc1.failure,

'process2\_processed': proc2.quantity,

'process2\_failed': proc2.failure,

'process3\_processed': proc3.quantity,

'process3\_failed': proc3.failure,

'distribution': distribution[i]}

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

# file xlsx

file\_name = 'ModelData.xlsx'

# import to Excel

df = df.append(rows)

df.to\_excel(file\_name)

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

Після запуску програми отримуємо результати і наступну таблицю exel



Як можемо бачити, при зміні параметрів моделі відбуваються очікувані зміни у результатах. Так, при піднятті значень черг, значення не опрацьованих елементів буде знижуватися. Або при піднятті затримки, кількість не опрацьованих елементів на поточному пристрої буде високою, тоді як на наступних – низькою. Також це працює і в зворотньому напрямі, тому можна зробити висновок, що модель працює правильно.

Далі ми модифікували клас PROCESS, щоб його можна було використовувати для моделювання процесу обслуговування кількома ідентичними пристроями. Для цього було обрано метод системи каналів - у одного Process може бути декілька каналів які будуть виконувати роботу.

Разом із цим ми також модифікуємо клас PROCESS, щоб можна було організовувати вихід в два і більше наступних блоків, в тому числі з поверненням у попередні блоки. Для цього ми зробили властивість next\_element масивом у якому будуть зберігатися наступні елементи і в залежності від деякої ймовірності буде обиратися деяка наступна дія.

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

Дещо змінимо алгоритм імітації простої моделі обслуговування одним пристроєм, для перевірки модифікацій класу Process.

c = Create(5)

proc = Process(5,2)

proc.max\_queue = 5

c.distribution = 'exp'

proc.distribution = 'exp'

c.name = 'Creator'

proc.name = 'Process 1'

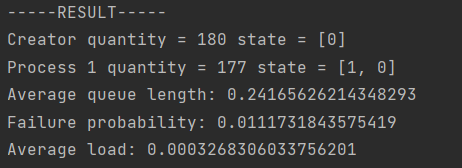
c.next\_element = [proc]

elements = [c, proc]

model = Model(elements)

res = model.simulate(1000)

Після виконання програми отримаємо наступні результати



Можна бачити що із використанням каналів ми отримали в майже нульове значення завантаження пристрою.

Додаймо ще два пристрої. Тепер Process1 буде виходити у декілька пристроїв (Process2 та Process3) одночасно. Шанси потрапляння у кожний пристрій будуть 50 на 50:

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'

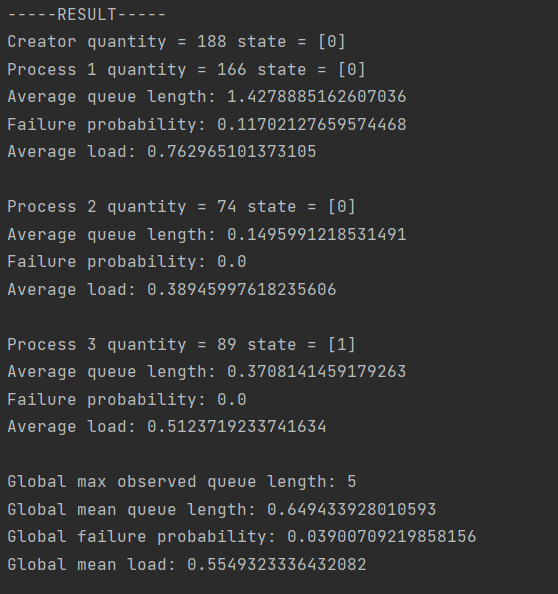
p2.name = 'Process 2'

p3.name = 'Process 3'

elements = [c, p1, p2, p3]

model = Model(elements)

res = model.simulate(1000)



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

**Висновок**

У даній лабораторній роботі ми успішно побудували алгоритм імітації простої моделі обслуговування одним пристроєм з використанням об’єктно-орієнтованого підходу. Модифікували його для можливості будувати більш складні моделі. Провели верифікацію моделі, змінюючи значення вхідних змінних та параметрів моделі та зрозуміли, що модель побудована та працює вірно.

Посилання на гітхаб: <https://github.com/RomBik17/modeling.git>