Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра программного обеспечения информационных технологий

Дисциплина: Системный анализ и машинное моделирование (САиММод)

ОТЧЕТ

по лабораторной работе №4

Тема работы: Построение и исследование аналитической модели системы массового обслуживания

Выполнил

студент: гр.651004 Ласый И. Е.

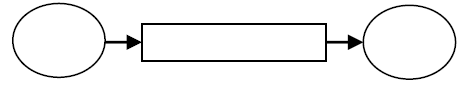
Проверил: Мельник Н. И.

Минск 2019

# Постановка задачи

**Вариант 1б**

λ=2.5 µ=3



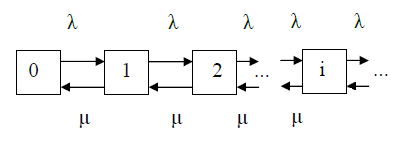
Исследовать значения средней длины очереди, среднего времени ожидания в очереди, среднего времени пребывания заявки в системе при следующих дисциплинах обслуживания заявок из очереди:

a) FIFO;

б) первыми обслуживаются заявки, требующие меньшего времени обслуживания.

Построить аналитическую и имитационную модели и сравнить результаты исследования для дисциплины а); построить имитационную модель для дисциплины б).

# Диаграмма интенсивности переходов



# Вычисления

Так как по условию λ < µ, можно вывести формулы финальных вероятностей системы:

Отношение интенсивностей:

Необходимые характеристики эффективности:

* Среднее число заявок в очереди

* Среднее время нахождения заявки в очереди

* Среднее время нахождения заявки в системе

# Результаты имитационной модели

**FIFO:**

L=4.182472486616473

Wq=1.6721517868089777

Ws=2.005956813325098

Job time: 39978.90928199255

Output order amount: 99994

**Каждая заявка имеет своё время обработки:**

L=1.7904986043805222

Wq=0.7171029803304049

Ws=1.0503299117987115

Job time: 40049.543676662004

Output order amount: 99997

# Исходный код программы

class Generator:

def \_\_init\_\_(self,intensity):

self.intensity = intensity

def get\_time(self):

return np.random.exponential(1 / self.intensity)

def get\_events(self, count):

timings = []

timings.append(Event('GENERATOR', self.get\_time()))

for i in range(1, count):

timings.append(Event('GENERATOR', timings[-1].time\_value + self.get\_time()))

return timings

class Queue:

def \_\_init\_\_(self, pick\_fastest=False):

self.queue = []

self.len\_times = []

self.time\_of\_change = 0

self.pick\_fastest = pick\_fastest

@property

def count(self):

return len(self.queue)

def update\_times(self,time):

self.len\_times.append((self.count, time - self.time\_of\_change))

self.time\_of\_change = time

def add(self, order, time):

self.update\_times(time)

self.queue.append(order)

info(f"QUEUE - added order at {time}; ({self.count})")

def pop(self, time):

info(f"QUEUE - poped order at {time}; ({self.count-1})")

self.update\_times(time)

idx = 0

if self.pick\_fastest:

idx = self.queue.index(min(self.queue, key=lambda x: x.time\_needed))

return self.queue.pop(idx)

def inc(self, time):

for order in self.queue:

order.inc\_queue(time)

def get\_avg\_len(self, job\_time):

result = 0

for length, time in self.len\_times:

result += length \* time

info("--List of queue times--")

info(self.len\_times[:100])

return result / job\_time

class Processor:

def \_\_init\_\_(self, intensity):

self.intensity = intensity

self.current\_order = None

self.current\_time = 0

self.process\_time = sys.float\_info.max

def start(self, order, time=None):

self.current\_order = order

interval = np.random.exponential(1 / self.intensity) if order.time\_needed == -1 else order.time\_needed

self.current\_time = self.current\_time if time is None else time

self.process\_time = self.current\_time + interval

self.current\_order.inc\_system(interval)

info(f"PROCESSOR - started work at {self.current\_time}")

return Event('PROCESSOR', self.process\_time)

def end(self):

self.current\_time = self.process\_time

order = self.current\_order

self.current\_order = None

return order

def check\_end(self, time):

if time > self.process\_time:

self.current\_time = self.process\_time

order = self.current\_order

self.current\_order = None

info(f"PROCESSOR - ended work at {self.current\_time}")

return order

return None

def is\_free(self):

return self.current\_order is None

class Event:

def \_\_init\_\_(self, event\_type, time\_value):

self.type = event\_type

self.time\_value = time\_value

def \_\_lt\_\_(self, other):

return self.time\_value < other.time\_value

class Order:

def \_\_init\_\_(self, time, generate\_time = False):

self.entry\_time = time

self.time\_in\_queue = 0

self.time\_in\_system = 0

self.time\_needed = -1

if generate\_time:

self.time\_needed = np.random.exponential(1 / 3)

def inc\_queue(self, time):

self.time\_in\_queue += time

self.time\_in\_system = self.time\_in\_queue

def inc\_system(self, time):

self.time\_in\_system += time

import bisect

count = 100000

input\_intensity = 2.5

output\_intensity = 3

class System:

def \_\_init\_\_(self, input\_intensity=input\_intensity,output\_intensity=output\_intensity, orders\_with\_time=False,seed=None):

self.time = 0

self.orders\_with\_time = orders\_with\_time

self.output\_orders = []

self.generator = Generator(input\_intensity)

self.processor = Processor(output\_intensity)

self.queue = Queue(self.orders\_with\_time)

np.random.seed(seed)

def on(self, count=count):

events = self.generator.get\_events(count)

self.time = events[0].time\_value

final\_value = events[-1].time\_value

process\_event = self.processor.start(Order(self.time, self.orders\_with\_time), self.time)

bisect.insort\_left(events, process\_event)

i = 1

while self.time < final\_value:

self.time = events[i].time\_value

info(f"SYSTEM - current time: {self.time} -- {events[i].type}")

self.queue.inc(self.time - events[i-1].time\_value)

if events[i].type == 'GENERATOR':

new\_order = Order(self.time, self.orders\_with\_time)

if self.processor.is\_free():

process\_event = self.processor.start(new\_order, self.time)

bisect.insort\_left(events, process\_event)

else:

self.queue.add(new\_order, self.time)

if events[i].type == 'PROCESSOR':

processed\_order = self.processor.end()

self.output\_orders.append(processed\_order)

if self.queue.count > 0:

process\_event = self.processor.start(self.queue.pop(self.time),self.time)

bisect.insort\_left(events, process\_event)

i += 1

print(f"L={self.queue.get\_avg\_len(self.time)}")

print(f"Wq={self.get\_avg\_order\_time\_in\_queue()}")

print(f"Ws={self.get\_avg\_order\_time\_in\_system()}")

print(f"Job time: {self.time}")

print(f"Output order amount: {len(self.output\_orders)}")

def get\_avg\_order\_time\_in\_queue(self):

result = 0

for order in self.output\_orders:

result += order.time\_in\_queue

return result / len(self.output\_orders)

def get\_avg\_order\_time\_in\_system(self):

result = 0

for order in self.output\_orders:

result += order.time\_in\_system

return result / len(self.output\_orders)