МІНІСТЕРСТВО ОСВІТИ І НАУКИ, МОЛОДІ ТА СПОРТУ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО» КАФЕДРА АСОІУ

ЗВІТ

про виконання лабораторної роботи № 3

з дисципліни «Сучасні операційні системи»

*Виконала:*

*студентка групи ІС-72*

*Красножон К.Г.*

*Перевірив:*

*Сімоненко А.В.*

|  |  |
| --- | --- |
|  |  |

Київ 2020

**Тема**: Дослідження дисциплін обслуговування заявок при обмежених ресурсах

**Мета**: розробити програму обслуговування заявок за алгоритмом Foregraund Backgraund.

**Теорія**

Алгоритм має N черг. Вхідний потік заявок поступає в першу чергу. Із черг заявки поступають на виконання.

Якщо заявка за відведений квант часу не встигла завершитися, то вона повертається в чергу i+1, де i – черга з якої заявка була взята. З найбільш високим пріоритетом черга No1. Черга i обслуговується, якщо порожні всі черги котрі менші за i. Починаючи з 2-ої черги заявки сортуються за пріоритетом (0 пріоритет < 1 пріоритету).

Квант часу для заявки визначається по формулі 2i-1, заявка з останньої черги обслуговується стільки часу, скільки їй необхідно до завершення.

**Лістинг коду мовою Python**

----- **task.py** -------------------------------------------------------------------------------------------------------------------------------------------------------------

from random import randint

TIME\_UNIT = 10

class Task:

def \_\_init\_\_(self, max\_runtime):

self.done = False

self.arrival = randint(0, TIME\_UNIT - 1)

self.runtime = randint(1, max\_runtime)

def \_\_str\_\_(self):

done = self.done

arrival = self.arrival

runtime = self.runtime

return f'{{ arrival: {arrival}, runtime: {runtime}, done: {done} }}'

---- **queue.py** -----------------------------------------------------------------------------------------------------------------------------------------------------------

from math import ceil, factorial, exp

from random import randrange

from task import \*

TIME\_UNIT = 10

class Queue:

def \_\_init\_\_(self, tasks\_amount, arrival\_rate, max\_runtime):

timeline\_stock = max\_runtime \* tasks\_amount

self.sum\_await\_time = 0

self.downtime = 0

self.intervals = set()

self.x\_await\_time = []

self.y\_tasks\_amount = []

self.x\_interval\_probability = []

self.y\_interval\_probability = []

self.x\_tasks\_unit\_probability = []

self.y\_tasks\_unit\_probability = []

self.busy = 0

self.max\_runtime = max\_runtime

self.tasks\_amount = tasks\_amount

self.arrival\_rate = arrival\_rate

self.arrival\_density = arrival\_rate / TIME\_UNIT

self.units = ceil(tasks\_amount / arrival\_rate)

self.beats = self.units \* TIME\_UNIT + timeline\_stock

self.timeline = [[] for i in range(self.beats)]

self.last\_active\_beat = self.beats - 1

def \_\_str\_\_(self):

beat\_tasks = lambda bt: ', '.join(map(lambda t: str(t), bt))

beat\_to\_string = lambda bt: '\t[{}]'.format(beat\_tasks(bt))

timeline = ',\n'.join(map(beat\_to\_string, self.timeline))

return f'[\n{timeline}\n]'

def init\_timeline(self):

for unit in range(self.units):

remainder = self.tasks\_amount - unit \* self.arrival\_rate

tasks\_unit\_amount = self.arrival\_rate \

if remainder > self.arrival\_rate \

else remainder

for i in range(tasks\_unit\_amount):

task = Task(self.max\_runtime)

beat\_index = task.arrival + unit \* TIME\_UNIT

beat\_tasks = self.timeline[beat\_index]

task.abs\_arrival = beat\_index

beat\_tasks.append(task)

def execute(self, task):

task.done = True

self.busy = task.runtime - 1

def pick\_task(self, beat):

timeline = self.timeline[0:beat]

tasks = [task for beat in timeline for task in beat if not task.done]

if len(tasks):

random\_task = tasks[randrange(0, len(tasks))]

await\_time = beat - random\_task.arrival

abs\_await\_time = beat - random\_task.abs\_arrival

self.sum\_await\_time += abs\_await\_time

self.execute(random\_task)

self.x\_await\_time.append(await\_time)

self.y\_tasks\_amount.append(len(tasks) - 1)

self.intervals.add(random\_task.runtime)

else:

timeline = self.timeline[beat:len(self.timeline)]

upcoming\_tasks = [task for beat in timeline for task in beat]

if len(upcoming\_tasks):

self.downtime += 1

else:

self.last\_active\_beat = beat

def calculate\_interval\_probability(self):

for interval in self.intervals:

p = self.arrival\_density \* exp(-self.arrival\_density \* interval)

self.y\_interval\_probability.append(p)

self.x\_interval\_probability.append(interval)

def calculate\_tasks\_per\_unit\_probability(self):

for task in range(self.tasks\_amount):

p = (

((self.arrival\_density \* TIME\_UNIT) \*\* task) /

factorial(task)

) \* exp(-self.arrival\_density \* TIME\_UNIT)

self.x\_tasks\_unit\_probability.append(task)

self.y\_tasks\_unit\_probability.append(p)

def start(self):

for beat in range(1, self.beats + 1):

if self.busy:

self.busy -= 1

continue

else:

self.pick\_task(beat)

self.calculate\_interval\_probability()

# self.calculate\_tasks\_per\_unit\_probability()

self.avg\_await\_time = self.sum\_await\_time / self.tasks\_amount

self.downtime\_percent = (self.downtime / self.last\_active\_beat) \* 100

---- **graph.py** -----------------------------------------------------------------------------------------------------------------------------------------------------------

import scipy as sp

import matplotlib.pyplot as plt

class Graph:

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

self.x = []

self.y = []

def title(self, title):

plt.title(title)

def set\_values(self, x, y):

self.x.append(x)

self.y.append(y)

def y\_label(self, label):

plt.ylabel(label)

def x\_label(self, label):

plt.xlabel(label)

def show(self, smooth = False):

if (smooth):

p = sp.polyfit(self.x, self.y, deg=5)

y\_ = sp.polyval(p, self.x)

plt.plot(self.x, y\_)

plt.show()

else:

plt.plot(self.x, self.y)

plt.show()

def show\_hist(self, values):

plt.hist(values, bins=15)

plt.show()

---- **cli.py** -----------------------------------------------------------------------------------------------------------------------------------------------------------

import sys

from queue import \*

from graph import \*

help = """Commands:

- exit: exit CLI

- <number of graph>: select plot to build

Available graphs:

1. plot of probability of interval's between task processing versus interval

2. plot of tasks amount versus awaiting time

3. plot of average awaiting time versus arrival rate

4. plot of percent of downtime versus arrival rate

5. plot of tasks per unit probability versus tasks\_amount

"""

def emulation\_params():

tasks\_amount = input('Enter tasks amount: ')

arrival\_rate = input('Enter arrival rate: ')

max\_runtime = input('Enter max task runtime: ')

return tasks\_amount, arrival\_rate, max\_runtime

def cli():

print(help)

while True:

command = input('Enter command: ')

if command == 'exit':

return

elif command == '1':

graph = Graph()

graph.title("Probability of interval's between task processing versus interval")

graph.y\_label('Probability p(t)')

graph.x\_label('Interval t')

tasks\_amount, arrival\_rate, max\_runtime = emulation\_params()

queue = Queue(int(tasks\_amount), int(arrival\_rate), int(max\_runtime))

queue.init\_timeline()

queue.start()

for i in range(len(queue.x\_interval\_probability)):

graph.set\_values(

queue.x\_interval\_probability[i],

queue.y\_interval\_probability[i]

)

graph.show()

elif command == '2':

graph = Graph()

graph.title("Tasks amount versus awaiting time")

graph.y\_label('Tasks amount')

graph.x\_label('Awaiting time')

tasks\_amount, arrival\_rate, max\_runtime = emulation\_params()

queue = Queue(int(tasks\_amount), int(arrival\_rate), int(max\_runtime))

queue.init\_timeline()

queue.start()

for i in range(len(queue.x\_await\_time)):

graph.set\_values(queue.x\_await\_time[i], queue.y\_tasks\_amount[i])

graph.show(True)

# graph.show\_hist(queue.x\_await\_time)

elif command == '3':

graph = Graph()

graph.title("Average awaiting time versus arrival rate")

graph.y\_label('Average awaiting time')

graph.x\_label('Arrival rate (tasks per 10 beats)')

tasks\_amount = input('Enter tasks amount: ')

min\_rate = input('Enter min arrival rate: ')

max\_rate = input('Enter max arrival rate: ')

max\_runtime = input('Enter max task runtime: ')

for rate in range(int(min\_rate), int(max\_rate) + 1):

queue = Queue(int(tasks\_amount), rate, int(max\_runtime))

queue.init\_timeline()

queue.start()

graph.set\_values(rate, queue.avg\_await\_time)

graph.show()

elif command == '4':

graph = Graph()

graph.title("Percent of downtime versus arrival rate")

graph.y\_label('Percent of downtime')

graph.x\_label('Arrival rate')

tasks\_amount = input('Enter tasks amount: ')

min\_rate = input('Enter min arrival rate: ')

max\_rate = input('Enter max arrival rate: ')

max\_runtime = input('Enter max task runtime: ')

for rate in range(int(min\_rate), int(max\_rate) + 1):

queue = Queue(int(tasks\_amount), rate, int(max\_runtime))

queue.init\_timeline()

queue.start()

graph.set\_values(rate, queue.downtime\_percent)

graph.show()

elif command == '5':

graph = Graph()

graph.title("Tasks per unit probability versus tasks amount")

graph.y\_label('Tasks per unit probability')

graph.x\_label('Tasks amount')

tasks\_amount, arrival\_rate, max\_runtime = emulation\_params()

queue = Queue(int(tasks\_amount), int(arrival\_rate), int(max\_runtime))

queue.init\_timeline()

queue.start()

for i in range(len(queue.x\_tasks\_unit\_probability)):

graph.set\_values(

queue.x\_tasks\_unit\_probability[i],

queue.y\_tasks\_unit\_probability[i]

)

graph.show(True)

cli()