НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Кафедра обчислювальної техніки \_

(повна назва кафедри, циклової комісії)

**РОЗРАХУНКОВО-ГРАФІЧНА РОБОТА**

з дисципліни «Інтелектуальні вбудовані системи»

(назва дисципліни)

на тему: «Дослідження роботи планувальників роботи систем реального часу»

Студента 3 курсу групи ІП-83

спеціальності

121 «Інженерія програмного

забезпечення»

Карпюк І.В. \_

(прізвище та ініціали)

Керівник доцент Волокіта А.Н.

Київ – 2021 рік

**Дослідження роботи планувальників роботи систем реального часу**

**Мета роботи** - змоделювати роботу планувальника задач у системі реального часу

**Основні теоретичні відомості**

**Планування виконання завдань** є однією з ключових концепцій в багатозадачності і багатопроцесорних систем, як в операційних системах загального призначення, так і в операційних системах реального часу. Планування полягає в призначенні пріоритетів процесам в черзі з пріоритетами.

Найважливішою метою планування завдань є якнайповніше завантаження доступних ресурсів. Для забезпечення загальної продуктивності системи планувальник має опиратися на:

* Використання процесора(-ів) — дати завдання процесору, якщо це можливо.
* Пропускна здатність — кількість процесів, що виконуються за одиницю часу.
* Час на завдання — кількість часу, для повного виконання певного процесу.
* Очікування — кількість часу, який процес очікує в черзі готових.
* Час відповіді — час, який проходить від подання запиту до першої відповіді на запит.
* Справедливість — Рівність процесорного часу для кожної ниті

У середовищах обчислень реального часу, наприклад, на пристроях, призначених для автоматичного управління в промисловості (наприклад, робототехніка), планувальник завдань повинен забезпечити виконання процесів в перебігу заданих часових проміжків (час відгуку); це критично для підтримки коректної роботи системи реального часу.

**Завдання на лабораторну роботу**

1. Змоделювати планувальник роботи системи реального часу. Дві дисципліни планування: перша – RR, друга задається викладачем або обирається самостійно.

2. Знайти наступні значення:

1) середній розмір вхідної черги заявок, та додаткових черг (за їх наявності);

2) середній час очікування заявки в черзі;

3) кількість прострочених заявок та її відношення до загальної кількості заявок

3. Побудувати наступні графіки:

1) Графік залежності кількості заявок від часу очікування при фіксованій інтенсивності вхідного потоку заявок.

2) Графік залежності середнього часу очікування від інтенсивності вхідного потоку заявок.

3) Графік залежності проценту простою ресурсу від інтенсивності вхідного потоку заявок

**Лістинг програми**

earliest\_deadline\_first.py

import random  
import pylab  
import math  
  
"""edf"""  
  
  
class Task:  
 def \_\_init\_\_(self, time, la):  
 self.time\_to\_solve = MX  
 self.deadline = random.randint(2, MAX\_K) \* self.time\_to\_solve  
 self.appearing\_time = time - 1 / la \* math.log(random.random())  
  
  
def get\_new\_task(t):  
 ret = 0  
  
 for i in range(len(queue)):  
 if getattr(queue[i], "deadline") < getattr(queue[ret], "deadline") and t > getattr(queue[i], "appearing\_time"):  
 ret = i  
 return queue.pop(ret)  
  
  
def last\_appeared\_element(t):  
 return getattr(queue[-1], "appearing\_time") if len(queue) != 0 and getattr(queue[-1], "appearing\_time") > t else t  
  
  
response\_to\_lambda = []  
LAMBDA\_COUNT = 100  
MAX\_K = 5  
task\_made\_by\_025 = []  
sleep\_to\_lambda = []  
MX = 3  
lambda\_list\_x\_for\_plot = []  
MODEL\_TIME = 10000  
tasks\_in\_time = []  
  
for lam\_scale in range(1, LAMBDA\_COUNT + 1, 1):  
 lam = 0.01 \* lam\_scale  
  
 T = 0  
 queue = []  
 response\_time = []  
 sleep = 0  
  
 task = Task(0, lam)  
 task.appearing\_time = 0.0  
 while T < MODEL\_TIME:  
 if T < getattr(task, "appearing\_time"):  
 sleep += getattr(task, "appearing\_time") - T  
 T = getattr(task, "appearing\_time")  
  
 if T < getattr(task, "appearing\_time") + getattr(task, "deadline"):  
 if T + getattr(task, "time\_to\_solve") < getattr(task, "appearing\_time") + getattr(task, "deadline"):  
 response\_time.append(T - getattr(task, "appearing\_time"))  
 if lam == 0.25:  
 task\_made\_by\_025.append(T - task.appearing\_time)  
 T += getattr(task, "time\_to\_solve")  
 else:  
 T += getattr(task, "appearing\_time") + getattr(task, "deadline") - T  
  
 while T > last\_appeared\_element(getattr(task, "appearing\_time")):  
 queue.append(Task(last\_appeared\_element(getattr(task, "appearing\_time")), lam))  
 if lam == 0.25:  
 tasks\_in\_time.append(getattr(queue[-1], "appearing\_time"))  
 task = get\_new\_task(T)  
  
 lambda\_list\_x\_for\_plot.append(lam)  
 response\_to\_lambda.append(sum([i for i in response\_time]) / len(response\_time))  
 sleep\_to\_lambda.append(sleep)  
  
 print('\r{}%'.format(lam\_scale), end='')  
 if lam\_scale == LAMBDA\_COUNT:  
 print()  
pylab.plot(lambda\_list\_x\_for\_plot, response\_to\_lambda)  
pylab.xlabel('lambda')  
pylab.ylabel('average response time')  
pylab.show()  
pylab.plot(lambda\_list\_x\_for\_plot, sleep\_to\_lambda)  
pylab.xlabel('lambda')  
pylab.ylabel('sleep time')  
pylab.show()  
  
GROUP = 100  
bar\_x = [0]  
for i in range(len(tasks\_in\_time)):  
 if tasks\_in\_time[i] < GROUP \* (len(bar\_x)):  
 bar\_x[-1] += 1  
 else:  
 bar\_x.append(1)  
  
pylab.plt.bar(tasks\_in\_time, [1 for i in range(len(tasks\_in\_time))], align='center', alpha=1)  
pylab.plt.show()  
  
pylab.plt.bar([i for i in range(len(bar\_x))], bar\_x, align='center', alpha=1)  
pylab.plt.show()  
  
task\_made\_by\_025.sort()  
tmp = [0]  
for i in range(len(task\_made\_by\_025)):  
 if task\_made\_by\_025[i] < len(tmp):  
 tmp[-1] += 1  
 else:  
 tmp.append(0)  
  
pylab.plot([i for i in range(len(tmp))], tmp)  
pylab.xlabel('response time')  
pylab.ylabel('tasks')  
pylab.show()

round\_robin.py

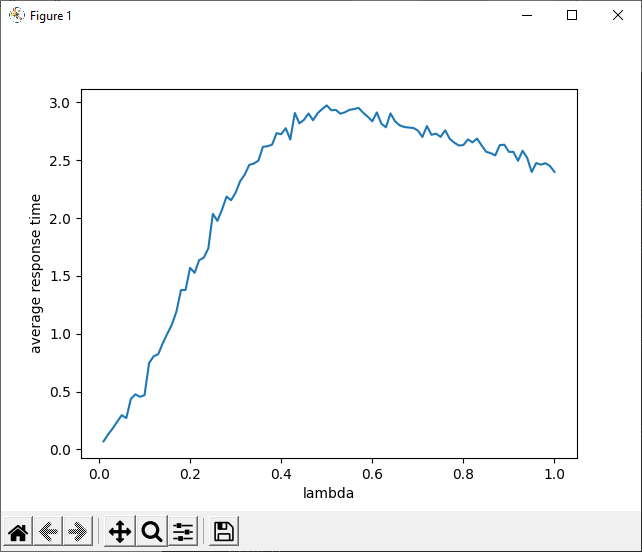
import math  
import random  
import pylab  
  
  
class Task:  
 def \_\_init\_\_(self, time, lampd):  
 self.time\_to\_solve = TIME\_OF\_FINISHING\_TASK  
 self.appearing\_time = time - 1 / lampd \* math.log(random.random())  
 self.deadline = random.randint(2, MAX\_K) \* self.time\_to\_solve  
 self.time\_after\_break = self.appearing\_time  
 self.reaction = None  
  
 def get\_appeartime(self):  
 return self.appearing\_time  
  
 def get\_solvetime(self):  
 return self.time\_to\_solve  
  
 def get\_deadline(self):  
 return self.deadline  
  
 def get\_break\_time(self):  
 return self.time\_after\_break  
  
  
def insert\_in\_stack(full\_task):  
 if len(stack) == 0 or stack[-1].get\_break\_time() < full\_task.get\_break\_time():  
 stack.append(full\_task)  
 else:  
 for i in range(len(stack)):  
 if full\_task.get\_break\_time() < stack[i].get\_break\_time():  
 stack.insert(i, full\_task)  
 break  
  
  
lampda = 0.01  
breaking\_time = 0.1  
TIME\_OF\_FINISHING\_TASK = 3 # регулярний потік  
  
lambda\_list\_x\_for\_plot = []  
response\_to\_lambda = []  
sleep\_to\_lambda = []  
tasks\_in\_time = []  
task\_made\_by\_025 = []  
  
MAX\_K = 5  
MODEL\_TIME = 10000  
AMOUNT\_OF\_LAMBDA = 100  
for scale\_lambda in range(1, AMOUNT\_OF\_LAMBDA+1, 1):  
 lampda = 0.01 \* scale\_lambda  
 response\_time = []  
 sleep = 0  
  
 T = 0  
 stack = [Task(0, lampda)]  
 stack[-1].appearing\_time = 0  
 while T < MODEL\_TIME:  
 task = stack.pop(0)  
 if T < task.get\_break\_time():  
 sleep += task.get\_break\_time() - T  
 T = task.get\_break\_time()  
  
 if task.reaction is None: # формирование след задачи  
 next\_task = task.get\_appeartime() - 1 / lampda \* math.log(random.random())  
 insert\_in\_stack(Task(task.get\_appeartime(), lampda))  
 if lampda == 0.25:  
 tasks\_in\_time.append(stack[-1].get\_appeartime())  
  
 if task.get\_solvetime() > breaking\_time and T - task.get\_appeartime() + breaking\_time < task.get\_deadline(): # задача прерывеается  
 task.time\_to\_solve = task.get\_solvetime() - breaking\_time  
 task.time\_after\_break = T + breaking\_time  
  
 if task.reaction is None: # время реакции  
 task.reaction = T - task.get\_appeartime()  
  
 T += breaking\_time  
 insert\_in\_stack(task)  
  
 elif task.get\_solvetime() <= breaking\_time and T - task.get\_appeartime() + task.get\_solvetime() < task.get\_deadline(): # задача выполнится  
 if lampda == 0.25:  
 task\_made\_by\_025.append(T - task.appearing\_time)  
 T += task.get\_solvetime()  
 response\_time.append(task.reaction)  
  
 elif T - task.get\_break\_time() < task.get\_deadline(): # потеряное время из за наступления дедлайга  
 T += task.get\_deadline() - (T - task.get\_break\_time())  
  
 lambda\_list\_x\_for\_plot.append(lampda)  
 print(len(response\_time))  
 if len(response\_time) != 0:  
 response\_to\_lambda.append(sum([i for i in response\_time]) / len(response\_time))  
 else:  
 response\_to\_lambda.append(0)  
 sleep\_to\_lambda.append(sleep)  
  
 print('\r{}%'.format(scale\_lambda), end='')  
 if scale\_lambda == AMOUNT\_OF\_LAMBDA:  
 print()  
  
pylab.plot(lambda\_list\_x\_for\_plot, response\_to\_lambda)  
pylab.xlabel('lambda')  
pylab.ylabel('average response time')  
pylab.show()  
  
pylab.plot(lambda\_list\_x\_for\_plot, sleep\_to\_lambda)  
pylab.xlabel('lambda')  
pylab.ylabel('sleep time')  
pylab.show()  
  
bar\_x = [0]  
GROUP = 100  
for i in range(len(tasks\_in\_time)):  
 if tasks\_in\_time[i] < GROUP\*(len(bar\_x)):  
 bar\_x[-1] += 1  
 else:  
 bar\_x.append(1)  
  
  
pylab.plt.bar(tasks\_in\_time,[1 for i in range(len(tasks\_in\_time))], align='center', alpha=1)  
pylab.plt.show()  
  
pylab.plt.bar([i for i in range(len(bar\_x))], bar\_x, align='center', alpha=1)  
pylab.plt.show()  
  
pylab.plot(lambda\_list\_x\_for\_plot, [i/MODEL\_TIME\*110 for i in sleep\_to\_lambda])  
pylab.xlabel('lambda')  
pylab.ylabel('sleep time 100%')  
pylab.show()  
  
task\_made\_by\_025.sort()  
tmp = [0]  
for i in range(len(task\_made\_by\_025)):  
 if task\_made\_by\_025[i] < len(tmp):  
 tmp[-1] += 1  
 else:  
 tmp.append(0)  
  
pylab.plot([i for i in range(len(tmp))], tmp)  
pylab.xlabel('response time')  
pylab.ylabel('tasks')  
pylab.show()

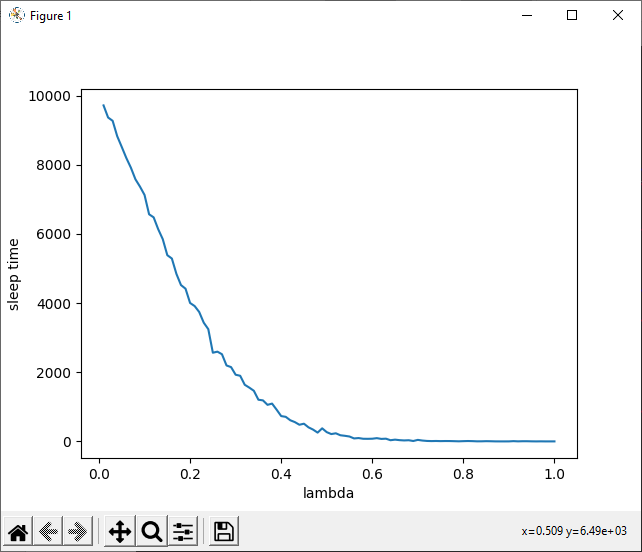
rate\_monotonic.py

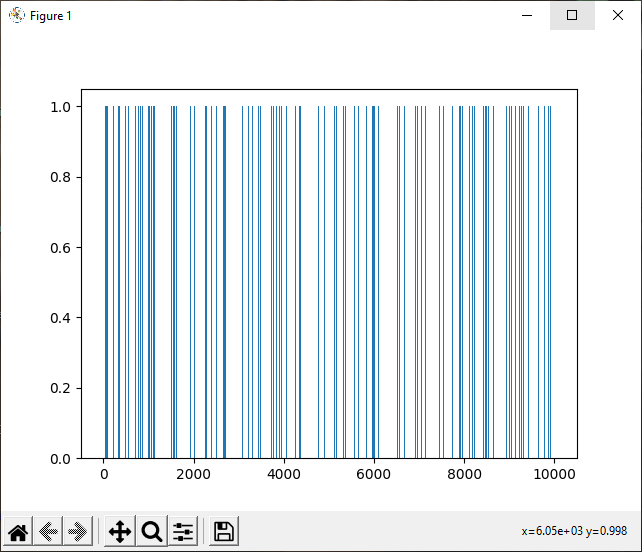
import math  
import random  
import pylab  
"""RM"""  
  
  
class Task:  
 def \_\_init\_\_(self, time, lampd):  
 self.time\_to\_solve = MX  
 self.appearing\_time = time - 1 / lampd \* math.log(random.random())  
 self.deadline = random.randint(2, MAX\_K) \* self.time\_to\_solve  
 self.priority = random.choice([1, 2, 3])  
 self.reaction = None  
 self.time\_after\_break = self.appearing\_time  
  
 def get\_appeartime(self):  
 return self.appearing\_time  
  
 def get\_solvetime(self):  
 return self.time\_to\_solve  
  
 def get\_deadline(self):  
 return self.deadline  
  
 def get\_priority(self):  
 return self.priority  
  
 def get\_break\_time(self):  
 return self.time\_after\_break  
  
  
def find\_last\_appeared\_time(t):  
 return queue[-1].get\_appeartime() if len(queue) != 0 and queue[-1].get\_appeartime() > t else t  
  
  
def get\_new\_task(t):  
 ret = 0  
 for i in range(len(queue)):  
 if queue[i].get\_deadline() < queue[ret].get\_deadline() and t > queue[i].get\_appeartime():  
 ret = i  
 return queue.pop(ret)  
  
  
def choose\_next\_task():  
 ret = 0  
 if len(queue) == 1:  
 pass  
 else:  
 min\_deadline = queue[0].get\_deadline()  
 for i in range(len(queue)):  
 if queue[i].get\_deadline() < min\_deadline:  
 min\_deadline = queue[i].get\_deadline()  
 ret = i  
 return ret  
  
  
def check\_if\_break():  
 if task.get\_deadline() > queue[-1].get\_deadline():  
 return True  
 else:  
 return False  
  
  
def break\_curr\_task():  
 task.time\_to\_solve /= 2  
  
  
  
MAX\_K = 3  
MX = 3  
MODEL\_TIME = 10000  
AMOUNT\_OF\_LAMBDA = 100  
lambda\_list\_x\_for\_plot = []  
response\_to\_lambda = []  
sleep\_to\_lambda = []  
tasks\_in\_time = []  
task\_made\_by\_025 = []  
  
for scale\_lambda in range(1, AMOUNT\_OF\_LAMBDA+1, 1):  
 lampda = 0.01 \* scale\_lambda  
  
 T = 0  
 queue = [Task(0, lampda)]  
 response\_time = []  
 sleep = 0  
  
 queue[0].appearing\_time = 0.0  
 while T < MODEL\_TIME:  
 task = queue.pop(choose\_next\_task())  
 if T < task.get\_appeartime(): # простой процесора  
 sleep += task.get\_appeartime() - T  
 T = task.get\_appeartime()  
  
 if task.reaction is None: # формирование след задачи  
 task.reaction = T - task.get\_appeartime()  
 next\_task = task.get\_appeartime() - 1 / lampda \* math.log(random.random())  
 queue.append(Task(task.get\_appeartime(), lampda))  
 if check\_if\_break():  
 break\_curr\_task()  
 queue.append(task)  
 continue  
 if lampda == 0.25:  
 tasks\_in\_time.append(queue[-1].get\_appeartime())  
  
 if task.get\_solvetime() > task.get\_deadline() or task.get\_break\_time() > task.get\_appeartime(): # задача прерывеается  
 task.time\_to\_solve = task.get\_solvetime() - task.get\_break\_time()  
 task.time\_after\_break = T + task.get\_break\_time()  
  
 if task.reaction is None: # время реакции  
 task.reaction = T - task.get\_appeartime()  
  
 T += task.get\_deadline()  
 queue.append(task)  
 if check\_if\_break():  
 break\_curr\_task()  
 queue.append(task)  
 continue  
  
 elif task.get\_solvetime() <= task.get\_deadline() and T - task.get\_appeartime() + task.get\_solvetime() < task.get\_deadline(): # задача выполнится  
 if lampda == 0.25:  
 task\_made\_by\_025.append(T - task.appearing\_time)  
 T += task.get\_solvetime()  
 response\_time.append(task.reaction)  
  
 elif T - task.get\_break\_time() < task.get\_deadline(): # потеряное время из за наступления дедлайга  
 T += task.get\_deadline() - (T - task.get\_break\_time())  
  
 lambda\_list\_x\_for\_plot.append(lampda)  
 #print(len(response\_time))  
 #print(response\_time)  
 if len(response\_time) != 0:  
 response\_to\_lambda.append(sum([i for i in response\_time]) / len(response\_time))  
 else:  
 response\_to\_lambda.append(0)  
 sleep\_to\_lambda.append(sleep)  
  
 print('\r{}%'.format(scale\_lambda), end='')  
 if scale\_lambda == AMOUNT\_OF\_LAMBDA:  
 print()  
  
  
pylab.plot(lambda\_list\_x\_for\_plot, response\_to\_lambda)  
pylab.xlabel('lambda')  
pylab.ylabel('average response time')  
pylab.show()  
pylab.plot(lambda\_list\_x\_for\_plot, sleep\_to\_lambda)  
pylab.xlabel('lambda')  
pylab.ylabel('sleep time')  
pylab.show()  
  
bar\_x = [0]  
GROUP = 100  
for i in range(len(tasks\_in\_time)):  
 if tasks\_in\_time[i] < GROUP\*(len(bar\_x)):  
 bar\_x[-1] += 1  
 else:  
 bar\_x.append(1)  
  
pylab.plt.bar(tasks\_in\_time, [1 for i in range(len(tasks\_in\_time))], align='center', alpha=1)  
pylab.plt.show()  
  
pylab.plt.bar([i for i in range(len(bar\_x))], bar\_x, align='center', alpha=1)  
pylab.plt.show()  
  
task\_made\_by\_025.sort()  
tmp = [0]  
for i in range(len(task\_made\_by\_025)):  
 if task\_made\_by\_025[i] < len(tmp):  
 tmp[-1] += 1  
 else:  
 tmp.append(0)  
  
pylab.plot([i for i in range(len(tmp))], tmp)  
pylab.xlabel('response time')  
pylab.ylabel('tasks')  
pylab.show()

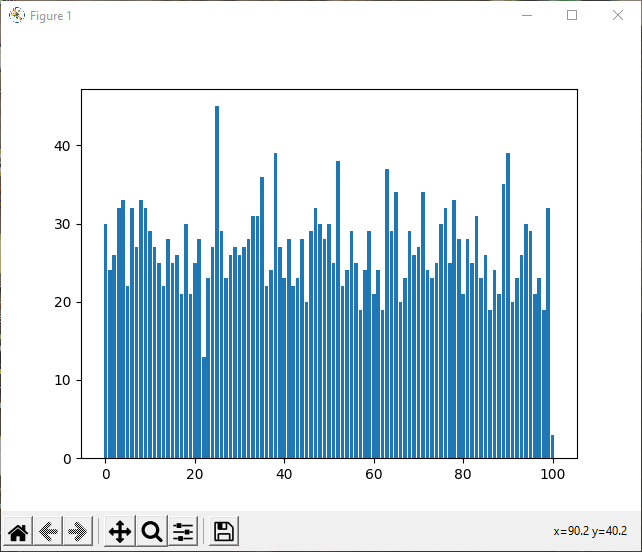
**Результати роботи програми**

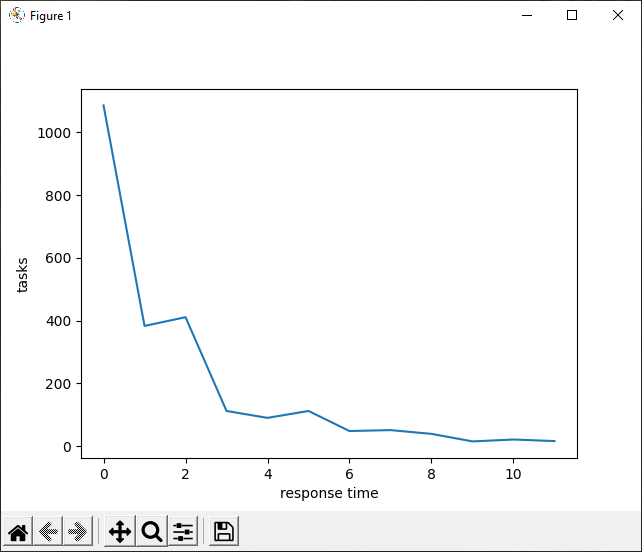
earliest\_deadline\_first

****

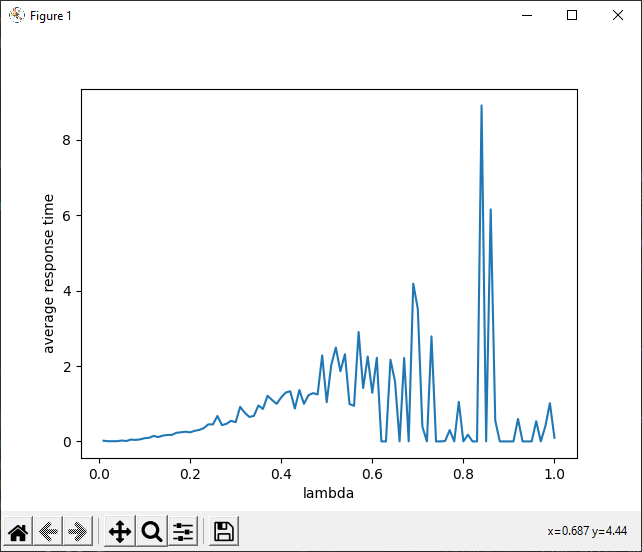
****

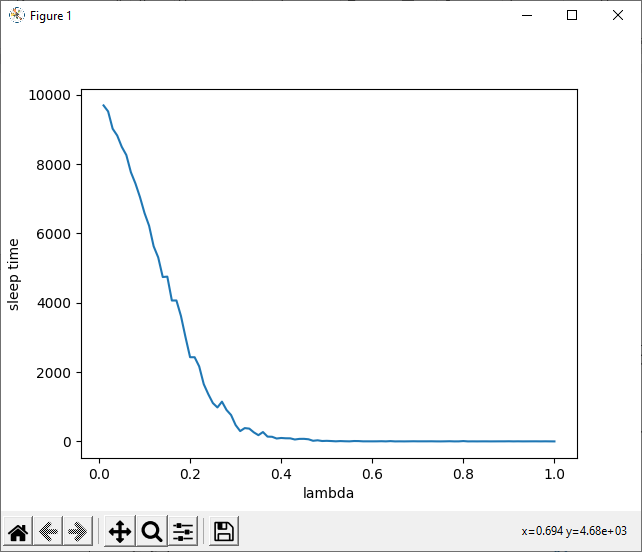
****

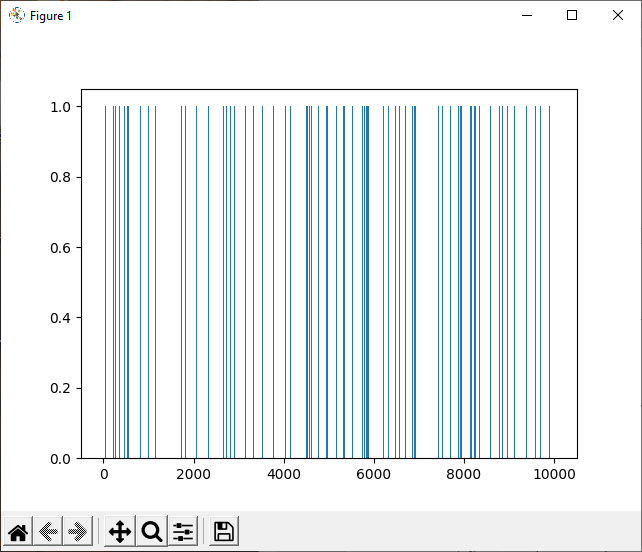
****

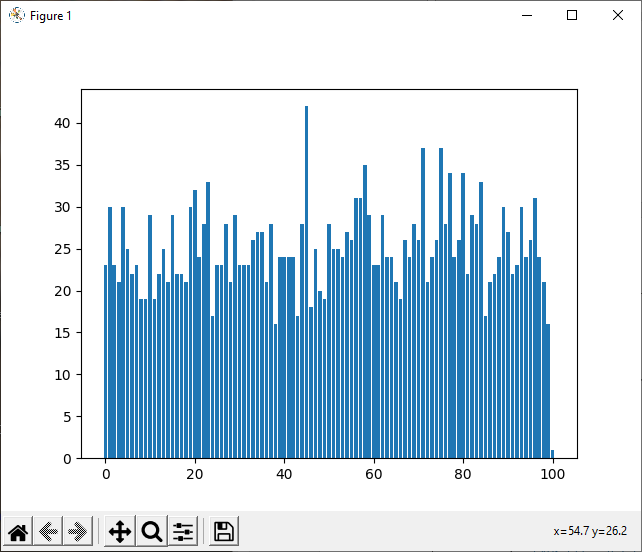
****

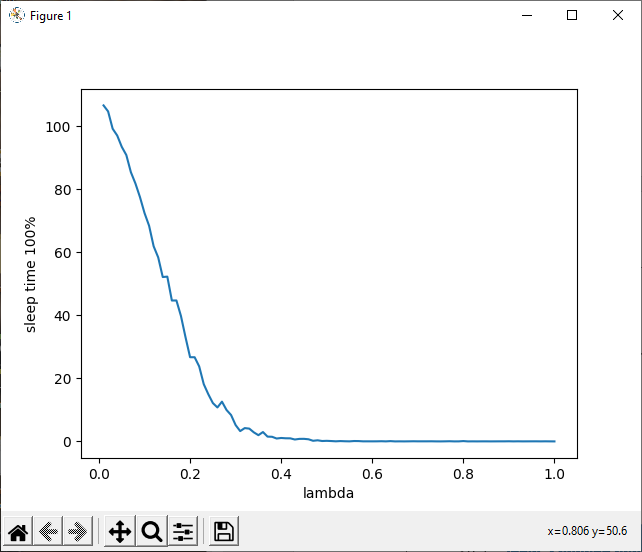
**Round robin**

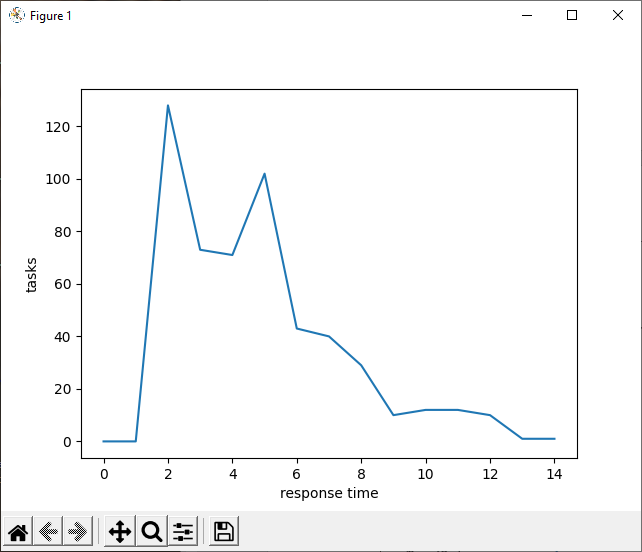
****

****

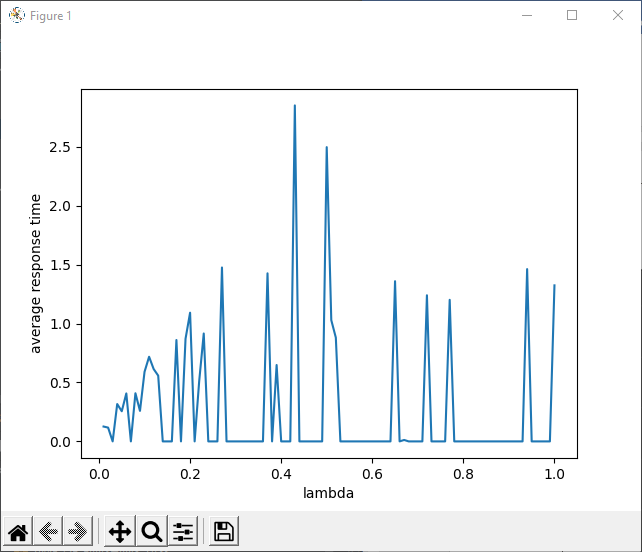
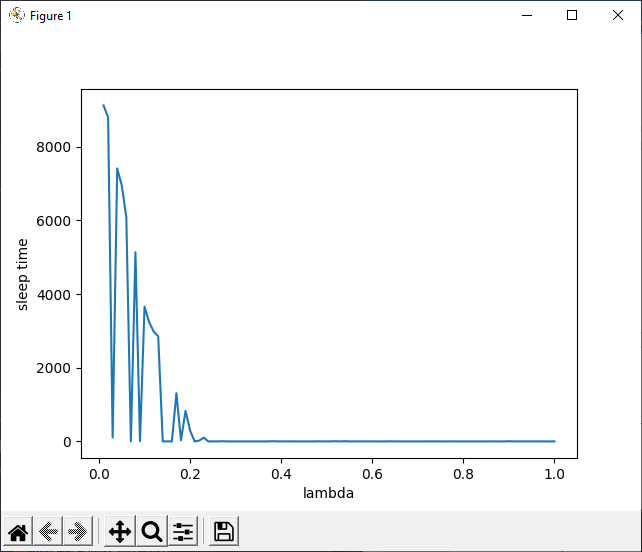
****

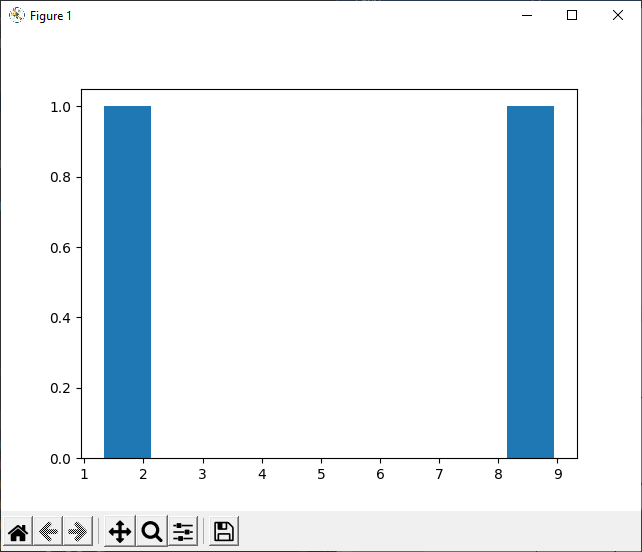
****

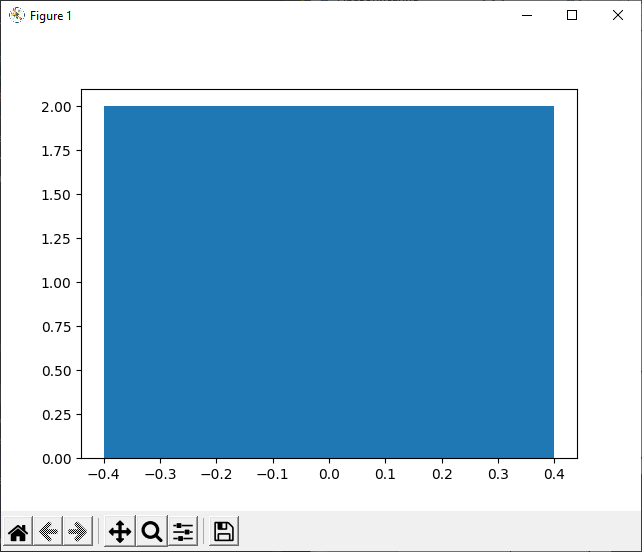
****

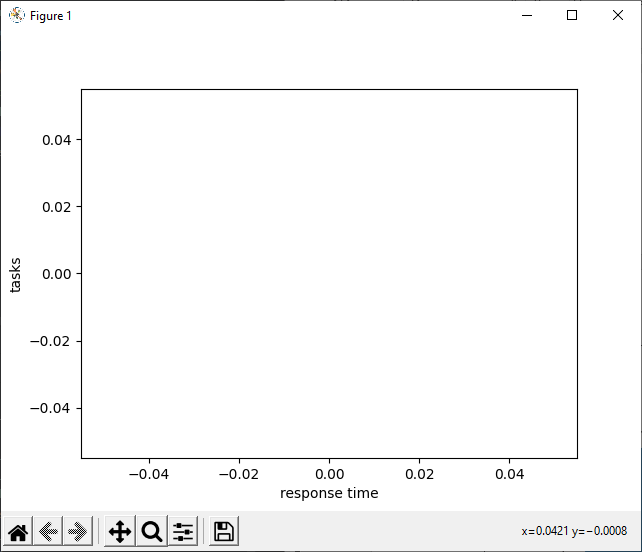
****

**Rate Monotonic**

**** ****

****

****

****

**Висновки**

Ми – змоделювали роботу планувальника задач у системі реального часу Визначили середній час очікування від інтенсивності вхідного потоку, залежність простою ресурсу від інтенсивності вхідного потоку заявок, кількість відмов вхідних заявок залежно від інтенсивності вхідного потоку заявок. Дані знання знадобляться нам у нашій подальшій кар’єрі.