НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО”
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

Розрахунково-графічна робота
з дисципліни
“Системи Реального Часу”

Виконав:
студент групи ІП-84
ЗК ІП-8523
Тимофеєнко Павло

Київ 2021

Мета роботи: змоделювати роботу планувальника задач у системі реального часу.

Дисципліни обслуговування –RM і EDF.

Основні теоретичні відомості

Планування виконання завдань (англ. Scheduling) 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 нитi

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.

Вибір заявки з черги на обслуговування здійснюється за допомогою так званої дисципліни обслуговування. Їх прикладами є FIFO (прийшов першим - обслуговується першим), LIFO (прийшов останнім -обслуговується першим), RANDOM (випадковий вибір). У системах з очікуванням накопичувач в загальному випадку може мати складну структуру.

Система масового обслуговування (СМО) — система, яка виконує обслуговування вимог (заявок), що надходять до неї. Обслуговування вимог у СМО проводиться обслуговуючими приладами. Класична СМО містить від одного до нескінченного числа приладів. В залежності від наявності можливості очікування вхідними вимогами початку обслуговування СМО (наявності черг) поділяються на:

1) системи з втратами, в яких вимоги, що не знайшли в момент надходження жодного вільного приладу, втрачаються;

2) системи з очікуванням, в яких є накопичувач нескінченної ємності для буферизації надійшли вимог, при цьому очікують вимоги утворюють чергу;

3) системи з накопичувачем кінцевої ємності (чеканням і обмеженнями), в яких довжина черги не може перевищувати ємності накопичувача; при цьому вимога, що надходить в переповнену СМО (відсутні вільні місця для очікування), втрачається.

Дисципліна RM

Rate monotonic (RM) - це просте правило, яке призначає пріоритети різних завдань відповідно до їх періодом часу. Тобто завдання з найменшим

періодом часу буде мати найвищий пріоритет, а завдання з найменшим періодом часу буде мати найменший пріоритет для виконання.

Алгоритм монотонного планування швидкості (RM) важливий для розробників систем реального часу, тому що він дозволяє гарантувати, що набір завдань є плановаим. Набір завдань називається плановаим, якщо всі завдання можуть укластися в терміни. Служба управління правами надає набір правил, які можна використовувати для виконання аналізу гарантованої планованого для набору завдань. Цей аналіз визначає, чи є набір завдань плановаим в умовах найгіршого випадку, і підкреслює передбачуваність поведінки системи. Було доведено, що:

RMS - це оптимальний алгоритм статичного пріоритету для планування незалежних, усували періодичних завдань на одному процесорі.

Середньоквадратичне відхилення є оптимальним в тому сенсі, що якщо який-небудь набір завдань може бути запланований за допомогою будь-якого алгоритму статичного пріоритету, тоді середньоквадратичне відхилення зможе планувати цей набір завдань. RM засновує свій аналіз планованих на рівні використання процесора, нижче якого можуть дотримуватися всі терміни.

RM вимагає статичного призначення пріоритетів завдань на основі їх періоду. Чим коротший період завдання, тим вище її пріоритет. Наприклад, завдання з періодом 1 мілісекунда має більш високий пріоритет, ніж завдання з періодом 100 мілісекунд. Якщо два завдання мають однаковий період, то служба управління правами не розрізняє завдання. Однак RTEMS вказує, що при завданні задач з рівним пріоритетом завдання, яка була готова найдовше, буде виконуватися першої. Схема призначення пріоритетів RM не надає точних числових значень для пріоритетів завдань. Наприклад, розглянемо наступний набір завдань і призначення пріоритетів:

Таблиця 1 – Дані пріорітетів планування задач на RM

Задача	Період (в мілісекундах)	Пріоритет
1	100	Низький
2	50	Середній
3	50	Середній
4	25	Високий

Служба управління правами вимагає, щоб задача 1 мала найнижчий пріоритет, завдання 4 мала найвищий пріоритет, а завдання 2 і 3 мали рівний пріоритет з пріоритетами завдань 1 і 4. Фактичні пріоритети RTEMS, призначені завданням, повинні відповідати тільки ці керівні принципи.

Дисципліна EDF

EDF (earliest deadline first) - пріоритет завдань призначається за принципом "в кожний момент часу найвищий пріоритет має та завдання, у якій залишилося найменше часу до крайнього терміну".

При цьому є 2 модифікації алгоритму EDF:

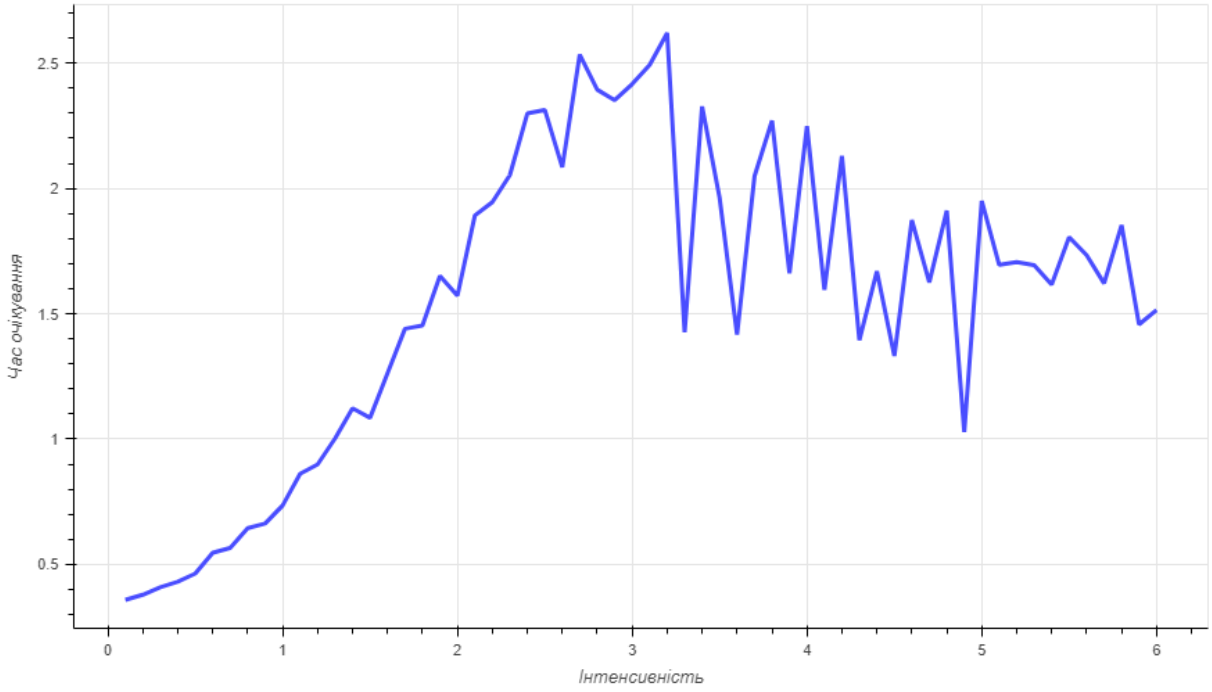
- з витісненням завдань
- без витіснення завдань

Під витісненням мається на увазі те, що якщо під час роботи якогось завдання виникає робота іншого завдання з пріоритетом вище, ніж у виконуваного системою у цей момент, то управління передається знову виникла задачі.

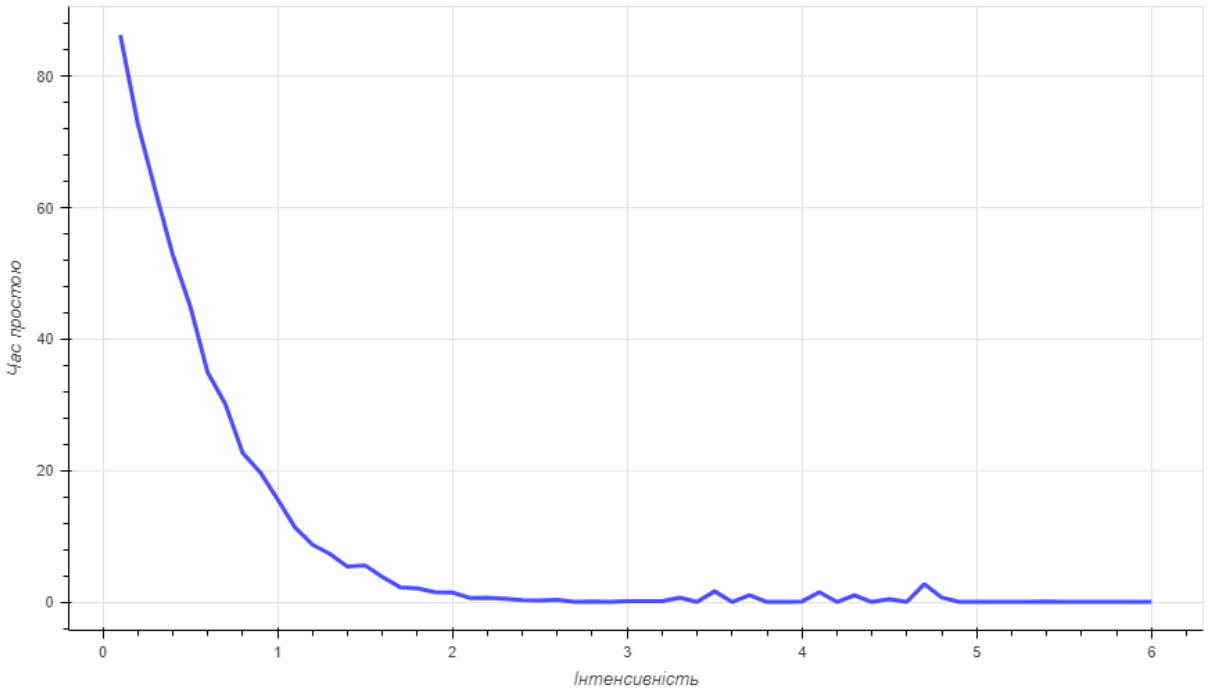
При невитісняючому EDF завдання завжди дороблює свою чергову роботу до кінця, незалежно від того, чи з'явилися під час роботи цього завдання інші завдання з більш високим пріоритетом або не з'явились.

Результати роботи

RM:

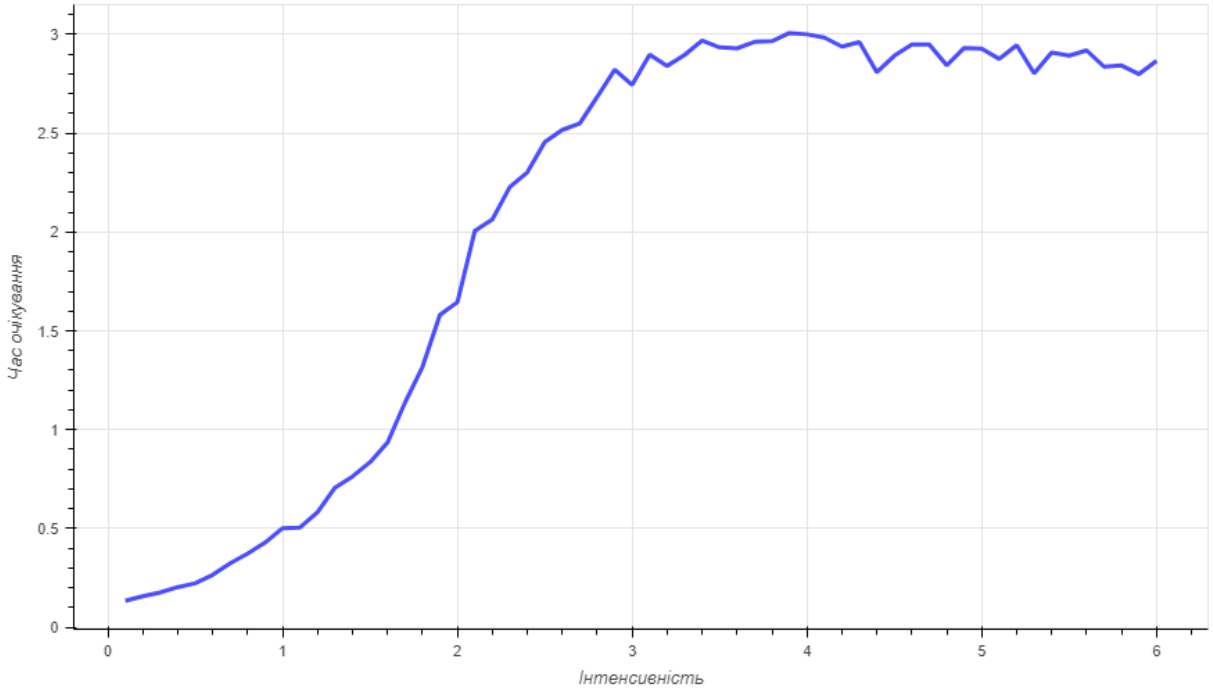


Графік середнього часу очікування від інтенсивності RM

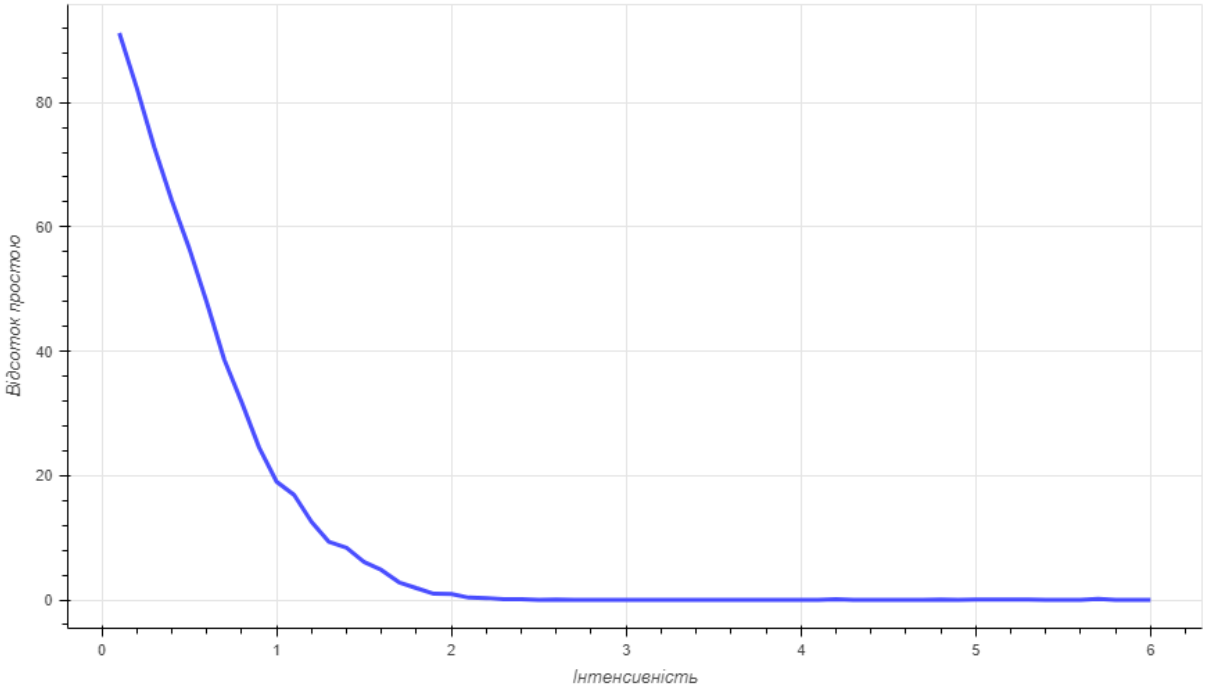


Графік відсотку простою пристрою від інтенсивності RM

EDF:



Графік середнього часу очікування від інтенсивності EDF



Графік відсотку простою пристрою від інтенсивності EDF

Додаток:

Mss.py

```
import random
import numpy
from queue import Queue, PriorityQueue

N = 6
m = 6

class Process:
    def __init__(self, pid=None, t_prev=0, priorities=(1, N), lamb=1, Td=(1 / m)
* 20):

        self.pid = pid
        self.priority = random.randint(*priorities)
        self.Td = Td

        self.T_execution = 1 / m
        self.T_end = None
        self.wait = 0
        self.T_in_system = self.T_execution
        self.lamb = lamb
        self.approach = -1/self.lamb * numpy.log(random.random())
        self.T_approach = t_prev + self.approach

        self.T_cur_exec = 0
```

```
    def in_sys_count(self):
        return self.T_in_system + self.wait

    def __str__(self):
        return "Process " + str(self.pid)

    def __lt__(self, other):
        return self.priority < other.priority


class MSS:
    def __init__(self, lamb):
        self.requests = 2000
        self.lamb = lamb
        self.queue = Queue()
        self.handled = []
        self.garbage = []
        self.counter = 1

        """ Initial state """
        self.first_e = Process(pid=0, lamb=self.lamb)
        self.first_e.T_approach = 0

        self.event_next = None
        self.event_current = None
        self.saved_next = None
        self.in_proc = self.first_e
        self.processor = 0

        self.T_appr = 0
        self.T_end = self.first_e.T_execution
        self.downtime = 0
        self.Td = (1 / m) * 20
        self.tau = (1 / m) / 2

        self.downtime_all = 0


    def generate(self, eid):
        """ Generating Task """
        return Process(pid=eid, lamb=self.lamb, t_prev=self.T_appr)

    def generate_req(self, rid):
        """ Generating Request """
        return [Process(
                        pid=int(str(rid) + str(i)),
                        lamb=self.lamb,
                        t_prev=self.T_appr
                        ) for i in range(random.randint(1, 5))]
```

```
    def save_event_current(self, eve):
        self.event_current = eve

    def save_event_next(self, eve):
        self.event_next = eve


def main():
    r1 = Process(0, lamb=5)

if __name__ == '__main__':
    main()
```

EDF.py

```
import random
import numpy
from typing import List
from queue import Empty, PriorityQueue
from mss import Process, MSS, m


class EDFProcess(Process):
    def __lt__(self, other):
        return self.Td - self.wait < self.Td - other.wait


class System(MSS):
    def __init__(self, lamb):
        super().__init__(lamb)
        self.handled.append(self.in_proc)
        self.queue = PriorityQueue()

        self.moments = []
        self.thrown = []

    def handling(self):

        while self.counter < self.requests:

            if self.processor:
                self.unpack_req(self.saved_next)
                self.in_proc = self.queue.get_nowait()
                in_proc_tsys = self.in_proc.in_sys_count()

                if in_proc_tsys >= self.Td:
                    self.processor = 0
                    self.T_end += in_proc_tsys - self.Td + self.downtime
                    self.garbage.append(self.in_proc)
```

```
                self.counter += 1
                self.saved_next = None
                continue

            self.processor = 0
            self.T_end += self.in_proc.T_execution + self.downtime
            self.handled.append(self.in_proc)

            self.counter += 1
            self.saved_next = None
            continue

        if not self.saved_next:

            e = self.generate_req(self.counter)

            self.save_event_next(eve=e)
            self.T_appr = self.event_next[0].T_approach

            if (self.T_appr - self.T_end) > 0 and self.queue.empty():
                self.downtime = self.T_appr - self.T_end

            else:
                self.downtime = 0

            self.downtime_all += self.downtime

        else:
            self.T_appr = self.event_next[0].T_approach

            if (self.T_appr - self.T_end) > 0 and self.queue.empty():
                self.downtime = self.T_appr - self.T_end

            else:
                self.downtime = 0

            self.downtime_all += self.downtime

        A = self.T_appr < self.T_end
        B = self.T_appr > self.T_end

        if A:
            for i in self.event_next:
                i.wait += self.T_end - self.T_appr
            self.unpack_req(self.event_next)
            self.saved_next = None

            self.counter += 1
            continue

        if B:
```

```
                self.saved_next = self.event_next

                if self.queue.empty():
                    self.processor = 1
                else:
                    eve = self.queue.get_nowait()

                    try:
                        while eve.wait > self.Td:
                            self.garbage.append(eve)
                            eve = self.queue.get_nowait()

                        while (eve.T_in_system + eve.wait) > self.Td:
                            self.garbage.append(eve)
                            self.T_end += (eve.T_in_system +
                                           eve.wait) - self.Td
                            eve = self.queue.get_nowait()
                    except Empty:
                        self.processor = 1
                        continue

                    self.moments.append(self.T_end)
                    self.thrown.append(len(self.garbage))

                    self.in_proc = eve
                    self.T_end += self.in_proc.T_execution
                    self.handled.append(self.in_proc)
                    for i in self.queue.queue:
                        i.wait += self.in_proc.T_execution

        return {
            "queue": self.queue,
            "handled": self.handled,
            'garbage': self.garbage,
        }

    def unpack_req(self, request: List[EDFProcess]):
        for i in request:
            self.queue.put_nowait(i)

    def generate_req(self, rid):

        return [EDFProcess(
            pid=int(str(rid) + str(i)),
            lamb=self.lamb,
            t_prev=self.T_appr
        ) for i in range(random.randint(1, 5))]


if __name__ == "__main__":
    from pprint import pprint
```

```
    from bokeh.models.widgets import Panel, Tabs
    from bokeh.io import output_file, show
    from bokeh.plotting import figure
    from collections import Counter

    intensities = numpy.linspace(0.1, 6, 60)
    attempts = [System(i) for i in intensities]
    results = [s.handling()['handled'] for s in attempts]
    waitings = []
    downtime_part = []
    processor_working = []

    att_count = len(attempts)

    for s in range(att_count):
        waitings.append(sum([i.wait for i in results[s]]) / len(results[s]))
        processor_working.append(
            sum([i.T_in_system+i.wait for i in results[s]]))

    for i in range(att_count):
        downtime_part.append(
            (attempts[i].downtime_all / (attempts[i].downtime_all + processor_wor
king[i]))*100)

    waiting_times_rounded = [[round(j.wait, 1) for j in i] for i in results]
    req_from_wt = [Counter(i) for i in waiting_times_rounded]

    output_file("EDF.html")
    tabs = []

    p1 = figure(plot_width=900, plot_height=500,
                x_axis_label='Інтенсивність', y_axis_label='Час очікування')
    p1.line(intensities, waitings, line_width=3, color="#494dff")
    tabs.append(Panel(child=p1, title="Очікування/Інтенсивність"))

    p2 = figure(plot_width=900, plot_height=500,
                x_axis_label='Інтенсивність', y_axis_label='Відсоток простою')
    p2.line(intensities, downtime_part, line_width=3, color="#494dff")
    tabs.append(Panel(child=p2, title="Простій/Інтенсивність"))

    tabs = Tabs(tabs=tabs)

    show(tabs)
```

RM.py

```
import random
import numpy
from typing import List
```

```
from queue import Empty
from mss import Process, MSS, m


class System(MSS):
    def __init__(self, lamb):
        super().__init__(lamb)
        if self.in_proc.T_execution <= self.tau:
            self.in_proc.T_end = self.in_proc.T_execution
            self.in_proc.T_cur_exec += self.in_proc.T_execution
            self.handled.append(self.in_proc)

        else:
            self.in_proc.T_execution = self.first_e.T_execution - self.tau
            self.in_proc.T_end = self.tau
            self.in_proc.T_cur_exec += self.tau
            self.queue.put_nowait(self.in_proc)
        self.moments = []
        self.thrown = []

    def handling(self):
        while self.counter < self.requests:
            if self.processor:
                self.unpack_req(self.saved_next)
                self.in_proc = self.queue.get_nowait()
                if self.in_proc.T_execution <= self.tau:
                    self.in_proc.T_cur_exec += self.in_proc.T_execution
                    in_proc_tsys = self.in_proc.T_cur_exec + self.in_proc.wait
                    if in_proc_tsys > self.Td:
                        self.processor = 0
                        self.T_end += in_proc_tsys - self.Td + self.downtime
                        self.garbage.append(self.in_proc)
                        self.counter += 1
                        self.saved_next = None
                        continue
                    self.processor = 0
                    self.T_end += self.in_proc.T_execution + self.downtime
                    self.handled.append(self.in_proc)
                    self.counter += 1
                    self.saved_next = None
                else:
                    self.in_proc.T_cur_exec += self.tau
                    in_proc_tsys = self.in_proc.T_cur_exec + self.in_proc.wait
                    if in_proc_tsys > self.Td:
                        self.processor = 0
                        self.T_end += in_proc_tsys - self.Td + self.downtime
                        self.garbage.append(self.in_proc)
                        self.counter += 1
                        self.saved_next = None
                        continue
                    self.processor = 0
```

```
                    self.in_proc.T_execution -= self.tau
                    self.T_end += self.tau + self.downtime
                    self.queue.put_nowait(self.in_proc)
                    self.counter += 1
                    self.saved_next = None
                continue
            if not self.saved_next:
                e = self.generate_req(self.counter)
                self.save_event_next(eve=e)
                self.T_appr = self.event_next[0].T_approach
                if (self.T_appr - self.T_end) > 0 and self.queue.empty():
                    self.downtime = self.T_appr - self.T_end
                else:
                    self.downtime = 0
                self.downtime_all += self.downtime

            else:
                self.save_event_next(eve=self.saved_next)
                self.T_appr = self.event_next[0].T_approach
                if (self.T_appr - self.T_end) > 0 and self.queue.empty():
                    self.downtime = self.T_appr - self.T_end
                else:
                    self.downtime = 0
                self.downtime_all += self.downtime
            A = self.T_appr < self.T_end
            B = self.T_appr > self.T_end
            if A:
                for i in self.event_next:
                    i.wait += self.T_end - self.T_appr
                self.unpack_req(self.event_next)
                self.saved_next = None
                self.counter += 1
                continue
            if B:
                self.saved_next = self.event_next
                if self.queue.empty():
                    self.processor = 1
                else:
                    eve = self.queue.get_nowait()
                    try:
                        while (eve.wait + eve.T_cur_exec) > self.Td:
                            self.garbage.append(eve)
                            eve = self.queue.get_nowait()
                        while (eve.T_execution + (eve.wait + eve.T_cur_exec)) > s
elf.Td:
                            self.garbage.append(eve)
                            self.T_end += (eve.T_in_system +
                                           eve.wait) - self.Td
                            eve = self.queue.get_nowait()
                    except Empty:
                        self.processor = 1
```

```
                        continue
                    self.moments.append(self.T_end)
                    self.thrown.append(len(self.garbage))
                    self.in_proc = eve
                    if self.in_proc.T_execution <= self.tau:
                        self.in_proc.T_cur_exec += self.in_proc.T_execution
                        self.T_end += self.in_proc.T_execution
                        self.handled.append(self.in_proc)
                        for i in self.queue.queue:
                            i.wait += self.in_proc.T_execution
                    else:
                        self.in_proc.T_cur_exec += self.tau
                        self.in_proc.T_execution -= self.tau
                        self.T_end += self.tau
                        self.queue.put_nowait(self.in_proc)
                        for i in self.queue.queue:
                            i.wait += self.tau

        return {
            "queue": self.queue,
            "handled": self.handled,
            'garbage': self.garbage,
        }

    def unpack_req(self, request: List[Process]):
        for i in request:
            self.queue.put_nowait(i)


if __name__ == "__main__":
    from pprint import pprint
    from bokeh.models.widgets import Panel, Tabs
    from bokeh.io import output_file, show
    from bokeh.plotting import figure
    from collections import Counter
    intensities = numpy.linspace(0.1, 6, 60)
    attempts = [System(i) for i in intensities]
    ah = [s.handling() for s in attempts]
    results = [s['handled'] for s in ah]
    waitings = []
    downtime_part = []
    processor_working = []
    att_count = len(attempts)
    for s in range(att_count):
        waitings.append(sum([i.wait for i in results[s]]) / len(results[s]))
        processor_working.append(
            sum([i.T_in_system + i.wait for i in results[s]]))
    for i in range(att_count):
        downtime_part.append(
            (attempts[i].downtime_all / (attempts[i].downtime_all + processor_wor
king[i])) * 100)
```

```
waiting_times_rounded = [[round(j.wait, 1) for j in i] for i in results]
req_from_wt = [Counter(i) for i in waiting_times_rounded]

output_file("RM.html")

tabs = []
p1 = figure(plot_width=900, plot_height=500,
            x_axis_label='Інтенсивність', y_axis_label='Час очікування')
p1.line(intensities, waitings, line_width=3, color="#494dff")
tabs.append(Panel(child=p1, title="Очікування/Інтенсивність"))
p2 = figure(plot_width=900, plot_height=500,
            x_axis_label='Інтенсивність', y_axis_label='Час простою')
p2.line(intensities, downtime_part, line_width=3, color="#494dff")
tabs.append(Panel(child=p2, title="Простой/Інтенсивність"))

tabs = Tabs(tabs=tabs)
show(tabs)
from collections import Counter
from pprint import pprint
pprint([(len(results[i]), intensities[i]) for i in range(len(results))])
pprint([len(i['garbage']) for i in ah])
```