Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

СИСТЕМИ РЕАЛЬНОГО ЧАСУ

Розрахунково-графічна робота

«Дослідження роботи планувальників роботи систем реального часу»

Виконав:

студент групи ІП-84

Кучін В. Д.

Залікова книжка № 8415

Перевірив:

Регіда П. Г.

Київ

2021 р.

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

**Завдання:** змоделювати планувальник роботи системи реального часу. Три дисципліни планування: FIFO, RM, EDF. В кожній дві черги – загальна та пріоритетна.

# 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 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.

Вибір заявки 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.

***Дисципліна FIFO***

FIFO (англ. first in, first out) — перший прийшов перший вийшов — є загальний принцип накопичення та обробки завдань (об'єктів). Принцип пов'язаний з поняттям черги: хто перший прийшов — той перший отримав обслуговування. Чергу можна представити у вигляді труби — з однієї сторони щось входить (стає в чергу) з іншої сторони виходить (оброблюється або обслуговується).

***Дисципліна EDF***

Earliest deadline first (EDF) - це алгоритм динамічного планування пріоритетів для вбудованих систем реального часу. EDF спочатку вибирає завдання згідно її крайнього терміну так, щоб завдання з найближчим крайнім терміном мала вищий пріоритет, ніж інші. Це означає, що пріоритет завдання обернено пропорційний її абсолютному терміну. Оскільки абсолютний термін виконання завдання залежить від поточного моменту часу, тому кожну мить є подією планування в EDF, оскільки термін виконання завдання змінюється з часом. Завдання, яке має більш високий пріоритет через самого раннього терміну в один момент, може мати низький пріоритет в наступний момент з-за більш раннього терміну виконання іншої задачі. EDF зазвичай виконується в режимі з витісненням, тобто виконуване зараз завдання переривається всякий раз, коли активується інша задача з найбільш раннім терміном.

EDF є оптимальним алгоритмом, який означає, що якщо задати виконання, то він обов'язково запланований EDF. Інше діло, що EDF спеціально не робить ніяких попередніх періодичних задач, тому він не залежить від періоду і, можливо, також використовувався для планування аперіодичних задач.

***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. Оскільки період часу завдання не змінюється, тому її пріоритет не змінюється з плином часу, тому швидкість монотонна є алгоритмом з фіксованим пріоритетом. Пріоритети визначаються до початку виконання і не змінюються з часом.

RM працює за принципом попередження. Витіснення відбувається на даному процесорі, коли завдання з більш високим пріоритетом блокує виконання завдання з більш низьким пріоритетом. Це блокування відбувається через рівень пріоритету різних завдань в даному наборі завдань. RM - це алгоритм з витісненням, який означає, що якщо завдання з більш коротким періодом настане під час виконання, вона отримає більш високий пріоритет і може блокувати або витісняти виконуються в даний момент завдання. У RM пріоритети призначаються відповідно до періодом часу. Пріоритет завдання обернено пропорційний її періоду таймера. Завдання з найнижчим періодом часу має найвищий пріоритет, а завдання з найнижчим періодом матиме найнижчий пріоритет.

***Код програми***

**index.py**

import scheduler\_class as sch  
import entries\_generator as gen  
import matplotlib.pyplot as plt  
import queue\_class as q  
import numpy as np  
  
MIN\_INTERVAL = 7  
MAX\_INTERVAL = 36  
INTERVAL\_STEP = 1  
ENTRY\_DURATION = 8  
ENTRIES\_NUMBER = 75000  
  
  
def single\_test(interval, entries\_number, entry\_duration, common\_queue, priority\_queue):  
 frequency = 1 / interval  
 total\_time = entries\_number \* interval  
 generator = gen.Generator(total\_time, frequency, 0.2, 0.3, 0.5, 0.5, entry\_duration)  
 entries = generator.generate()  
 scheduler = sch.Scheduler(common\_queue, priority\_queue, entries)  
 scheduler.run()  
 return scheduler  
  
  
def test(min\_interval, max\_interval, interval\_step, entry\_duration, entries\_number, common\_queue, priority\_queue):  
 frequencies = list()  
 idle\_percentages = list()  
 expired\_entries\_percentages = list()  
  
 common\_waiting\_times = list()  
 priority\_waiting\_times = list()  
 schedulers = list()  
  
 for interval in np.arange(min\_interval, max\_interval, interval\_step):  
 scheduler = single\_test(interval, entries\_number, entry\_duration, common\_queue, priority\_queue)  
  
 avg\_common\_waiting\_time = sum(scheduler.common\_waiting\_times) / len(scheduler.common\_waiting\_times)  
 avg\_priority\_waiting\_time = sum(scheduler.priority\_waiting\_times) / len(scheduler.priority\_waiting\_times)  
  
 print(f"Entries interval: {interval}. Entries avg duration: {entry\_duration}. Entries number: {entries\_number}")  
 print(f"Avg length queue. Common: {scheduler.avg\_common\_queue\_len}, priority: {scheduler.avg\_priority\_queue\_len}")  
 print(f"Avg waiting time. Common: {avg\_common\_waiting\_time}, priority: {avg\_priority\_waiting\_time}")  
 print(f"Idle percentage: {scheduler.idle\_percentage}%\n")  
  
 total\_common\_expired\_entries = scheduler.total\_common\_expired\_entries  
 percentage\_common\_expired\_entries = total\_common\_expired\_entries \* 100 / len(scheduler.common\_waiting\_times)  
  
 total\_priority\_expired\_entries = scheduler.total\_priority\_expired\_entries  
 percentage\_priority\_expired\_entries = total\_priority\_expired\_entries \* 100 / len(scheduler.priority\_waiting\_times)  
  
 print(f"Expired entries.\n"  
 f"Common (total / percentage): {total\_common\_expired\_entries} / {percentage\_common\_expired\_entries}%\n"  
 f"Priority (total / percentage): {total\_priority\_expired\_entries} / {percentage\_priority\_expired\_entries}%\n")  
 print("---------------------------------------------")  
  
 total\_expired\_entries = total\_common\_expired\_entries + total\_priority\_expired\_entries  
 expired\_entries\_percentages.append(total\_expired\_entries \* 100 / ENTRIES\_NUMBER)  
  
 frequency = 1 / interval  
 frequencies.append(frequency)  
 idle\_percentages.append(scheduler.idle\_percentage)  
 common\_waiting\_times.append(avg\_common\_waiting\_time)  
 priority\_waiting\_times.append(avg\_priority\_waiting\_time)  
  
 schedulers.append(scheduler)  
  
 return frequencies, idle\_percentages, common\_waiting\_times, priority\_waiting\_times, schedulers, expired\_entries\_percentages  
  
  
common\_fifo\_queue = q.FIFOQueue(False)  
priority\_fifo\_queue = q.FIFOQueue(True)  
  
common\_rm\_queue = q.RMQueue(False)  
priority\_rm\_queue = q.RMQueue(True)  
  
common\_edf\_queue = q.EDFQueue(False)  
priority\_edf\_queue = q.EDFQueue(True)  
  
fifo\_result = test(MIN\_INTERVAL, MAX\_INTERVAL, INTERVAL\_STEP, ENTRY\_DURATION, ENTRIES\_NUMBER, common\_fifo\_queue, priority\_fifo\_queue)  
rm\_result = test(MIN\_INTERVAL, MAX\_INTERVAL, INTERVAL\_STEP, ENTRY\_DURATION, ENTRIES\_NUMBER, common\_rm\_queue, priority\_rm\_queue)  
edf\_result = test(MIN\_INTERVAL, MAX\_INTERVAL, INTERVAL\_STEP, ENTRY\_DURATION, ENTRIES\_NUMBER, common\_edf\_queue, priority\_edf\_queue)  
  
plt.subplot(231)  
plt.plot(fifo\_result[0], fifo\_result[2], label="FIFO")  
plt.plot(rm\_result[0], rm\_result[2], label="RM")  
plt.plot(edf\_result[0], edf\_result[2], label="EDF")  
plt.legend(loc='lower right')  
plt.xlabel(f"Частота вхідних заявок.\nЗагальна черга")  
plt.ylabel(f"Середній час очікування в черзі")  
  
plt.subplot(232)  
plt.plot(fifo\_result[0], fifo\_result[3], label="FIFO")  
plt.plot(rm\_result[0], rm\_result[3], label="RM")  
plt.plot(edf\_result[0], edf\_result[3], label="EDF")  
plt.legend(loc='lower right')  
plt.xlabel(f"Частота вхідних заявок.\nПріоритетна черга")  
plt.ylabel(f"Середній час очікування в черзі")  
  
plt.subplot(233)  
plt.plot(fifo\_result[0], fifo\_result[1], label="FIFO")  
plt.plot(rm\_result[0], rm\_result[1], label="RM")  
plt.plot(edf\_result[0], edf\_result[1], label="EDF")  
plt.legend(loc='upper right')  
plt.xlabel(f"Частота вхідних заявок")  
plt.ylabel(f"Відсоток простою ресурсу")  
  
plt.subplot(234)  
plt.plot(fifo\_result[0], fifo\_result[5], label="FIFO")  
plt.plot(rm\_result[0], rm\_result[5], label="RM")  
plt.plot(edf\_result[0], edf\_result[5], label="EDF")  
plt.legend(loc='upper left')  
plt.xlabel(f"Частота вхідних заявок")  
plt.ylabel(f"Відсоток протермінованих заявок")  
  
plt.subplot(235)  
plt.hist([fifo\_result[4][8].common\_waiting\_times,  
 rm\_result[4][8].common\_waiting\_times,  
 edf\_result[4][8].common\_waiting\_times],  
 25, label=["FIFO", "RM", "EDF"])  
plt.legend(loc='upper right')  
plt.xlabel(f"Середній час очікування в черзі.\nІнтервал між заявками {1 / fifo\_result[0][8]}.\nЗагальна черга.")  
plt.ylabel(f"Кількість заявок")  
  
plt.subplot(236)  
plt.hist([fifo\_result[4][8].priority\_waiting\_times,  
 rm\_result[4][8].priority\_waiting\_times,  
 edf\_result[4][8].priority\_waiting\_times],  
 25, label=["FIFO", "RM", "EDF"])  
plt.legend(loc='upper right')  
plt.xlabel(f"Середній час очікування в черзі.\nІнтервал між заявками {1 / fifo\_result[0][8]}.\nПріоритетна черга")  
plt.ylabel(f"Кількість заявок")  
  
plt.show()

**scheduler\_class.py**

import numpy as np  
import entry\_class as etr  
  
  
class Scheduler:  
 def \_\_init\_\_(self, queue, priority\_queue, entries):  
 self.common\_queue = queue  
 self.priority\_queue = priority\_queue  
 self.entries = entries  
  
 self.time\_counter = 0  
 self.current\_entry = None  
  
 self.idle\_percentage = 0  
 self.avg\_common\_queue\_len = 0  
 self.avg\_priority\_queue\_len = 0  
 self.common\_waiting\_times = list()  
 self.priority\_waiting\_times = list()  
  
 self.closest\_deadline\_entry = etr.Entry(0, False, np.Infinity, 0, 0)  
 self.total\_common\_expired\_entries = 0  
 self.total\_priority\_expired\_entries = 0  
  
 def run(self):  
 common\_queue\_len\_sum = 0  
 priority\_queue\_len\_sum = 0  
  
 idle\_time = 0  
  
 while True:  
 self.check\_times\_list()  
  
 if self.current\_entry is not None and self.current\_entry.is\_completed():  
 if self.closest\_deadline\_entry == self.current\_entry:  
 self.closest\_deadline\_entry = None  
 if self.current\_entry.priority:  
 self.priority\_waiting\_times.append(self.current\_entry.waiting\_time)  
 else:  
 self.common\_waiting\_times.append(self.current\_entry.waiting\_time)  
 self.current\_entry = None  
  
 is\_deadline = self.check\_deadline()  
 if is\_deadline:  
 self.process\_expired\_entry()  
 self.closest\_deadline\_entry = self.find\_closest\_deadline\_entry()  
  
 self.current\_entry = self.dequeue\_entry\_from\_queues()  
  
 if len(self.entries) == 0 and self.common\_queue.is\_empty() and self.priority\_queue.is\_empty() \  
 and self.current\_entry is None:  
 break  
  
 next\_action\_time = self.find\_next\_action\_time()  
 time\_to\_next\_action = next\_action\_time - self.time\_counter  
 if self.current\_entry is not None:  
 self.current\_entry.ms\_left -= time\_to\_next\_action  
 else:  
 idle\_time += time\_to\_next\_action  
  
 self.common\_queue.add\_waiting\_time(time\_to\_next\_action)  
 self.priority\_queue.add\_waiting\_time(time\_to\_next\_action)  
  
 common\_queue\_len\_sum += self.common\_queue.get\_queue\_length() \* time\_to\_next\_action  
 priority\_queue\_len\_sum += self.priority\_queue.get\_queue\_length() \* time\_to\_next\_action  
  
 self.time\_counter = next\_action\_time  
  
 self.idle\_percentage = idle\_time / self.time\_counter \* 100  
 self.avg\_common\_queue\_len = common\_queue\_len\_sum / self.time\_counter  
 self.avg\_priority\_queue\_len = priority\_queue\_len\_sum / self.time\_counter  
  
 def check\_times\_list(self):  
 if len(self.entries) == 0:  
 return False  
 closest\_time = self.entries[0].enqueue\_time  
 if closest\_time == self.time\_counter:  
 entry = self.entries.popleft()  
 if entry.priority:  
 self.priority\_queue.enqueue(entry)  
 else:  
 self.common\_queue.enqueue(entry)  
 if self.closest\_deadline\_entry is None or entry.deadline < self.closest\_deadline\_entry.deadline:  
 self.closest\_deadline\_entry = entry  
 return True  
 return False  
  
 def dequeue\_entry\_from\_queues(self):  
 next\_entry = self.priority\_queue.deque(self.current\_entry)  
 if next\_entry != self.current\_entry:  
 if self.current\_entry is not None:  
 if self.current\_entry.priority:  
 self.priority\_queue.retrieve(self.current\_entry)  
 else:  
 self.common\_queue.retrieve(self.current\_entry)  
 return next\_entry  
 next\_entry = self.common\_queue.deque(self.current\_entry)  
 if next\_entry != self.current\_entry:  
 if self.current\_entry is not None:  
 self.common\_queue.retrieve(self.current\_entry)  
 return next\_entry  
  
 def find\_next\_action\_time(self):  
 enqueue\_time = np.Infinity  
 if len(self.entries) > 0:  
 enqueue\_time = self.entries[0].enqueue\_time  
  
 entry\_complete\_time = np.Infinity  
 if self.current\_entry is not None:  
 entry\_complete\_time = self.time\_counter + self.current\_entry.ms\_left  
  
 if self.closest\_deadline\_entry is not None:  
 next\_action\_time = min(enqueue\_time, entry\_complete\_time, self.closest\_deadline\_entry.deadline)  
 return next\_action\_time  
  
 return min(enqueue\_time, entry\_complete\_time)  
  
 def check\_deadline(self):  
 if self.closest\_deadline\_entry is None:  
 return False  
 if self.time\_counter == self.closest\_deadline\_entry.deadline:  
 return True  
 return False  
  
 def process\_expired\_entry(self):  
 if self.closest\_deadline\_entry.priority:  
 if self.closest\_deadline\_entry == self.current\_entry:  
 self.current\_entry = None  
 else:  
 self.priority\_queue.remove\_entry(self.closest\_deadline\_entry)  
 self.total\_priority\_expired\_entries += 1  
 self.priority\_waiting\_times.append(self.closest\_deadline\_entry.waiting\_time)  
 else:  
 if self.closest\_deadline\_entry == self.current\_entry:  
 self.current\_entry = None  
 else:  
 self.common\_queue.remove\_entry(self.closest\_deadline\_entry)  
 self.total\_common\_expired\_entries += 1  
 self.common\_waiting\_times.append(self.closest\_deadline\_entry.waiting\_time)  
  
 def find\_closest\_deadline\_entry(self):  
 priority\_min = self.priority\_queue.get\_closest\_deadline\_entry()  
 common\_min = self.common\_queue.get\_closest\_deadline\_entry()  
 entries = [self.current\_entry, priority\_min, common\_min]  
 min\_entry = entries[0]  
 for index in range(len(entries)):  
 if entries[index] is None:  
 continue  
 if min\_entry is None:  
 min\_entry = entries[index]  
 if entries[index].deadline < min\_entry.deadline:  
 min\_entry = entries[index]  
 return min\_entry

**entry\_class.py**

class Entry:  
 def \_\_init\_\_(self, ms\_left, priority, deadline, frequency, enqueue\_time):  
 self.ms\_left = ms\_left  
 self.waiting\_time = 0  
 self.priority = priority  
 self.deadline = deadline  
 self.frequency = frequency  
 self.enqueue\_time = enqueue\_time  
  
 def is\_completed(self):  
 return True if abs(self.ms\_left) < 0.00001 else False

**entries\_generator.py**

import entry\_class as etr  
import numpy as np  
import collections  
  
  
class Generator:  
  
 def \_\_init\_\_(self, time, frequency, fq\_1, fq\_2, fq\_3, priority\_fraction, duration):  
 self.time = time  
 self.frequency = frequency  
 self.fq\_1 = fq\_1  
 self.fq\_2 = fq\_2  
 self.fq\_3 = fq\_3  
 self.priority\_fraction = priority\_fraction  
 self.duration = duration  
  
 def generate(self):  
 frequency\_1 = self.frequency \* self.fq\_1  
 frequency\_2 = self.frequency \* self.fq\_2  
 frequency\_3 = self.frequency \* self.fq\_3  
  
 interval\_1 = 1.0 / frequency\_1  
 interval\_2 = 1.0 / frequency\_2  
 interval\_3 = 1.0 / frequency\_3  
  
 amount\_1 = round(frequency\_1 \* self.time)  
 amount\_2 = round(frequency\_2 \* self.time)  
 amount\_3 = round(frequency\_3 \* self.time)  
  
 generated\_intervals\_1 = np.random.exponential(interval\_1, int(amount\_1))  
 generated\_intervals\_2 = np.random.exponential(interval\_2, int(amount\_2))  
 generated\_intervals\_3 = np.random.exponential(interval\_3, int(amount\_3))  
  
 entries = np.concatenate((self.create\_entries(generated\_intervals\_1, frequency\_1, self.duration),  
 self.create\_entries(generated\_intervals\_2, frequency\_2, self.duration),  
 self.create\_entries(generated\_intervals\_3, frequency\_3, self.duration)))  
  
 return collections.deque(sorted(entries, key=lambda entry: entry.enqueue\_time))  
  
 def create\_entries(self, intervals, frequency, avg\_duration):  
 result = list()  
 time = 0  
 for i in intervals:  
 duration = np.random.uniform(avg\_duration - 5, avg\_duration + 5)  
 deadline = time + duration \* 10  
 priority\_random = np.random.uniform(0, 1)  
 priority = True if priority\_random < self.priority\_fraction else False  
  
 entry = etr.Entry(duration, priority, deadline, frequency, time)  
 result.append(entry)  
 time += i  
 return np.array(result)

**queue\_class.py**

import collections  
  
  
class Queue:  
 def \_\_init\_\_(self, priority):  
 self.priority = priority  
 self.queue = collections.deque()  
  
 def enqueue(self, entry):  
 self.queue.append(entry)  
  
 def deque(self, entry):  
 pass  
  
 def retrieve(self, entry):  
 pass  
  
 def is\_empty(self):  
 return True if len(self.queue) == 0 else False  
  
 def add\_waiting\_time(self, waiting\_time):  
 for entry in self.queue:  
 entry.waiting\_time += waiting\_time  
  
 def get\_queue\_length(self):  
 return len(self.queue)  
  
 def base\_dequeue(self, entry):  
 if entry is not None and entry.priority and not self.priority:  
 return True  
 if len(self.queue) == 0:  
 return True  
 return False  
  
 def remove\_entry(self, entry):  
 self.queue.remove(entry)  
  
 def get\_closest\_deadline\_entry(self):  
 if self.is\_empty():  
 return None  
 closest\_deadline\_entry = self.queue[0]  
 for index in range(len(self.queue)):  
 if self.queue[index].deadline < closest\_deadline\_entry.deadline:  
 closest\_deadline\_entry = self.queue[index]  
 return closest\_deadline\_entry  
  
  
class FIFOQueue(Queue):  
 def deque(self, entry):  
 if self.base\_dequeue(entry):  
 return entry  
 if entry is None or (not entry.priority and self.priority):  
 return self.queue.popleft()  
  
 return entry  
  
 def retrieve(self, entry):  
 super(FIFOQueue, self).retrieve(entry)  
 self.queue.appendleft(entry)  
  
  
class RMQueue(Queue):  
 def deque(self, entry):  
 if self.base\_dequeue(entry):  
 return entry  
 max\_frequency\_entry\_index = 0 if entry is None or (not entry.priority and self.priority) else -1  
 max\_frequency\_entry = self.queue[0] if max\_frequency\_entry\_index == 0 else entry  
 for index in range(len(self.queue)):  
 if self.queue[index].frequency > max\_frequency\_entry.frequency:  
 max\_frequency\_entry\_index = index  
 max\_frequency\_entry = self.queue[index]  
 if max\_frequency\_entry\_index != -1:  
 del self.queue[max\_frequency\_entry\_index]  
 return max\_frequency\_entry  
  
 def retrieve(self, entry):  
 super(RMQueue, self).retrieve(entry)  
 self.queue.append(entry)  
  
  
class EDFQueue(Queue):  
 def deque(self, entry):  
 if self.base\_dequeue(entry):  
 return entry  
 earliest\_deadline\_entry\_index = 0 if entry is None or (not entry.priority and self.priority) else -1  
 earliest\_deadline\_entry = self.queue[0] if earliest\_deadline\_entry\_index == 0 else entry  
 for index in range(len(self.queue)):  
 if self.queue[index].deadline < earliest\_deadline\_entry.deadline:  
 earliest\_deadline\_entry\_index = index  
 earliest\_deadline\_entry = self.queue[index]  
 if earliest\_deadline\_entry\_index != -1:  
 del self.queue[earliest\_deadline\_entry\_index]  
 return earliest\_deadline\_entry  
  
 def retrieve(self, entry):  
 super(EDFQueue, self).retrieve(entry)  
 self.queue.append(entry)

***Результати роботи програми***

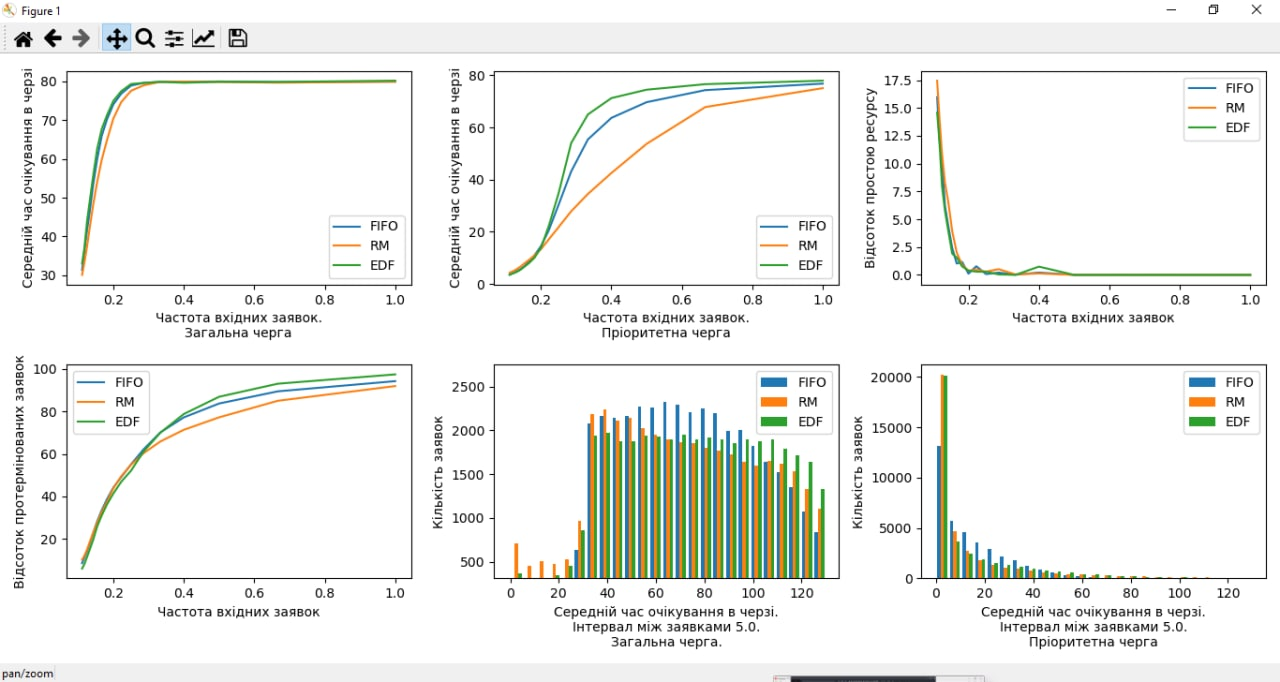
При генерації заявок вони розподілялися у три групи за частотою, що задавалися як частки від загальної частоти. Таким чином було створено три групи заявок з частотами 0.2, 0.3 та 0.5 від базової частоти, що задається для кожного експерименту окремо. Також кожна заявка може належати до однієї з двох категорій – загальної та пріоритетної. Заявки з пріоритетної черги обробляються перед заявками з загальної, а також витісняють їх, якщо в поточний момент обробляється заявка з загальної черги. Частка пріоритетних заявок задається перед моделюванням. При виконанні цієї роботи частка пріоритетних заявок становила 0.5. Час надходження заявок генерується за законом Пуассона. Тривалість виконання заявок при виконанні роботи обиралася випадково в інтервалі **(avg – 5, avg + 5)**, де avg – середнє значення тривалості виконання заявки, що в даній роботі становило 8 мс. Дедлайн кожної заявки визначається за формулою:

**deadline = enqueue\_time + duration \* 10**,

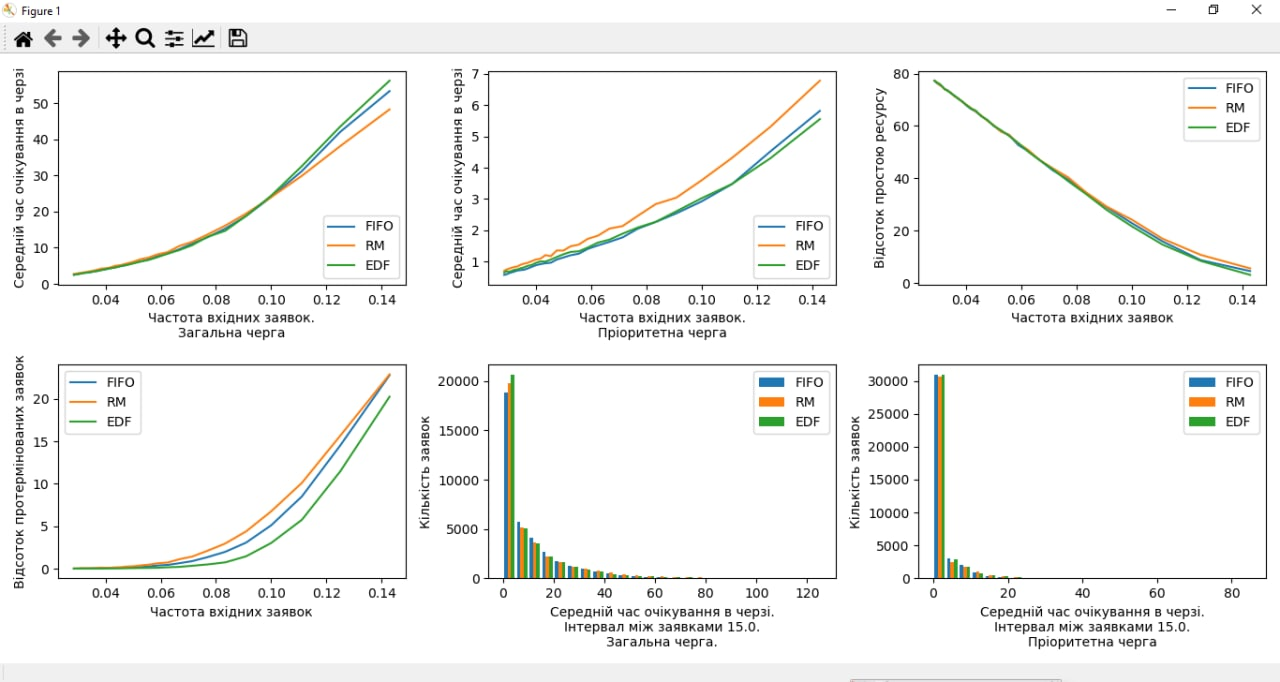
де enqueue\_time – час надходження заявки до черги, duration – тривалість виконання заявки.

В кожному моделюванні дисципліни обслуговування генерувалося 75000 заявок за наведеними вище правилами. Було проведено декілька моделювань з різним інтервалом надходження заявок та побудовано відповідні графіки. В першому варіанті інтервал змінювався в межах від 1 мс до 9 мс з кроком 0.5 мс. В другому варіанті – від 7 мс до 35 мс з кроком 1 мс.

Окремо було побудовано графіки при фіксованому інтервалі (частоті) надходження заявок (5 мс та 15 мс), де встановлено залежність між кількістю заявок та їх часом очікуванням в черзі.



*Рис. 1. Графіки моделювання СМО (FIFO, RM, EDF) при інтервалі надходження заявок у межах 1-9 мс (крок 0.5 мс)*



*Рис. 2. Графіки моделювання СМО (FIFO, RM, EDF) при інтервалі надходження заявок у межах 7-35 мс (крок 1 мс)*

***Висновок***

***Переваги EDF над RM***

* Не потрібно визначати пріоритети в режимі офлайн
* Вона має менше перемикання контексту, ніж RM
* EDF використовує процесор максимум до 100% коефіцієнта використання порівняно з RM

***Недоліки EDF над RM***

* EDF менш передбачуваний. Оскільки час відгуку завдань є змінним, то час відгуку завдань є постійним у випадку RM
* EDF надає менше контролю над виконанням
* EDF має високі накладні витрати