НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

\_\_\_\_\_\_\_\_\_\_\_\_\_\_Кафедра обчислювальної техніки\_\_\_\_\_\_\_

(повна назва кафедри, циклової комісії)

**РОЗРАХУНКОВО-ГРАФІЧНА РОБОТА**

з дисципліни «Системи реального часу»

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

Студента 3 курсу групи ІП-83

спеціальності

123 «Комп’ютерна інженерія»

\_\_\_\_\_Каплинського Р.П. \_\_\_\_

(прізвище та ініціали)

Керівник доцент Волокіта А.Н.

Київ – 2021 рік

З А В Д А Н Н Я

НА РОЗРАХУНКОВО-ГРАФІЧНУ РОБОТУ

Каплинському Роману Павловичу

Тема роботи «Дослідження роботи планувальників роботи систем реального часу»

Керівник роботи Волокіта Артем Миколайович к.т.н., доцент

1. Змоделювати планувальник роботи системи реального часу. Дві дисципліни планування: перша – RR, друга задається викладачем або обирається самостійно (була обрана дисципліна EDF).
2. Знайти наступні значення:
3. середній розмір вхідної черги заявок, та додаткових черг (за їх наявності);
4. середній час очікування заявки в черзі;
5. кількість прострочених заявок та її відношення до загальної кількості заявок
6. Побудувати наступні графіки:
7. Графік залежності кількості заявок від часу очікування при фіксованій інтенсивності вхідного потоку заявок.
8. Графік залежності середнього часу очікування від інтенсивності вхідного потоку заявок.
9. Графік залежності проценту простою ресурсу від інтенсивності вхідного потоку заявок

# [Основні](#_Toc533528592) 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.

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;

1. системи з очікуванням, в яких є накопичувач нескінченної ємності

для буферизації надійшли вимог, при цьому очікують вимоги

утворюють чергу;

1. 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.

Основні поняття СМО:

* Вимога (заявка) — запит на обслуговування.
* Вхідний потік вимог — сукупність вимог, що надходять у СМО.
* Час обслуговування - період часу, протягом якого обслуговується

вимогу.

# Вимоги до системи

# Вхідні задачі

Вхідними заявками є обчислення, які проводилися в лабораторних

роботах 1-3, а саме обчислення математичного очікування, дисперсії,

автокореляції, перетворення Фур’є.

Вхідні заявки характеризуються наступними параметрами:

1. час приходу в систему – Tp – потік заявок є потоком Пуассона або

потоком Ерланга k-го порядку (інтенсивність потоків та їх порядок

задаються варіантом);

1. час виконання (обробки) – То; математичним очікуванням часу

виконання є середнє значення часу виконання відповідних

обчислень в попередніх лабораторних роботах;

1. крайній строк завершення (дедлайн) – Тd –задається (випадково?);

якщо заявка залишається необробленою в момент часу t = Td, то її

обробка припиняється і вона покидає систему.

# Потік вхідних задач

Потоком Пуассона є послідовність випадкових подій, середнє значення

інтервалів між настанням яких є сталою величиною, що дорівнює 1/λ , де λ –

інтенсивність потоку.

Потоком Ерланга k-го порядку називається потік, який отримується з

потоку Пуассона шляхом збереження кожної (k+1)-ї події (решта

відкидаються). Наприклад, якщо зобразити на часовій осі потік Пуассона,

поставивши у відповідність кожній події деяку точку, і відкинути з потоку

кожну другу подію (точку на осі), то отримаємо потік Ерланга 2-го порядку.

Залишивши лише кожну третю точку і відкинувши дві проміжні, отримаємо

потік Ерланга 3-го порядку і т.д. Очевидно, що потоком Ерланга 0-го порядку

є потік Пуассона.

# Пристрій обслуговування

Пристрій обслуговування складається з P незалежних рівноправних

обслуговуючих приладів - обчислювальних ресурсів (процесорів). Кожен

ресурс обробляє заявки, які йому надає планувальник та може перебувати у

двох станах – вільний та зайнятий. Обробка заявок може виконуватися

повністю (заявка перебуває на обчислювальному ресурсі доти, доки не

обробиться повністю) або поквантово (ресурс обробляє заявку лише

протягом певного часу – кванту обробки – і переходить до обробки наступної

заявки).

# Пріоритети заявок

Заявки можуть мати пріоритети – явно задані, або обчислені системою

(в залежності від алгоритму обслуговування або реалізації це може бути час

обслуговування (обчислення), час до дедлайну і т.д.). Заявки в чергах

сортуються за пріоритетом. Є два види обробки пріоритетів заявок:

1. без витіснення – якщо в чергу до ресурсу потрапляє заявка з

більшим пріоритетом, ніж та, що в даний момент часу обробляється

ним, то вона чекає завершення обробки ресурсом його задачі.

1. з витісненням – якщо в чергу до ресурсу потрапляє заявка з більшим

пріоритетом, ніж та, що в даний момент часу обробляється ним, то

вона витісняє її з обробки; витіснена задача стає в чергу.

# Дисципліна обслуговування

Вибір заявки з черги на обслуговування здійснюється за допомогою так

званої дисципліни обслуговування. Їх прикладами є RoundRobin (Алгоритм циклічного розподілу навантаження), та EDF - Earliest Deadline First (планування за найближчим терміном завершення). У системах з очікуванням накопичувач в загальному випадку може мати складну структуру.

# Дисципліна RR

Алгоритм Round-Robin (від англ. round-robin — циклічний) – алгоритм

розподілу навантаження на розподілену (або паралельну) обчислювальну

систему методом перебору і впорядкування її заявок по круговому циклу.

Даний алгоритм не враховує пріоритети вхідних заявок.

Нехай є Р ресурсів (з порядковими номерами р) та X заявок (з

порядковими номерами x), які необхідно виконати. Тоді перша заявка (x = 1)

назначається для виконання на першому ресурсі (р = 1), друга (х = 2) –

другому і т.д., до досягнення зайнятості останнього ресурсу (р = Р, х = Р) або

до вичерпування необроблюваних заявок (х = Х). Усі наступні заявки будуть

розподілені по ресурсах аналогічно до попередніх, починаючи з першого

ресурсу (х = Р + 1 → р = 1, х = Р + 2 → р = 2 і т.д.). Іншими словами

відбувається перебір ресурсів по циклу (по колу – round).

Обчислення задач розділене на кванти часу, причому по закінченню

кванту завершені та прострочені задачі виходять з системи, незавершені –

здвигаються по колу на 1 ресурс (тобто задача першого об’єкта передається

другому, другого – третьому і т.д., останнього – першому).

# Дисципліна EDF

Алгоритм планування Earliest Deadline First (по найближчому строку

завершення) використовується для встановлення черги заявок в операційних системах реального часу.

При наставанні події планування (завершився квант часу, прибула нова

заявка, завершилася обробка заявки, заявка прострочена) відбувається пошук

найближчої до крайнього часу виконання (дедлайну) заявки і призначення її

виконання на перший вільний ресурс або на той, який звільниться найшвидше.

# Розробка програми

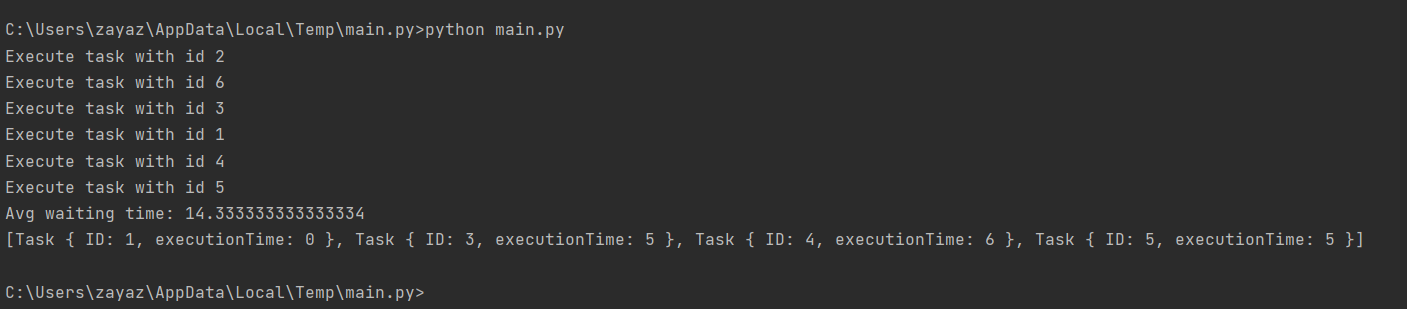
# Earliest Deadline First

Програма написана на мові «Python». В ній реалізовано стратегію Earliest Deadline First тобто стратегію яка передбачає розподіл пріоритетності задач залежно від часу дедлайну з яким приходить задача. Дана стратегія використовується в операційних системах реального часу для розміщення процесів у черзі пріоритетів. У EDF ми отримуємо на вхід масив завдань, з даними про ID цього завдання, його дедлайн та час необхідний для виконання цього завдання (це необхідно в даній реалізації щоб симулювати процес приходу і виконання завдань в реальності). Щоразу, коли відбувається подія планування (завдання закінчується, випускається нове завдання тощо), у черзі буде здійснюватися пошук процесу, найближчого до кінцевого терміну. Цей процес є наступним, який планується виконати. Таким чином, при поступанні на обробку великої кількості процесів з різними дедлайнами та часом виконання, існує ймовірність прострочування дедлайнів. Однак при великій кількості малих заявок ця затримка буде доволі не значною і в решті решт всі заявки будуть оброблені.

Лістнінг програми:

import threading  
import random  
  
  
class Task:  
 def \_\_init\_\_(self, ID, deadLine, executionTime) -> None:  
 self.ID = ID  
 self.deadLine = deadLine  
 self.executionTime = executionTime  
  
 def reduceDeadLine(self, number):  
 self.deadLine -= number  
  
 def \_\_str\_\_(self):  
 return f"Task {{ ID: {self.ID}, executionTime: {self.executionTime} }}"  
  
 def \_\_repr\_\_(self):  
 return f"Task {{ ID: {self.ID}, executionTime: {self.executionTime} }}"  
  
  
class Processor:  
 def \_\_init\_\_(self) -> None:  
 self.tasks = [] # Task array  
 self.expiredDeadLine = []  
 self.timeCounter = 0  
  
 def addTask(self, task: Task) -> None:  
 self.tasks.append(task)  
  
 def execute(self):  
 taskListLen = self.tasks.\_\_len\_\_()  
 while self.tasks.\_\_len\_\_():  
 earliestTask = self.findEarliestDeadLine()  
 print(f"Execute task with id {earliestTask.ID}")  
 self.tasks.remove(earliestTask)  
 for task in self.tasks:  
 task.reduceDeadLine(earliestTask.executionTime)  
 self.timeCounter += earliestTask.executionTime  
 if task.deadLine <= 0 and task not in self.expiredDeadLine:  
 self.expiredDeadLine.append(task)  
 print(f"Avg waiting time: {self.timeCounter/taskListLen}")  
  
 def findEarliestDeadLine(self) -> Task:  
 closestToDeadLineTask = self.tasks[0]  
 for task in self.tasks:  
 if (task.deadLine < closestToDeadLineTask.deadLine): closestToDeadLineTask = task  
 return closestToDeadLineTask  
  
  
def appendTaskThread(processor):  
 for i in range(6):  
 processor.addTask(Task(i + 1, random.randint(0, 25), random.randint(0, 15)))  
  
  
def main():  
 processor = Processor()  
 thread = threading.Thread(target=appendTaskThread, args=(processor,))  
 executor = threading.Thread(target=processor.execute)  
 thread.start()  
 executor.start()  
 thread.join()  
 executor.join()  
 print(processor.expiredDeadLine)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

Приклад виводу результатів виконання програми:



Переваги EDF над RR

• Гарантує виконання більшої кількості завдань

• Має менший середній час очікування задачі в черзі

Недоліки EDF над RR

• Потрібно визначати пріоритет задачі

• EDF має високі накладні витрати

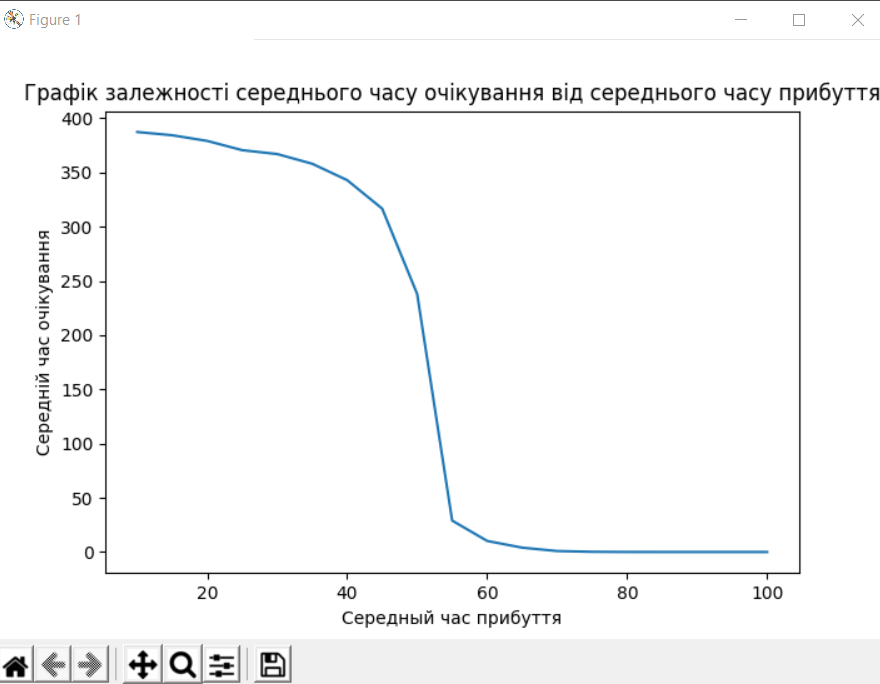
Очевидно, що в системах реального часу доцільніше використання системи EDF ніж RR.

# Round Robin

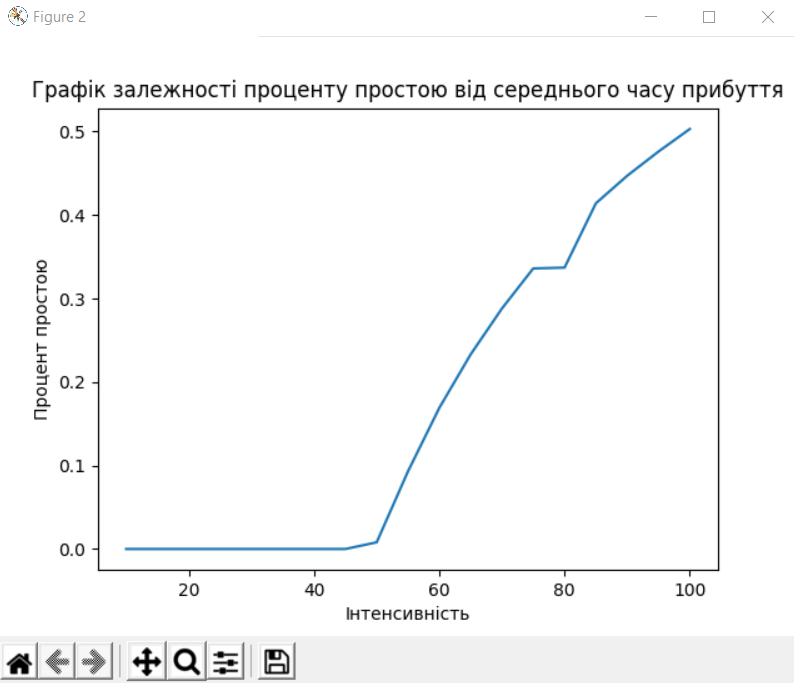
Програма написана на мові «С». В ній реалізовано стратегію Round Robin або карусельну стратегію планування з пріоритетним витісненням. Дана стратегія може застосовуватись в системах розподілу часу. У RR ми визначаємо невеликий відрізок часу, квантом (10..100 мс), квантом ми можемо задавати самі. Черга готових заявок розглядається як кільцева. Заявки циклічно переміщаються по черзі, отримуючи CPU на деякий час, що рівний одному кванту. При необхідності додання нового процесу, ми дивимось на його пріоритет і додаємо в чергу, на місце згідно пріоритету (чим менший пріоритет, тим скоріше обробиться заява). Якщо процес не завершився в межах виділеного йому кванту часу, то його робота примусово переривається, і він переміщується в хвіст черги. Якщо ж виконання процесу завершене, то він видаляється з черги.

Для прикладу додамо в чергу 32 заявки, з різними пріоритетами і різними потребами. При обробці цих заяв вони будуть розміщені в чергу відповідно пріорітетам. Однак це дасть їм перевагу тільки на першому колі, що є значним недоліком RR. Однак все ж всі заяви будуть оброблені і виконані.

Перший графік демонструє залежність середнього часу очікування від середнього часу прибуття. Можна помітити, що зі збільшенням середнього часу прибуття зменшується час очікування.

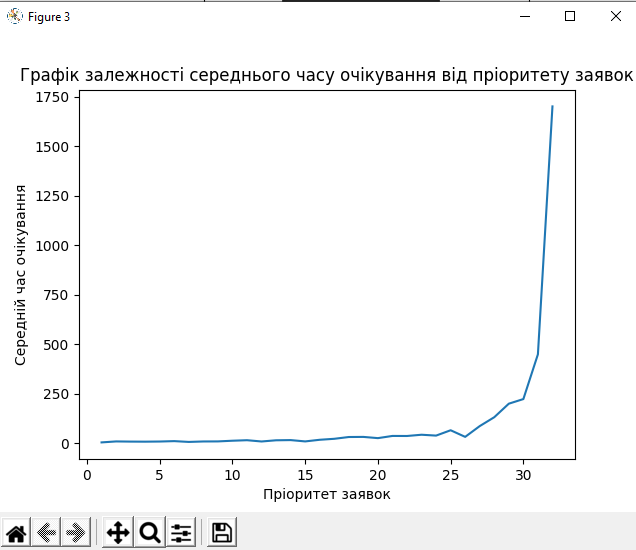


Другий графік зображує залежність проценту простою від середнього часу прибуття. Зі збільшенням середнього часу прибуття, не враховуючи погрішності процент простою зростає:



Третій графік демонструє залежність середнього часу очікування від пріорітету заявок. З графіку видно, що, враховуючи погрішності, середній час очікування у заяв з вищим пріоритетом – набагато менший (найвищий

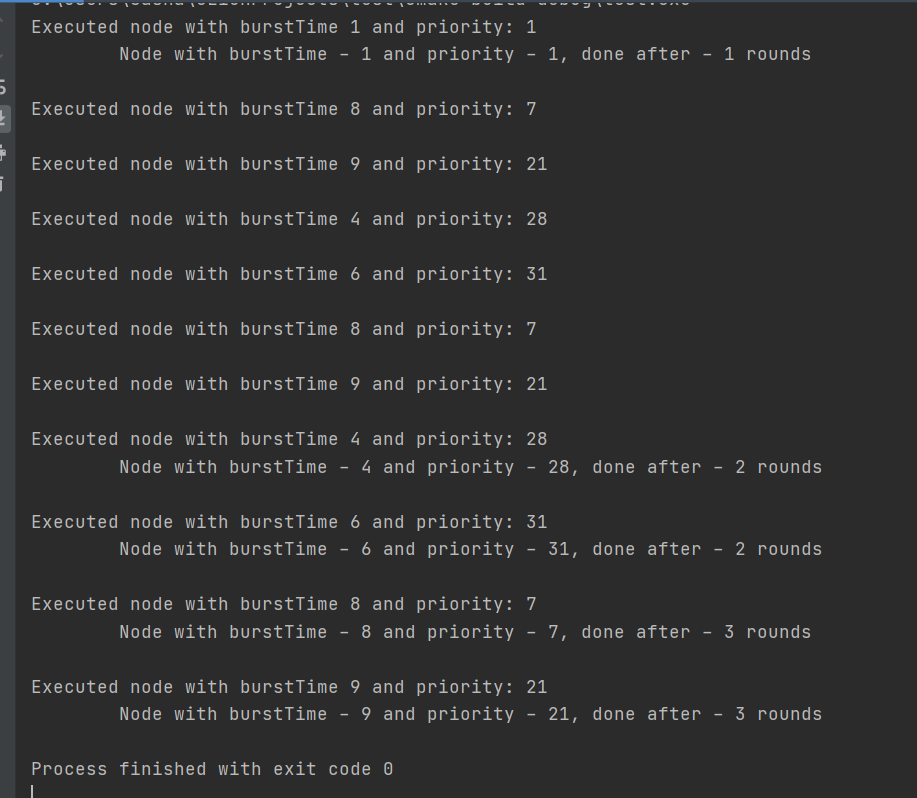
пріоритет -1, найнижчий - 32):



Лістнінг програми:

#include <stdio.h>  
#include <stdlib.h>  
#define PATH "results.txt"  
#define QUANTUM 3  
  
typedef struct queue{  
 struct node\* head;  
 int length;  
 int lastID;  
 int processed;  
 double waitTimeSum;  
} Queue;  
  
typedef struct node {  
 int priority;  
 int burstTime;  
 int timeLeft;  
 int id;  
 double avgWaitTimeOnAppend;  
 struct node\* next;  
 struct node\* prev;  
} Node;  
  
double getWaitTime (Node\* node) {  
 double res = 0;  
 while (node != NULL){  
 res += node->timeLeft;  
 node = node->prev;  
 }  
 return res;  
}  
  
Node\* push(Queue\* queue, int priority, int burstTime){  
 if(queue->head == NULL){  
  
 Node\* newNode = (Node\*)malloc(sizeof(Node));  
 newNode->burstTime = burstTime;  
 newNode->timeLeft = burstTime;  
 newNode->priority = priority;  
 newNode->next = NULL;  
 newNode->prev = NULL;  
 newNode->id = 1;  
 newNode->avgWaitTimeOnAppend = 0;  
  
 queue->head = newNode;  
 queue->length = 1;  
 queue->lastID = 1;  
 queue->waitTimeSum = 0;  
 queue->processed = 0;  
  
 return newNode;  
 }  
 Node \*node = queue->head;  
 while (node != NULL){  
 if (priority > node->priority && node->next != NULL) {  
 node = node->next;  
 } else {  
 Node\* newNode = (Node\*)malloc(sizeof(Node));  
 newNode->burstTime = burstTime;  
 newNode->priority = priority;  
 newNode->timeLeft = burstTime;  
 newNode->id = queue->lastID + 1;  
  
 queue->lastID = queue->lastID + 1;  
 queue->waitTimeSum += burstTime;  
  
 if (node->prev == NULL && node->priority > priority) {  
 newNode->next = node;  
 node->prev = newNode;  
 newNode->prev = NULL;  
 queue->head = newNode;  
 } else if (node->next == NULL && node->priority < priority) {  
 newNode->prev = node;  
 node->next = newNode;  
 newNode->next = NULL;  
 } else {  
 Node\* prevNode = node->prev;  
 node->prev = newNode;  
 newNode->next = node;  
 newNode->prev = prevNode;  
 prevNode->next = newNode;  
 }  
 newNode->avgWaitTimeOnAppend = getWaitTime(node) / queue->lastID;  
   
 queue->length++;  
 return newNode;  
 }  
 }  
 return NULL;  
}  
  
Node\* pop (Queue\* queue,Node\* node) {  
 if (node->next == NULL && node->prev == NULL) {  
 queue->length = 0;  
 queue->head = NULL;  
 return NULL;  
 }  
 Node\* nextNode = node->next;  
 Node\* prevNode = node->prev;  
  
 if (prevNode == NULL) {  
 queue->head = nextNode;  
 nextNode->prev = NULL;  
 queue->length--;  
 return node;  
 }  
  
 if (nextNode == NULL) {  
 prevNode->next = NULL;  
 queue->length--;  
 return node;  
 }  
  
 prevNode->next = nextNode;  
  
 if (nextNode != NULL)  
 nextNode->prev = prevNode;  
 queue->length--;  
 return node;  
}  
  
  
  
void roundRobin(Queue \*queue, int quantum, FILE\* file) {  
 Node \*node = queue->head;  
 while (queue->length != 0) {  
 queue->processed += quantum;  
 fprintf(file, "{ \"burstTime\": %d, \"priority\": %d, \"id\": %d, \"waitTime\": %f, \"avgWaitTimeOnAppend\": %f, \"processes\": %d, \"processed\": %d, \"totalWaitTime\": %f }", node->burstTime, node->priority, node->id, getWaitTime(node), node->avgWaitTimeOnAppend, queue->length, queue->processed, queue->waitTimeSum);  
 node->timeLeft = node->timeLeft - quantum;  
 printf("Executed node with burstTime %d and priority: %d \n", node->burstTime, node->priority);  
 if(node->timeLeft <= 0) {  
 int round = ((node->burstTime % quantum) != 0) ? (node->burstTime / quantum) + 1 : (node->burstTime / quantum);  
 printf("\tNode with burstTime - %d and priority - %d, done after - %d rounds \n", node->burstTime, node->priority, round);  
 Node\* poppedNode = pop(queue, node);  
 if (queue->length == 0) return;  
 fputs(",\n", file);  
 node = poppedNode->next == NULL ? queue->head : poppedNode->next;  
 } else if (queue->length == 0){  
 return;  
 } else {  
 node = node->next == NULL ? queue->head : node->next;  
 fputs(",\n", file);  
 }  
 printf("\n");  
 }  
  
}  
  
int main()  
{  
 Queue queue = {NULL};  
 FILE\* file = fopen(PATH, "w");  
 fputs("[", file);  
 // push(&queue, 1,1);  
 // push(&queue, 28,4);  
 // push(&queue, 13,2);  
 // push(&queue, 31,6);  
 // push(&queue, 7,8);  
 // push(&queue, 21,9);  
 // push(&queue, 9,3);  
 // push(&queue, 2,7);  
 for (int i = 1; i < 33; ++i) {  
 push(&queue, i, (int)(rand() % 50 + 1) );  
 }  
 roundRobin(&queue, QUANTUM, file);  
  
 fputs("]", file);  
 fclose(file);  
  
}

Результат виконання програми:



Лістнінг програми на Piton, для візуалізації графіків:

import matplotlib.pyplot as plt  
import json  
  
QUANTUM = 3  
  
  
def parseJson(text):  
 return json.loads(text)  
  
  
def main():  
 data = None  
 with open('results.txt') as file:  
 buildPlot(parseJson(file.read()))  
  
  
def buildPlot(data):  
  
 avgTime = {"id": [], 'avgWaitTimeOnAppend': [], 'waitTime': []}  
 avgWaitTimeOnAppend = sorted(data, key=lambda item: item['avgWaitTimeOnAppend'], reverse=False)  
 waitTimes = []  
 idle = [(process['totalWaitTime'] - process['processed']) / process['totalWaitTime'] for process in data]  
 #idle = [(process['totalWaitTime'] - process['waitTime']) / process['totalWaitTime'] for process in data]  
 processes = [process['processes'] for process in data]  
  
 for item in avgWaitTimeOnAppend:  
 if item['avgWaitTimeOnAppend'] not in avgTime['avgWaitTimeOnAppend']:  
 avgTime['id'].append(item['id'])  
 avgTime['avgWaitTimeOnAppend'].append(item['avgWaitTimeOnAppend'])  
 avgTime['waitTime'].append(item['waitTime'])  
  
 for process in data:  
 if process['id'] == max(avgTime['id']):  
 waitTimes.append(process['waitTime'])  
 break  
 waitTimes.append(process['waitTime'])  
  
 plt.plot(list(range(1, len(avgTime['id']) + 1)), avgTime['avgWaitTimeOnAppend'])  
 plt.xlabel("Інтенсивність вхідного потоку")  
 plt.ylabel("Середній час очіквання")  
 plt.title("Графік залежності середнього часу очікування від інтенсивночті входного потоку")  
 plt.figure()  
  
 plt.plot(list(idle), list(reversed(processes)))  
 plt.xlabel("Процент простою")  
 plt.ylabel("Інтенсивність вхідного потоку")  
 plt.title("Графік залежності проценту простою від інтенсивночті входного потоку")  
 plt.figure()  
  
 plt.plot(list( range(1, max(avgTime['id']) + 1) ), waitTimes)  
 plt.xlabel("Пріорітет")  
 plt.ylabel("Середній час очіквання")  
 plt.title("Графік залежності середнього часу очікування від пріорітету")  
  
 plt.show()  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

*Переваги і недоліки дисципліни обслуговування Round Robin:*

Серед переваг даної дисципліни можна виділити:

* Легкий в реалізації з використанням базових структур даних таких як черга.
* Є ефективним підходом в системах управління розподілу часу.
* Оптимальна ефективність може бути встановлена шляхом контролю кванта часу.
* Всі процеси вступають в обробку найближчим часом.

А серед недоліків:

* Є те, що пріоритетність має сенс тільки на першій ітерації, оскільки в подальшому заявки обробляються в ході черги.
* Також при збільшенні кількості заявок, час на обробку кожної з них буде зростати. А отже і в цілому час який буде зайнятий CPU буде збільшуватись.
* Чим менше квант часу тим вище час відклику в системі.

Крім того продуктивність процесора дорівнює 1/N, де N – кількість заявок, а отже чим менше заявок тим продуктивніше, а отже тим швидше вони обробляються.