Міністерство освіти і науки України
Національний технічний університет України "Київський політехнічний інститут імені Ігоря Сікорського"
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

ЗВІТ

Розрахунково-графічна робота

з дисципліни
«Інтелектуальні вбудовані системи»

Виконав:
Василиненко Д.Д.
Студент групи ІП-84
Перевірив:
Регіда Павло Геннадійович

Київ 2021

Мета роботи

Змоделювати роботу планувальника задач у системі реального часу.

Завдання

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

Основні теоретичні відомості

Планування виконання завдань

Протягом існування процесу виконання його потоків може бути багаторазово перерване і продовжене. Перехід від виконання одного потоку до іншого здійснюється в результаті планування і диспетчеризації. Робота по визначенню того, в який момент необхідно перервати виконання поточного активного потоку і якому потоку дати можливість виконуватися, називається плануванням.

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 — дати завдання процесору, якщо це можливо.

• Пропускну 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

Системи масового обслуговування

Система масового обслуговування (СМО) - це система, яка обслуговує вимоги, що надходять до неї (заявки). Основними елементами системи є вхідний потік вимог, канали обслуговування, черга вимог та вихідний потік вимог.

Вимоги (заявки) на обслуговування надходять через дискретні (постійні або випадкові) інтервали часу. Важливо знати закон розподілу вхідного потоку. Обслуговування триває деякий час, постійний або випадковий. Випадковий характер потоку заявок та часу обслуговування призводить до того, що в деякі моменти часу на вході СМО може виникнути черга, в інші моменти – канали можуть бути недозавантаженими або взагалі простоювати.

Задача теорії масового обслуговування полягає в побудові моделей, які пов‘язують задані умови роботи СМО з показниками ефективності системи, що описують її спроможність впоратися з потоком вимог. Під ефективністю, що обслуговує системи розуміють характеристику рівня виконання цією системою функцій, для яких вона призначена.

В залежності від наявності можливості очікування вступниками вимогами початку обслуговування СМО поділяються на:

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

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

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

Алгоритми планування процесів

Найчастіше зустрічаються такі дві групи алгоритмів планування: побудовані на принципі квантування або на принципі пріоритетів. В першому випадку зміна активного процесу відбувається, якщо:

• процес закінчився і покинув систему;

• процес перейшов в стан Очікування;

• закінчився квант процесорного часу, відведений даному процесові.

Процес, для якого закінчився його квант, переводиться в стан готовності і очікує, коли йому буде надано новий квант процесорного часу, а на виконання відповідно до певного правила вибирається новий процес з черги готових. Жодний процес не захоплює процесор надовго, тому квантування широко використовується в системах розподілу часу. По-різному може бути організована черга готових процесів:

• циклічно;

• FIFO (перший прийшов — перший обслуговується);

• LIFO (останній прийшов — перший обслуговується).

В другому випадку використовується поняття ”пріоритет”. Пріоритет — це число, яке характеризує ступінь привілейованості процесу при використанні ресурсів комп’ютеру, зокрема, процесорного часу. Чим вище пріоритет, тим вище привілеї, тим менше часу він буде проводити в чергах.

Пріоритетами можуть призначатись адміністратором системи в залежності від важливості роботи, або внесеної плати, або обчислюватись самою ОС за певними правилами. Він може залишатись фіксованим упродовж всього життя процесу або мінятись в часі відповідно до деякого закону. В останньому випадку пріоритети називають динамічними. Є алгоритми, які використовують:

• відносні пріоритети;

• абсолютні пріоритети.

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 P ресурсів (з порядковими номерами p) та X заявок (з порядковими номерами x), які необхідно виконати. Тоді перша заявка (x = 1) назначається для виконання на першому ресурсі (p = 1), друга (x = 2) – другому і т.д., до досягнення зайнятості останнього ресурсу (p = P, x = P) або до вичерпування необроблюваних заявок (x = X). Усі наступні заявки будуть розподілені по ресурсах

аналогічно до попередніх, починаючи з першого ресурсу (x = P + 1 → p = 1, x = P + 2 → p = 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.

Аналіз отриманих графіків

Залежність середнього часу очікування заявок від інтенсивності:

При збільшенні інтенсивності середній час очікування спершу різко зростає, а згодом лишається на відносно сталому рівні, коли система є завантаженою і певна частина заявок відкидається.

Залежність відсотку простою системи від інтенсивності:

З наведеного графіка видно, що відсоток простою системи стрімко спадає і наближається до 0 зі зростанням інтенсивності незалежно від обраного алгоритму планування.

Залежність відсотку пропущених дедлайнів від інтенсивності:

Найбільший відсоток пропущених дедлайнів має алгоритм FIFO. Алгоритм RM має невеликий відсоток пропущених заявок навіть при незначній інтенсивності. EDF має найменший відсоток пропущених дедлайнів.

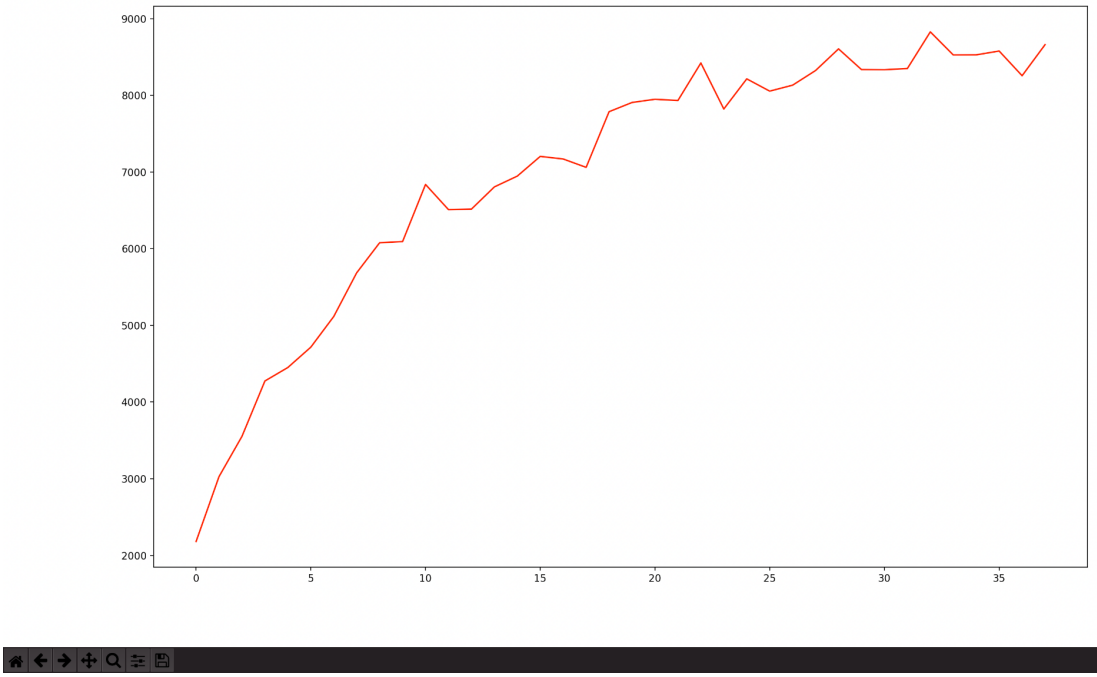
Залежність відсотку пропущених дедлайнів від кількості заявок:

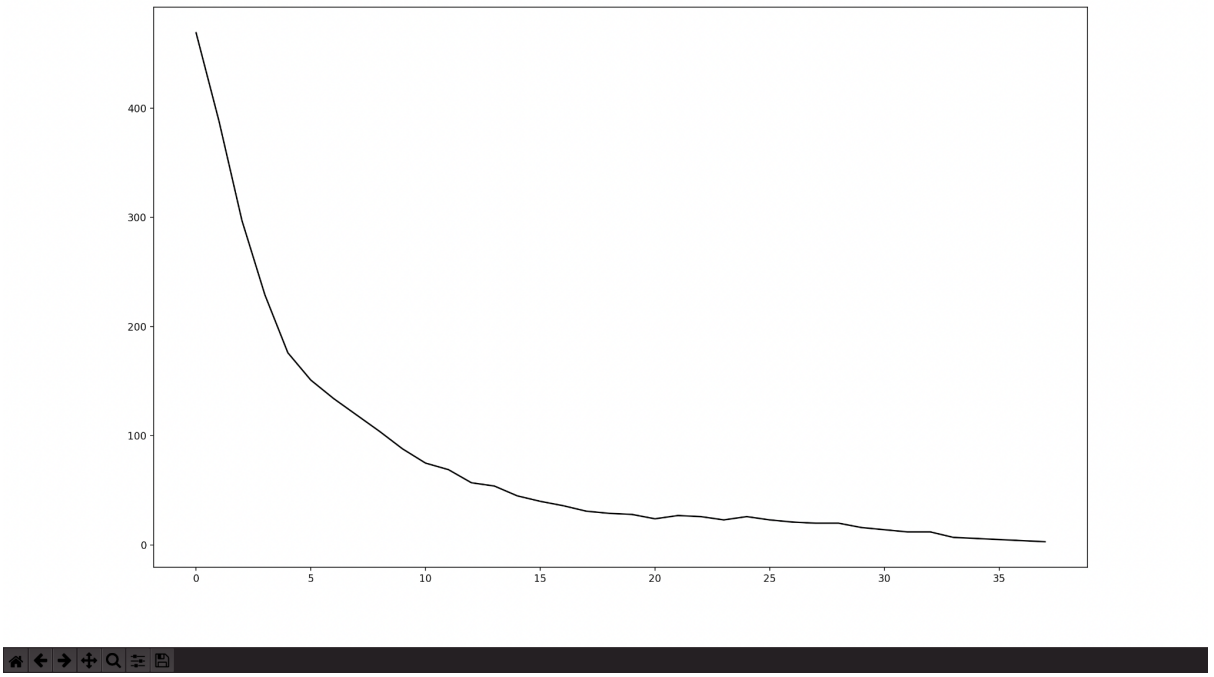
При збільшенні кількості заявок, що надходять до системи, відсоток пропущених дедлайнів зростає.

Залежність середнього часу очікування від кількості заявок:

Середній найбільший час очікування має FIFO, а найменший – RM.

Результати Роботи





Висновки

Під час виконання розрахунково-графічної роботи я ознайомився з теорією планування виконання завдань та систем масового обслуговування. У результаті роботи було створено планувальник з 2 доступними дисциплінами: Rate Monotonic та Earliest Deadline First.

Додаток (Код програми)

```
import matplotlib.pyplot as plt
import random
import er as er
import task
import rm
import edf
import numpy

a = []
lam = 1
k = 2
E = er.erDistribution(k, lam)

def GenerateQ():
   Q = []
   i = 0
```

```
    time = GenerateTime()
    while i < 10000:
        i += E.GenerateNextInternal() * 8
        t = task.task(i, time)
        Q.append(t)
    return Q

def GenerateTime():
    rnd = random.random()
    if rnd < 0.3:
        ans = random.randrange(7)  # Rxy
    elif 0.3 <= rnd < 0.6:
        ans = random.randrange(5)  # Rxx
    elif 0.6 <= rnd < 0.8:
        ans = random.randrange(3)  # Dx
    else:
        ans = random.randrange(2)  # Mx
    return ans + 40



if __name__ == "__main__":
    QuEDF = []
    QuRM = []
    smos = []
    RMs = []
    Tw = []
    Tww =[]
    Tn = []
    t = [x for x in range(10000)]
    faults = []
    # Changes
    faultschange = []
    Tnchange = []
    Twchange = []
    FullWaitTime = []

    for lam in numpy.arange(1, 20, 0.5):
        E.ChangeLambda(lam)
        temp = GenerateQ()
        QuEDF.append(temp)

    QuRM = QuEDF[:]

    for i in range(len(QuEDF)):
        smos.append(edf.EDF(QuEDF[i]))
```

```
        RMs.append(rm.RM(QuRM[i]))

    buf1 = []
    buf2 = []
    buf3 = []

    def calculate(arr):
        for elem in arr:
            elem.work()
            buf1 = elem.GetFaults()
            buf2 = elem.GetWaitTimes()
            buf3 = elem.GetProcessorFreeTime()
            faults.append(buf1[:])
            Tw.append(buf2[:])
            Tn.append(buf3[:])

        for elem in range(len(arr)):
            faultschange.append(faults[elem][9999])
            Tnchange.append(Tn[elem][9999])

        for o in range(len(arr)):
            time = 0
            for elem in range(10000):
                time += Tw[o][elem]
            FullWaitTime.append(time)

        for elem in range(len(arr)):
            Tww.append(FullWaitTime[elem])

        plt.plot(Tnchange, 'r')
        plt.show()
        plt.plot(faultschange, 'g')
        plt.show()
        plt.plot(Tww, 'k')
        plt.show()

        buf1.clear()
        buf2.clear()
        buf3.clear()
        Tw.clear()
        Tn.clear()
        faults.clear()
        faultschange.clear()
        Tnchange.clear()
        Twchange.clear()
```

```
        Tww.clear()

    calculate(smos)

    calculate(RMs)
```

RM.py

```
class RM:
    currentTime = 0
    Tw = [0 for x in range(10000)]
    Tn = [0 for y in range(10000)]
    faults = [0 for z in range(10000)]
    Q = []
    Qready = []
    currentTask = None

    def __init__(self, Q):
        self.Q = Q

    def getEDTask(self, removeFromQ):
        timeBuf = 9999999
        taskwED = None
        for i in range(len(self.Qready)):
            newTime = self.Qready[i].getDeadline()
            if timeBuf > newTime:
                timeBuf = newTime
                taskwED = self.Qready[i]
        if removeFromQ:
            self.Qready.remove(taskwED)
        return taskwED

    def ToReadyQueue(self):
        for i in range(len(self.Q)):
            if self.Q[i].getCreationTime() == self.currentTime:
                self.Qready.append(self.Q[i])

    def CheckForDeadlines(self):
        flt = 0
        flti = []
        for i in range(len(self.Qready)):
```

```
            if self.Qready[i].getDeadline() < self.currentTime:
                flt += 1
                flti.append(i)
        if self.currentTime == 0:
            self.faults[self.currentTime] = flt
        else:
            self.faults[self.currentTime] = self.faults[self.currentTime - 1] + flt
        for i in range(len(flti)):
            del self.Qready[flti[i]]
            for j in range(i, len(flti)):
                flti[j] -= 1

    # Modelling
    def work(self):
        self.currentTime = 0
        for self.currentTime in range(10000):
            if self.currentTime != 0:
                self.Tn[self.currentTime] = self.Tn[self.currentTime - 1]
            timewait = 0
            self.CheckForDeadlines()
            self.ToReadyQueue()
            if self.currentTask is not None and self.currentTask.getExecutionTime()
== 0:
                self.currentTask = None
            elif self.currentTask is not None:
                self.currentTask.workedOn()
            if self.getEDTask(False) is None and self.currentTask is None:
                self.Tn[self.currentTime] += 1
                continue
            elif self.currentTask is None:
                self.currentTask = self.getEDTask(True)
            for task in self.Qready:
                task.wait()
                timewait += 1
            self.Tw[self.currentTime] = timewait

    def GetWaitTimes(self):
        return self.Tw

    def GetFaults(self):
        return self.faults

    def GetProcessorFreeTime(self):
        return self.Tn
```

er.py

```
import random
import math

class erDistribution:

   k = 0           # order
   lam = 0         # rate

   def __init__(self, k, lam):
       self.lam = lam
       self.k = k
       if k == 0:
           raise Exception("Order parameter can't be less than 1!")
       if lam <= 0:
           raise Exception("Streaming rate can't be less or equal 0!")

   def GenerateNext(self):
       res = 0
       for n in range(self.k - 1):
           if res != 0:
               res = random.random() * res
           else:
               res = random.random()
       res = 0 - (math.log(res) / self.lam)
       return res

   def GenerateNextInternal(self):
       return random.gammavariate(self.k, self.lam)

   def ChangeLambda(self, lam):
       self.lam = lam
```

edf.py

```
class EDF:
   currentTime = 0
   Tw = [0 for x in range(10000)]
   Tn = [0 for y in range(10000)]
   faults = [0 for z in range(10000)]
   Q = []
   Qready = []
   currentTask = None
```

```
def __init__(self, Q):
    self.Q = Q

def GetEDTask(self, removeFromQ):
    timeBuf = 9999999
    taskwED = None
    for i in range(len(self.Qready)):
        newTime = self.Qready[i].getDeadline()
        if timeBuf > newTime:
            timeBuf = newTime
            taskwED = self.Qready[i]
    if removeFromQ:
        self.Qready.remove(taskwED)
    return taskwED

def ToReadyQueue(self):
    for i in range(len(self.Q)):
        if self.Q[i].getCreationTime() == self.currentTime:
            self.Qready.append(self.Q[i])

def CheckForDeadlines(self):
    flt = 0
    flti = []
    for i in range(len(self.Qready)):
        if self.Qready[i].getDeadline() < self.currentTime:
            flt += 1
            flti.append(i)
    if self.currentTime == 0:
        self.faults[self.currentTime] = flt
    else:
        self.faults[self.currentTime] = self.faults[self.currentTime - 1] + flt
    for i in range(len(flti)):
        del self.Qready[flti[i]]
        for j in range(i, len(flti)):
            flti[j] -= 1

# Modelling
def work(self):
    self.currentTime = 0
    for self.currentTime in range(10000):
        if self.currentTime != 0:
            self.Tn[self.currentTime] = self.Tn[self.currentTime - 1]
        timewait = 0
        self.CheckForDeadlines()
        self.ToReadyQueue()
```

```
            if self.currentTask is not None and self.currentTask.getExecutionTime()
== 0:
                self.currentTask = None
            elif self.currentTask is not None and self.GetEDTask(False) is not None:
                if self.GetEDTask(False).getExecutionTime() <
self.currentTask.getExecutionTime():
                    self.Qready.append(self.currentTask)
                    self.currentTask = self.GetEDTask(True)
            elif self.currentTask is not None:
                self.currentTask.workedOn()
            if self.GetEDTask(False) is None and self.currentTask is None:
                self.Tn[self.currentTime] += 1
                continue
            elif self.currentTask is None:
                self.currentTask = self.GetEDTask(True)
            for task in self.Qready:
                task.wait()
                timewait += 1
            self.Tw[self.currentTime] = timewait

    def GetWaitTimes(self):
        return self.Tw

    def GetFaults(self):
        return self.faults

    def GetProcessorFreeTime(self):
        return self.Tn
```