**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО**

Факультет інформатики та обчислювальної техніки Кафедра інформатики та програмної інженерії

Звіт з комп’ютерного практикуму №2

«Об’єктно-орієнтований підхід до побудови імітаційних моделей дискретно-подійних систем.»

роботи з дисципліни: « Моделювання систем »

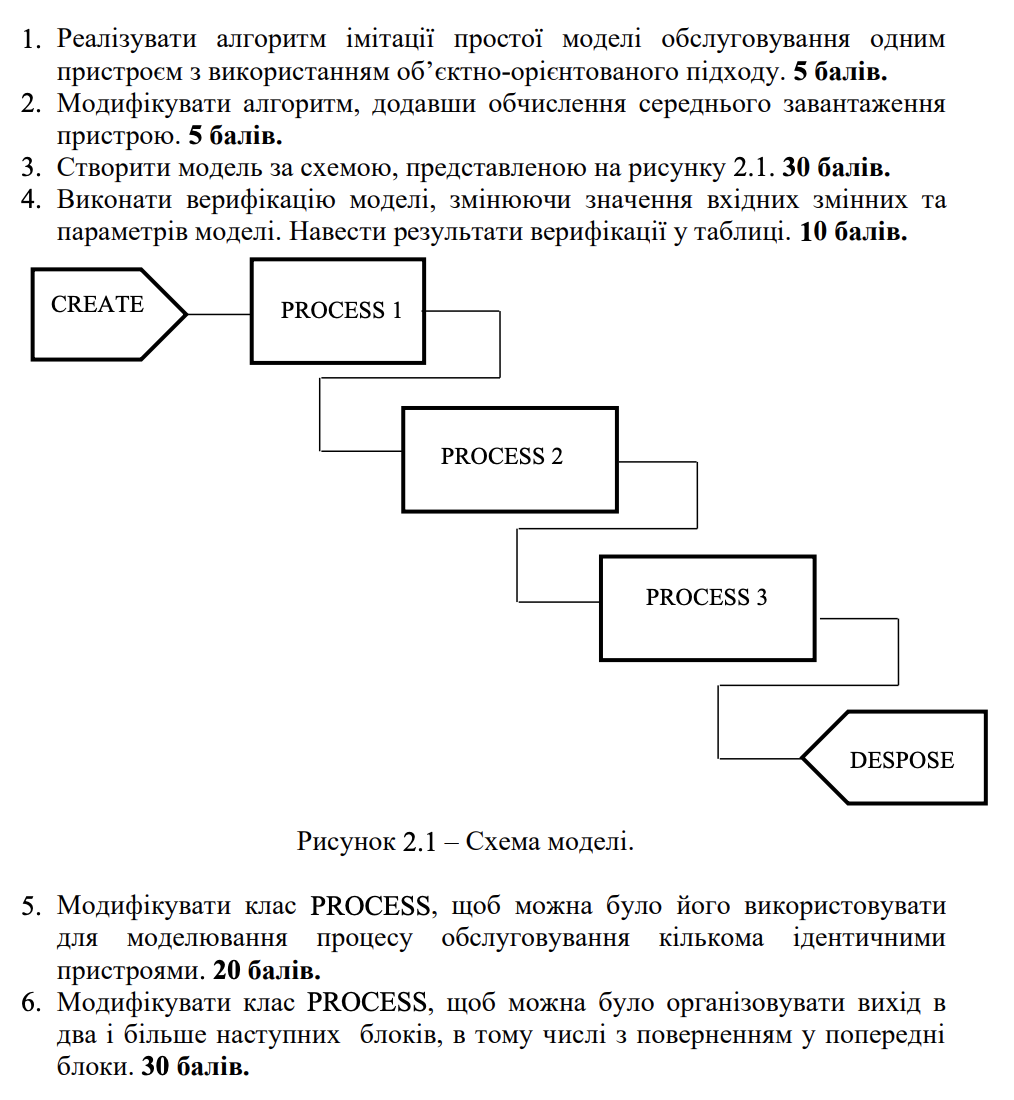
Студент: Мєшков Андрій Ігорович\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Група: ІП-15\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Викладач: асистент Дифучин А. Ю.\_\_\_\_\_\_\_\_\_\_

Київ, 2024

# Завдання



# Хід роботи

Було створено модель.

*import* Foundation

let create = Create(name: "create", delay: 1)

let process1 = Process(name: "process1", delays: [5, 5, 5, 5, 5])

let process2 = Process(name: "process2", delays: [1.8, 1.8])

let process3 = Process(name: "process3", delays: [1])

create.transfer = SoloTransfer(nextElement: process1)

process1.transfer = SoloTransfer(nextElement: process2)

process2.transfer = SoloTransfer(nextElement: process3)

process1.maxQueue = 1

process2.maxQueue = 2

process3.maxQueue = 5

create.distribution = .exp

process1.distribution = .exp

process2.distribution = .exp

process3.distribution = .exp

let model = Model(elements: [create, process1, process2, process3])

model.simulate(timeModeling: 1000)

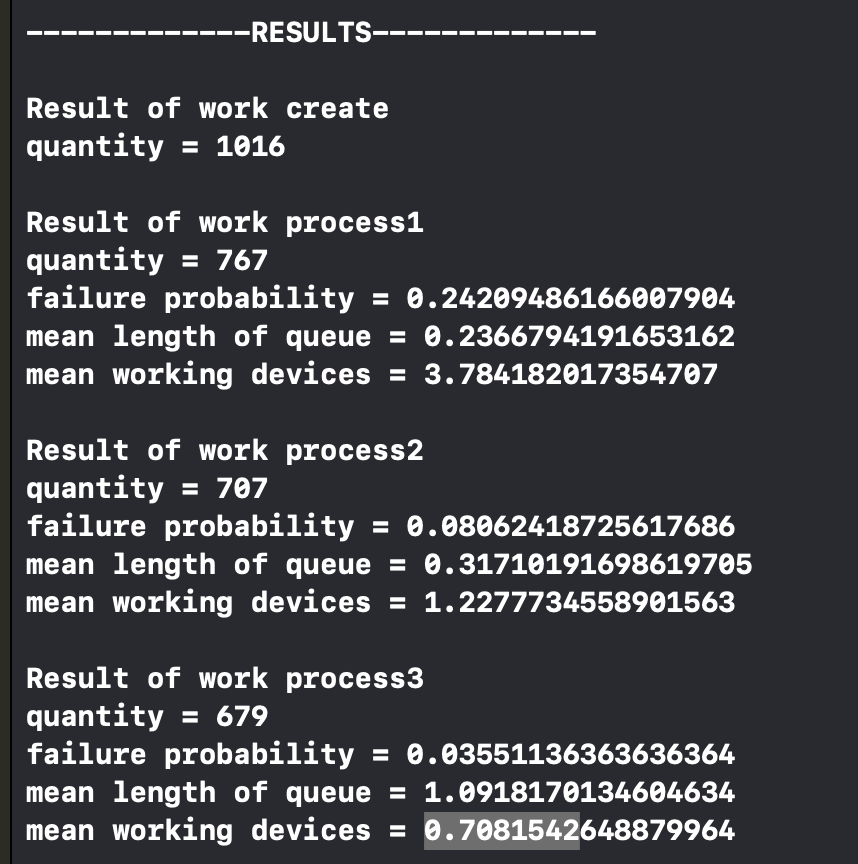
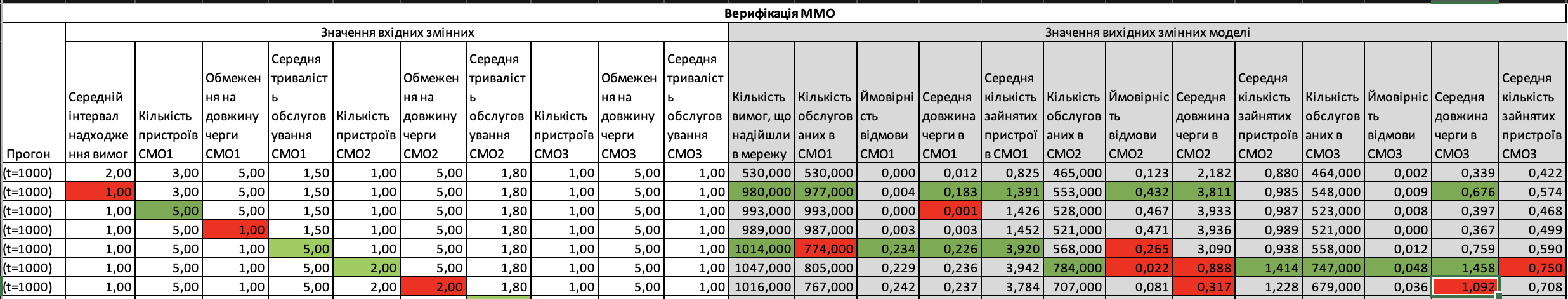


Рисунок 2.1. Результат імітації

Перевіримо працездатність моделі:



Як бачимо, модель відповідає всім правилам, а отже пройшла верифікацію.

Модифікуємо програму.

*import* Foundation

let create = Create(name: "create", delay: 1)

let process1 = Process(name: "process1", delays: [5, 5, 5, 5, 5])

let process2 = Process(name: "process2", delays: [1.8, 1.8])

let process3 = Process(name: "process3", delays: [1])

let process4 = Process(name: "process4", delays: [0.1])

create.transfer = SoloTransfer(nextElement: process1)

process1.transfer = SoloTransfer(nextElement: process2)

process2.transfer = TranferWithProbability(probabilities: [

TransferProbability(probability: 0.5, nextElement: process1),

TransferProbability(probability: 0.5, nextElement: process3),

TransferProbability(probability: 0.5, nextElement: process4)

])

process1.maxQueue = 1

process2.maxQueue = 2

process3.maxQueue = 5

process4.maxQueue = 1

create.distribution = .exp

process1.distribution = .exp

process2.distribution = .exp

process3.distribution = .exp

process4.distribution = .exp

let model = Model(elements: [create, process1, process2, process3, process4])

model.simulate(timeModeling: 1000)

# ВИСНОВКИ

У результаті виконання практичної роботи було розроблено модель масового обслуговування, що підтримує роботу декількох пристороїв та переходи з ймовірностями. Було проведено верифікацію моделі

ЛІСТИНГ КОДУ

--- main.swift ---

import Foundation

let create = Create(name: "create", delay: 1)

let process1 = Process(name: "process1", delays: [5, 5, 5, 5, 5])

let process2 = Process(name: "process2", delays: [1.8, 1.8])

let process3 = Process(name: "process3", delays: [1])

let process4 = Process(name: "process4", delays: [0.1])

create.transfer = SoloTransfer(nextElement: process1)

process1.transfer = SoloTransfer(nextElement: process2)

//process2.transfer = SoloTransfer(nextElement: process3)

process2.transfer = TranferWithProbability(probabilities: [

TransferProbability(probability: 0.5, nextElement: process1),

TransferProbability(probability: 0.5, nextElement: process3),

TransferProbability(probability: 0.5, nextElement: process4)

])

process1.maxQueue = 1

process2.maxQueue = 2

process3.maxQueue = 5

process4.maxQueue = 1

create.distribution = .exp

process1.distribution = .exp

process2.distribution = .exp

process3.distribution = .exp

process4.distribution = .exp

// let model = Model(elements: [create, process1, process2, process3])

let model = Model(elements: [create, process1, process2, process3, process4])

model.simulate(timeModeling: 1000)

--- Element/Process.swift ---

import Foundation

class Process: Element {

private var queue = 0

var maxQueue = Int.max

private(set) var failure = 0

private(set) var meanQueue = 0.0

private(set) var loadTime = 0.0

private(set) var workingDevicesCount = 0.0

init(name: String, delays: [Double]) {

super.init(nameOfElement: name, delays: delays)

}

override func inAct() {

switch state {

case .free:

devices.first(where: {$0.state == .free})?.inAct()

case .working:

if queue < maxQueue {

queue += 1

} else {

failure += 1

}

}

}

override func outAct() {

super.outAct()

let outCount = devices.filter({ $0.tNext == tCurr }).count

devices.filter({ $0.tNext == tCurr }).forEach({ $0.outAct() })

(0..<outCount).forEach({ \_ in

if queue > 0 {

queue -= 1

devices.first(where: {$0.state == .free})?.inAct()

} else {

devices.filter({ $0.state == .free }).forEach({ $0.tNext = .infinity })

}

transfer?.goNext()

})

}

override func printInfo() {

super.printInfo()

print("failure = \(failure)")

}

override func doStatisctic(delta: Double) {

meanQueue += Double(queue) \* delta

if devices.filter({ $0.state == .working }).count > 0 {

loadTime += delta

}

workingDevicesCount += Double(devices.filter({ $0.state == .working }).count) \* delta

}

}

--- Element/Element.swift ---

import Foundation

class Element {

private static var nextID = 0

let name: String

var distribution = Method.exp

private(set) var quantity = 0

var tCurr = 0.0 {

didSet {

devices.forEach({ $0.tCurr = tCurr })

}

}

var transfer: Transfer?

let id: Int

var devices: [Device]

init(nameOfElement: String, delays: [Double]) {

id = Element.nextID

Element.nextID += 1

name = nameOfElement

devices = []

delays.forEach({ devices.append(Device(delay: $0, distribution: distribution)) })

}

var tNext: Double {

devices.map({ $0.tNext }).min()!

}

var state: State {

if devices.filter({ $0.state == .free }).count > 0 {

return .free

} else {

return .working

}

}

func inAct() {

}

func outAct() {

quantity += 1

}

func printResult() {

print("Result of work \(name) \nquantity = \(quantity)")

}

func printInfo() {

print("\(name) state = \(state.rawValue) quantity = \(quantity) tNext = \(tNext)")

}

func doStatisctic(delta: Double) {

}

}

enum Method {

case exp, norm, unif

}

enum State: String {

case working, free

}

--- Element/Create.swift ---

import Foundation

class Create: Element {

init(name: String, delay: Double) {

super.init(nameOfElement: name, delays: [delay])

devices[0].tNext = 0

}

override func outAct() {

super.outAct()

devices[0].outAct()

devices[0].tNext += devices[0].delay

transfer?.goNext()

}

}

--- Transfer/SoloTransfer.swift ---

import Foundation

class SoloTransfer: Transfer {

let nextElement: Element

init(nextElement: Element) {

self.nextElement = nextElement

}

func goNext() {

nextElement.inAct()

}

}

--- Transfer/TranferWithProbability.swift ---

import Foundation

class TranferWithProbability: Transfer {

let probabilities: [TransferProbability]

let maxProbability: Double

init(probabilities: [TransferProbability]) {

self.probabilities = probabilities

maxProbability = probabilities.reduce(0.0, { $0 + $1.probability })

}

func goNext() {

let number = Double.random(in: 0..<maxProbability)

var currentProbability = 0.0

for probability in probabilities {

currentProbability += probability.probability

if currentProbability > number {

probability.nextElement.inAct()

break

}

}

}

}

struct TransferProbability {

let probability: Double

let nextElement: Element

}

--- Transfer/Transfer.swift ---

import Foundation

protocol Transfer {

func goNext()

}

--- FunRand/FunRand.swift ---

import Foundation

class FunRand {

private class var generateA: Double {

var a = Double.random(in: 0..<1)

while a == 0 {

a = Double.random(in: 0..<1)

}

return a

}

class func exp(timeMean: Double) -> Double {

var a = generateA

a = -timeMean \* log(a)

return a

}

class func unif(timeMin: Double, timeMax: Double) -> Double {

var a = generateA

a = timeMin + a \* (timeMax - timeMin)

return a

}

class func norm(timeMean: Double, timeDeviation: Double) -> Double {

timeMean + timeDeviation \* Double.random(in: -Double.infinity...Double.infinity)

}

}

--- Model/Model.swift ---

import Foundation

class Model {

private var tNext = 0.0

private var tCurr = 0.0

private var event = 0

private let elements: [Element]

init(elements: [Element]) {

self.elements = elements

}

func simulate(timeModeling: Double) {

while tCurr < timeModeling {

tNext = Double.infinity

elements.forEach { element in

if element.tNext < tNext {

tNext = element.tNext

}

}

elements.forEach({ $0.doStatisctic(delta: tNext - tCurr) })

tCurr = tNext

elements.forEach({ $0.tCurr = tCurr })

elements.forEach { element in

if element.tNext == tCurr {

element.outAct()

print("It's time for event in \(element.name) time = \(tCurr)")

}

}

elements.forEach({ $0.printInfo() })

}

printResult()

}

private func printResult() {

print("\n-------------RESULTS-------------\n")

elements.forEach { element in

element.printResult()

if let process = element as? Process {

print("failure probability = \(Double(process.failure) / Double(process.quantity + process.failure))")

print("mean length of queue = \(process.meanQueue / tCurr)")

print("mean working devices = \(process.workingDevicesCount / tCurr)")

}

print("")

}

}

}

--- Device/Device.swift ---

import Foundation

class Device {

var tNext = 0.0

private let delayMean: Double

var tCurr = 0.0

var state = State.free

let distribution: Method

private let delayDev = 0.0

var delay: Double {

var delay = delayMean

switch distribution {

case .exp:

delay = FunRand.exp(timeMean: delayMean)

case .norm:

delay = FunRand.norm(timeMean: delayMean, timeDeviation: delayDev)

case .unif:

delay = FunRand.unif(timeMin: delayMean, timeMax: delayDev)

}

return delay

}

init(delay: Double, distribution: Method) {

self.delayMean = delay

self.distribution = distribution

}

func inAct() {

state = .working

tNext = tCurr + delay

}

func outAct() {

state = .free

}

}