**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 №3

«Побудова імітаційної моделі системи з використанням формалізму моделі масового обслуговування.»

роботи з дисципліни: « Моделювання систем »

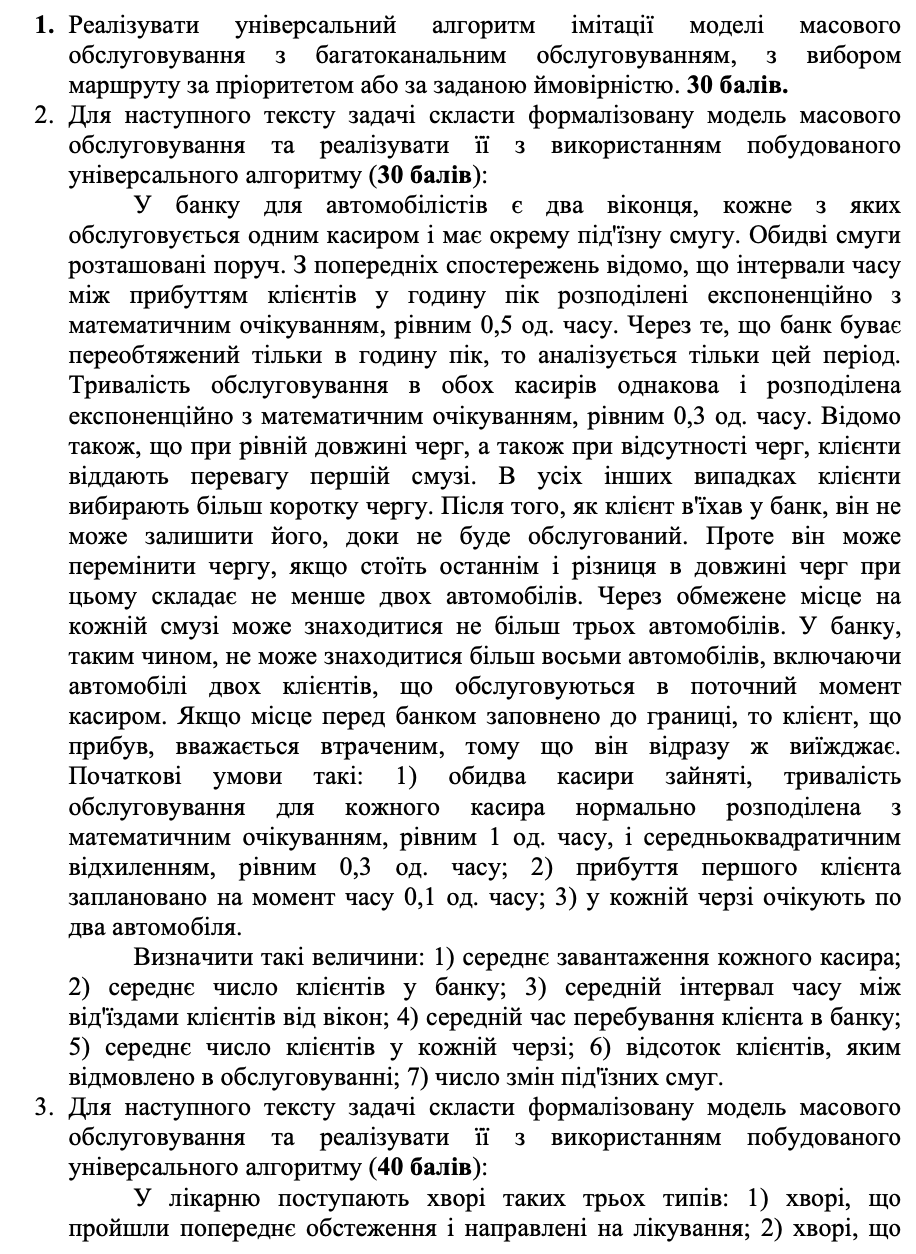
Студент: Мєшков Андрій Ігорович\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

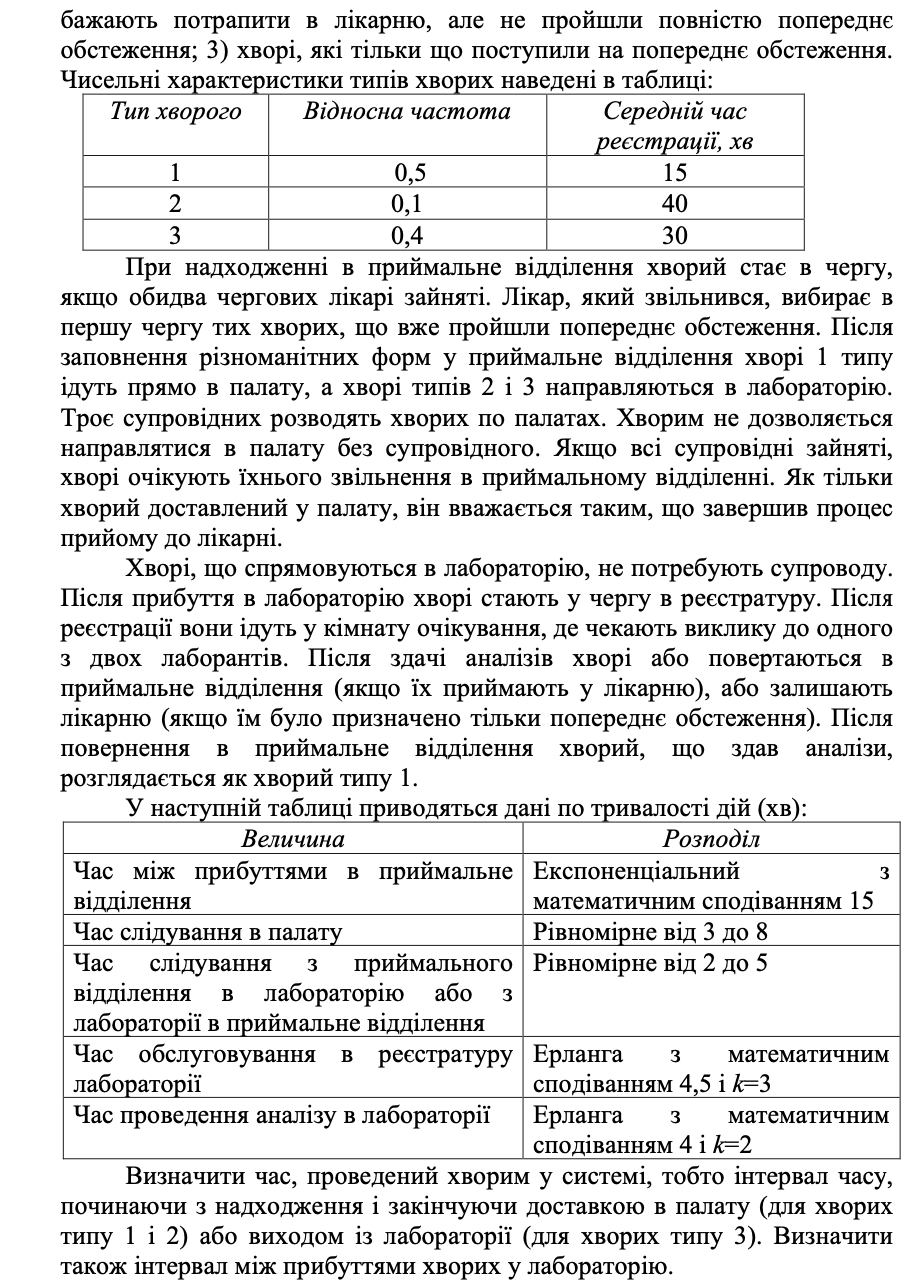
Група: ІП-15\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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

Київ, 2024

# Завдання

****

****

# Хід роботи

Задача 2:

Модифікуємо модель та створимо екземпляр, що вирішує задачу.

private func bank() -> Model {

let create = Create(name: "create", delay: 0.5)

let window1 = Process(name: "window1", delays: [0.3])

let window2 = Process(name: "window2", delays: [0.3])

create.transfer = PriorityTransferWithQueueCheck(elements: [

ElementWithPriority(element: window1, priority: PriorityConstant.hight),

ElementWithPriority(element: window2, priority: PriorityConstant.low)

])

window1.queue = RelatedQueue(relatedProcesses: [window2], minimumTransitionDifference: 2, maxLength: 3)

window2.queue = RelatedQueue(relatedProcesses: [window1], minimumTransitionDifference: 2, maxLength: 3)

create.distribution = .exp

window1.distribution = .exp

window2.distribution = .exp

let firstTimeNext = abs(FunRand.norm(timeaverage: 1, timeDeviation: 0.3))

window1.devices[0].tNext = firstTimeNext

window1.devices[0].state = .working

let secondTimeNext = abs(FunRand.norm(timeaverage: 1, timeDeviation: 0.3))

window2.devices[0].tNext = secondTimeNext

window2.devices[0].state = .working

create.devices[0].tNext = 0.1

*try*! window1.queue.add(Task())

*try*! window1.queue.add(Task())

*try*! window2.queue.add(Task())

*try*! window2.queue.add(Task())

*return* Model(elements: [create, window1, window2], resultsPrinter: bankResultsPrinter)

}

Реалізуємо перехід з пріоритетом.

class PriorityTransferWithQueueCheck: Transfer {

let elements: [ElementWithPriority]

init(elements: [ElementWithPriority]) {

*self*.elements = elements

}

func goNext(for task: Task) {

let elementsWithEmptyDevice = *self*.elements.filter({ *$0*.element.state == .free })

*if* let maxElement = elementsWithEmptyDevice.max(by: { *$0*.priority.value < *$1*.priority.value }) {

elementsWithEmptyDevice.filter({ *$0*.priority.value == maxElement.priority.value }).randomElement()?.element.inAct(task: task)

*return*

}

let minQueueLength = *self*.elements.compactMap { element *in*

*if* element.element.canAccept {

*return* element.element.queueLength

} *else* {

*return* nil

}

}.min()

let elementsWithEmptyQueue = *self*.elements.filter({ *$0*.element.queueLength == minQueueLength && *$0*.element.canAccept })

*if* let maxElement = elementsWithEmptyQueue.max(by: { *$0*.priority.value < *$1*.priority.value }) {

elementsWithEmptyQueue.filter({ *$0*.priority.value == maxElement.priority.value }).randomElement()?.element.inAct(task: task)

*return*

}

elements.max(by: { *$0*.priority.value < *$1*.priority.value })?.element.inAct(task: task)

}

}

Реалізуємо чергу, що дозволяє переїзди машин.

class RelatedQueue: Queue {

let relatedProcesses: [Process]

let minimumTransitionDifference: Int

init(relatedProcesses: [Process], minimumTransitionDifference: Int, maxLength: Int) {

*self*.relatedProcesses = relatedProcesses

*self*.minimumTransitionDifference = minimumTransitionDifference

*super*.init(maxLength: maxLength)

}

override func remove() throws -> Task {

let result = *try* *super*.remove()

outerloop: *while* status != .full {

*for* process *in* relatedProcesses {

*if* process.queue.status != .full && currentLength + minimumTransitionDifference <= process.queue.currentLength {

let newTask = *try* process.queue.remove()

*try* add(newTask)

relatedCount += 1

*continue* outerloop

}

}

*break*

}

*return* result

}

}

Винесемо виведення результатів в окрему логіку.

*import* Foundation

class ResultsPrinter {

private let processResultsConfig: [ProcessResultsOption]

private let createResultsConfig: [CreateResultsOption]

private let modelResultsConfig: [ModelResultsOption]

init(processResultsConfig: [ProcessResultsOption], createResultsConfig: [CreateResultsOption], modelResultsConfig: [ModelResultsOption]) {

*self*.processResultsConfig = processResultsConfig

*self*.modelResultsConfig = modelResultsConfig

*self*.createResultsConfig = createResultsConfig

}

func printModelResults(\_ model: Model) {

print("[Result of work model]")

modelResultsConfig.forEach({ printModelResult(model, option: *$0*) })

}

private func printModelResult(\_ model: Model, option: ModelResultsOption) {

*switch* option {

*case* .averageTasksInModel:

print("average tasks in model = \(model.tasksInModel / model.tCurr)")

*case* .averageTimeBetweenTaskCompletions:

print("average time between task completions = \(Double(model.processCount) \* model.tCurr / Double(model.tasksCompleted))")

*case* .averageTimeTasksSpendsInModel:

print("average time tasks spends in model = \(model.tasksInModel / Double(model.tasksCompleted))") *// Формула Літтла*

*case* .failureProbability:

print("failure probability = \(model.failureProbability)")

*case* .relatedCount:

print("related count = \(model.relatedCount)")

}

}

func printCreateResults(\_ create: Create) {

print("[Result of work \(create.name)]")

createResultsConfig.forEach({ printCreateResult(create, option: *$0*) })

}

private func printCreateResult(\_ create: Create, option: CreateResultsOption) {

*switch* option {

*case* .quantity:

print("quantity = \(create.quantity)")

}

}

func printProcessResults(\_ process: Process) {

print("[Result of work \(process.name)]")

processResultsConfig.forEach({ printProcessResult(process, option: *$0*) })

}

private func printProcessResult(\_ process: Process, option: ProcessResultsOption) {

*switch* option {

*case* .quantity:

print("quantity = \(process.quantity)")

*case* .averageQueueLength:

print("average length of queue = \(process.averageQueue / process.tCurr)")

*case* .failureProbability:

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

*case* .averageLoadDevice:

print("average load device = \(process.loadTime / process.tCurr)")

*case* .averageWorkingDevice:

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

*case* .averageTasksInWork:

print("average tasks in work = \(process.tasksInWorkCount / process.tCurr)")

*case* .averageTimeBetweenTaskCompletions:

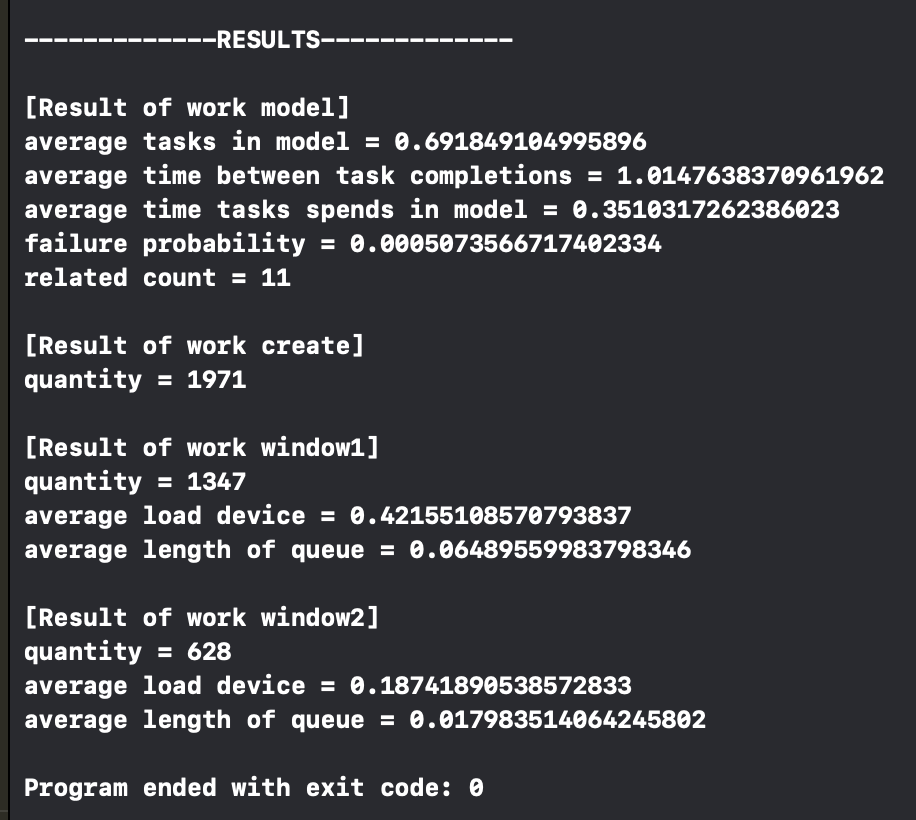
print("average time between task completions = \(Double(process.devices.count) \* process.tCurr / Double(process.quantity))")

}

}

}

Отримаємо результат.



Задача 3

private func hospital() -> Model {

let create = MultitypeCreator(name: "create", delays: [

MultuitypeCreatorProbabilities(delay: 1.5, probability: 0.5),

MultuitypeCreatorProbabilities(delay: 4, probability: 0.1),

MultuitypeCreatorProbabilities(delay: 3, probability: 0.4)

])

let receptionDepartment = Process(name: "reception department", delays: [1.5, 1.5])

let wayToWard = Process(name: "way to ward", delays: [0.55, 0.55, 0.55])

let wayToLaboratory = UnlimitedProcess(name: "way to laboratory", delay: 0.35)

let registry = Process(name: "registry", delays: [0.45])

let laboratory = Process(name: "laboratory", delays: [0.4, 0.4])

let wayToReception = UnlimitedProcess(name: "way to reception", delay: 0.35)

create.transfer = SoloTransfer(nextElement: receptionDepartment)

receptionDepartment.transfer = CustomTransfer { task *in*

*if* task.typeId == 1 {

*return* wayToWard

} *else* {

*return* wayToLaboratory

}

}

wayToLaboratory.transfer = SoloTransfer(nextElement: registry)

registry.transfer = SoloTransfer(nextElement: laboratory)

laboratory.transfer = CustomTransfer { task *in*

*if* task.typeId == 2 {

task.typeId = 1

*return* wayToReception

}

*return* nil

}

wayToReception.transfer = SoloTransfer(nextElement: receptionDepartment)

receptionDepartment.queue = PriorityQueue(maxLength: .max, priorityCheck: { newTask, oldTask *in*

newTask.typeId == 1 && oldTask.typeId != 1

})

wayToWard.queue = Queue(maxLength: .max)

wayToLaboratory.queue = Queue(maxLength: .max)

registry.queue = Queue(maxLength: .max)

laboratory.queue = Queue(maxLength: .max)

wayToReception.queue = Queue(maxLength: .max)

create.distribution = .exp

wayToWard.distribution = .norm

wayToWard.setDelayDev(0.25)

wayToLaboratory.distribution = .norm

wayToLaboratory.setDelayDev(0.15)

registry.distribution = .erlanga

registry.setErlangaValue(3)

laboratory.distribution = .erlanga

laboratory.setErlangaValue(2)

wayToReception.distribution = .norm

wayToReception.setDelayDev(0.15)

*return* Model(elements: [create, receptionDepartment, wayToWard, wayToLaboratory, registry, laboratory, wayToReception], resultsPrinter: hospitalResultsPrinter)

}

Реалізуємо Create, що генерує різні типи задач.

class MultitypeCreator: Create {

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

*super*.init(name: name, delay: delays[0].delay)

devices = [MultuitypeCreatorDevice(delays: delays, distribution: distribution)]

devices[0].tNext = 0

}

}

class MultuitypeCreatorDevice: Device {

private let delays: [MultuitypeCreatorProbabilities]

private var nextType = 1

init(delays: [MultuitypeCreatorProbabilities], distribution: Method) {

*self*.delays = delays

*super*.init(delay: delays[0].delay, distribution: distribution)

}

override var delay: Double {

let newAverageDelay = generatedNextDelay

var delay = newAverageDelay

*switch* distribution {

*case* .exp:

delay = FunRand.exp(timeaverage: newAverageDelay)

*case* .norm:

delay = FunRand.norm(timeaverage: newAverageDelay, timeDeviation: delayDev)

*case* .unif:

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

*case* .erlanga:

delay = FunRand.erlanga(timeaverage: newAverageDelay, k: erlangaValue)

}

*return* delay

}

private var generatedNextDelay: Double {

var randomValue = Double.random(in: 0..<delays.reduce(0.0, { *$0* + *$1*.probability }))

*for* index *in* 0..<delays.count {

let probability = delays[index]

*if* randomValue < probability.probability {

nextType = index + 1

*return* probability.delay

} *else* {

randomValue -= probability.probability

}

}

*return* 0

}

override func outAct() -> Task {

state = .free

*return* Task(typeId: nextType)

}

}

struct MultuitypeCreatorProbabilities {

let delay: Double

let probability: Double

}

Реалізуємо процес з необмеженою кількістю пристроїв

class UnlimitedProcess: Process {

private let constDelay: Double

init(name: String, delay: Double) {

constDelay = delay

*super*.init(name: name, delays: [constDelay])

}

override var canAccept: Bool {

*return* true

}

override var state: State {

.free

}

override func inAct(task: Task) {

*if* !devices.contains(where: { *$0*.state == .free }) {

devices.append(Device(delay: constDelay, distribution: distribution))

}

*super*.inAct(task: task)

}

}

Реалізуємо перехід з гнучкою умовою

class CustomTransfer: Transfer {

private let nextElement: (Task) -> Element?

init(nextElement: @escaping (Task) -> Element?) {

*self*.nextElement = nextElement

}

func goNext(for task: Task) {

let element = nextElement(task)

element?.inAct(task: task)

}

}

Реалізуємо чергу з пріоритетом

class PriorityQueue: Queue {

private let priorityCheck: (Task, Task) -> Bool

init(maxLength: Int, priorityCheck: @escaping (Task, Task) -> Bool) {

*self*.priorityCheck = priorityCheck

*super*.init(maxLength: maxLength)

}

override func add(\_ task: Task) throws {

*guard* currentLength < maxLength *else* {

*throw* QueueError.queueFill

}

*if* tasks.isEmpty {

tasks.append(task)

*return*

}

var isInsert = false

*for* taskIndex *in* 0..<tasks.count {

*if* priorityCheck(task, tasks[taskIndex]) {

isInsert = true

tasks.insert(task, at: taskIndex)

*break*

}

}

*if* !isInsert {

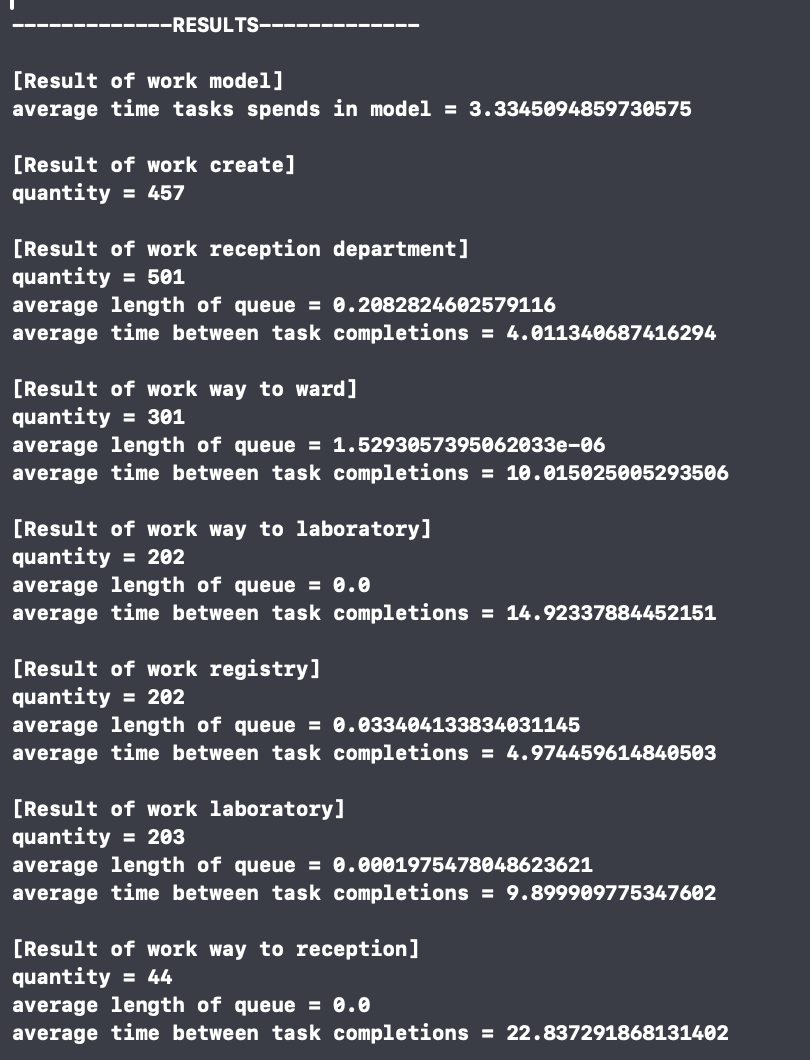
tasks.append(task)

}

}

}

Отримаємо результати



# ВИСНОВКИ

У результаті виконання практичної роботи було модифіковано модель. Створено моделі для розв’язання поставлених задач. Протестовано моделі.

ЛІСТИНГ КОДУ

--- main.swift ---

import Foundation

let model = ModelFactory().makeModel(.priorityTest)

model.simulate(timeModeling: 1000)

--- Model/.DS\_Store ---

--- Model/Element/Process.swift ---

import Foundation

class Process: Element {

var queue = Queue(maxLength: .max)

private(set) var failure = 0

private(set) var averageQueue = 0.0

private(set) var loadTime = 0.0

private(set) var workingDevicesCount = 0.0

private(set) var tasksInWorkCount = 0.0

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

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

}

override var canAccept: Bool {

queue.status != .full

}

override var queueLength: Int {

queue.currentLength

}

override var queueRelatedCount: Int {

return queue.relatedCount

}

override func inAct(task: Task) {

switch state {

case .free:

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

case .working:

do {

try queue.add(task)

} catch {

failure += 1

}

}

}

override func outAct() {

super.outAct()

let outTasks = devices.filter({ $0.tNext == tCurr }).map({ $0.outAct() })

(0..<outTasks.count).forEach({ index in

do {

if queue.currentLength > 0 {

let newTask = try queue.remove()

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

} else {

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

}

transfer?.goNext(for: outTasks[index])

} catch {

print(error)

}

})

}

override func printInfo() {

super.printInfo()

print("failure = \(failure)")

}

override func doStatisctic(delta: Double) {

averageQueue += Double(queue.currentLength) \* delta

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

loadTime += delta

}

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

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

}

}

--- Model/Element/UnlimitedProcess.swift ---

import Foundation

class UnlimitedProcess: Process {

private let constDelay: Double

init(name: String, delay: Double) {

constDelay = delay

super.init(name: name, delays: [constDelay])

}

override var canAccept: Bool {

return true

}

override var state: State {

.free

}

override func inAct(task: Task) {

if !devices.contains(where: { $0.state == .free }) {

devices.append(Device(delay: constDelay, distribution: distribution))

}

super.inAct(task: task)

}

}

--- Model/Element/Element.swift ---

import Foundation

class Element {

private static var nextID = 0

let name: String

var distribution = Method.exp {

didSet {

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

}

}

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 canAccept: Bool {

return true

}

var queueLength: Int {

return 0

}

var queueRelatedCount: Int {

return 0

}

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(task: Task) {

}

func outAct() {

quantity += 1

}

func printInfo() {

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

}

func doStatisctic(delta: Double) {

}

func setDelayDev(\_ value: Double) {

devices.forEach({ $0.delayDev = value })

}

func setErlangaValue(\_ value: Int) {

devices.forEach({ $0.erlangaValue = value })

}

}

enum Method {

case exp, norm, unif, erlanga

}

enum State: String {

case working, free

}

--- Model/Element/MultitypeCreator.swift ---

import Foundation

class MultitypeCreator: Create {

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

super.init(name: name, delay: delays[0].delay)

devices = [MultuitypeCreatorDevice(delays: delays, distribution: distribution)]

devices[0].tNext = 0

}

}

--- Model/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()

let newTask = devices[0].outAct()

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

transfer?.goNext(for: newTask)

}

}

--- Model/Transfer/PriorityTransferWithQueueCheck.swift ---

import Foundation

class PriorityTransferWithQueueCheck: Transfer {

let elements: [ElementWithPriority]

init(elements: [ElementWithPriority]) {

self.elements = elements

}

func goNext(for task: Task) {

let elementsWithEmptyDevice = self.elements.filter({ $0.element.state == .free })

if let maxElement = elementsWithEmptyDevice.max(by: { $0.priority.value < $1.priority.value }) {

elementsWithEmptyDevice.filter({ $0.priority.value == maxElement.priority.value }).randomElement()?.element.inAct(task: task)

return

}

let minQueueLength = self.elements.compactMap { element in

if element.element.canAccept {

return element.element.queueLength

} else {

return nil

}

}.min()

let elementsWithEmptyQueue = self.elements.filter({ $0.element.queueLength == minQueueLength && $0.element.canAccept })

if let maxElement = elementsWithEmptyQueue.max(by: { $0.priority.value < $1.priority.value }) {

elementsWithEmptyQueue.filter({ $0.priority.value == maxElement.priority.value }).randomElement()?.element.inAct(task: task)

return

}

elements.max(by: { $0.priority.value < $1.priority.value })?.element.inAct(task: task)

}

}

--- Model/Transfer/SoloTransfer.swift ---

import Foundation

class SoloTransfer: Transfer {

let nextElement: Element

init(nextElement: Element) {

self.nextElement = nextElement

}

func goNext(for task: Task) {

nextElement.inAct(task: task)

}

}

--- Model/Transfer/CustomTransfer.swift ---

import Foundation

class CustomTransfer: Transfer {

private let nextElement: (Task) -> Element?

init(nextElement: @escaping (Task) -> Element?) {

self.nextElement = nextElement

}

func goNext(for task: Task) {

let element = nextElement(task)

element?.inAct(task: task)

}

}

--- Model/Transfer/PriorotyTransfer.swift ---

import Foundation

class PriorotyTransfer: Transfer {

let elements: [ElementWithPriority]

init(elements: [ElementWithPriority]) {

self.elements = elements

}

func goNext(for task: Task) {

let elementsWithEmptyDevice = self.elements.filter({ $0.element.state == .free })

if let maxElement = elementsWithEmptyDevice.max(by: { $0.priority.value < $1.priority.value }) {

elementsWithEmptyDevice.filter({ $0.priority.value == maxElement.priority.value }).randomElement()?.element.inAct(task: task)

return

}

let elementsWithEmptyQueue = self.elements.filter({ $0.element.canAccept })

if let maxElement = elementsWithEmptyQueue.max(by: { $0.priority.value < $1.priority.value }) {

elementsWithEmptyQueue.filter({ $0.priority.value == maxElement.priority.value }).randomElement()?.element.inAct(task: task)

return

}

elements.max(by: { $0.priority.value < $1.priority.value })?.element.inAct(task: task)

}

}

--- Model/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(for task: Task) {

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

var currentProbability = 0.0

for probability in probabilities {

currentProbability += probability.probability

if currentProbability > number {

probability.nextElement.inAct(task: task)

break

}

}

}

}

--- Model/Transfer/Transfer.swift ---

import Foundation

protocol Transfer {

func goNext(for task: Task)

}

--- Model/Transfer/Model/Probability.swift ---

import Foundation

struct TransferProbability {

let probability: Double

let nextElement: Element

}

--- Model/Transfer/Model/Priority.swift ---

import Foundation

struct ElementWithPriority {

let element: Element

let priority: Prioroty

}

protocol Prioroty {

var value: Int { get }

}

extension Int: Prioroty {

var value: Int {

self

}

}

enum PriorityConstant: Prioroty {

case low, medium, hight

var value: Int {

switch self {

case .low:

return 0

case .medium:

return 50

case .hight:

return 100

}

}

}

--- Model/Calc/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(timeaverage: Double) -> Double {

var a = generateA

a = -timeaverage \* 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(timeaverage: Double, timeDeviation: Double) -> Double {

timeaverage + timeDeviation \* Double.random(in: -1...1)

}

class func erlanga(timeaverage: Double, k: Int) -> Double {

(0..<k).reduce(0.0, { prevValue, \_ in

prevValue + exp(timeaverage: timeaverage)

})

}

}

--- Model/Task/Task.swift ---

import Foundation

class Task {

let id = UUID()

var typeId: Int

init(typeId: Int = 1) {

self.typeId = typeId

}

}

--- Model/Model/Model.swift ---

import Foundation

class Model {

private var tNext = 0.0

private(set) var tCurr = 0.0

private var event = 0

private let elements: [Element]

private(set) var tasksInModel = 0.0

private let resultsPrinter: ResultsPrinter

init(elements: [Element], resultsPrinter: ResultsPrinter) {

self.resultsPrinter = resultsPrinter

self.elements = elements

}

var tasksCompleted: Int {

return elements.filter({ $0 is Create }).reduce(0, { $0 + $1.quantity })

}

var processCount: Int {

return elements.filter({ $0 is Process }).count

}

var failureProbability: Double {

let failures = elements.filter({ $0 is Process }).reduce(0, { $0 + ($1 as! Process).failure })

let createdElements = elements.filter({ $0 is Create }).reduce(0, { $0 + $1.quantity })

return Double(failures) / Double(createdElements)

}

var relatedCount: Int {

return elements.reduce(0, { $0 + $1.queueRelatedCount })

}

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) })

doStatistic()

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 doStatistic() {

calculateTasksInModel()

}

private func calculateTasksInModel() {

var tasks = elements.reduce(0, { $0 + $1.queueLength })

elements.forEach { element in

tasks += element.devices.filter({ $0.state == .working }).count

}

tasksInModel += Double(tasks) \* (tNext - tCurr)

}

private func printResult() {

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

resultsPrinter.printModelResults(self)

print("")

elements.forEach { element in

if let process = element as? Process {

resultsPrinter.printProcessResults(process)

} else if let create = element as? Create {

resultsPrinter.printCreateResults(create)

}

print("")

}

}

}

--- Model/Queue/RelatedQueue.swift ---

import Foundation

class RelatedQueue: Queue {

let relatedProcesses: [Process]

let minimumTransitionDifference: Int

init(relatedProcesses: [Process], minimumTransitionDifference: Int, maxLength: Int) {

self.relatedProcesses = relatedProcesses

self.minimumTransitionDifference = minimumTransitionDifference

super.init(maxLength: maxLength)

}

override func remove() throws -> Task {

let result = try super.remove()

outerloop: while status != .full {

for process in relatedProcesses {

if process.queue.status != .full && currentLength + minimumTransitionDifference <= process.queue.currentLength {

let newTask = try process.queue.remove()

try add(newTask)

relatedCount += 1

continue outerloop

}

}

break

}

return result

}

}

--- Model/Queue/PriorityQueue.swift ---

import Foundation

class PriorityQueue: Queue {

private let priorityCheck: (Task, Task) -> Bool

init(maxLength: Int, priorityCheck: @escaping (Task, Task) -> Bool) {

self.priorityCheck = priorityCheck

super.init(maxLength: maxLength)

}

override func add(\_ task: Task) throws {

guard currentLength < maxLength else {

throw QueueError.queueFill

}

if tasks.isEmpty {

tasks.append(task)

return

}

var isInsert = false

for taskIndex in 0..<tasks.count {

if priorityCheck(task, tasks[taskIndex]) {

isInsert = true

tasks.insert(task, at: taskIndex)

break

}

}

if !isInsert {

tasks.append(task)

}

}

}

--- Model/Queue/Queue.swift ---

import Foundation

class Queue {

var tasks = [Task]()

let maxLength: Int

var relatedCount = 0

init(maxLength: Int) {

self.maxLength = maxLength

}

var currentLength: Int {

tasks.count

}

var status: QueueStatus {

if currentLength == 0 {

return .empty

}

if currentLength < maxLength {

return .withTasks

}

return .full

}

func add(\_ task: Task) throws {

guard currentLength < maxLength else {

throw QueueError.queueFill

}

tasks.append(task)

}

@discardableResult

func remove() throws -> Task {

guard currentLength > 0 else {

throw QueueError.queryEmpty

}

return tasks.remove(at: 0)

}

}

enum QueueError: Error {

case queueFill, queryEmpty

}

enum QueueStatus {

case full, empty, withTasks

}

--- Model/Device/MultuitypeCreatorDevice.swift ---

import Foundation

class MultuitypeCreatorDevice: Device {

private let delays: [MultuitypeCreatorProbabilities]

private var nextType = 1

init(delays: [MultuitypeCreatorProbabilities], distribution: Method) {

self.delays = delays

super.init(delay: delays[0].delay, distribution: distribution)

}

override var delay: Double {

let newAverageDelay = generatedNextDelay

var delay = newAverageDelay

switch distribution {

case .exp:

delay = FunRand.exp(timeaverage: newAverageDelay)

case .norm:

delay = FunRand.norm(timeaverage: newAverageDelay, timeDeviation: delayDev)

case .unif:

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

case .erlanga:

delay = FunRand.erlanga(timeaverage: newAverageDelay, k: erlangaValue)

}

return delay

}

private var generatedNextDelay: Double {

var randomValue = Double.random(in: 0..<delays.reduce(0.0, { $0 + $1.probability }))

for index in 0..<delays.count {

let probability = delays[index]

if randomValue < probability.probability {

nextType = index + 1

return probability.delay

} else {

randomValue -= probability.probability

}

}

return 0

}

override func outAct() -> Task {

state = .free

return Task(typeId: nextType)

}

}

struct MultuitypeCreatorProbabilities {

let delay: Double

let probability: Double

}

--- Model/Device/Device.swift ---

import Foundation

class Device {

var tNext = 0.0

private let delayaverage: Double

var tCurr = 0.0

var state = State.free

var currentTask: Task?

var distribution: Method

var delayDev = 0.0

var erlangaValue = 0

var delay: Double {

var delay = delayaverage

switch distribution {

case .exp:

delay = FunRand.exp(timeaverage: delayaverage)

case .norm:

delay = FunRand.norm(timeaverage: delayaverage, timeDeviation: delayDev)

case .unif:

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

case .erlanga:

delay = FunRand.erlanga(timeaverage: delayaverage, k: erlangaValue)

}

return delay

}

init(delay: Double, distribution: Method) {

self.delayaverage = delay

self.distribution = distribution

}

func inAct(task: Task) {

state = .working

tNext = tCurr + delay

currentTask = task

}

func outAct() -> Task {

state = .free

if let result = currentTask {

currentTask = nil

return result

} else {

return Task()

}

}

}

--- Model/ResultsPrinter/ResultsOption.swift ---

import Foundation

enum ProcessResultsOption {

case quantity, averageQueueLength, failureProbability, averageLoadDevice, averageWorkingDevice, averageTasksInWork, averageTimeBetweenTaskCompletions

}

enum CreateResultsOption {

case quantity

}

enum ModelResultsOption {

case averageTasksInModel, averageTimeBetweenTaskCompletions, averageTimeTasksSpendsInModel, failureProbability, relatedCount

}

--- Model/ResultsPrinter/ResultsPrinter.swift ---

import Foundation

class ResultsPrinter {

private let processResultsConfig: [ProcessResultsOption]

private let createResultsConfig: [CreateResultsOption]

private let modelResultsConfig: [ModelResultsOption]

init(processResultsConfig: [ProcessResultsOption], createResultsConfig: [CreateResultsOption], modelResultsConfig: [ModelResultsOption]) {

self.processResultsConfig = processResultsConfig

self.modelResultsConfig = modelResultsConfig

self.createResultsConfig = createResultsConfig

}

func printModelResults(\_ model: Model) {

print("[Result of work model]")

modelResultsConfig.forEach({ printModelResult(model, option: $0) })

}

private func printModelResult(\_ model: Model, option: ModelResultsOption) {

switch option {

case .averageTasksInModel:

print("average tasks in model = \(model.tasksInModel / model.tCurr)")

case .averageTimeBetweenTaskCompletions:

print("average time between task completions = \(Double(model.processCount) \* model.tCurr / Double(model.tasksCompleted))")

case .averageTimeTasksSpendsInModel:

print("average time tasks spends in model = \(model.tasksInModel / Double(model.tasksCompleted))") // Формула Літтла

case .failureProbability:

print("failure probability = \(model.failureProbability)")

case .relatedCount:

print("related count = \(model.relatedCount)")

}

}

func printCreateResults(\_ create: Create) {

print("[Result of work \(create.name)]")

createResultsConfig.forEach({ printCreateResult(create, option: $0) })

}

private func printCreateResult(\_ create: Create, option: CreateResultsOption) {

switch option {

case .quantity:

print("quantity = \(create.quantity)")

}

}

func printProcessResults(\_ process: Process) {

print("[Result of work \(process.name)]")

processResultsConfig.forEach({ printProcessResult(process, option: $0) })

}

private func printProcessResult(\_ process: Process, option: ProcessResultsOption) {

switch option {

case .quantity:

print("quantity = \(process.quantity)")

case .averageQueueLength:

print("average length of queue = \(process.averageQueue / process.tCurr)")

case .failureProbability:

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

case .averageLoadDevice:

print("average load device = \(process.loadTime / process.tCurr)")

case .averageWorkingDevice:

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

case .averageTasksInWork:

print("average tasks in work = \(process.tasksInWorkCount / process.tCurr)")

case .averageTimeBetweenTaskCompletions:

print("average time between task completions = \(Double(process.devices.count) \* process.tCurr / Double(process.quantity))")

}

}

}

--- Model/Factory/ModelFactory.swift ---

import Foundation

class ModelFactory {

private let standartResultsPrinter = ResultsPrinter(

processResultsConfig: [.quantity, .averageQueueLength, .failureProbability, .averageLoadDevice, .averageWorkingDevice, .averageTasksInWork],

createResultsConfig: [.quantity],

modelResultsConfig: []

)

private let bankResultsPrinter = ResultsPrinter(

processResultsConfig: [.quantity, .averageLoadDevice, .averageQueueLength],

createResultsConfig: [.quantity],

modelResultsConfig: [.averageTasksInModel, .averageTimeBetweenTaskCompletions, .averageTimeTasksSpendsInModel, .failureProbability, .relatedCount]

)

private let hospitalResultsPrinter = ResultsPrinter(

processResultsConfig: [.quantity, .averageQueueLength, .averageTimeBetweenTaskCompletions],

createResultsConfig: [.quantity],

modelResultsConfig: [.averageTimeTasksSpendsInModel]

)

func makeModel(\_ type: ModelType) -> Model {

switch type {

case .priorityTest:

priorityTest()

case .bank:

bank()

case .hospital:

hospital()

}

}

private func priorityTest() -> Model {

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

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

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

let process3 = Process(name: "process3", delays: [2])

let process4 = Process(name: "process4", delays: [1])

create.transfer = SoloTransfer(nextElement: process1)

process1.transfer = SoloTransfer(nextElement: process2)

process2.transfer = PriorityTransferWithQueueCheck(elements: [

ElementWithPriority(element: process3, priority: PriorityConstant.hight),

ElementWithPriority(element: process4, priority: PriorityConstant.low)

])

process1.queue = Queue(maxLength: 1)

process2.queue = Queue(maxLength: 5)

process3.queue = Queue(maxLength: 5)

process4.queue = Queue(maxLength: 1)

create.distribution = .exp

process1.distribution = .exp

process2.distribution = .exp

process3.distribution = .exp

process4.distribution = .exp

return Model(elements: [create, process1, process2, process3, process4], resultsPrinter: standartResultsPrinter)

}

private func bank() -> Model {

let create = Create(name: "create", delay: 0.5)

let window1 = Process(name: "window1", delays: [0.3])

let window2 = Process(name: "window2", delays: [0.3])

create.transfer = PriorityTransferWithQueueCheck(elements: [

ElementWithPriority(element: window1, priority: PriorityConstant.hight),

ElementWithPriority(element: window2, priority: PriorityConstant.low)

])

window1.queue = RelatedQueue(relatedProcesses: [window2], minimumTransitionDifference: 2, maxLength: 3)

window2.queue = RelatedQueue(relatedProcesses: [window1], minimumTransitionDifference: 2, maxLength: 3)

create.distribution = .exp

window1.distribution = .exp

window2.distribution = .exp

let firstTimeNext = abs(FunRand.norm(timeaverage: 1, timeDeviation: 0.3))

window1.devices[0].tNext = firstTimeNext

window1.devices[0].state = .working

let secondTimeNext = abs(FunRand.norm(timeaverage: 1, timeDeviation: 0.3))

window2.devices[0].tNext = secondTimeNext

window2.devices[0].state = .working

create.devices[0].tNext = 0.1

try! window1.queue.add(Task())

try! window1.queue.add(Task())

try! window2.queue.add(Task())

try! window2.queue.add(Task())

return Model(elements: [create, window1, window2], resultsPrinter: bankResultsPrinter)

}

private func hospital() -> Model {

let create = MultitypeCreator(name: "create", delays: [

MultuitypeCreatorProbabilities(delay: 1.5, probability: 0.5),

MultuitypeCreatorProbabilities(delay: 4, probability: 0.1),

MultuitypeCreatorProbabilities(delay: 3, probability: 0.4)

])

let receptionDepartment = Process(name: "reception department", delays: [1.5, 1.5])

let wayToWard = Process(name: "way to ward", delays: [0.55, 0.55, 0.55])

let wayToLaboratory = UnlimitedProcess(name: "way to laboratory", delay: 0.35)

let registry = Process(name: "registry", delays: [0.45])

let laboratory = Process(name: "laboratory", delays: [0.4, 0.4])

let wayToReception = UnlimitedProcess(name: "way to reception", delay: 0.35)

create.transfer = SoloTransfer(nextElement: receptionDepartment)

receptionDepartment.transfer = CustomTransfer { task in

if task.typeId == 1 {

return wayToWard

} else {

return wayToLaboratory

}

}

wayToLaboratory.transfer = SoloTransfer(nextElement: registry)

registry.transfer = SoloTransfer(nextElement: laboratory)

laboratory.transfer = CustomTransfer { task in

if task.typeId == 2 {

task.typeId = 1

return wayToReception

}

return nil

}

wayToReception.transfer = SoloTransfer(nextElement: receptionDepartment)

receptionDepartment.queue = PriorityQueue(maxLength: .max, priorityCheck: { newTask, oldTask in

newTask.typeId == 1 && oldTask.typeId != 1

})

wayToWard.queue = Queue(maxLength: .max)

wayToLaboratory.queue = Queue(maxLength: .max)

registry.queue = Queue(maxLength: .max)

laboratory.queue = Queue(maxLength: .max)

wayToReception.queue = Queue(maxLength: .max)

create.distribution = .exp

wayToWard.distribution = .norm

wayToWard.setDelayDev(0.25)

wayToLaboratory.distribution = .norm

wayToLaboratory.setDelayDev(0.15)

registry.distribution = .erlanga

registry.setErlangaValue(3)

laboratory.distribution = .erlanga

laboratory.setErlangaValue(2)

wayToReception.distribution = .norm

wayToReception.setDelayDev(0.15)

return Model(elements: [create, receptionDepartment, wayToWard, wayToLaboratory, registry, laboratory, wayToReception], resultsPrinter: hospitalResultsPrinter)

}

}

enum ModelType {

case priorityTest, bank, hospital

}