Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Кафедра ПОИТ

Курс «Системный анализ и машинное моделирование»

Лабораторная работа №5-6

«Построение и исследование аналитической модели непрерывно-стохастической системы массового обслуживания»

Вариант 35

Выполнил:

Воривода М.А.

Проверил:

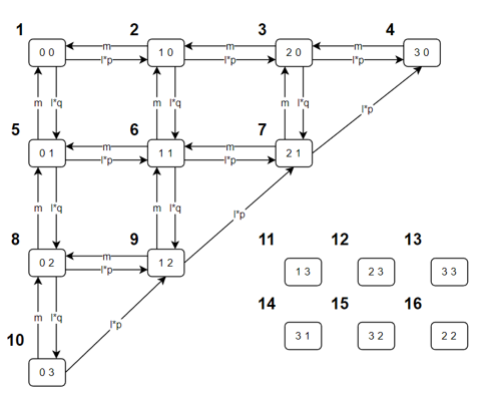
Мельник Н.И.

Минск 2022

**Задание**

В СМО вида М/М/2/1 поступают заявки двух видов. Заявка первого вида появляется на входе с вероятностью р, второго с вероятностью (1 - р). Заявка первого вида имеет более высокий приоритет и может вытеснить заявку второго вида из канала в очередь, если место в очереди свободно или из системы, если место занято. В случае, когда заявка первого вида застает систему в состоянии обслуживания заявок первого вида, то она ставится в очередь, если место ожидания свободно или занято заявкой второго вида (менее приоритетная заявка теряется). Найти относительные пропускные способности Q1 и Q2. µ=0.5, λ=0.9, р=0.5.  
  
M – Пуассоновский источник  
M – Экспоненциальный канал  
2 - Количество каналов  
1 - Длина очереди

**Диаграмма интенсивности переходов**



**Система уравнений**

**Расчёты**

**Иммитация**

Q1 = 1.0

Q2 = 0.581774971244887

fun time(factor: Double): Double {  
 return (-1 / factor) \* *ln*(Random.nextDouble())  
}

class PoissonSource(  
 private val lambda: Double,  
 private val requestFactory: Supplier<Request>  
) : Source {  
  
 private var remainTime = 0.0  
  
 init {  
 update()  
 }  
  
 override val state: String get() = ""  
  
 override fun tick(deltaT: Double): Request? {  
 remainTime -= deltaT  
 if (remainTime > 0) return null  
 update()  
 return requestFactory.get()  
 }  
  
 private fun update() {  
 remainTime += *time*(lambda)  
 }  
}

class ExponentialChannel(index: Int, private val mu: Double) : Channel() {  
  
 private var remainTime = 0.0  
  
 override fun internalHandle(request: Request?) {  
 if (request != null) update()  
 }  
  
 override fun internalTick(deltaT: Double): Request? {  
 if (request != null) {  
 remainTime -= deltaT  
 if (remainTime > 0) return null  
 update()  
 }  
 return request?.*also* **{** request = null **}**  
}  
  
 private fun update() {  
 remainTime += *time*(mu)  
 }  
}

class Channels(  
 private val parallel: Boolean,  
 private val queue: Queue,  
 val list: List<Channel>  
) : Component<List<Int>> {  
  
 override val state get() = list.*map* **{ it**.state **}**  
  
fun tick(deltaT: Double) {  
 if (parallel) {  
 list.*forEach* **{ it**.tick(deltaT)?.handled = true **}**  
} else {  
 var next = list.*last*()  
  
 next.tick(deltaT)?.*let* **{**  
 **it**.handled = true  
 **}**  
for (i in (list.size - 2) *downTo* 0) {  
 val current = list[i]  
  
 current.tick(deltaT)?.*also* **{**  
if (next.isFree()) {  
 next.handle(**it**)  
 return@also  
 }  
 if (next.request!!.priority >= **it**.priority) {  
 **it**.refusedOnChannel = current  
 return@also  
 }  
  
 val repressed = next.handle(**it**)!!  
 if (!queue.enqueue(repressed)) repressed.repressed = true  
 **}**  
  
next = current  
 }  
 }  
 }  
  
 fun handle(request: Request?): Boolean {  
 if (request == null) return false  
  
 (if (parallel) {  
 list.*find* **{ it**.isFree() **}** ?: list.*find* **{ it**.request!!.priority < request.priority **}**  
} else {  
 if (list[0].isFree() || list[0].request!!.priority < request.priority) list[0] else null  
 })  
 .*also* **{** if (**it** == null) return false **}**!!  
 .handle(request)  
 .*also* **{**  
if (**it** != null && !queue.enqueue(**it**)) **it**.repressed = true  
 return true  
 **}**  
}  
}

class Queue() : Component<Int> {  
  
 private val list = LinkedList<Request>()  
  
 val size get() = list.size  
  
 fun enqueue(request: Request?): Boolean {  
 if (request == null) throw Exception()  
 if (isFull()) return false  
 list.add(request)  
 return true  
 }  
  
 fun dequeue(): Request? {  
 return list.pollFirst()  
 }  
  
 fun peek(): Request? {  
 return list.peekFirst()  
 }  
  
 fun tick(ignored: Double) {  
 }  
  
 private fun isFull(): Boolean {  
 return list.size >= 1  
 }  
  
 override val state get() = if (list.isEmpty()) 0 else list[0].type  
}

class Request(val priority: Int, val type: Int) {  
  
 var handled = false  
 var refusedOnSource = false  
 var refusedOnChannel: Channel? = null  
 var repressed = false  
  
 fun isRefused(): Boolean {  
 return refusedOnSource || refusedOnChannel != null || repressed  
 }  
}

class System(  
 private val source: Source,  
 private val queue: Queue,  
 private val channels: Channels  
) {  
  
 var totalTime = 0.0  
 private set  
  
 private val state get() = State(queue.state, channels.state)  
  
 private fun tick(deltaT: Double): Pair<State, Request?> {  
 totalTime += deltaT  
 channels.tick(deltaT)  
 queue.tick(deltaT)  
 if (channels.handle(queue.peek())) queue.dequeue()  
  
 val emitted = source.tick(deltaT)  
 if (emitted != null) {  
 if (!channels.handle(emitted) && !queue.enqueue(emitted)) {  
 emitted.refusedOnSource = true  
 }  
 }  
  
 return state *to* emitted  
 }  
  
 fun simulate(tickCount: Int, deltaT: Double): Statistics {  
 val requests = *mutableListOf*<Request>()  
 val states = *mutableMapOf*<State, Int>()  
 for (i in 0 *until* tickCount) {  
 val tickResult = tick(deltaT)  
  
 states.compute(tickResult.first) **{** \_, count **->**  
if (count == null) return@compute 1  
 return@compute count + 1  
 **}**  
  
if (tickResult.second != null) requests.add(tickResult.second!!)  
 }  
  
 return Statistics.from(requests.*toList*())  
 }  
  
 class State(  
 val queueSize: Int,  
 val channelBusiness: List<Int>  
 ) {  
  
 override fun hashCode(): Int {  
 return Objects.hash(queueSize, channelBusiness)  
 }  
  
 override fun equals(other: Any?): Boolean {  
 if (this === other) return true  
 if (*javaClass* != other?.*javaClass*) return false  
  
 other as State  
  
 if (queueSize != other.queueSize) return false  
 if (channelBusiness != other.channelBusiness) return false  
  
 return true  
 }  
  
 override fun toString(): String {  
 return "" + queueSize + channelBusiness.*joinToString*("") **{ it**.toString() **}**  
}  
 }  
}

data class Statistics(  
 val q1: Double, *// Q1*  
val qa1: Double, *// QA1*  
val q2: Double, *// Q2*  
val qa2: Double, *// QA2*  
*) {*  
  
companion object {  
  
 fun from(requests: List<Request>): Statistics {  
 return Statistics(  
 q(requests, 1, true),  
 q(requests, 1, false),  
 q(requests, 2, true),  
 q(requests, 2, false)  
 )  
 }  
  
 private fun q(requests: List<Request>, type: Int, checkOnSource: Boolean): Double {  
 val entered =  
 (if (checkOnSource) requests.*filter* **{** !**it**.refusedOnSource **}** else requests).*filter* **{ it**.type == type **}**  
val refused = entered.*filter* **{ it**.isRefused() **}**  
val handled = entered.*filter* **{ it**.handled **}**  
  
return handled.size.toDouble() / (refused.size + handled.size)  
 }  
 }  
}