БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Кафедра информатики

Факультет КСИС

Специальность ИиТП

Индивидуальная практическая работа №2

по дисциплине «Математическое моделирование»

Выполнил студент: Драгун О.В.

группа 893551

Зачетная книжка № 2520050

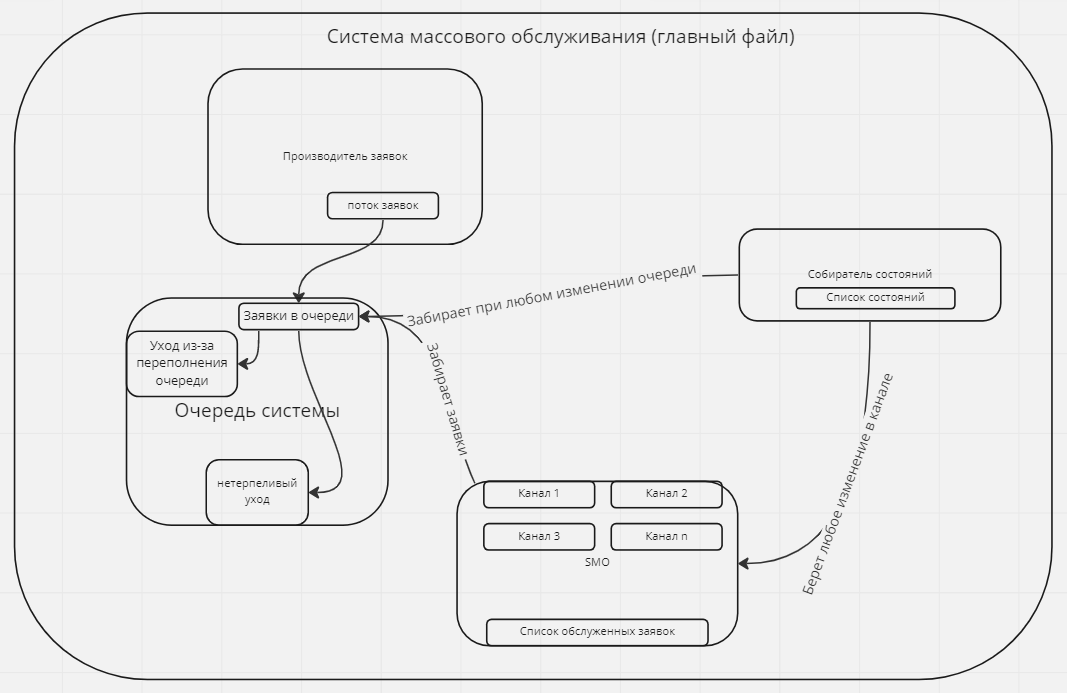
Минск 2022

# Постановка задачи

Имеется простейшая n-канальная СМО с m местами в очереди; интенсивность потока заявок λ, потока обслуживания μ. Время пребывания заявки в очереди ограничено некоторым случайным сроком T, распределенным по показательному закону с параметром ν. Если по истечению времени T заявка не была передана на обслуживание, она покидает СМО необслуженной.   
**Задание**   
Построить имитационную модель СМО. Сравнить характеристики СМО, полученные в результате моделирования, с теоретическими характеристиками (финальные вероятности состояний, абсолютная пропускная способность, вероятность отказа, средние число заявок в СМО, среднее число заявок в очереди, среднее время пребывания заявки в СМО, среднее время пребывания заявки в очереди, среднее число занятых каналов). Продемонстрировать работоспособность модели с помощью графиков и построить графики, показывающие установку стационарного режима СМО. Исследовать СМО при различных параметрах n, m, λ, μ,

# Архитектура системы

График элементов системы и краткое описание:

  
Главный файл – Знает обо всех элементах системы, синхронизирует многопоточноть и общение между элементами

Производитель заявок – генерирует непрерывный поток заявок

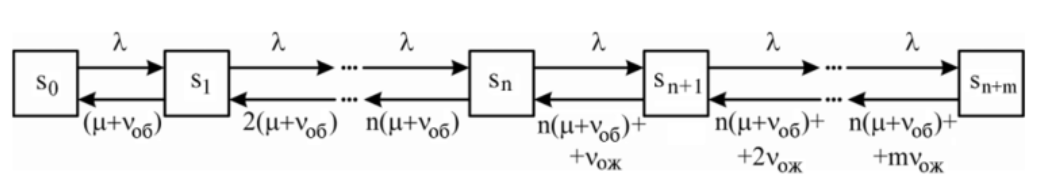
Очередь – синхронизирует многопоточность через mutex, собирает заявки из производителя, управляет уставшими заявками в очереди, собирает список ушедших для статистики

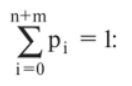
Канал – обслуживает одну заявку

SMO (набор каналов) – синхронизирует работу каналов, при освобождении любого канала забирает заявки из очереди, собирает список обслуженных заявок

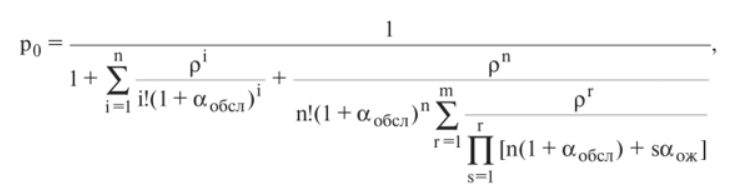
# Подсчет характеристик

## Финальные вероятности

Переходы из состояния в состояние в рассматриваемой СМО происходят под действием входящего потока заявок и потоков уходов «нетерпеливых» заявок из очереди, с игнорированием возможности ухода из каналов обслуживания. На рисунке vоб можно игнорировать  
  
Вероятность р0 определяется с учетом нормировочного условия

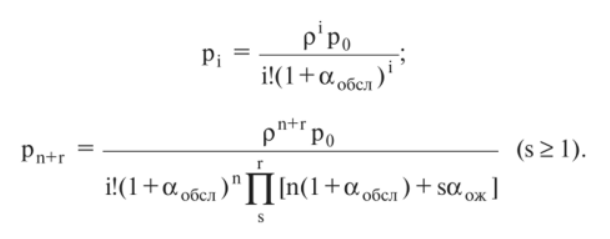


Alpha обслуживания в формуле стоит принять за 0



Где 

Для остальных вероятностей воспользуемся нижеприведенными формулами, с учетом Alpha обслуживания = 0



Реализация подсчета теоретических значений на kotlin

//1.55  
val p0 = 1 /  
 (1 +  
 (nList.*fold*(0.0) **{** sum, i **->** sum + (myY.*pow*(i) / *factorial*(i))  
 **}**)  
 +  
 (myY.*pow*(n) / (*factorial*(n) \* MList.*fold*(0.0) **{** sum, r **->** val rList = *List*(r) **{ it** + 1 **}** //myY.pow(r)  
 sum + (myY.*pow*(r) /  
 rList.*fold*(1.0) **{** mult, s **->** mult \* (n + s \* alphaOj)  
 **}**)  
 **}**)))  
  
val pList = *mutableListOf*(p0)  
//1.56  
for (i in 1..n) {  
 val pIndexed = myY.*pow*(i) \* p0 / *factorial*(i)  
 pList.add(pIndexed)  
}  
  
//1.57  
for (i in 1..M) {  
 val rList = *List*(i) **{ it** + 1 **}** val pIndexed = (myY.*pow*(n + i) \* p0 /  
 (*factorial*(i) \* rList.*fold*(1.0) **{** mult, s **->** mult \* (n + s \* alphaOj)  
 **}**))  
  
 pList.add(pIndexed)  
}

while (pList.*sum*() !in 0.9999..1.0001) {  
 *println*("производим нормировку")  
 val x = 1 / pList.*sum*()  
 val mapped = pList.*map* **{ it** \* x **}** pList.clear()  
 pList.addAll(mapped)  
 *println*("p sum = ${pList.*sum*()}")  
}

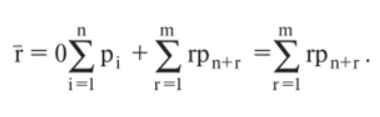
Подсчет практических значений будем брать из фактически проведенного времени системы во всех состояниях

val \_pList = *mutableListOf*<Double>()  
for (i in 0..n) {  
 validStates  
 .*filter* **{ it**.queueSize == 0 && **it**.busyChannels == i **}** .*fold*(0L) **{** sum, element **->** sum + element.stateTime **}** .*apply* **{** \_pList.add(this.toDouble() / validStatesTime) **}**}  
  
for (i in 1..M) {  
 validStates  
 .*filter* **{ it**.queueSize == i && **it**.busyChannels == n **}** .*fold*(0L) **{** sum, element **->** sum + element.stateTime **}** .*apply* **{** \_pList.add(this.toDouble() / validStatesTime) **}**}

## Вероятность отказа

Теоретическая равна последней вероятности p, или p(n+m), соответствующая всем занятым каналам и всем занятым местам в очереди.  
Практическая также равна \_p(n+m), или времени, фактически проведенному системой в данном состоянии, деленному на общее время работы системы.

## Среднее число заявок, находящихся в очереди



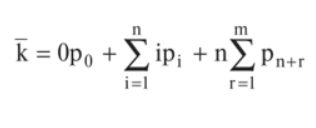
Теоретическое на kotlin

val l = MList.*fold*(0.0) **{** sum, r **->** sum + r \* pList[n + r]  
**}***println*("l среднее число заявок, находящихся в очереди = $l")

Практическое на kotlin

var \_l = 0.0  
for (i in 0..M) {  
 \_l += i \*  
 (validStates  
 .*filter* **{ it**.queueSize == i **}** .*sumOf* **{ it**.stateTime.toDouble() **}** / validStatesTime)  
}  
*println*("\_l среднее число заявок, находящихся в очереди = $\_l")

## Среднее число занятых каналов



Теоретическое на kotlin

val kzan = nList.*fold*(0.0) **{** sum, i **->** sum + i \* pList[i]  
**}** + n \* MList.*fold*(0.0) **{** sum, r **->** sum + pList[n + r]  
**}**

*println*("kzan среднее число занятых каналов = $kzan")

Практическое:

var \_kzan = 0.0  
for (i in 0..n) {  
 \_kzan += i \*  
 (validStates  
 .*filter* **{ it**.busyChannels == i **}** .*sumOf* **{ it**.stateTime.toDouble() **}** / validStatesTime)  
}  
*println*("\_kzan среднее число занятых каналов = $\_kzan")

## Среднее число заявок в СМО



И для теоретического, и для практического значения это будет сумма двух предыдущих показателей

## Среднее время ожидания в очереди



Теоретическое на kotlin c поправкой на ню

val w = l / (lambda - nu)  
*println*("w среднее время ожидания в очереди = $w")

Практическое на kotlin

val \_w = allFinishedList.*map* **{ it**.queueWaitingTime **}**.*average*() / *MILLIS\_IN\_SECOND  
println*("\_w среднее время ожидания в очереди = $\_w")

## Среднеe время пребывания заявки в СМО

Изображение выглядит как текст

Автоматически созданное описание

Теоретическое на kotlin с поправкой на ню

val m = l + kzan  
*println*("m среднее число заявок в СМО = $m")  
val u = m / (lambda - nu)  
*println*("u среднеe время пребывания заявки в СМО = $u")

Практическое на kotlin

val \_w = allFinishedList.*map* **{ it**.queueWaitingTime **}**.*average*() / *MILLIS\_IN\_SECOND  
println*("\_w среднее время ожидания в очереди = $\_w")  
  
val \_p = allFinishedList  
 .*map* **{ it**.serviceWaitingTime **}** .*average*() / *MILLIS\_IN\_SECOND  
println*("\_p среднее время обслуживания в канале = $\_p")  
  
val \_u = \_w + \_p  
*println*("среднеe время пребывания заявки в СМО = $\_u")

## Абсолютная пропускная способность

Выведем из среднего числа занятых каналов

Изображение выглядит как текст

Автоматически созданное описание

Теоретическое на kotlin

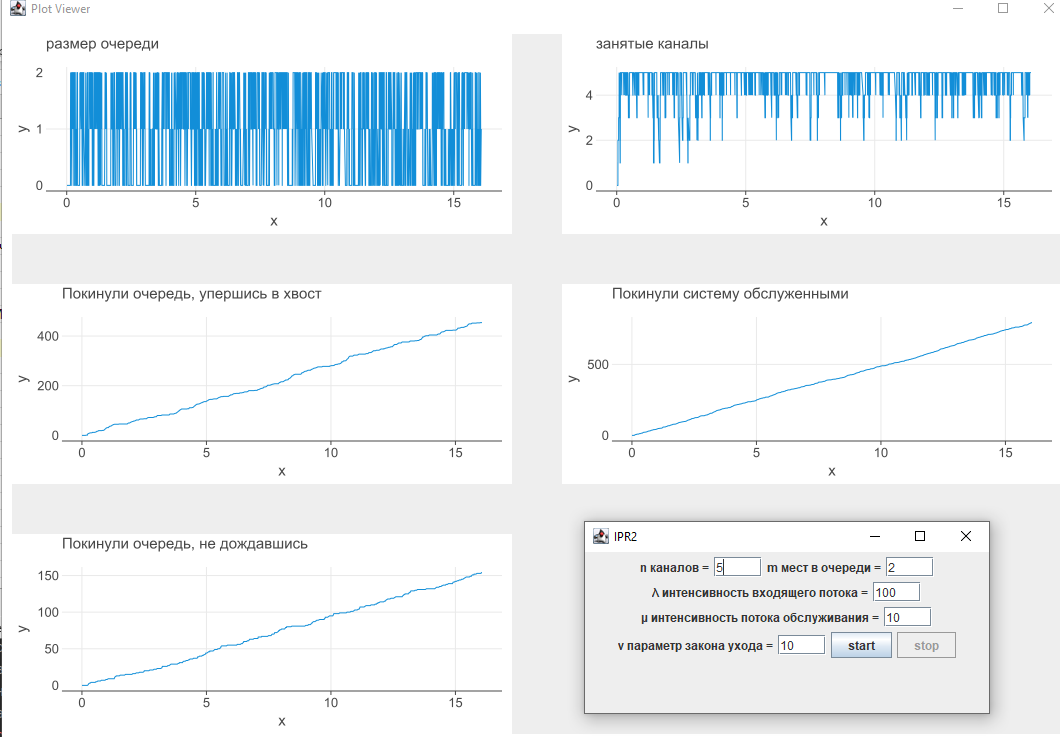
val kzan = nList.*fold*(0.0) **{** sum, i **->** sum + i \* pList[i]  
**}** + n \* MList.*fold*(0.0) **{** sum, r **->** sum + pList[n + r]  
**}***println*("kzan среднее число занятых каналов = $kzan")  
  
val lambda\_ = kzan \* mu  
*println*("λ` абсолютная пропускная способность = $lambda\_")

Практическое на kotlin

val \_lambda\_ = allFinishedList.size.toDouble() / validStatesTime \* *MILLIS\_IN\_SECOND  
println*("\_λ` абсолютная пропускная способность = $\_lambda\_")

# Прогон системы

Длительность 15с



Изображение выглядит как стол

Автоматически созданное описание

# Листинг кода

## Главный файл

package ipr\_two  
  
import kotlinx.coroutines.\*  
import java.awt.FlowLayout  
import java.awt.event.ActionListener  
import java.lang.Exception  
import javax.swing.\*  
import kotlin.coroutines.CoroutineContext  
  
class IprTwoInputGetter : JFrame("IPR2"), CoroutineScope {  
  
 val job = *Job*()  
  
 override val coroutineContext: CoroutineContext  
 get() = job + Dispatchers.Main  
  
 companion object {  
 @JvmStatic  
 fun main(args: Array<String>) {  
 //remove annoying warning "Graphics2D from BufferedImage lacks BUFFERED\_IMAGE hint", was actual for 1/2 PC  
 System.setProperty("org.apache.batik.warn\_destination", "false")  
  
 SwingUtilities.invokeLater **{** IprTwoInputGetter() **}** }  
 }  
  
 var startButton = JButton("start")  
 var stopButton = JButton("stop")  
  
 val labelChannelsN = JLabel("n каналов =")  
 val textFieldChannelsN = JTextField(IprTwoParamsSaver.loadIprTwoParams().channelsN.toString(), 4)  
  
 val labelQueueM = JLabel("m мест в очереди =")  
 val textFieldQueueN = JTextField(IprTwoParamsSaver.loadIprTwoParams().queueM.toString(), 4)  
  
 val labelInputFlowLambda = JLabel("λ интенсивность входящего потока =")  
 val textFieldInputFlowLambda = JTextField(IprTwoParamsSaver.loadIprTwoParams().inputFlowLambda.toString(), 4)  
  
 val labelServiceFlowMu = JLabel("μ интенсивность потока обслуживания =")  
 val textFieldServiceFlowMu = JTextField(IprTwoParamsSaver.loadIprTwoParams().serviceFlowMu.toString(), 4)  
  
 val labelLeavingNu = JLabel("ν параметр закона ухода =")  
 val textFieldLeavingNu = JTextField(IprTwoParamsSaver.loadIprTwoParams().leavingNu.toString(), 4)  
  
 var nChannels: Int? = null  
 var mQueue: Int? = null  
 var lambdaInputFlow: Double? = null  
 var muServiceFlow: Double? = null  
 var nuLeaving: Double? = null  
  
 lateinit var stateCollector: StateCollector  
 lateinit var smo: SMO  
 lateinit var queueSMO: QueueSMO  
 lateinit var requestProducer: RequestProducer  
 var epochStartTime: Long = 0L  
  
 fun getLeftQueueSize() = queueSMO.leftSize()  
 fun getAllImpatient() = queueSMO.getAllImpatient()  
 fun getRequestProducedSize() = requestProducer.getRequestProducedSize()  
 fun getAllFinishedList() = smo.getAllFinishedList()  
  
 init {  
 *layout* = FlowLayout()  
 *defaultCloseOperation* = *EXIT\_ON\_CLOSE* setLocationRelativeTo(null)  
 *isVisible* = true  
 setSize(420, 200)  
  
 startButton.addActionListener **{** startButtonClicked()  
 **}** stopButton.*isEnabled* = false  
  
 add(labelChannelsN)  
 add(textFieldChannelsN)  
 add(labelQueueM)  
 add(textFieldQueueN)  
 add(labelInputFlowLambda)  
 add(textFieldInputFlowLambda)  
 add(labelServiceFlowMu)  
 add(textFieldServiceFlowMu)  
 add(labelLeavingNu)  
 add(textFieldLeavingNu)  
 add(startButton)  
 add(stopButton)  
 }  
  
 private fun startButtonClicked() {  
 try {  
 nChannels = textFieldChannelsN.*text*.*toInt*().*also* **{** *println*("n каналов = $**it**") **}** mQueue = textFieldQueueN.*text*.*toInt*().*also* **{** *println*("m мест в очереди = $**it**") **}** lambdaInputFlow = textFieldInputFlowLambda.*text*.*toDouble*().*also* **{** *println*("λ вход = $**it**") **}** muServiceFlow = textFieldServiceFlowMu.*text*.*toDouble*().*also* **{** *println*("μ обслуживания = $**it**") **}** nuLeaving = textFieldLeavingNu.*text*.*toDouble*().*also* **{** *println*("ν параметр закона ухода = $**it**") **}** if (nuLeaving!! >= lambdaInputFlow!!) {  
 throw Exception("Невозможно корректно подсчитать теоретические значения! Задайте ν меньше λ")  
 }  
  
  
 *launch* **{** epochStartTime = System.currentTimeMillis()  
 stateCollector = StateCollector(epochStartTime)  
 queueSMO = QueueSMO(  
 capacityM = mQueue!!,  
 epochTime = epochStartTime,  
 reportSizeChanged = **{** queueSize, leftSize, impatientLeftSize**->** *println*("queue size changed to $queueSize")  
 *launch*(Dispatchers.Main) **{** stateCollector.stateChanged(  
 newQueueSize = queueSize,  
 newBusyChannelsSize = null,  
 newQueueLeftSize = leftSize,  
 newFinishedRequests = null,  
 newImpatientLeftSize = impatientLeftSize  
 )  
 **}  
 }**)  
 smo = SMO(  
 queueSMO,  
 this,  
 muServiceFlow!!,  
 nChannels!!,  
 epochStartTime,  
 reportSizeChanged = **{** busyChannels: Int, allFinishedListSize: Int **->** *println*("SMO busy size changed to $busyChannels")  
 *launch*(Dispatchers.Main) **{** stateCollector.stateChanged(  
 newQueueSize = null,  
 newBusyChannelsSize = busyChannels,  
 newQueueLeftSize = null,  
 newFinishedRequests = allFinishedListSize,  
 newImpatientLeftSize = null  
 )  
 **}  
 }** )  
 *launch* **{** requestProducer = RequestProducer(lambdaInputFlow!!,nuLeaving!!, epochStartTime)  
 requestProducer.requestsFlow().collect **{** request **->** *println*("collecting ${request.id}, deltaFromEpoch = ${request.deltaFromEpoch}, deltaFromLast = ${request.deltaFromLastRequest}")  
 queueSMO.addRequest(request)  
  
 **}  
  
 }  
 }**.*setUpCancellation*()  
  
  
 } catch (e: Exception) {  
 JOptionPane.showMessageDialog(  
 this@IprTwoInputGetter,  
 """  
 Ошибка во время работы системы:  
 ${e.message}  
 """.*trimIndent*()  
 )  
 }  
 }  
  
 private fun stopButtonClicked() {  
 try {  
 StatisticsCalculator.calculate(this)  
 IprTwoParamsSaver.saveIprTwoParams(  
 IprTwoParams(  
 channelsN = nChannels!!,  
 queueM = mQueue!!,  
 inputFlowLambda = lambdaInputFlow!!,  
 serviceFlowMu = muServiceFlow!!,  
 leavingNu = nuLeaving!!  
 )  
 )  
  
 } catch (e: Exception) {  
 JOptionPane.showMessageDialog(  
 this@IprTwoInputGetter,  
 """  
 Ошибка во время обработки результатов:  
 ${e.message}  
 """.*trimIndent*()  
 )  
 }  
 }  
  
 private fun Job.setUpCancellation() {  
 val processingJob = this  
 val listener = *ActionListener* **{** processingJob.cancel()  
 stopButtonClicked()  
 **}** stopButton.addActionListener(listener)  
 stopButton.*isEnabled* = true  
 // update the status and remove the listener after the loading job is completed  
 *launch* **{** processingJob.join()  
 stopButton.removeActionListener(listener)  
 stopButton.*isEnabled* = false  
 **}** }  
}

## Модели заявки и состояния

package ipr\_two.model  
  
data class Request(  
 val id: Long,  
 //дельта времени появления между заявками  
 val deltaFromLastRequest: Long,  
 //прошло времени со старта системы до появления заявки  
 val deltaFromEpoch: Long,  
 //время появления заявки  
 val issueTime: Long,  
 //сколько времени готова простоять в очереди  
 val impatientQueueLeavingTime: Long,  
 //сколько находилась в очереди до попадания на обслуживание  
 var queueWaitingTime: Long = 0L,  
 //сколько находилась на обслуживании в системе  
 var serviceWaitingTime: Long = 0L,  
 //появилась при заполненной очереди и не пошла дальше  
 var isFaceFullQueue: Boolean = false  
)

package ipr\_two.model  
  
data class State(  
 //сколько сейчас в очереди  
 val queueSize: Int,  
 //сколько сейчас занято каналов  
 val busyChannels: Int,  
 //сколько времени провисело это состояние, прежде чем сменилось на новое  
 var stateTime: Long,  
 // systemtimemillis на момент перехода системы в это состояние  
 val stateTimeStamp: Long,  
 //сколько уже ушло из очереди упершись в максимум  
 val queueLeftSize: Int,  
 //сколько уже обслужено  
 val finishedRequests: Int,  
 //сколько нетерпеливо ушло из очереди  
 val impatientLeftSize: Int  
)

## Генерация заявок

package ipr\_two  
  
import ipr\_two.model.Request  
import kotlinx.coroutines.delay  
import kotlinx.coroutines.flow.Flow  
import kotlinx.coroutines.flow.flow  
import org.apache.commons.math3.distribution.ExponentialDistribution  
import org.jetbrains.letsPlot.geom.geomHistogram  
import org.jetbrains.letsPlot.letsPlot  
  
const val *MILLIS\_IN\_SECOND* = 1000  
  
class RequestProducer(  
 val lambdaInputFlow: Double,  
 val nuLeaving: Double,  
 val epochTime: Long  
) {  
 private val lambdaDistributionGenerator = ExponentialDistribution(1.0 / lambdaInputFlow)  
 private val nuDistributionGenerator = ExponentialDistribution(1.0 / nuLeaving)  
 private var i: Long = 1  
  
 fun getRequestProducedSize() = i  
  
 fun requestsFlow(): Flow<Request> = *flow* **{** while (true) {  
 //через это кол-во миллисикунд мы добавляем заявку в поток  
 var t = (lambdaDistributionGenerator.sample() \* *MILLIS\_IN\_SECOND*).toLong()  
 //сколько времени готова простоять в очереди  
 var impatientQueueLeavingTime = (nuDistributionGenerator.sample() \* *MILLIS\_IN\_SECOND*).toLong()  
 //при очень высокой интенсивности не позволяем времени быть нулями, чтобы не было одновременно выпущенных заявок  
 if (t == 0L) {  
 t = 1  
 }  
 //даем процессору немного времени пройти сквозь пустую очередь  
 if (impatientQueueLeavingTime <=4L) {  
 impatientQueueLeavingTime = 5L  
 }  
 delay(t)  
 val systemTime: Long = System.currentTimeMillis()  
 *println*("emiting id $i, impatientTime = $impatientQueueLeavingTime")  
 emit(  
 Request(  
 id = i,  
 deltaFromLastRequest = t,  
 deltaFromEpoch = systemTime - epochTime,  
 issueTime = systemTime,  
 impatientQueueLeavingTime = impatientQueueLeavingTime  
 )  
 )  
 i++  
 }  
 **}** fun testDistribution() {  
 val data = *mapOf*<String, List<\*>>(  
 "x" *to List*(100000) **{** lambdaDistributionGenerator.sample() **}** )  
 val p = *letsPlot*(data) **{** x = "x" **}** (p + geomHistogram(boundary = 0.0)).show()  
 }  
}  
  
fun main() {  
 RequestProducer(  
 lambdaInputFlow = 1.0,  
 nuLeaving = 0.0,  
 epochTime = System.currentTimeMillis()  
 ).testDistribution()  
}

## Очередь

package ipr\_two  
  
import ipr\_two.model.Request  
import kotlinx.coroutines.sync.Mutex  
import kotlinx.coroutines.sync.withLock  
import kotlinx.html.currentTimeMillis  
  
class QueueSMO(  
 val capacityM: Int,  
 val epochTime: Long,  
 val reportSizeChanged: (queueSize: Int, leftSize: Int, impatientLeftSize: Int) -> Unit  
) {  
 private val mutex = *Mutex*()  
 private val queueList = *mutableListOf*<Request>()  
 private val leftList = *mutableListOf*<Request>()  
 //нетерпливые заявки попадают сюда  
 private val impatientLeftList = *mutableListOf*<Request>()  
  
 fun leftSize() = leftList.size  
 fun leftImpatientSize() = impatientLeftList.size  
 fun getAllImpatient() = impatientLeftList.*toList*()  
  
 suspend fun getRequest(): Request? {  
 var result: Request? = null  
 modifyQueue **{** //удаляем уставшие заявки  
 val currentTime = *currentTimeMillis*()  
 val (impatient, valid) = this.*partition* **{** currentTime > (**it**.issueTime + **it**.impatientQueueLeavingTime) **}** if (impatient.*isNotEmpty*()) {  
 impatientLeftList += impatient  
 this.clear()  
 this.addAll(valid)  
 reportSizeChanged(this.size, leftSize(), leftImpatientSize())  
 }  
  
 result = this.*firstOrNull*()?.*also* **{  
 it**.queueWaitingTime = *currentTimeMillis*() - **it**.issueTime  
 this.*removeFirst*()  
 reportSizeChanged(this.size, leftSize(), leftImpatientSize())  
 **}  
 }** return result  
 }  
  
 suspend fun addRequest(request: Request) {  
 modifyQueue **{** //удаляем уставшие заявки  
 val currentTime = *currentTimeMillis*()  
 val (impatient, valid) = this.*partition* **{** currentTime > (**it**.issueTime + **it**.impatientQueueLeavingTime) **}** if (impatient.*isNotEmpty*()) {  
 impatientLeftList += impatient  
 this.clear()  
 this.addAll(valid)  
 reportSizeChanged(this.size, leftSize(), leftImpatientSize())  
 }  
  
 if (this.size < capacityM) {  
 this.add(request)  
 reportSizeChanged(this.size, leftSize(), leftImpatientSize())  
 } else {  
 request.isFaceFullQueue = true  
 leftList.add(request)  
 reportSizeChanged(this.size, leftSize(), leftImpatientSize())  
 *println*("leftQueue = ${leftList.size}")  
 }  
 **}** }  
  
 private suspend fun modifyQueue(block: MutableList<Request>.()-> Unit) {  
 mutex.withLock **{** queueList.block()  
 **}** }  
}

## Канал

package ipr\_two  
  
import ipr\_two.model.Request  
import kotlinx.coroutines.CoroutineScope  
import kotlinx.coroutines.channels.Channel  
import kotlinx.coroutines.delay  
import kotlinx.coroutines.launch  
import org.apache.commons.math3.distribution.ExponentialDistribution  
  
class ChannelSMO(  
 private val id: Int,  
 muServiceFlow: Double,  
 private val scope: CoroutineScope,  
 val epochTime: Long,  
 //сюда кладем заявку при выходе обслуженной  
 val finishedRequests : Channel<Request>  
) {  
 private val distributionGenerator = ExponentialDistribution(1.0/muServiceFlow)  
 var isAvailable: Boolean = true  
 private var millisLoaded = 0L  
  
 fun putRequest(request: Request) {  
 isAvailable = false  
 scope.*launch* **{** var t = (distributionGenerator.sample() \* *MILLIS\_IN\_SECOND*).toLong()  
 //при очень высокой интенсивности не позволяем времени быть нулями, чтобы не было одновременно выпущенных заявок  
 if (t == 0L) {  
 t = 1  
 }  
 delay(t)  
 millisLoaded += t  
 request.serviceWaitingTime = t  
 isAvailable = true  
 finishedRequests.send(request)  
 **}** }  
}

## Набор каналов

package ipr\_two  
  
import ipr\_two.model.Request  
import kotlinx.coroutines.CoroutineScope  
import kotlinx.coroutines.channels.Channel  
import kotlinx.coroutines.delay  
import kotlinx.coroutines.flow.receiveAsFlow  
import kotlinx.coroutines.launch  
  
  
class SMO(  
 private val queueSMO: QueueSMO,  
 private val coroutineScope: CoroutineScope,  
 private val muServiceFlow: Double,  
 nChannels: Int,  
 private val epochTime: Long,  
 //число занятых каналов поменялось  
 val reportSizeChanged: (busyChannels: Int, allFinishedListSize: Int) -> Unit  
) {  
 //список каналов обслуживания  
 private val channels: List<ChannelSMO>  
 //канал с завершенными заявками, все каналы обсуживания сбрасывают заявки сюда  
 private val finishedRequests = *Channel*<Request>()  
 private val allFinishedList = *mutableListOf*<Request>()  
 fun getAllFinishedList(): List<Request> = allFinishedList.*toList*()  
  
 init {  
 channels = *List*(nChannels) **{** ChannelSMO(id = **it**, muServiceFlow, coroutineScope, epochTime, finishedRequests) **}** subscribeToFinished()  
 coroutineScope.*launch* **{** while (true) {  
 delay(1)  
 channels.*firstOrNull* **{ it**.isAvailable **}**?.*also* **{** channelsSMO **->** queueSMO.getRequest()?.*let* **{** request **->** channelsSMO.putRequest(request)  
 reportSizeChanged(channels.*filter* **{** !**it**.isAvailable **}**.size, allFinishedList.size)  
 **}  
 }** }  
 **}** }  
  
 private fun subscribeToFinished() {  
 coroutineScope.*launch* **{** finishedRequests.*receiveAsFlow*().collect **{** request **->** *println*("request id ${request.id} finished service time: ${request.serviceWaitingTime}")  
 allFinishedList += request  
 reportSizeChanged(channels.*filter* **{** !**it**.isAvailable **}**.size, allFinishedList.size)  
 *println*("finishedListSize = ${allFinishedList.size}")  
 **}  
 }** }  
}

## Сбор состояний системы

package ipr\_two  
  
import ipr\_two.model.State  
import kotlinx.coroutines.sync.Mutex  
import kotlinx.coroutines.sync.withLock  
import kotlinx.html.currentTimeMillis  
  
class StateCollector(  
 val epochTime: Long  
) {  
 private val mutex = *Mutex*()  
 private val stateList = *mutableListOf*<State>()  
 private var lastState: State = State(0, 0, 0L, epochTime, 0, 0, 0)  
  
 suspend fun stateChanged(newQueueSize: Int?, newBusyChannelsSize: Int?, newQueueLeftSize: Int?, newFinishedRequests: Int?, newImpatientLeftSize: Int?) {  
 mutex.withLock **{** val currentTime = *currentTimeMillis*()  
 lastState.stateTime = currentTime - lastState.stateTimeStamp  
 stateList += lastState  
 val newLastState = State(  
 queueSize = newQueueSize ?: lastState.queueSize,  
 busyChannels = newBusyChannelsSize ?: lastState.busyChannels,  
 stateTime = 0L,  
 stateTimeStamp = currentTime,  
 queueLeftSize = newQueueLeftSize ?: lastState.queueLeftSize,  
 finishedRequests = newFinishedRequests ?: lastState.finishedRequests,  
 impatientLeftSize = newImpatientLeftSize ?: lastState.impatientLeftSize  
 )  
 lastState = newLastState  
 **}** }  
  
 fun getStateList(): List<State> = stateList.*toList*()  
  
}

## Подсчет статистики и генерация excel файла

package ipr\_two  
  
import kr\_two.OSDetector  
import kr\_two.factorial  
import org.apache.poi.hssf.usermodel.HSSFWorkbook  
import org.apache.poi.ss.usermodel.FillPatternType  
import org.apache.poi.ss.usermodel.IndexedColors  
import org.apache.poi.ss.util.CellRangeAddress  
import org.jetbrains.letsPlot.GGBunch  
import org.jetbrains.letsPlot.geom.geomLine  
import org.jetbrains.letsPlot.label.ggtitle  
import org.jetbrains.letsPlot.letsPlot  
import java.io.File  
import java.io.FileOutputStream  
import kotlin.math.pow  
  
object StatisticsCalculator {  
 fun calculate(getter: IprTwoInputGetter) {  
 val n = getter.nChannels!!  
 val M = getter.mQueue!!  
 val lambda = getter.lambdaInputFlow!!  
 val mu = getter.muServiceFlow!!  
 val nu = getter.nuLeaving!!  
 val nList = *List*(n) **{ it** + 1 **}** val MList = *List*(M) **{ it** + 1 **}** *println*("~~~~~~~~~~~~~~~~~~~~~ТЕОРЕТИЧЕСКИЕ~~~~~~~~~~~~~~~~~~~~~~~~~~~")  
 val myY = lambda / mu  
 val alphaOj = nu / mu  
 *println*("y нагрузка = $myY")  
  
 //1.55  
 val p0 = 1 /  
 (1 +  
 (nList.*fold*(0.0) **{** sum, i **->** sum + (myY.*pow*(i) / *factorial*(i))  
 **}**)  
 +  
 (myY.*pow*(n) / (*factorial*(n) \* MList.*fold*(0.0) **{** sum, r **->** val rList = *List*(r) **{ it** + 1 **}** //myY.pow(r)  
 sum + (myY.*pow*(r) /  
 rList.*fold*(1.0) **{** mult, s **->** mult \* (n + s \* alphaOj)  
 **}**)  
 **}**)))  
  
 val pList = *mutableListOf*(p0)  
 //1.56  
 for (i in 1..n) {  
 val pIndexed = myY.*pow*(i) \* p0 / *factorial*(i)  
 pList.add(pIndexed)  
 }  
  
 //1.57  
 for (i in 1..M) {  
 val rList = *List*(i) **{ it** + 1 **}** val pIndexed = (myY.*pow*(n + i) \* p0 /  
 (*factorial*(i) \* rList.*fold*(1.0) **{** mult, s **->** mult \* (n + s \* alphaOj)  
 **}**))  
  
 pList.add(pIndexed)  
 }  
  
 *println*("p sum = ${pList.*sum*()}")  
  
 while (pList.*sum*() !in 0.9999..1.0001) {  
 *println*("производим нормировку")  
 val x = 1 / pList.*sum*()  
 val mapped = pList.*map* **{ it** \* x **}** pList.clear()  
 pList.addAll(mapped)  
 *println*("p sum = ${pList.*sum*()}")  
 }  
  
 pList.*forEachIndexed* **{** index, d **->** *println*("p${index}= $d")  
 **}** val potk = pList.*last*()  
 *println*("P отказа = $potk")  
  
 val kzan = nList.*fold*(0.0) **{** sum, i **->** sum + i \* pList[i]  
 **}** + n \* MList.*fold*(0.0) **{** sum, r **->** sum + pList[n + r]  
 **}** *println*("kzan среднее число занятых каналов = $kzan")  
  
 val lambda\_ = kzan \* mu  
 *println*("λ` абсолютная пропускная способность = $lambda\_")  
  
 val l = MList.*fold*(0.0) **{** sum, r **->** sum + r \* pList[n + r]  
 **}** *println*("l среднее число заявок, находящихся в очереди = $l")  
 val w = l / (lambda - nu)  
 *println*("w среднее время ожидания в очереди = $w")  
 val m = l + kzan  
 *println*("m среднее число заявок в СМО = $m")  
 val u = m / (lambda - nu)  
 *println*("u среднеe время пребывания заявки в СМО = $u")  
  
 *println*("~~~~~~~~~~~~~~~~~~~~~ПРАКТИЧЕСКИЕ~~~~~~~~~~~~~~~~~~~~~~~~~~~")  
  
 val epochStartTime = getter.epochStartTime  
 val allFinishedList = getter.getAllFinishedList()  
 val allImpatientList = getter.getAllImpatient()  
 val \_potk2 = getter.getLeftQueueSize().toDouble() / getter.getRequestProducedSize()  
 val (invalidStates, validStates) = getter.stateCollector.getStateList().*toMutableList*()  
 .*partition* **{ it**.busyChannels < n && **it**.queueSize > 0 **}** val validStatesTime = validStates.*sumOf* **{ it**.stateTime **}**.*also* **{** *println*("validStatesTime = $**it**") **}** val invalidStatesTime = invalidStates.*sumOf* **{ it**.stateTime **}**.*also* **{** *println*("invalidStatesTime = $**it**") **}** val \_pList = *mutableListOf*<Double>()  
 for (i in 0..n) {  
 validStates  
 .*filter* **{ it**.queueSize == 0 && **it**.busyChannels == i **}** .*fold*(0L) **{** sum, element **->** sum + element.stateTime **}** .*apply* **{** \_pList.add(this.toDouble() / validStatesTime) **}** }  
  
 for (i in 1..M) {  
 validStates  
 .*filter* **{ it**.queueSize == i && **it**.busyChannels == n **}** .*fold*(0L) **{** sum, element **->** sum + element.stateTime **}** .*apply* **{** \_pList.add(this.toDouble() / validStatesTime) **}** }  
 \_pList.*forEachIndexed* **{** index, d **->** *println*("\_p${index}= $d")  
 **}** *println*("\_p sum= ${\_pList.*sum*()}")  
  
 val \_potk1 = \_pList.*last*()  
 *println*("\_p отказа = $\_potk1")  
 *println*("\_p отказа (по ушедшим) = $\_potk2")  
  
 var \_kzan = 0.0  
 for (i in 0..n) {  
 \_kzan += i \*  
 (validStates  
 .*filter* **{ it**.busyChannels == i **}** .*sumOf* **{ it**.stateTime.toDouble() **}** / validStatesTime)  
 }  
 *println*("\_kzan среднее число занятых каналов = $\_kzan")  
  
 var \_l = 0.0  
 for (i in 0..M) {  
 \_l += i \*  
 (validStates  
 .*filter* **{ it**.queueSize == i **}** .*sumOf* **{ it**.stateTime.toDouble() **}** / validStatesTime)  
 }  
 *println*("\_l среднее число заявок, находящихся в очереди = $\_l")  
  
 val \_m = \_l + \_kzan  
 *println*("\_m среднее число заявок в СМО = $\_m")  
  
 //ПРАКТИЧЕСКОE среднее число заявок, обслуженное в единицу времени  
 val \_lambda\_ = allFinishedList.size.toDouble() / validStatesTime \* *MILLIS\_IN\_SECOND  
 println*("\_λ` абсолютная пропускная способность = $\_lambda\_")  
  
  
 val \_w = allFinishedList.*map* **{ it**.queueWaitingTime **}**.*average*() / *MILLIS\_IN\_SECOND  
 println*("\_w среднее время ожидания в очереди = $\_w")  
  
 val \_p = allFinishedList  
 .*map* **{ it**.serviceWaitingTime **}** .*average*() / *MILLIS\_IN\_SECOND  
 println*("\_p среднее время обслуживания в канале = $\_p")  
  
  
 val \_u = \_w + \_p  
 *println*("среднеe время пребывания заявки в СМО = $\_u")  
  
 */\*\*  
 \* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~Графики~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
 \*/* val dataQueue = *mapOf*<String, List<\*>>(  
 "x" *to* validStates.*map* **{** (**it**.stateTimeStamp - epochStartTime).toDouble() / *MILLIS\_IN\_SECOND* **}**,  
 "y" *to* validStates.*map* **{ it**.queueSize **}** )  
 val pDataQueue = *letsPlot*(dataQueue, mapping = **{** x = "x"; y = "y" **}**)  
 val plotDataQueue = (pDataQueue +  
 geomLine **{** y = "y" **}** +  
 *ggtitle*("размер очереди"))  
  
 val dataChannels = *mapOf*<String, List<\*>>(  
 "x" *to* validStates.*map* **{** (**it**.stateTimeStamp - epochStartTime).toDouble() / *MILLIS\_IN\_SECOND* **}**,  
 "y" *to* validStates.*map* **{ it**.busyChannels **}** )  
 val pDataChannels = *letsPlot*(dataChannels, mapping = **{** x = "x"; y = "y" **}**)  
 val plotDataChannels = (pDataChannels +  
 geomLine **{** y = "y" **}** +  
 *ggtitle*("занятые каналы"))  
  
 val dataLeftRequests = *mapOf*<String, List<\*>>(  
 "x" *to* validStates.*map* **{** (**it**.stateTimeStamp - epochStartTime).toDouble() / *MILLIS\_IN\_SECOND* **}**,  
 "y" *to* validStates.*map* **{ it**.queueLeftSize **}** )  
 val pDataLeftRequests = *letsPlot*(dataLeftRequests, mapping = **{** x = "x"; y = "y" **}**)  
 val plotDataLeftRequests = (pDataLeftRequests +  
 geomLine **{** y = "y" **}** +  
 *ggtitle*("Покинули очередь, упершись в хвост"))  
  
 val dataFinishedRequests = *mapOf*<String, List<\*>>(  
 "x" *to* validStates.*map* **{** (**it**.stateTimeStamp - epochStartTime).toDouble() / *MILLIS\_IN\_SECOND* **}**,  
 "y" *to* validStates.*map* **{ it**.finishedRequests **}** )  
 val pDataFinishedRequests = *letsPlot*(dataFinishedRequests, mapping = **{** x = "x"; y = "y" **}**)  
 val plotDataFinishedRequests = (pDataFinishedRequests +  
 geomLine **{** y = "y" **}** +  
 *ggtitle*("Покинули систему обслуженными"))  
  
 val dataImpationedFinishedRequests = *mapOf*<String, List<\*>>(  
 "x" *to* validStates.*map* **{** (**it**.stateTimeStamp - epochStartTime).toDouble() / *MILLIS\_IN\_SECOND* **}**,  
 "y" *to* validStates.*map* **{ it**.impatientLeftSize **}** )  
 val pDataImpationedFinishedRequests = *letsPlot*(dataImpationedFinishedRequests, mapping = **{** x = "x"; y = "y" **}**)  
 val plotDataImpationedFinishedRequests = (pDataImpationedFinishedRequests +  
 geomLine **{** y = "y" **}** +  
 *ggtitle*("Покинули очередь, не дождавшись"))  
  
 GGBunch()  
 .addPlot(  
 plotDataQueue, 0, 0, 500, 200  
 )  
 .addPlot(  
 plotDataLeftRequests, 0, 250, 500, 200  
 )  
 .addPlot(  
 plotDataChannels, 550, 0, 500, 200  
 )  
 .addPlot(  
 plotDataFinishedRequests, 550, 250, 500, 200  
 )  
 .addPlot(  
 plotDataImpationedFinishedRequests, 0, 500, 500, 200  
 )  
 .show()  
  
 */\*\*  
 \* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~Excel~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  
 \*/* val workbook = HSSFWorkbook()  
 val sheet = workbook.createSheet(SHEET\_NAME).*apply* **{** setColumnWidth(0, 46 \* 256)  
 setColumnWidth(1, 16 \* 256)  
 setColumnWidth(2, 16 \* 256)  
  
 **}** val headerCellStyle = workbook.createCellStyle().*apply* **{** *fillForegroundColor* = IndexedColors.*LEMON\_CHIFFON*.getIndex()  
 *fillPattern* = FillPatternType.*SOLID\_FOREGROUND* **}** sheet.createRow(0).*apply* **{** createCell(0).setCellValue("n каналов =")  
 createCell(1).setCellValue(n.toDouble())  
 **}** sheet.createRow(1).*apply* **{** createCell(0).setCellValue("M мест в очереди =")  
 createCell(1).setCellValue(M.toDouble())  
 **}** sheet.createRow(2).*apply* **{** createCell(0).setCellValue("λ интенсивность входящего потока =")  
 createCell(1).setCellValue(lambda)  
 **}** sheet.createRow(3).*apply* **{** createCell(0).setCellValue("μ интенсивность потока обслуживания =")  
 createCell(1).setCellValue(mu)  
 **}** sheet.createRow(4).*apply* **{** createCell(0).setCellValue("ν параметр закона ухода =")  
 createCell(1).setCellValue(nu)  
 **}** sheet.createRow(5).*apply* **{** createCell(0).*apply* **{** setCellValue("Параметры:")  
 setCellStyle(headerCellStyle)  
 **}** createCell(1).*apply* **{** setCellValue("Теоретическое:")  
 setCellStyle(headerCellStyle)  
 **}** createCell(2).*apply* **{** setCellValue("Практическое:")  
 setCellStyle(headerCellStyle)  
 **}  
 }** sheet.createRow(6).*apply* **{** createCell(0).setCellValue("P отказа =")  
 createCell(1).setCellValue(potk)  
 createCell(2).setCellValue(\_potk1)  
 createCell(3).setCellValue(\_potk2)  
 **}** sheet.createRow(7).*apply* **{** createCell(0).setCellValue("l среднее число заявок, находящихся в очереди =")  
 createCell(1).setCellValue(l)  
 createCell(2).setCellValue(\_l)  
 **}** sheet.createRow(8).*apply* **{** createCell(0).setCellValue("kzan среднее число занятых каналов =")  
 createCell(1).setCellValue(kzan)  
 createCell(2).setCellValue(\_kzan)  
 **}** sheet.createRow(9).*apply* **{** createCell(0).setCellValue("m среднее число заявок в СМО =")  
 createCell(1).setCellValue(m)  
 createCell(2).setCellValue(\_m)  
 **}** sheet.createRow(10).*apply* **{** createCell(0).setCellValue("w среднее время ожидания в очереди =")  
 createCell(1).setCellValue(w)  
 createCell(2).setCellValue(\_w)  
 **}** sheet.createRow(11).*apply* **{** createCell(0).setCellValue("u среднеe время пребывания заявки в СМО =")  
 createCell(1).setCellValue(u)  
 createCell(2).setCellValue(\_u)  
 **}** sheet.createRow(12).*apply* **{** createCell(0).setCellValue("λ` абсолютная пропускная способность =")  
 createCell(1).setCellValue(lambda\_)  
 createCell(2).setCellValue(\_lambda\_)  
 **}** sheet.createRow(13).*apply* **{** sheet.addMergedRegion(CellRangeAddress(13, 13, 0, 2))  
 createCell(0).*apply* **{** setCellValue("финальные вероятности состояний")  
 setCellStyle(headerCellStyle)  
 **}  
 }** pList.*forEachIndexed* **{** index, d **->** sheet.createRow(14 + index).*apply* **{** createCell(0).setCellValue("p$index =")  
 createCell(1).setCellValue(d)  
 createCell(2).setCellValue(\_pList[index])  
 **}  
 }** val path = String.*format*(PATH, System.currentTimeMillis().toString())  
 FileOutputStream(path).*use* **{** fileOut **->** workbook.write(fileOut)  
 workbook.close()  
 fileOut.close()  
 **}** OSDetector.openWithSystem(File(path))  
 }  
  
  
 private const val PATH = "src/main/kotlin/ipr\_two/reports/%s\_JavaBooks.xls"  
 private const val SHEET\_NAME = "SMO report"  
  
}

## Сохранение параметров

package ipr\_two  
  
import java.util.prefs.Preferences  
  
object IprTwoParamsSaver {  
 private fun prefNode(): Preferences = Preferences.userRoot().node("IPR\_TWO")  
  
 fun loadIprTwoParams(): IprTwoParams {  
 return prefNode().run **{** IprTwoParams(  
 channelsN = getInt(IprTwoParams.N\_KEY, 3),  
 queueM = getInt(IprTwoParams.M\_KEY, 10),  
 inputFlowLambda = getDouble(IprTwoParams.LAMBDA\_KEY, 1.0),  
 serviceFlowMu = getDouble(IprTwoParams.MU\_KEY, 1.0),  
 leavingNu = getDouble(IprTwoParams.NU\_KEY, 1.0)  
 )  
 **}** }  
  
 fun saveIprTwoParams(params: IprTwoParams) {  
 prefNode().apply **{** putInt(IprTwoParams.N\_KEY, params.channelsN)  
 putInt(IprTwoParams.M\_KEY, params.queueM)  
 putDouble(IprTwoParams.LAMBDA\_KEY, params.inputFlowLambda)  
 putDouble(IprTwoParams.MU\_KEY, params.serviceFlowMu)  
 putDouble(IprTwoParams.NU\_KEY, params.leavingNu)  
 **}** }  
}  
  
*/\*\*  
 \** ***@param*** *inputFlowLambda λ интенсивность входящего потока  
 \** ***@param*** *serviceFlowMu μ интенсивность потока обслуживания  
 \** ***@param*** *leavingNu ν параметр закона ухода  
 \*/*data class IprTwoParams(  
 val channelsN: Int,  
 val queueM: Int,  
 val inputFlowLambda: Double,  
 val serviceFlowMu: Double,  
 val leavingNu: Double  
) {  
 companion object {  
 const val N\_KEY = "n"  
 const val M\_KEY = "m"  
 const val LAMBDA\_KEY = "λ"  
 const val MU\_KEY = "μ"  
 const val NU\_KEY = "ν"  
 }  
}

# Литература

1. Т. В. Черушева, Н. В. Зверовщикова ТЕОРИЯ МАССОВОГО ОБСЛУЖИВАНИЯ Учебное пособие Пенза Издательство ПГУ 2021  
   <https://dep_vipm.pnzgu.ru/files/dep_vipm.pnzgu.ru/books/cherusheva_2021_teoria_masobsl.pdf>