Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**

**(ФГБОУ ВО «КубГУ»)**

**Факультет компьютерных технологий и прикладной математики**

**Кафедра вычислительных технологий**

**ЛАБОРАТОРНАЯ РАБОТА №7**

**Дисциплина: Платформо-независимое программирование**

Работу выполнил: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ В. С. Таран

Направление подготовки: 02.03.02 Фундаментальная информатика и информационные технологии

Преподаватель: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ В. И. Шиян

**Тема:** Потоки Executable.

**Цель**:Изучить работу потоковExecutable и провести анализ процессов обслуживания в системах с несколькими каналами обслуживания.

**Задание:** используя потоки, выполнить задания.

Вариант 6.

Железнодорожная касса с двумя окошками продает билеты в пункты А и В. Интенсивность потока пассажиров, желающих купить билеты, для обоих пунктов одинакова: λА = λВ = 0,45 (чел./мин). Рассматриваются два варианта продажи билетов: 1 й — билеты продаются в одной кассе с двумя окошками одновременно в оба пункта А и В; 2 й — билеты продаются в двух специализированных кассах (по одному окошку в каждой): одна — только в пункт А, другая — только в пункт В. На обслуживание пассажиров кассир тратит в среднем две минуты.

Необходимо:

а) сравнить два варианта продажи билетов по основным характеристикам обслуживания;

б) определить, как надо изменить среднее время обслуживания одного пассажира, чтобы по второму варианту продажи пассажиры затрачивали на приобретение билетов в среднем меньше времени, чем по первому варианту.

**Ход работы:**

Реализуем сначала первый вариант продажи билетов. Сначала создадим класс Passenger, который создает пассажира с уникальным id, временем обслуживания и временем ожидания.

public class Passenger {  
 private final int id;  
 private final int serviceTime;  
 private final int arrivalTick;  
  
  
 public int getId() {  
 return id;  
 }  
  
 public int getServiceTime() {  
 return serviceTime;  
 }  
  
 public int getArrivalTick() {  
 return arrivalTick;  
 }  
  
 public Passenger(int id, int serviceTime, int arrivalTick) {  
 this.id = id;  
 this.serviceTime = serviceTime;  
 this.arrivalTick = arrivalTick;  
  
 }

Реализуем класс TicketWindow, в котором объект содержит основные поляname (название окна) и queue (для хранения идентификаторов пассажиров, ожидающих обслуживание). Также есть поле totalServiceTime для хранения общего времени обслуживания людей в течении всей симуляции. И объект rand, который используется для генерации случайных чисел для вычисления времени обслуживания.

Так как класс TicketWindow реализует интерфейс Runnable, то реализуем метод run(), который будет вызван, когда запустим поток. Код в этом методе будет выполняться в контексте отдельного потока, пока не будет прерван. В цикле извлекаем элемент из очереди, и если очередь пуста, поток будет ждать, пока в нее не добавится элемент. Так как среднее время обслуживания пассажира 2 минуты, то задаем случайное время от 1,9 мин до 2,1 мин. Когда выбрасывается исключение InterruptedExeption, выводится сообщение о завершении работы текущего окошка.

package ru.practicum;  
import java.util.concurrent.BlockingQueue;  
import java.util.Random;  
import java.util.concurrent.atomic.AtomicLong;  
import java.util.concurrent.atomic.AtomicInteger;  
  
public class TicketWindow implements Runnable {  
 private final String name;  
 private final BlockingQueue<Passenger> queue;  
 private final AtomicLong totalServiceTime;  
 private final AtomicLong totalWaitingTime;  
 private int lastServiceEndTick = 0;  
  
 public TicketWindow(String name, BlockingQueue<Passenger> queue, AtomicLong totalServiceTime, AtomicLong totalWaitingTime) {  
 this.name = name;  
 this.queue = queue;  
 this.totalServiceTime = totalServiceTime;  
 this.totalWaitingTime = totalWaitingTime;  
 }  
  
 @Override  
 public void run() {  
 try {  
 while (!Thread.*currentThread*().isInterrupted()) {  
 Passenger passenger = queue.take();  
   
 int startServiceTick = Math.*max*(passenger.getArrivalTick(), lastServiceEndTick);  
 int waitingTime = startServiceTick - passenger.getArrivalTick();  
   
 if (waitingTime < 0) waitingTime = 0;  
   
 totalWaitingTime.addAndGet(waitingTime);  
   
 synchronized (System.*out*) {  
 System.*out*.println(passenger.getId() + " имя " + name + " ждал " + waitingTime/1000.0);  
 }  
   
 int serviceTime = passenger.getServiceTime();  
   
 lastServiceEndTick = startServiceTick + serviceTime;  
 totalServiceTime.addAndGet(serviceTime);  
   
 synchronized (System.*out*) {  
 System.*out*.println(passenger.getId() + " имя " + name + " обслуживался " + serviceTime/1000.0);  
 }  
 }  
  
 } catch (InterruptedException e) {  
 System.*out*.println("[" + name + "] Работа завершена.");  
 Thread.*currentThread*().interrupt();  
 }  
 }  
}

Основной класс, показанный ниже, имитирует работу очередей. Задается очередь для хранения объектов и объект для отслеживания общего времени обслуживания. Затем создается пул потоков с двумя потоками, чтобы запустить 2 окошка одновременно.

BlockingQueue<Passenger> queue = new LinkedBlockingQueue<>();  
AtomicLong totalServiceTime = new AtomicLong(0);  
AtomicLong totalWaitingTime = new AtomicLong(0);  
  
ExecutorService executor = Executors.*newFixedThreadPool*(2);  
executor.submit(new TicketWindow("Окно 1", queue, totalServiceTime, totalWaitingTime));  
executor.submit(new TicketWindow("Окно 2", queue, totalServiceTime, totalWaitingTime));

Поток generator генерирует пассажиров с экспоненциальным интервалом времени. Каждый пассажир получает уникальный идентификатор, который помещается в очередь. Интервал между прибытием пассажиров генерируется с помощью распределения, связанного с параметром λ, который определяет интенсивность потока пассажиров.

Thread generator = new Thread(() -> {  
 int id = 1;  
 Random random = new Random();  
 double lambda = 0.9;  
 int currentTick = 0;  
  
 while(id <= 5) {  
 try {  
 synchronized (System.*out*) {  
 System.*out*.println("Пассажир " + id + " прибыл");  
 }  
  
 int serviceTime = 1900 + random.nextInt(200);  
 Passenger passenger = new Passenger(id, serviceTime, currentTick);  
  
 queue.put(passenger);  
 double nextInterval = -Math.*log*(1 - random.nextDouble()) / lambda;  
 nextInterval = Math.*max*(0.5, Math.*min*(2.0, nextInterval));  
 int arrivalInterval = (int) (nextInterval \* 1000);  
 currentTick+= arrivalInterval;  
 *globalTick* = currentTick;  
  
 id++;  
 } catch (InterruptedException e) {  
 break;  
 }  
 }  
});

Поток generator запускается start() и ожидает завершения с помощьюjoin(). Затем программа ждет обработку всех пассажиров, вызывает shutdownNow() для завершения всех потоков в пуле и awaitTermination(), чтобы дождаться завершения потоков.

В конце выводится общее время обслуживания всех пассажиров. Это все показано ниже.

generator.start();  
generator.join();  
  
if (!executor.awaitTermination(3, TimeUnit.*SECONDS*)) {  
 executor.shutdown();  
}  
  
double totalServiceMinutes = totalServiceTime.get() / 1000.0;  
double totalWaitingMinutes = totalWaitingTime.get() / 1000.0;  
  
synchronized (System.*out*) {  
 System.*out*.println("Общее время обслуживания всех пассажиров " + totalServiceMinutes + " мин");  
 System.*out*.println("Общее время ожидания всех пассажиров " + totalWaitingMinutes + " мин");  
}

На рисунке 1 представлен результат работы программы.

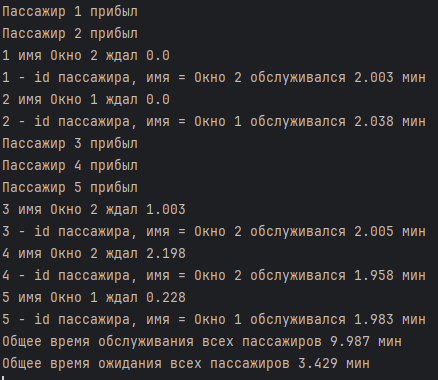


Рисунок 1 – результат работы 1 программы

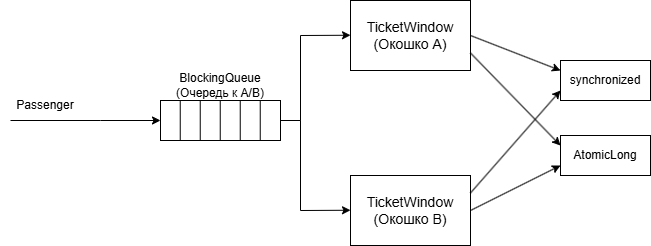


Рисунок 2 – Чертеж работы приложения для варианта 1

Теперь реализуем второй вариант продажи билетов. В классе Passenger добавим направление, куда хочет отправиться человек.

public class Passenger {  
 private final int id;  
 private final int serviceTime;  
 private final int arrivalTick;  
 private final String destination;  
  
 public int getId() {  
 return id;  
 }  
  
 public int getServiceTime() {  
 return serviceTime;  
 }  
  
 public int getArrivalTick() {  
 return arrivalTick;  
 }  
  
 public Passenger(int id, int serviceTime, int arrivalTick, String destination) {  
 this.id = id;  
 this.serviceTime = serviceTime;  
 this.arrivalTick = arrivalTick;  
 this.destination = destination;  
 }  
  
 public String getDestination() {  
 return destination;  
 }  
}

В TicketWindow ничего не изменяется.

Вgenerator создаем пассажиров с рандомными пунктами назначения и добавляем их в очереди.

Thread generator2 = new Thread(() -> {  
 int id = 1;  
 Random random = new Random();  
 double lambda = 0.9;  
 int currentTick = 0;  
  
 while (id <= 5) {  
 try {  
 int destinationInt = random.nextInt(2);  
 String destination = (destinationInt == 0) ? "A" : "B";  
  
 int serviceTime = 1900 + random.nextInt(200);  
 Passenger p = new Passenger(id, serviceTime, currentTick, destination);  
 synchronized (System.*out*) {  
 System.*out*.println("Пассажир с id " + id + " прибыл " + destination);  
 if (destination.equals("A")) {  
 queueA.put(p);  
 } else {  
 queueB.put(p);  
 }  
 double nextInterval = -Math.*log*(1 - random.nextDouble()) / lambda;  
 int arrivalInterval = (int) (nextInterval \* 1000);  
 currentTick += arrivalInterval;  
 *globalTick* = currentTick;  
 id++;  
 }  
 } catch (InterruptedException e) {  
 break;  
 }  
 }  
});  
  
generator2.start();  
generator2.join();

На рисунке 3 показан результат работы программы.

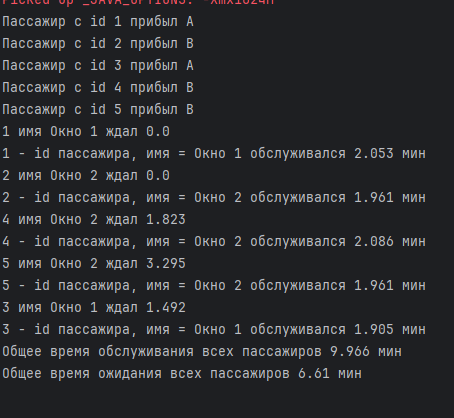
****

Рисунок 3– Результат работы 2 программы

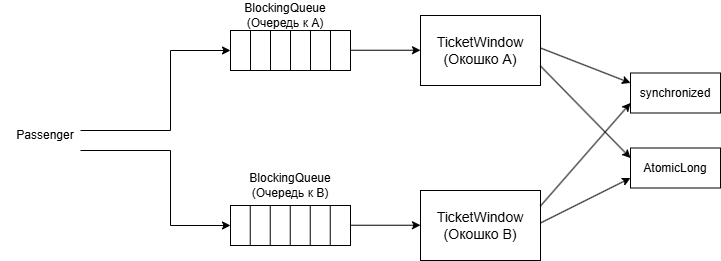


Рисунок 4 – Схема работы программы для варианта 2

Второй вариант обслуживания пассажиров будет работать хуже и дольше, потому что очереди в большинстве случаев не равны, а это значит, что одно из окон может простаивать, пока вся очередь находится у другого окна. То есть, здесь очень большое время ожидания пассажирами своего времени обслуживания. Чтобы этого избежать, нужно уменьшить время обслуживания хотя бы до 1,5 минуты.

Из синхронизаторов использовались BlockingQueue,AtomicLong и synchronised.

BlockingQueue – интерфейс из пакета java.util.concurrent, который расширяет стандартный Queue, добавляя блокирующие операции, а LinkedBlockingQueue – его конкретная реализация. В программе эта очередь используется для хранения идентификаторов пассажиров, которые будут обслуживаться. Когда пассажир появляется, он добавляется в очередь с помощью put(). Окощки будут извлекать элементы из этой очереди с помощью метода take(), что блокирует их, если очередь пуста, пока не появится новый пассажир.

AtomicLong – класс из пакета java.util.concurrent.atomic, который представляет собой потокобезопасное долгоживущее число. Это разновидность типа Long, которая гарантирует, что операции над этим числом будут атомарными, то есть, без возможности его изменения другими потоками в промежутке между операциями.

Synchronised – блок синхронизации, в котором в качестве монитора в моем случае используется объект потока вывода в консоль. Это гарантирует, что только один поток в данный момент выводит строку.

**Вывод:** проделав лабораторную работу, были изучены потоки Executableи продемонстрирована их работа на примере задачи обслуживания пассажиров.