Санкт-Петербургский Политехнический Университет Петра Великого

Институт прикладной математики и механики

**Кафедра “Телематика (при ЦНИИ РТК)”**

**К У Р С О В О Й П Р О Е К Т**

**Создание многопоточного приложения**

по дисциплине “Параллельные вычисления и алгоритмы”

Выполнил студент группы 13643/1 Маджуга П.И.

Руководитель  Лукашин А.А.

Санкт-Петербург 2018

**СОДЕРЖАНИЕ**

Введение 3

Задача 4

Описание программы 5

Код программы 6

Введение

**Параллельные вычисления** — способ организации [компьютерных вычислений](https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%82%D0%B5%D1%85%D0%BD%D0%B8%D0%BA%D0%B0), при котором [программы разрабатываются](https://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F) как набор взаимодействующих вычислительных процессов, работающих параллельно (одновременно). Термин охватывает совокупность вопросов [параллелизма в программировании](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%BB%D0%BB%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B9_%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC), а также создание эффективно действующих [аппаратных реализаций](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%BB%D0%BB%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D0%B2%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D1%8B). Теория параллельных вычислений составляет раздел [прикладной теории алгоритмов](https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D0%BE%D1%80%D0%B8%D1%8F_%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC%D0%BE%D0%B2).

Существуют различные способы реализации параллельных вычислений. Например, каждый вычислительный процесс может быть реализован в виде [процесса операционной системы](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)), либо же вычислительные процессы могут представлять собой набор [потоков выполнения](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%82%D0%BE%D0%BA_%D0%B2%D1%8B%D0%BF%D0%BE%D0%BB%D0%BD%D0%B5%D0%BD%D0%B8%D1%8F) внутри одного процесса ОС. Параллельные программы могут физически исполняться либо последовательно на единственном [процессоре](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80) — перемежая по очереди шаги выполнения каждого вычислительного процесса, либо параллельно — выделяя каждому вычислительному процессу один или несколько процессоров (находящихся [рядом](https://ru.wikipedia.org/wiki/%D0%9C%D0%BD%D0%BE%D0%B3%D0%BE%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80%D0%BD%D0%BE%D1%81%D1%82%D1%8C) или [распределённых](https://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D1%81%D0%BF%D1%80%D0%B5%D0%B4%D0%B5%D0%BB%D1%91%D0%BD%D0%BD%D1%8B%D0%B5_%D0%B2%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F) в компьютерную [сеть](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D1%81%D0%B5%D1%82%D1%8C)).

Основная сложность при проектировании параллельных программ — обеспечить правильную последовательность взаимодействий между различными вычислительными процессами, а также координацию ресурсов, разделяемых между процессами.

Задача

Американские горки. Система управления американскими горками позволяет отправлять тележку только когда она заполнена полностью. Пассажиры прибывают на платформу и регистрируются в контроллере с помощью турникетов. Когда набирается M пассажиров контроллер сигнализирует отправление тележки. По прибытии люди покидают платформу, и следующий пассажиры входят на борт. Не учитывать синхронизацию между погрузкой пассажиров и отправлением тележки. Американская горка состоит из трех процессов: турникет, контроллер, тележка. Турникет и контроллер взаимодействуют с помощью общего действия, обозначающего прибытие пассажира. Контроллер и тележка взаимодействуют с помощью действия отправления тележки.

Описание программы

В классе RollerCoaster 2 потока: пассажиры и тележка, запуск выполнения потоков осуществляется с помощью метода Thread.start()*.* В Java так же предусмотрен механизм, позволяющий одному потоку ждать завершения выполнения другого. Для этого используется метод Thread.join(). Метод Thread.sleep()делает поток спящим. Его выполнение будет приостановлено на указанное количество миллисекунд, после чего он снова начнет выполняться. Этот метод полезен при синхронизации потоков. Пассажиров 50, мест в тележке 10.

В классе Passenger реализован интерфейс Runnable для создания потока. Внутри интерфейса Runnable осуществляется прерывание - указание потоку прекратить текущую работу и сделать что-то ещё. Поток может послать прерывание вызовом метода interrupt () у объекта Thread, если нужно прервать ассоциированный с ним поток. Механизм прерывания реализован с использованием внутреннего флага interrupt status (флаг прерывания) класса Thread. Вызов Thread.interrupt() поднимает этот флаг. По соглашению, любой метод, завершающийся выбрасыванием InterruptedException, сбрасывает флаг прерывания.  Thread.interrupted() - статический метод класса Thread, и его вызов возвращает значение флага прерывания того потока, из которого он был вызван. Метод run() отправляет одного пассажира в тележку, один пассажир может проехать только один раз. Этот поток завершится, когда метод вернёт управление.

В классе Car реализован интерфейс Runnable для создания потока. Метод run() отправляет тележку, если пассажиры на местах.

В классе Monitor проверяется наличие места в тележке для нового пассажира (seatAvailable). Если место есть, то пассажир занимает свободное место (tryToGetOnCar), если мест больше нет, то тележка уезжает (passengerGetOn). Пока тележка находится в движении новые пассажиры не могут сесть в тележку и ожидают (carIsRunning). Как только тележка прекращает ехать, все параметры сбрасываются (passengerGetOff).

Код программы

import java.math.\*;

class RollerCoaster {

public static int PASSENGER\_NUM = 50;

public static int CAR\_NUM = 1;

public static int SEAT\_AVAIL = 10;

public static void main(String[] args) {

Monitor rcMon = new Monitor();

Car theCar;

Passenger aPassenger;

Thread t1[] = new Thread[PASSENGER\_NUM];

Thread t2[] = new Thread[CAR\_NUM];

for (int i = 0; i < PASSENGER\_NUM; i++) {

aPassenger = new Passenger(i, rcMon);

t1[i] = new Thread(aPassenger);

}

for (int i = 0; i < CAR\_NUM; i++) {

theCar = new Car(i, rcMon);

t2[i] = new Thread(theCar);

}

for(int i = 0; i < PASSENGER\_NUM; i++) {

t1[i].start();

}

for(int i = 0; i < CAR\_NUM; i++) {

t2[i].start();

}

try {

for (int i = 0; i < PASSENGER\_NUM; i++) {

t1[i].join();

}

} catch (InterruptedException e) {

}

try {

for (int i = 0; i < CAR\_NUM; i++) {

t2[i].join();

}

} catch (InterruptedException e) {

}

}

}

class Passenger implements Runnable {

private int id;

private Monitor passengerMon;

public Passenger(int i, Monitor monitorIn) {

id = i;

this.passengerMon = monitorIn;

}

public void run() {

for(int i=0; i<1; i++){ // Каждый пассажир ездит всего одну поездку.

try{

Thread.sleep((int)( Math.random()\*2000));

}catch(InterruptedException e){

}

passengerMon.tryToGetOnCar(id);

}

}

}

class Car implements Runnable {

private int id;

private Monitor carMon;

public Car(int i, Monitor monitorIn) {

id = i;

this.carMon = monitorIn;

}

public void run() {

while(true) {

carMon.passengerGetOn(id);

try{

Thread.sleep((int)(Math.random()\*2000));

}catch(InterruptedException e){

}

carMon.passengerGetOff(id);

}

}

}

class Monitor {

private int i, line\_length; // Количество пассажиров, ожидающих посадки

private int seats\_available = 0;

boolean coaster\_loading\_passengers = false;

boolean passengers\_riding = true;

private Object notifyPassenger = new Object(); // ввод / выход протокола обеспечивает взаимное исключение.

private Object notifyCar = new Object();

public void tryToGetOnCar(int i) {

synchronized (notifyPassenger) {

while (!seatAvailable()) {

try {

notifyPassenger.wait(); // Сообщите пассажиру, чтобы он подождал

} catch (InterruptedException e){}

}

}

System.out.println("Пассажир "+ i + " садится");

synchronized (notifyCar) {notifyCar.notify();}

}

private synchronized boolean seatAvailable() {

// Проверьте, доступно ли место для пассажира, который пытается попасть.

if ((seats\_available > 0)

&& (seats\_available <= RollerCoaster.SEAT\_AVAIL)

&& (!passengers\_riding)) {

seats\_available--;

return true;

} else return false;

}

public void passengerGetOn(int i) {

synchronized (notifyCar) {

while (!carIsRunning()) {

try {

notifyCar.wait();

} catch (InterruptedException e){}

}

}

System.out.println("Тележка заполнена");

synchronized(notifyPassenger) {notifyPassenger.notifyAll();}

}

private synchronized boolean carIsRunning() {

// Проверьте, работает ли автомобиль

if (seats\_available == 0) {

// Если нет места, автомобиль начинает работать и сбрасывает параметры.

seats\_available = RollerCoaster.SEAT\_AVAIL;

// Сбросить доступность места для следующей поездки

coaster\_loading\_passengers = true; // Индикатор автомобиля работает.

passengers\_riding = true; //пассажиры едут в машине.

return true;

} else return false;

}

public void passengerGetOff(int i) {

synchronized (this) {

// Параметры сброса

passengers\_riding = false;

coaster\_loading\_passengers = false;

}

synchronized(notifyPassenger) {notifyPassenger.notifyAll();}

}

}