**Лабораторная работа №6: Потоки**

**Лотарев Сергей**

**Постановка задачи.**

**1) Вычисление с синхронизацией.**  
Главный поток передаёт двум рабочим потокам значение n. Один из них вычисляет n! (числитель), второй — сумму Σ(n + i), где i = 0..n (знаменатель). Затем главный поток делит эти два значения и получает результат. После этого оба потока продолжают работу, поочерёдно выводя символы A и B n раз.

**2) Самокаты.**  
На стоянке изначально 10 самокатов. Каждые 4 часа происходит пополнение (в симуляции — 24 секунды), а каждые 10 минут приходит пользователь и берёт самокат (в симуляции — 1 секунда). Необходимо смоделировать поведение системы и определить, с какой частотой нужно пополнять стоянку, чтобы она не пустовала.

**Ход работы**

Задача 1:

Смоделирован многопоточный процесс, где:

* Поток A вычисляет факториал n! и выводит символ A n раз
* Поток B вычисляет сумму Σ(n + i) и выводит B n раз
* Потоки чередуются при выводе, используя wait/notifyAll

Класс SharedData содержит общее состояние для обмена между потоками: значение n, числитель (numerator), знаменатель (denominator), а также булевый флаг turnA, определяющий, какой поток в текущий момент должен вывести символ — A или B. Данный класс используется как синхронизируемый ресурс, на котором потоки вызывают wait и notifyAll для поочерёдного взаимодействия при печати символов.

public class SharedData {

public final int n;

public long numerator = 1;

public long denominator = 0;

public boolean turnA = true;

public SharedData(int n) {

this.n = n;

}

}

Класс NumeratorWorker реализует интерфейс Runnable и отвечает за вычисление факториала n!. После завершения вычислений поток входит в цикл, где поочерёдно печатает символ A n раз. Печать выполняется в синхронизированном блоке по объекту SharedData. Поток ожидает своей очереди с помощью конструкции while (!turnA) и метода wait. После вывода символа A он изменяет значение turnA на false и вызывает notifyAll, передавая управление потоку, печатающему символ B.

public class NumeratorWorker implements Runnable {

private final SharedData data;

public NumeratorWorker(SharedData data) {

this.data = data;

}

public void run() {

data.numerator = 1;

for (int i = 2; i <= data.n; i++) {

data.numerator \*= i;

}

for (int i = 0; i < data.n; i++) {

synchronized (data) {

while (!data.turnA) {

try { data.wait(); } catch (InterruptedException e) {}

}

System.out.println("A");

data.turnA = false;

data.notifyAll();

}

}

}

}

Класс DenominatorWorker также реализует интерфейс Runnable и вычисляет сумму Σ(n + i), где i меняется от 0 до n. После завершения вычислений поток переходит в цикл печати символа B n раз. Как и в предыдущем потоке, печать происходит в синхронизированном блоке с проверкой условия очередности. Если turnA установлен в true, поток вызывает wait и ожидает своей очереди. После вывода символа B устанавливает turnA в true и вызывает notifyAll, пробуждая поток A.

public class DenominatorWorker implements Runnable {

private final SharedData data;

public DenominatorWorker(SharedData data) {

this.data = data;

}

public void run() {

for (int i = 0; i <= data.n; i++) {

data.denominator += data.n + i;

}

for (int i = 0; i < data.n; i++) {

synchronized (data) {

while (data.turnA) {

try { data.wait(); } catch (InterruptedException e) {}

}

System.out.println("B");

data.turnA = true;

data.notifyAll();

}

}

}

}

Главный класс программы в методе main задаёт значение n и создаёт общий объект SharedData. Создаются и запускаются два потока — один для вычисления числителя и печати A, другой — для вычисления знаменателя и печати B. После запуска потоков вызывается join для ожидания их завершения. Когда оба потока завершают работу, в консоль выводится результат деления n! на сумму Σ(n + i), вычисленный на основе значений, сохранённых в объекте SharedData.

public class Main {

public static void main(String[] args) throws InterruptedException {

int n = 5;

SharedData data = new SharedData(n);

Thread t1 = new Thread(new NumeratorWorker(data));

Thread t2 = new Thread(new DenominatorWorker(data));

t1.start();

t2.start();

t1.join();

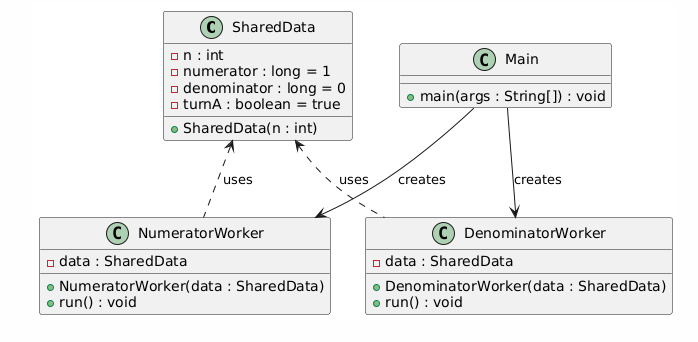
t2.join();

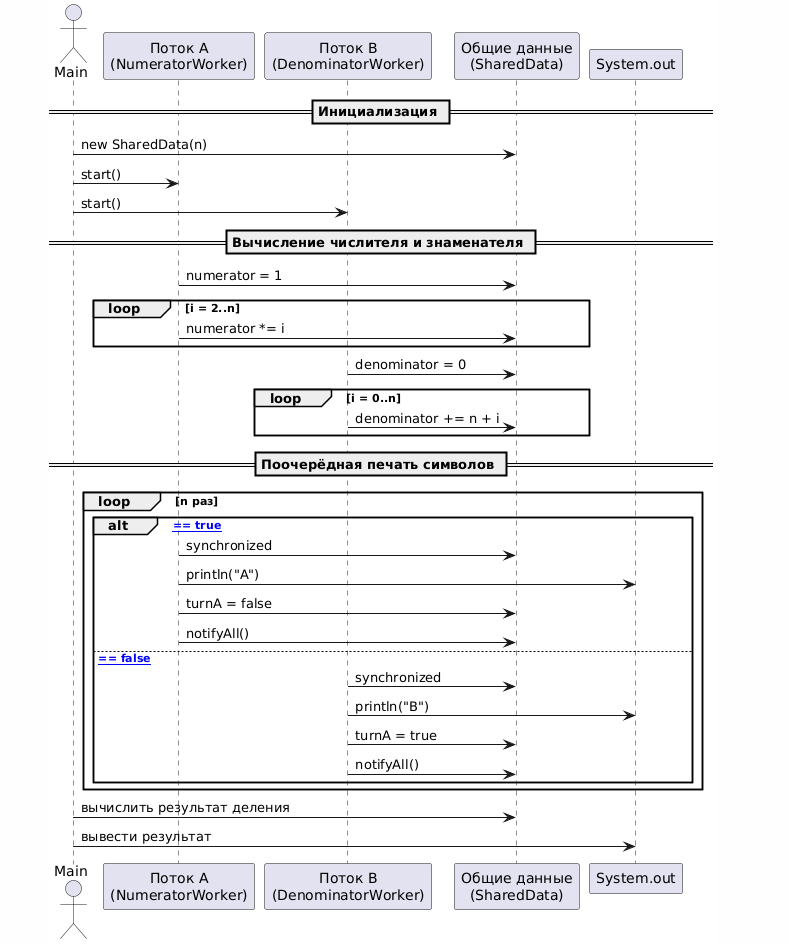
System.out.println("Результат деления: " + data.numerator + " / " + data.denominator +

" = " + ((double) data.numerator / data.denominator));

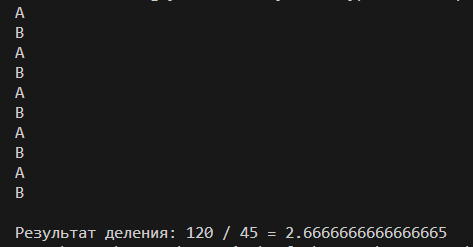
}

}

Диаграмма классов имеет вид:  


Действия, выполняемые в процессе выполнения каждого из потоков можно представить в виде диаграммы последовательности:  


Результат выполнения программы:



Задача 2: самокаты

Смоделирована стоянка с 10 самокатами. Пользователи приходят каждую секунду, пополнение — раз в 6 секунд. При нехватке самокатов пользователи уходят. Цель — найти минимальный безопасный интервал пополнения.

Класс ScooterStation cодержит текущее состояние стоянки самокатов. Переменная scooters определяет количество доступных самокатов. Константа capacity задаёт максимальное количество. Метод takeScooter вызывается пользовательским потоком и уменьшает счётчик, если самокаты есть. Если самокатов нет, увеличивается счётчик отказов refused. Метод refill вызывается пополняющим потоком и восстанавливает количество самокатов до максимума. Методы синхронизированы, так как вызываются из разных потоков.

public class ScooterStation {

private int scooters = 10;

private final int capacity = 10;

private int refused = 0;

public synchronized boolean takeScooter() {

if (scooters > 0) {

scooters--;

System.out.println("Пользователь взял самокат. Осталось: " + scooters);

return true;

} else {

refused++;

System.out.println("Самокатов нет! Пользователь ушел без самоката :(");

return false;

}

}

public synchronized void refill() {

int added = capacity - scooters;

scooters = capacity;

System.out.println("Пополнение стоянки. Добавлено: " + added + ", всего: " + scooters);

}

public synchronized int getRefusedCount() {

return refused;

}

}

Поток UserThread является наследником класса Thread. В методе run реализован цикл, в котором поток каждую секунду вызывает метод takeScooter у объекта ScooterStation. Таким образом моделируется пользователь, приходящий каждые 10 минут. Цикл продолжается до тех пор, пока не будет вызван метод stopRunning, устанавливающий флаг завершения.

public class UserThread extends Thread {

private final ScooterStation station;

private final long interval;

private volatile boolean running = true;

public UserThread(ScooterStation station, long interval) {

this.station = station;

this.interval = interval;

}

public void run() {

while (running) {

station.takeScooter();

try { Thread.sleep(interval); } catch (InterruptedException e) {}

}

}

public void stopRunning() { running = false; }

}

Поток RefillThread также наследует класс Thread. В методе run реализован бесконечный цикл, в котором поток раз в заданный интервал вызывает метод refill у объекта ScooterStation. Это моделирует регулярное пополнение самокатов. Цикл также завершает работу при установке флага через метод stopRunning.

public class RefillThread extends Thread {

private final ScooterStation station;

private final long interval;

private volatile boolean running = true;

public RefillThread(ScooterStation station, long interval) {

this.station = station;

this.interval = interval;

}

public void run() {

while (running) {

try { Thread.sleep(interval); } catch (InterruptedException e) {}

station.refill();

}

}

public void stopRunning() { running = false; }

}

Главный класс создаёт объект ScooterStation, два потока — UserThread и RefillThread — и запускает их. Интервалы задаются в миллисекундах: пользователь приходит каждые 1000 мс, пополнение — каждые 24000 мс. Главный поток спит 60 секунд, после чего завершает оба потока с помощью stopRunning и join. Затем выводится общее количество пользователей, которые не получили самокат.

public class Main {

public static void main(String[] args) throws InterruptedException {

ScooterStation station = new ScooterStation();

long userInterval = 1000L; // 1 сек = 10 минут

long refillInterval = 24000L; // 24 сек = 4 часа

UserThread user = new UserThread(station, userInterval);

RefillThread refill = new RefillThread(station, refillInterval);

user.start();

refill.start();

Thread.sleep(60000); // 1 минута симуляции

user.stopRunning();

refill.stopRunning();

user.join();

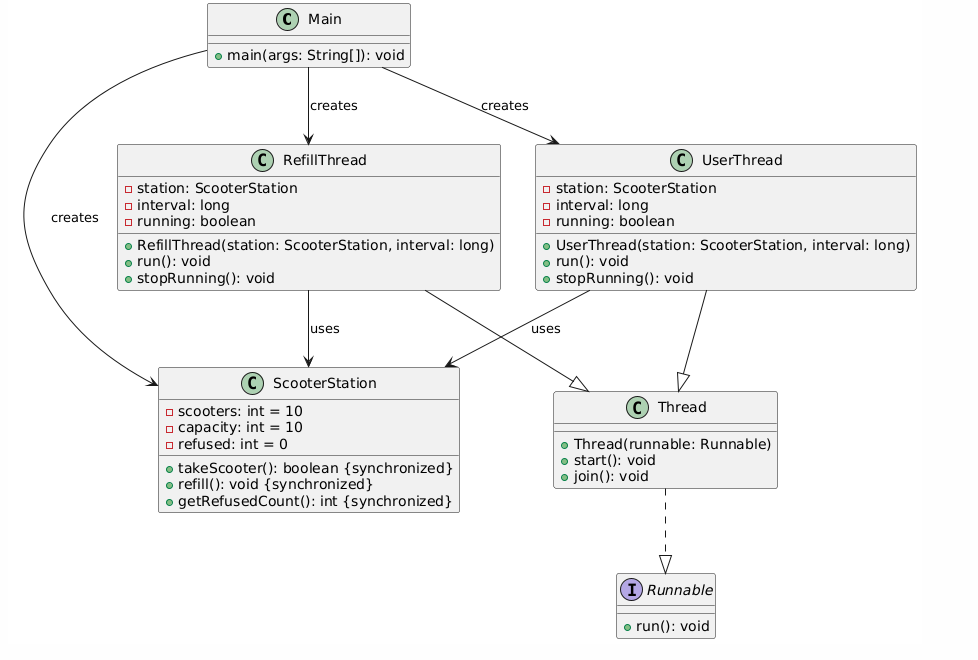
refill.join();

System.out.println("Пользователи, не получившие самокат: " + station.getRefusedCount());

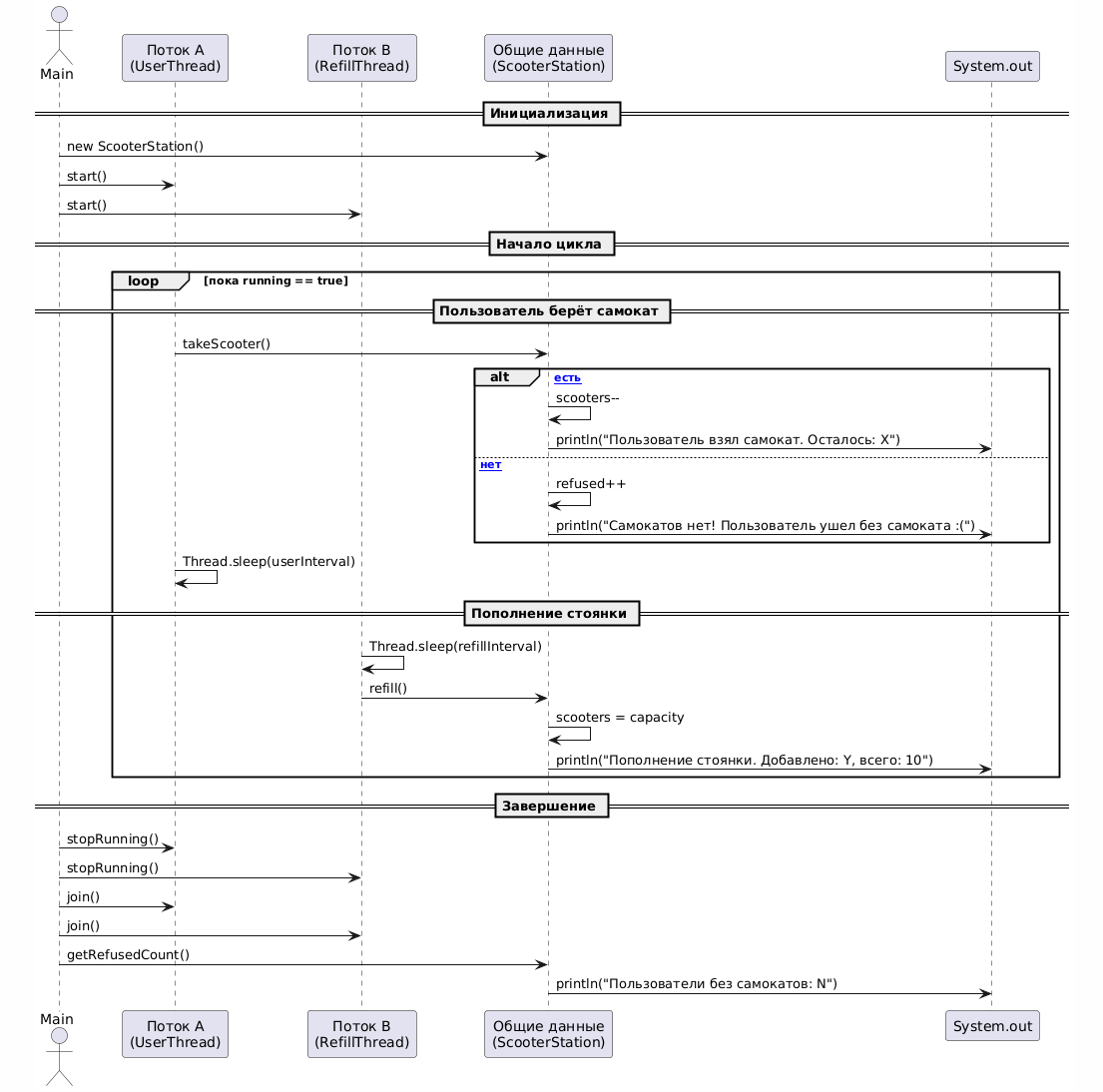
}

}

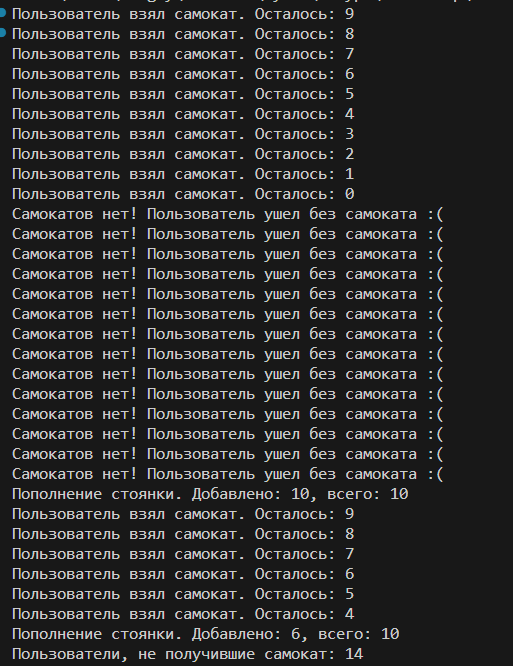
Диаграмма классов программы имеет вид:



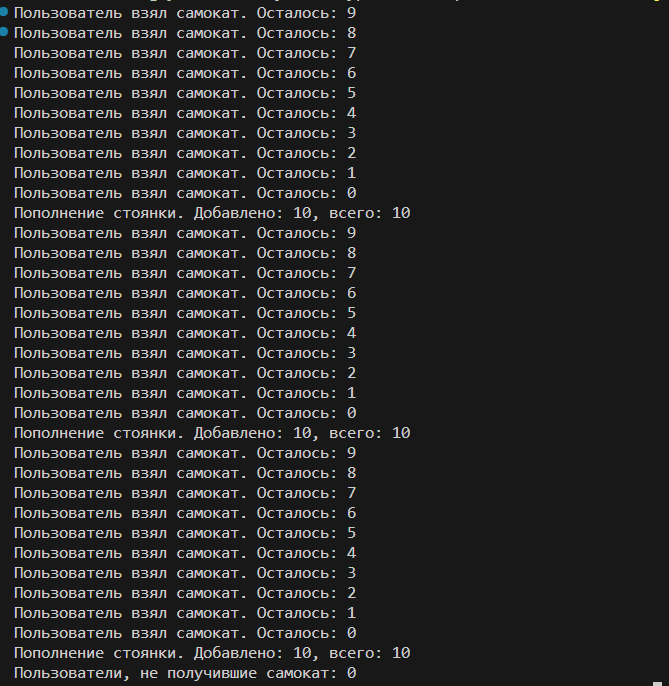
Процесс выполнения программы выражается следующей диаграммой последовательности:



Результат выполнения программы при пополнении станции раз в 4 часа (24 секунды в симуляции)



Можно увидеть, что при таком интервале пополнения самокатов много пользователей остаются без самокатов. Вот результат выполнения симуляции при пополнении каждые 100 минут вместо 4 часов:



Таким образом, найдено оптимальное время для пополнения стоянки, чтобы она не пустовала – каждые 100 минут.