МНОГОПОТОЧНОСТЬ В JAVA

1. ЦЕЛЬ РАБОТЫ

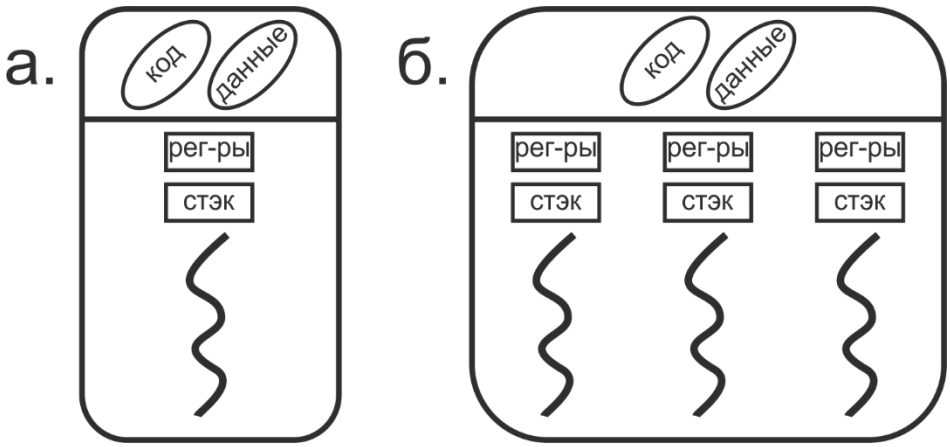
Изучить основные приемы работы с потоками в Java и получить практические навыки написания многопоточных Java-приложений.

2. КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

2.1 Основные понятия

Процесс — это совокупность кода и данных, разделяющих общее виртуальное адресное пространство. Чаще всего одна программа состоит из одного процесса, но бывают и исключения (например, браузер Chrome создает отдельный процесс для каждой вкладки, что дает ему некоторые преимущества, вроде независимости вкладок друг от друга). Процессы изолированы друг от друга, поэтому прямой доступ к памяти чужого процесса невозможен (взаимодействие между процессами осуществляется с помощью специальных средств). На рисунке 1(а) изображена общая структура процесса с одним потоком.

Поток («нить», «трэд») – это наименьшая единица обработки, исполнение которой может быть назначено планировщиком. Реализация потоков выполнения и процессов на разных платформах отличается друг от друга, но потоки всегда находятся внутри своего процесса. В соответствии с рисунком 1(б) несколько потоков выполнения разделяют общие ресурсы процесса в контексте общего виртуального адресного пространства. Также есть и данные, ограниченные контекстом потока – значения регистров, стэк.



а – простой процесс; б – процесс с несколькими потоками

Рисунок 1 – Структурная схема однопоточного и многопоточного процессов

Подразумевается, что несколько потоков могут работать параллельно, но как известно, одно ядро процессора за каждый такт может выполнить не более одной команды. То есть одноядерный процессор может обрабатывать команды только последовательно. Однако, запуск нескольких параллельных потоков возможен и в одноядерных системах. В этом случае система будет периодически переключаться между потоками, поочередно давая выполняться то одному, то другому потоку. Упрощенная схема этого процесса изображена на рисунке 2.

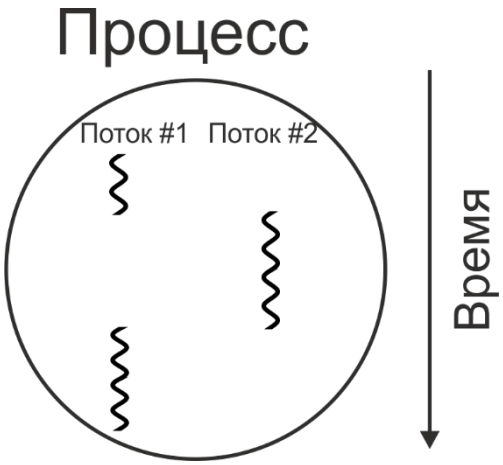


Рисунок 2 – Процесс с двумя потоками выполнения на одном процессоре

Как известно, Java-приложения выполняются на JVM, и с этим связана особенность Java-потоков – они не обязательно (по спецификации) эквивалентны потокам ОС, а поэтому Java-поток даже в состоянии RUNNING может фактически не выполняться на процессоре, так как процесс самой JVM ожидает получения доступа к процессору.

Когда запускается Java-приложение, оно создает поток, который называется *главным* (или *основным*). Также приложение может создавать несколько потоков, причем многие программы так и делают, например, приложения с графическим интерфейсом создают как минимум два потока.

Рассматривая модель многопоточной программы, следует заметить, что по ходу исполнения могут возникать ситуации, в которых один поток изменил состояние разделяемого (общего) ресурса, а другой поток все еще «наблюдает» предыдущее состояние этого объекта или даже частично измененное (некорректное) состояние (в случае если операция записи не атомарная). Поэтому, в Java существуют специальные механизмы для синхронизации потоков, которые будут рассмотрены далее.

2.2 Создание потока

В языке Java поток представляется в виде объекта класса Thread. Этот класс реализует стандартные механизмы работы с потоком. Следующая программа создает четыре потока, кроме главного:

```
import java.util.*;

public class Thread01 {
    public static void main(String[] args) {
        new Worker2();
        (new Thread(new Worker1())).start();
        (new Thread(new Runnable() {
```

```
            public void run() { ToDo.work(2, 5); }
        })).start();
        (new Thread(() -> ToDo.work(2, 5))).start();
        ToDo.work(5, 10);
    }
}

class ToDo {
    private static Random rand = new  Random();

    public static void work(int a, int  b) {
        print("started");
        for (int i = 0, n = rand.nextInt(b – a) + a; i < n; ++i) {
            print("working.");
            work();
        }
        print("terminated");
    }

    private static void print(String text) {
        long id = Thread.currentThread().getId();
        System.out.println("[" + id + "] " +  text);
    }

    private static void work() {
        double y;
        for (int i = 0; i < 1000000L; ++i) {
            y = Math.cos(Math.sqrt(rand.nextDouble()));
        }
    }
}
```

Рассмотрим класс *ToDo*, который в принципе ничего не делает с потоками. Он представляет некоторую задачу, которую выполняет компьютер, причем эта операция достаточно, следовательно, занимает процессор на некоторое время. Приватный метод *work()* выполняет работу, представляющую собой миллион вычислений, для которой используется много процессорного времени. Публичный же метод *work()* имеет два параметра и вызывает приватный метод несколько раз, в зависимости от переданных параметров. Метод выводит сообщения перед началом выполнения основной операции и после ее окончания, а также сообщение о выполнении работы в каждой итерации цикла.

```
long id = Thread.currentThread().getId(); // получение номера потока
```

Каждый поток идентифицируется по ID, который представляет собой порядковый номер, назначаемый потоку при создании. Главный поток имеет номер 1, а остальные потоки получают номера начиная с 10. Когда класс печатает какое-то сообщение, то он также выводит идентификатор потока, в котором он выполняется.

В Java потоки представлены классом *Thread*. Класс *Worker2* наследует этот класс и поэтому является потоком. Класс *Thread* имеет метод *run()*, который вызывается когда поток начинает работу, и в данном случае это происходит в конструкторе класса *Worker2*. Чтобы задать программу, которая будет выполняться потоком, программист должен переопределить метод *run()*. В случае потока класса *Worker2*, в процессе работы он просто вызывает метод *ToDo#work(…)*.

```
//Объявление класса потока
class Worker2 extends Thread {
    public Worker2() {
        start();
    }

    public void run() {
        ToDo.work(2, 5);
    }
}

//...
    public static void main(String[] args) {
        //...
        //запуск потока класса Worker2, (start вызывается в
конструкторе)
        new Worker2();
        //...
    }
```

Класс *Thread* реализует интерфейс Runnable, который определяет метод *run()*. Другим способом определения потока является передача объекта *Runnable* в качестве параметра в конструктор базовому классу *Thread*. Класс *Worker1* реализует интерфейс *Runnable* и таким образом реализация метода *run()* не отличается от класса *Worker2*.

```
class Worker1 implements Runnable {
    public void run() {
        ToDo.work(2, 5);
    }
}

//...

    public static void main(String[] args) {
        //...
        (new Thread(new Worker1())).start();
        //...
    }
```

В основной программе также можно увидеть способы создание потока путем передачи в конструктор класса *Thread* анонимного объекта, реализующего интерфейс *Runnable*, а также путем передачи лямбда-выражения с сигнатурой, соответствующей методу *run()*. Последним выражением метода *main(…)* является вызов метода *ToDo.work(…)*, и это означает, что после запуска четырех дополнительных потоков, главный поток продолжает работу в параллели с ними. Таким образом после запуска данной программы, будет создано 5 параллельных потоков.

```
...

    public static void main(String[] args) {
        //...
        (new Thread(new Runnable() {
            public void run() { ToDo.work(2, 5); }
        })).start();
        (new Thread(() -> ToDo.work(2, 5))).start();
        ToDo.work(5, 10);
    }
```

2.3 Потоки-демоны

Кроме метода *run()* класс *Thread* также имеет различные методы для получения или установки свойств потоков, таких как, приоритет, имя, флаг потока-демона и т.д. Каждый поток находится в определенном

состоянии, которое можно получить с помощью метода *getState()*. Список возможных состояний:

- NEW, поток создан, но еще не запущен
- RUNNING, поток выполняется в виртуальной машине Java
- BLOCKED, поток ожидает блокировки монитора для входа в синхронизированный блок/метод
- WAITING, поток ожидает уведомления от другого потока
- TIMED_WAITING, поток ожидает истечения установленного времени или уведомления от другого потока
- TERMINATED, поток завершил выполнение

В Java потоки могут быть основными (*none-daemon*) или специальными потоками-демонами (*daemon*), это свойство проверяется и устанавливается с помощью методов *isDaemon()* и *setDaemon(boolean)* соответственно. Потоки-демоны рассматриваются как потоки-помощники для основных потоков, и их основное отличие в том, что они автоматически завершаются в момент, когда завершается работа последнего основного потока программы. То есть программа не должна ожидать завершения всех потоков-демонов.

2.4 Приоритеты потоков

Потоки могут работать с разными приоритетами. Поток с более высоким приоритетом имеет больше шансов на выполнение в конкретный момент времени, чем другие потоки. Для получения и установки приоритета используются методы *getPriority()* и *setPriority(int)* соответственно.

```
Thread t = new Thread(…);
t.setPriority(Thread.NORM_PRIORITY + 1);
t.setPriority(t.getPriority() - 1);
```

2.5 Режим ожидания

Класс *Thread* имеет метод *sleep(…)*, который предназначен для временного прерывания выполнения текущего потока (переход в состояние TIMED_WAITING) на определенное количество времени. Если в этот период времени какой-то другой поток вызовет *Thread.interrupt()* для данного потока, то будет выброшено исключение *InterruptedException*. По истечении времени «сна» потока, он продолжит работу. Этот режим ожидания относится к ожиданию с истекающим сроком (with timeout), он используется также при вызове методов *Object.wait(…)* и *Thread.join(…)* с параметрами *timeout*.

Метод *wait(…)* класса *Object* позволяет текущему потоку ожидать пока какой-нибудь другой поток не вызовет *notify()* или *notifyAll()* на этом же объекте или пока не истечет заданное время (timeout).

Также существует метод *join(…)* класса *Thread*, вызов которого на объекте какого-то потока, приведет к тому, что текущий поток будет ожидать завершения этого потока или истечения заданного времени (timout).

Методы *Object.wait()* и *Thread.join()* могут быть вызваны без параметров *timeout*, тогда время ожидания будет не ограничено и поток продолжит выполнение только после срабатывания событий, соответствующих логике этих методов. В этом случае, в отличие от рассмотренных ранее, поток будет находиться в состоянии WAITING.

```
public class Thread02 {
    private static long prime;
    public static void main(String[] args) {
        Thread th;
        (th = new Thread(() -> nextPrime(10000000))).start();
```

```
        System.out.println(prime);
    }

    private static void nextPrime(long t) {
        if (t < 2) {
            prime = 2;
        } else {
            for (prime = t % 2 == 0 ? t + 1 : t; !isPrime(prime); prime
                                        += 2);
        }
    }

    private static boolean isPrime(long t) {
        if (t == 2 || t == 3 || t == 5 || t == 7) return true;
        if (t < 11 || t % 2 == 0) return false;
        for (long k = 3, m = (long)Math.sqrt(t) + 1; k <= m; k += 2)
            if (t % k == 0) return false;
        return true;
    }
}
```

В данном примере объявлен метод *nextPrime()*, который устанавливает в поле prime значение ближайшего простого числа, большего, чем переданное в качестве параметра. Если запустить этот код, то можно увидеть, что вывод *System.out.println(prime)* будет содержать неправильное число. Это происходит потому, что не основной поток *th* не успевает произвести вычисления до того, как основной поток выполнит оператор вывода значения на экран. То есть, основной поток должен подождать, пока вычисления побочного потока будут завершены, и только тогда можно будет вывести на экран результат.

```
public static void main(String[] args) {
    Thread th;
    (th = new Thread(() -> nextPrime(10000000))).start();
    try{
        th.join();
    } catch (Exception ex) {}
        System.out.println(prime);
    }
}
```

Данные код реализует именно такое поведение. Теперь, прежде чем

вывести на экран результат работы, основной поток дожидается установки побочным потоком в поле *prime* правильного значения.

2.6 Ключевое слово volatile

В Java ключевое слово *volatile* используется для того, чтобы обозначить переменную как “хранящуюся в основной памяти”. Более точно это означает, что каждой чтение *volatile* переменной будет происходит из оперативной памяти, а не из кэша процессора, и каждая запись тоже будет произведена сразу в память, а не просто в кэш процессора.

В многопоточных приложениях, в которых потоки оперируют не-*volatile* переменными, каждый поток может копировать переменную из памяти в кэш процессора для работы с ней в целях повышения производительности. Если приложение работает на многопроцессорной системе, то каждый поток может выполняться на разных процессорах. Это значит, что каждый поток может копировать переменную в кэш своего процессора. Эта ситуация изображена на рисунке 3.

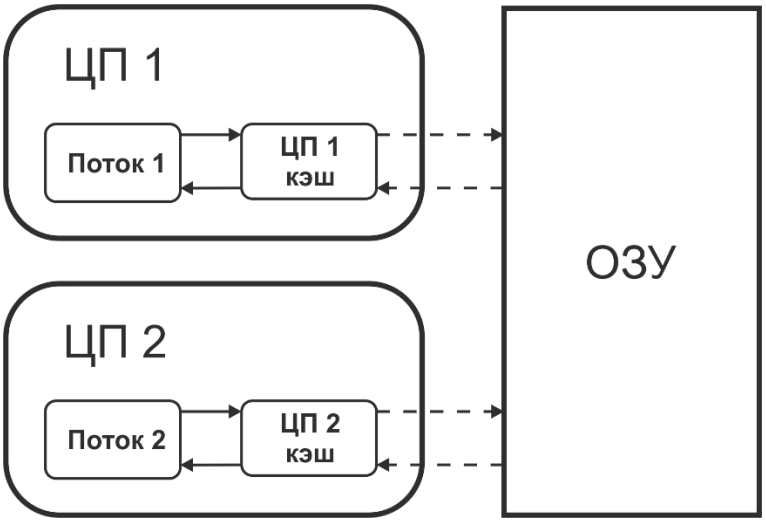


Рисунок 3 – Работа с не-*volatile* переменной

Рассмотрим ситуацию, когда только Поток 1 инкрементирует значение переменной *counter*, но периодически читать значение могут оба потока Поток 1 и Поток 2.

```
public class SharedObject {
    public int counter = 0;
}
```

Если *counter* не объявлена как *volatile*, то нет гарантий в какой момент времени значение переменной из кэша ЦП 1 запишется обратно в память, поэтому значение в кэше может отличаться от значения в основной памяти. В Java *volatile* гарантирует видимость изменений переменной для всех потоков, то есть разрешает чтение одной переменной из нескольких потоков. Объявление *volatile* переменной выглядит следующим образом:

```
public class SharedObject {
    public volatile int counter = 0;
}
```

Тем не менее, если оба Поток 1 и Поток 2 инкрементируют переменную *counter*, то объявления *volatile* не достаточно. В этом случае можно столкнуться с проблемой под названием «Состояние гонки», когда итоговое значение переменной зависит от порядка выполнения инструкций программы. Изменять, то есть записывать переменную из нескольких потоков нужно используя синхронизацию.

2.7 Синхронизация потоков

В некоторых случаях множество потоков может работать независимо и не использовать совместно какие-то объекты, но в большинстве случаев потокам требуются некоторые ограниченные ресурсы, и они должны разделять их, например, подключение к базе

данных, файлы и др. Также могут использоваться общие объекты, обеспечивающие согласованность данных в распределенной системе, например, счетчики.

Участки кода, в котором производится доступ к разделяемым ресурсам, называются критическими секциями (*critical section*). При этом для осуществления синхронизации между потоками используются мьютексы. Мьютекс (от англ. «mutex», «mutual exclusion» — «взаимное исключение») – аналог одноместного семафора, который обеспечивает доступ к ресурсу одновременно только одному потоку, таким образом, что если поток А захватил мьютекс (отметил ресурс занятым), то остальные потоки будут ждать освобождения мьютекса для его захвата и входа в критическую секцию.

В Java к каждому объекту прикреплен мьютекс, но у программиста нет прямого к нему. Работа с мьютексом происходит посредством монитора. Монитор – специальная надстройка над мьютексом, которая обеспечивает правильную работу с ним. Таким образом, JVM обеспечивает контроль за тем, чтобы потоки, не имеющие монитора не воспользовались общим объектом.

Для синхронизации потоков в Java существует ключевое слово synchronized. Весь код, находящийся в блоке synchronized рассматривается как неделимый и по сути является критической секцией.

```
synchronized (lock) {
    //statements
}
```

Синхронизированными также могут быть методы классов. Если метод не статический, то блокировка происходит на уровне объекта, то есть следующие два блока функционально эквивалентны (хотя компилятор генерирует для них различный байткод).

```
//Синхронизированный метод
```

```
public synchronized void someMethod() {
    //stuff
}

//Синхронизированный по this блок
public void someMethod() {
    synchronized (this) {
        //stuff
    }
}
```

В случае статического метода синхронизация будет произведена по объекту самого класса.

```
class ExampleClass {

    public static synchronized void someClassMethod() {
        //stuff
    }


    public static void normalMethod() {
        synchronized(ExampleClass.class) {
            //stuff
        }
    }

}
```

Далее приведена программа, в которой допущена ошибка. Несколько потоков в процессе работы инициализируют по 5 воображаемых объектов, каждый из которых должен иметь уникальный для потока номер.

```
class Program {
    public static int counter;
    public static void main(String[] args) {
        for (int i = 1; i < 6; i++) {
            Thread t = new Thread(new FactoryThread());
            t.setName("Thread "+ i);
            t.start();
        }
    }
}
class FactoryThread implements Runnable {
    public void run() {
        Program.counter = 1;
        for (int i = 1; i < 5; i++) {
            System.out.printf("%s New object withid: %d \n",
Thread.currentThread().getName(), Program.counter);
```

```
            Program.counter++;
            try{
                Thread.sleep(100);
            } catch(InterruptedException e){}
        }
    }
}
```

Результат работы программы представлен далее. По логике работы программы, максимальный id для каждого потока должен быть равен 4, но фактически счетчик инкрементируется из нескольких потоков неправильно и превышает это значение.

```
Thread 1 New object withid: 1
Thread 3 New object withid: 1
Thread 2 New object withid: 1
Thread 1 New object withid: 4
Thread 2 New object withid: 5
Thread 3 New object withid: 5
Thread 2 New object withid: 7
Thread 3 New object withid: 8
Thread 1 New object withid: 9
Thread 2 New object withid: 10
Thread 3 New object withid: 11
Thread 1 New object withid: 12
```

Для исправления данной проблемы необходимо синхронизировать код метода для общего использования счетчика. Далее представлен код, решающий обозначенную проблему.

```
class Program {
    public static int counter;
    public static final Object lock = new Object();
    public static void main(String[] args) {
        for (int i = 1; i < 4; i++){
            Thread t = new Thread(new FactoryThread());
            t.setName("Thread "+ i);
            t.start();
        }
    }
}
class FactoryThread implements Runnable {
    public void run() {
        synchronized (Program.lock) {
            Program.counter = 1;
            for (int i = 1; i < 5; i++) {
                System.out.printf("%s New object withid: %d \n",
Thread.currentThread().getName(), Program.counter);
```

```
                Program.counter++;
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {}
            }
        }
    }
}
```

3. ВАРИАНТЫ ЗАДАНИЙ И ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. В главном классе реализовать статический метод сортировки массива (направление и метод сортировки по варианту) и вывода результата сортировки на экран с помощью вызова *System.out.println(…)*, формат вывода: “имя_текущего_потока: [отсортированный_массив]”. Сигнатура функции сортировки *printSorted(int[] array)*.
2. В методе main реализовать цикл генерации 3 массивов случайных чисел длины N и для каждого сгенерированного массива вызывать метод сортировки в новом потоке. Установить для потоков приоритеты в соответствии с вариантом и после этого запустить потоки в произвольном порядке. Запустить программу несколько раз и пронаблюдать за соблюдением приоритетов.
3. В методе сортировки заменить вывод всего массива на поэлементный вывод с помощью вызова *System.out.print(…)*, причем массив должен выводиться по 100 элементов в строке и с двух сторон отделяться строками, состоящими из символов «*». Перед строками содержимого массива выводить имя текущего потока. Проверить корректность вывода при запуске этой программы.

4. Реализовать синхронизацию потоков в момент записи значений в поток вывод. Запустить исправленную программу и проверить корректность вывода.

Таблица 3.1 – Варианты заданий

№ Варианта	Метод сортировки	Направление сортировки	Приоритеты потоков	Размер массива N
1	Пузырька	по-убыванию	1-8 2-5 3-1	10000
2	Выбора	по-возрастанию	1-6 2-4 3-1	20000
3	Вставки	по-убыванию	1-5 2-7 3-2	15000
4	Шелла	по-возрастанию	1-3 2-5 3-7	12000

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое поток?
2. Что такое главный поток в Java?
3. Приведите примеры служебной операции JVM, выполняющейся в отдельном потоке.
4. Чем отличается состояние сна от состояния паузы потока?
5. Объясните, в каких случаях поток генерирует исключение InterruptedException.
6. Для чего нужен метод yield?
7. Объясните назначение метода Thread.join
8. Для чего нужны ключевые слова volatile и synchronized?
9. Объясните механизм замков в Java.
10. Почему применение ключевого слова volatile бесполезно для объектных типов данных?
11. Что выполняет роль замка для статических методов, помеченных ключевым словом synchronized?

СПИСОК ЛИТЕРАТУРЫ

1. Полное руководство по Java 8 Stream // Java Dev Blog URL: http://javadevblog.com/polnoe-rukovodstvo-po-java-8-stream.html (дата обращения: 18.03.2018).
2. Шпаргалка Java программиста. Streams // Хабрахабр URL: https://habrahabr.ru/company/luxoft/blog/270383/ (дата обращения: 18.03.2018).
3. Документация по пакету java.lang. // Официальный сайт Oracle URL: https://docs.oracle.com/javase/8/docs/api/java/lang/package-summary.html (дата обращения: 9.10.2019).
4. Документация по пакету java.util.stream // Официальный сайт Oracle URL: https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html (дата обращения: 18.03.2018).
5. Особенности Java 8. Максимальное руководство // Javarush URL: http://info.javarush.ru/translation/2014/10/09/Особенности-Java-8-максимальное-руководство-часть-2-.html (дата обращения: 18.03.2018).
6.