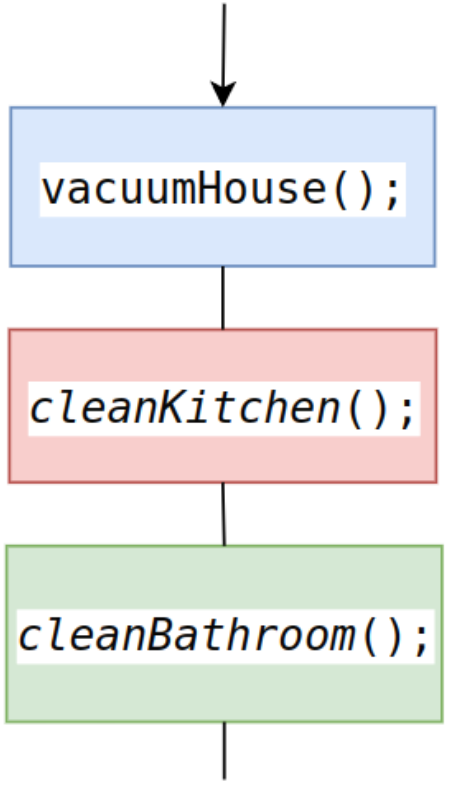
Основы многопоточных программ в Java

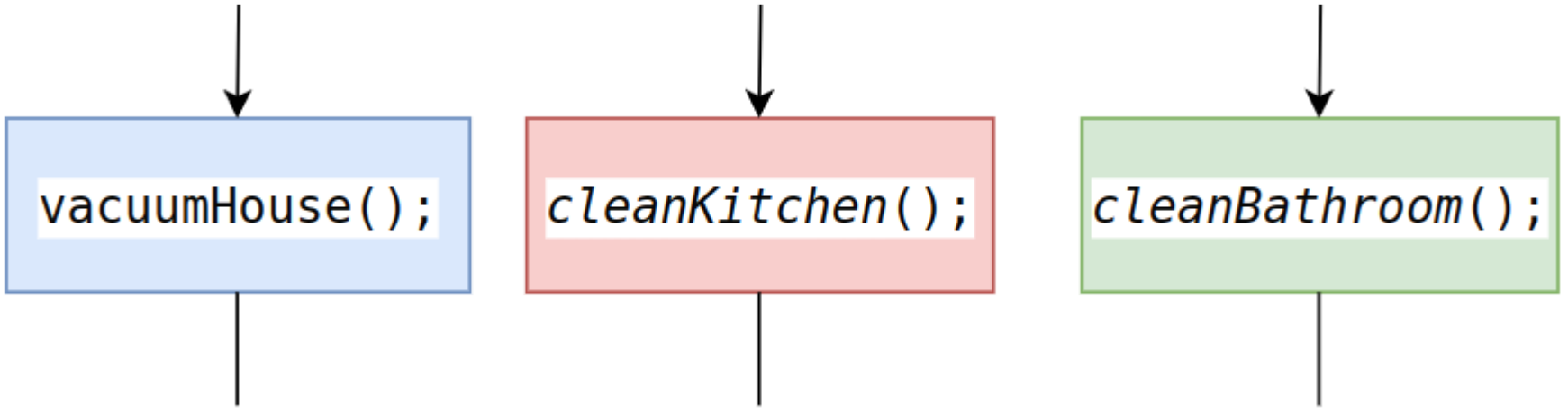
Сидоров Кирилл

Пример последовательного выполнения задачи




```java
// Уборка дома
public class HouseClean {
    public static void main(String[] args) {

        // Начало уборки
        vacuumHouse();
        cleanKitchen();
        cleanBathroom();
        // Конец уборки
    }
    private static void vacuumHouse() {
        // Уборка пылесосом
    }
    private static void cleanKitchen() {
        // Уборка кухни
    }
    private static void cleanBathroom() {
        // Уборка ванной комнаты
    }
}
```

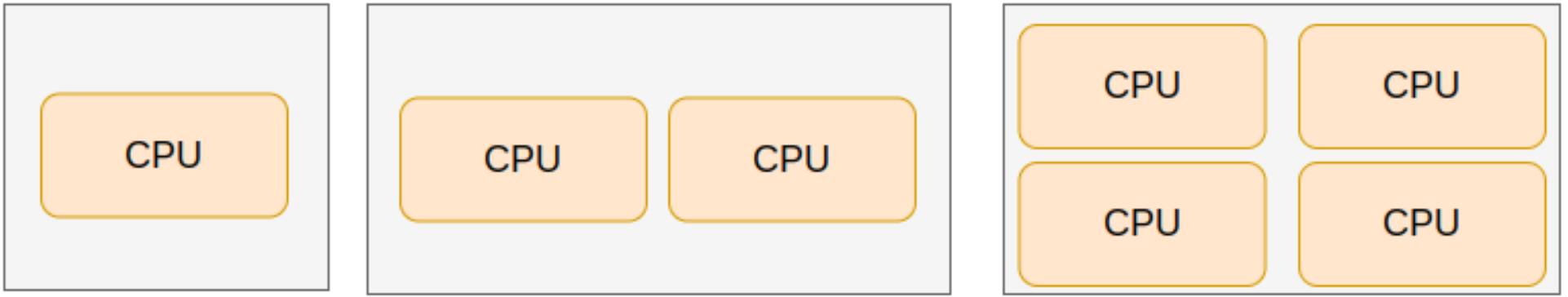
Пример параллельного выполнения задачи



Пример многоядерных процессоров

CPU - Central Processing Unit

В этом примере CPU - это ядро процессора



Определения

- **Программа** (program) – это последовательность команд, реализующая алгоритм решения задачи. Программа может быть записана на языке программирования (например, на Pascal, C++, BASIC); в этом случае она хранится на диске в виде текстового файла с расширением, соответствующим языку программирования (например, .PAS, .CPP, .VB). Также программа может быть представлена при помощи машинных команд; тогда она хранится на диске в виде двоичного исполняемого файла (executable file), чаще всего с расширением .EXE. Исполняемый файл генерируется из текста программы при компиляции.

Определения

- **Процесс** (process) – это программа (пользовательская или системная) в ходе выполнения.

- В современных операционных системах **процесс** представляет собой объект – структуру данных, содержащую информацию, необходимую для выполнения программы. Объект "Процесс" создается в момент запуска программы (например, пользователь дважды щелкает мышью на исполняемом файле) и уничтожается при завершении программы.

- Если операционная система умеет запускать в одно и то же время **несколько процессов**, она называется многозадачной (multitasking) (пример – Windows), иначе – однозадачной (пример – MS DOS).

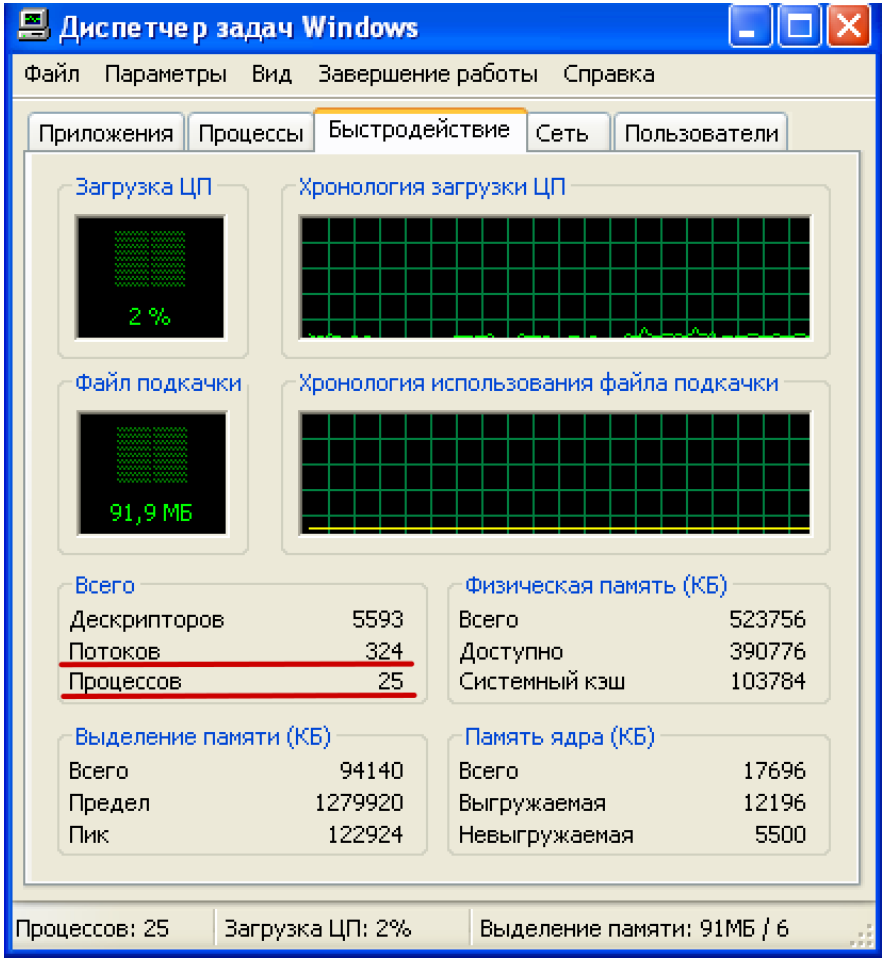
Определения

- **Процесс** может содержать один или несколько **потоков** (thread) – объектов, которым операционная система предоставляет процессорное время. Сам по себе **процесс** не выполняется – выполняются его **потоки**. Таким образом, машинные команды, записанные в исполняемом файле, выполняются на процессоре в составе потока. Если потоков несколько, они могут выполняться одновременно.

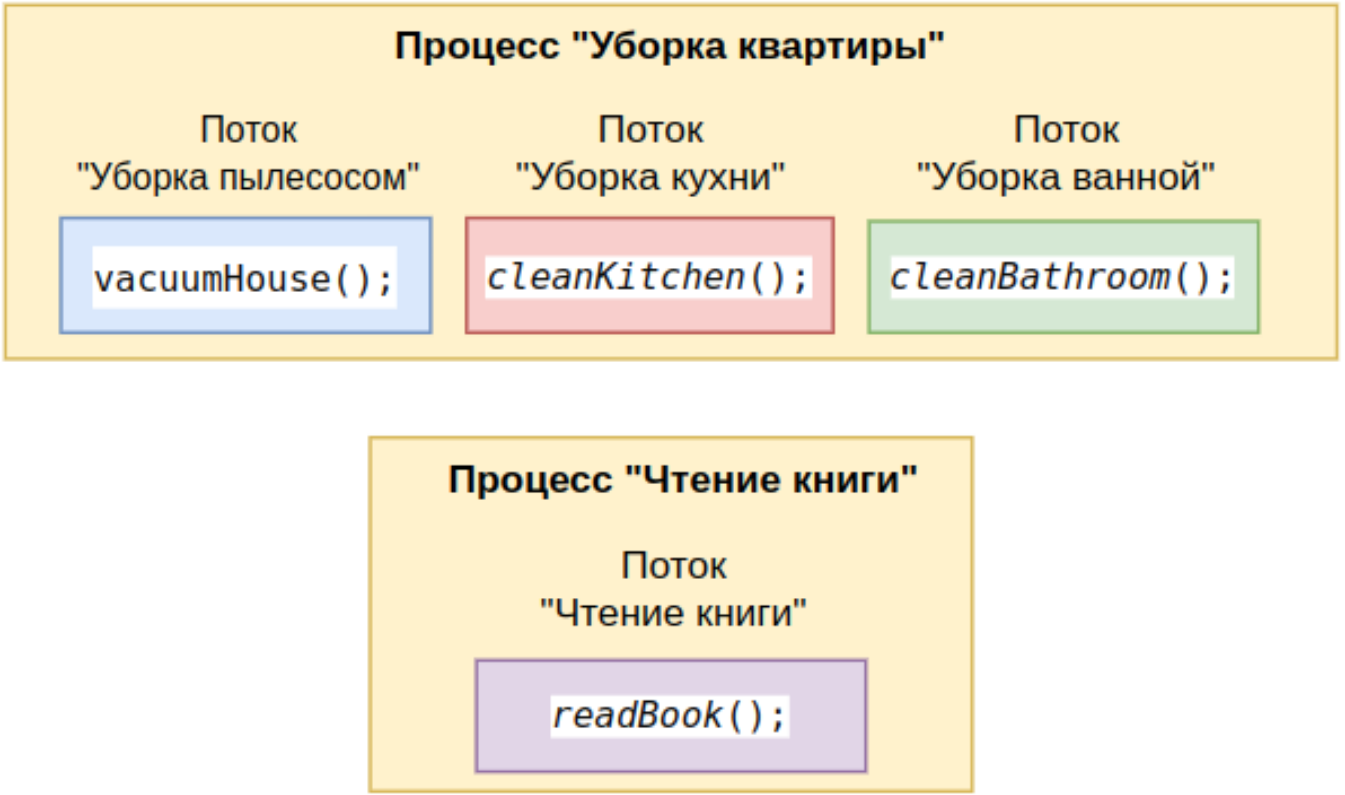
Определения

- "Одновременное" (или "параллельное") выполнение **потоков** подразумевает одну из двух принципиально разных ситуаций, зависящих от количества процессоров (ядер) на компьютере. В том случае, если имеется всего один процессор с одним ядром, в один момент времени может выполняться только один поток. Однако операционная система может быстро переключать процессор с выполнения одного потока на другой и вследствие высокой частоты процессоров у пользователя возникает иллюзия одновременной работы нескольких программ. Такая ситуация называется псевдопараллельное выполнение потоков. Если в компьютере установлен многоядерный процессор или количество процессоров больше одного, то возможно истинно параллельное или просто параллельное выполнение потоков.

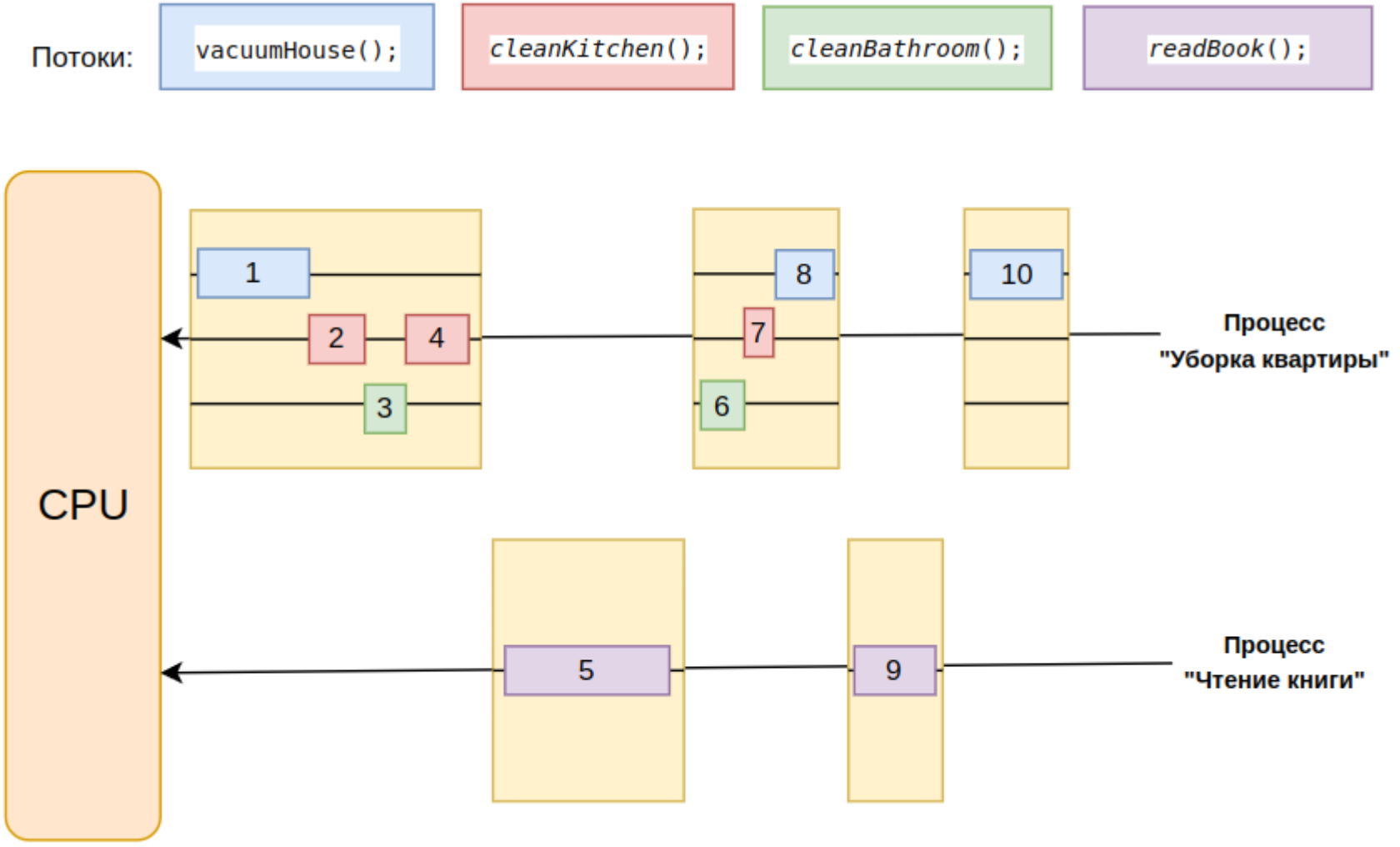
Диспетчер задач Windows XP. Система запущена на компьютере с одноядерным процессором



Наглядный пример процессов и потоков



Наглядный пример процессов и потоков



Различие между процессами и потоками

- **Поток представляет собой облегченную версию процесса.** Чтобы понять, в чем состоит его особенность, необходимо вспомнить основные характеристики процесса.

- **Процесс располагает определенными ресурсами.** Он размещен в некотором виртуальном адресном пространстве, содержащем образ этого процесса. Кроме того, процесс управляет другими ресурсами (файлы, устройства ввода / вывода и т.д.).

- **Процесс подвержен диспетчеризации.** Он определяет порядок выполнения одной или нескольких программ, при этом выполнение может перекрываться другими процессами. Каждый процесс имеет состояние выполнения и приоритет диспетчеризации.

Различие между процессами и потоками

Владельцу ресурса, обычно называемому **процессом** или задачей, присущи:

- виртуальное адресное пространство;
- индивидуальный доступ к процессору, другим процессам, файлам, и ресурсам ввода — вывода.

Модулю для диспетчеризации, обычно называемому **потоком** или облегченным процессом, присущи:

- состояние выполнения (активное, готовность и т.д.);
- сохранение контекста потока в неактивном состоянии;
- стек выполнения и некоторая статическая память для локальных переменных;
- доступ к пространству памяти и ресурсам своего процесса.

Различие между процессами и потоками

При корректной реализации **потоки** имеют определённые **преимущества** перед **процессами**. Им требуется:

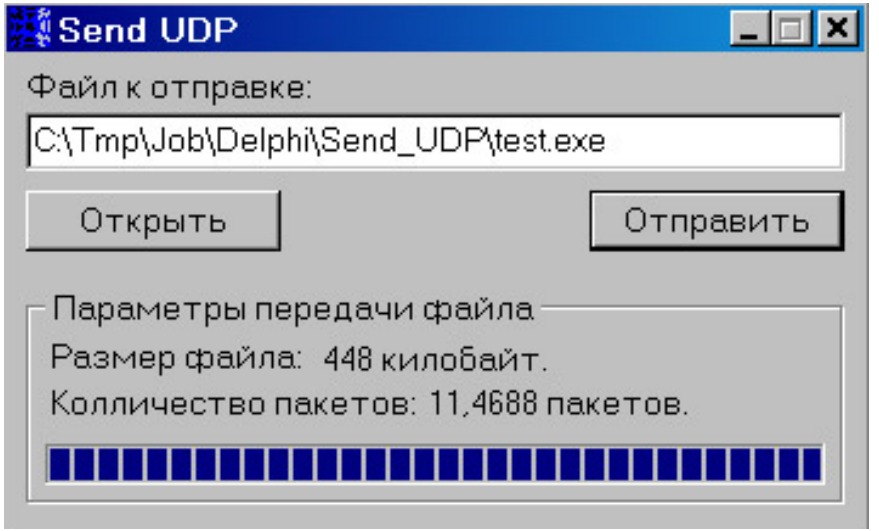
- меньше времени для создания нового потока, поскольку создаваемый поток использует адресное пространство текущего процесса;
- меньше времени для завершения потока;
- меньше времени для переключения между двумя потоками в пределах процесса;
- меньше коммуникационных расходов, поскольку потоки разделяют все ресурсы, и в частности адресное пространство.

Проблемы, которые решает многопоточность

1. Одновременное выполнение нескольких действий (например, отрисовка пользовательского интерфейса, передача файлов по сети и т. д.)

2. Рациональное использование ресурсов компьютера (например, если один поток ожидает ответа на запрос, отправленный по сети, то другой поток может тем временем использовать ЦП для выполнения чего-то другого)

3. Ускорение вычислений (при наличии нескольких вычислительных ядер)



Создание и запуск потоков в Java

```java
public class Main {

    public static void main(String[] args) {
        Thread myThread = new MyThread();

        myThread.start();
    }
}
```

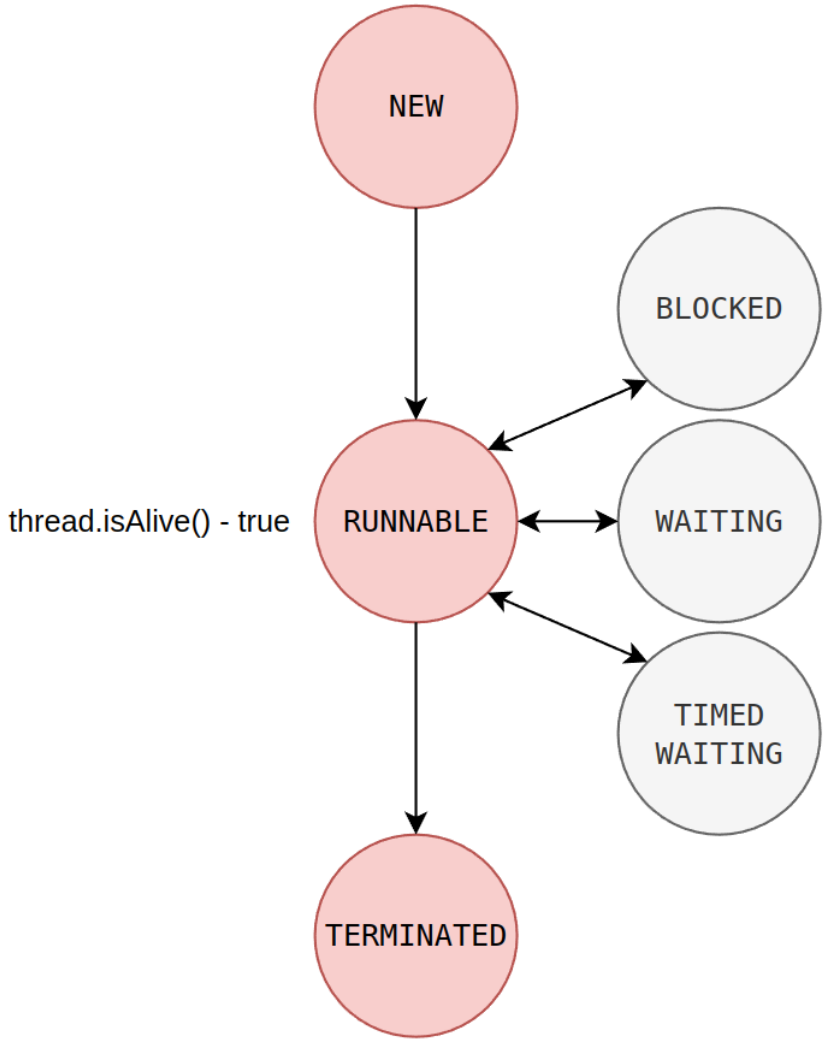
```java
public class MyThread extends Thread {

    @Override
    public void run() {
        System.out.println("MyThread running");
        // Выполнение какой-то работы
        System.out.println("MyThread finished");
    }
}
```

Создание и запуск потоков в Java

```java
public class Main {

    public static void main(String[] args) {
        Runnable myRunnable = new MyRunnable();
        Thread thread = new Thread(myRunnable);

        thread.start();
    }
}
```

```java
public class MyRunnable implements Runnable {

    @Override
    public void run() {
        System.out.println("MyRunnable running");
        // Выполнение какой-то работы
        System.out.println("MyRunnable finished");
    }
}
```

Жизненный цикл потока в Java



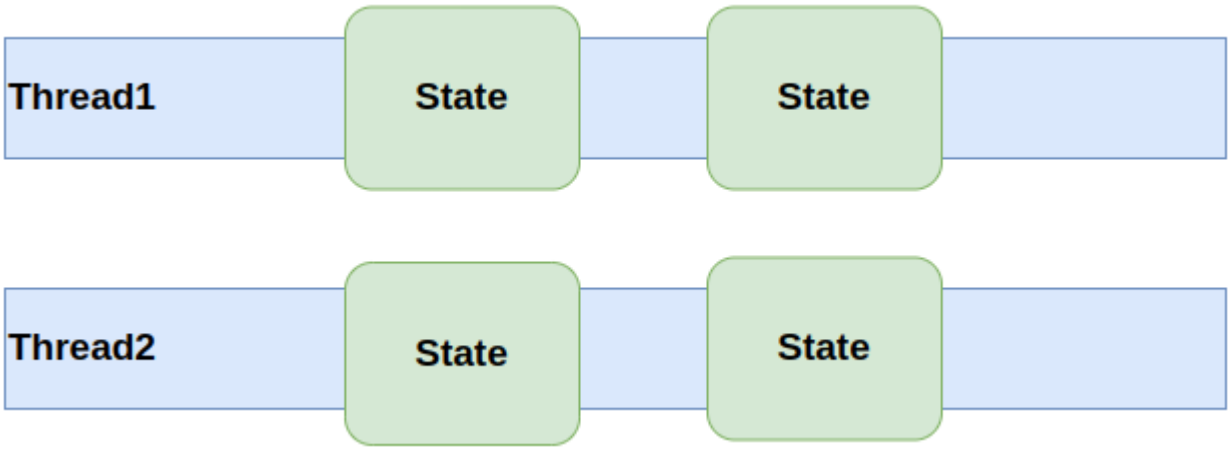
Планировщик потоков

Планировщик потоков – это часть JVM, которая решает какой поток должен выполняться в каждый конкретный момент времени и какой поток нужно приостановить.

Что гарантирует Java?

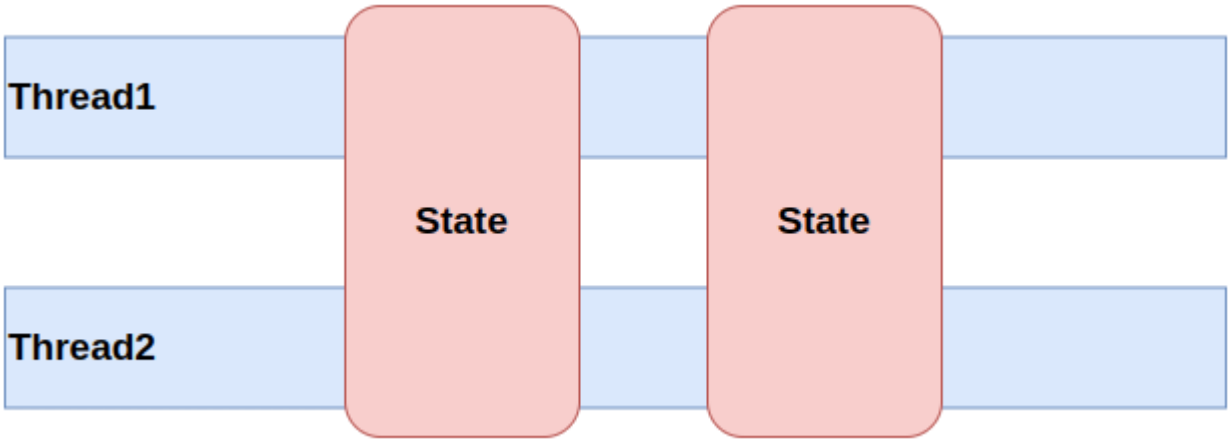
1. **Java НЕ гарантирует, что потоки будут выполнены в том порядке в котором они были запущены.**
2. Нет гарантии, что если поток начал свое выполнение, то он выполнит свою работу не прерываясь.
3. При каждом новом запуске программы, результат может быть разным.

Раздельное состояние (Separate State)



Проблем нет

Общее состояние (Shared State)



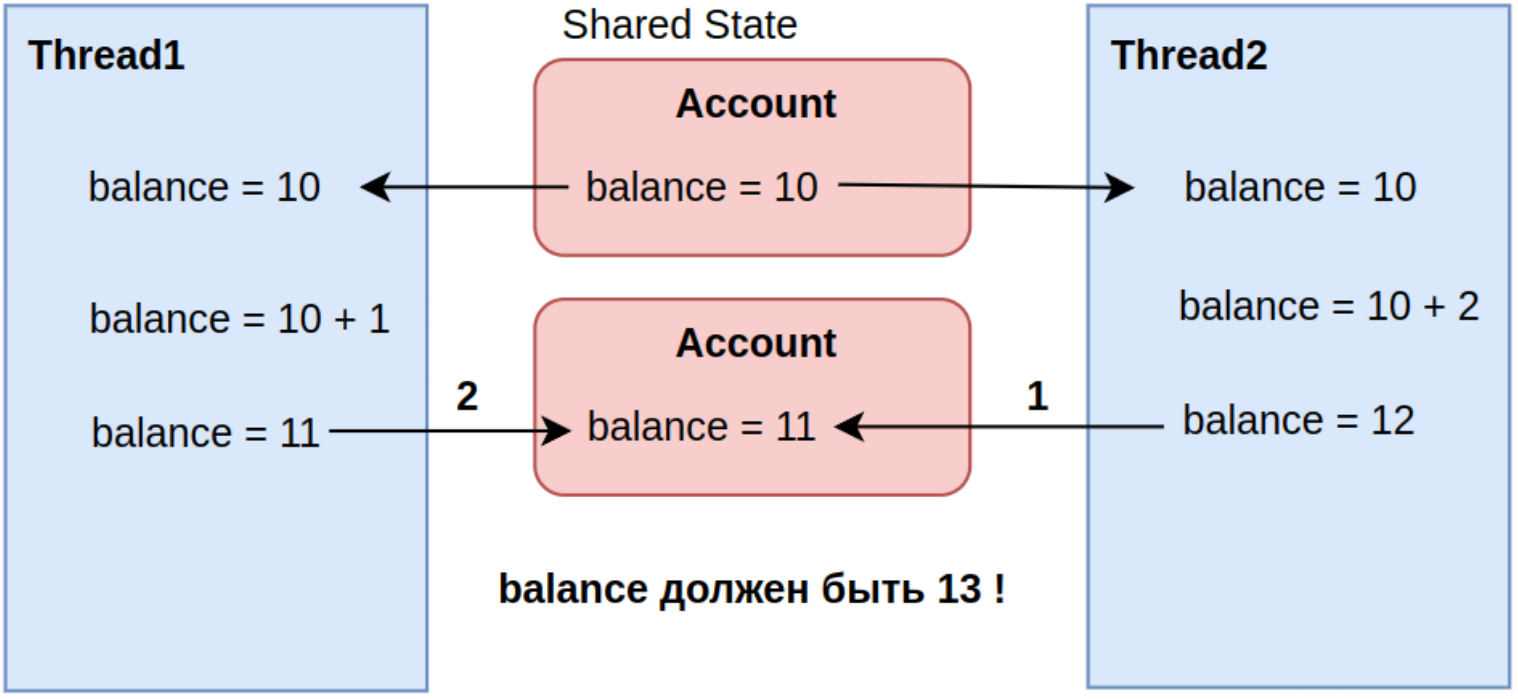
Могуть возникать следующие проблемы:

- Race condition (Состояние гонки);

- Deadlock (Взаимная блокировка).

Пример Race condition (Состояние гонки)

Состояние гонки (Race condition) — это проблема многопоточности, которая может возникать внутри критической секкции. Критическая секция (Critical Section) — это секция кода, в которой два или более потока **читают** или **записывают** данные **в один** и тот же **экземпляр объекта**.



Монитор объекта

Что происходит при выполнение строки **synchronized (объект)** ?

Каждый объект в Java имеет свой **монитор**. **Монитор** представляет собой инструмент для управления доступа к объекту. Когда выполнение кода доходит до оператора **synchronized**, **монитор объекта** класса **Account** блокируется, и на время его блокировки монопольный доступ к блоку кода имеет только один поток, который и произвел блокировку. После окончания работы блока кода, **монитор объекта** освобождается и становится доступным для других потоков.

```
public class Account {

    private long balance;

    public void add(long value) {

        synchronized (this) {
            balance += value;
        }
    }
}
```

Synchronized блок в **методах объекта**
Эквивалентная запись

```java
public class Account {

    private long balance;

    public synchronized void add(long value) {
        balance += value;
    }
}
```

```java
public class Account {

    private long balance;

    public void add(long value) {
        synchronized (this) {
            balance += value;
        }
    }
}
```

Synchronized блок в **статических методах**
Эквивалентная запись

```java
public class Account {

    private static long balance;

    public static synchronized void add(long value) {
        balance += value;
    }
}
```

```java
public class Account {

    private static long balance;

    public static void add(long value) {
        synchronized (Account.class) {
            balance += value;
        }
    }
}
```

Ограничения при использовании **synchronized**

Нужно помнить, что.

1. Использование **synchronized** увеличивает накладные расходы и в определенной степени снижает производительность работы программы.

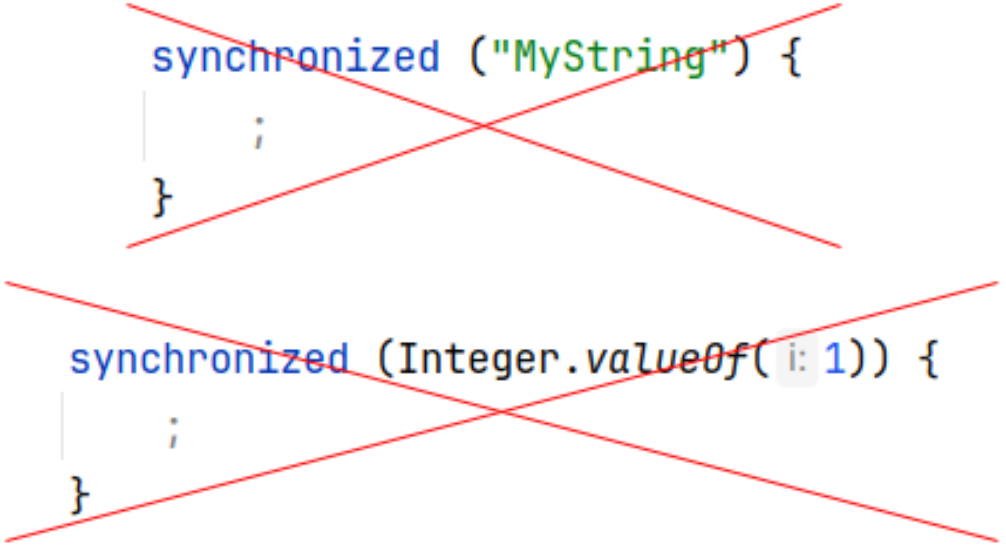
2. Старайтесь не использовать синхронизированные блоки большего размера, чем необходимо. Другими словами, синхронизируйте только те операции, которые действительно необходимы для синхронизации, чтобы избежать блокировки других потоков от выполнения операций, которые не нуждаются в синхронизации. Это повысить параллелизм вашего кода.

Ограничения при использовании **synchronized**

По каким объектам **нельзя** выполнять синхронизацию (захватывать монитор объекта).

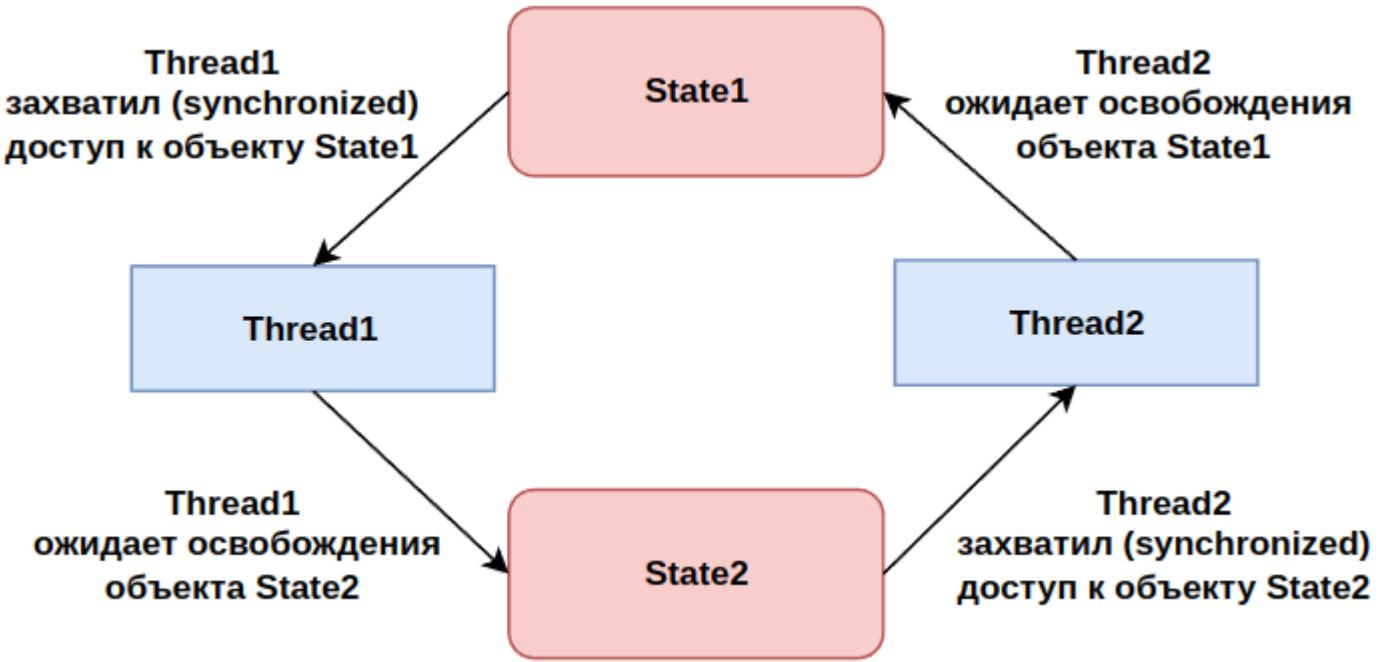
1. Объекты класса String.

2. Объекты-обертки примитивных типов: Integer, Double и т.д.



Пример Deadlock (Взаимная блокировка)

Deadlock (Взаимная блокировка) — ситуация в многопоточной программе, при которой несколько **протоков** находятся **в состоянии ожидания ресурсов**, занятых друг другом, и ни один из них не может продолжать свое выполнение.



Методы **wait()**, **notify()**, **notifyAll()**

wait(): **освобождает монитор** у объекта, по которому выполняется синхронизация**,** и переводит вызывающий поток в состояние ожидания до тех пор, пока другой поток не вызовет метод **notify()**.

notify(): отправляет сигнал для пробуждения **одного потока**, который ранее вызывал метод **wait()**.

notifyAll(): отправляет сигнал для пробуждения **всех потоков**, которые ранее вызвали метод **wait()**.

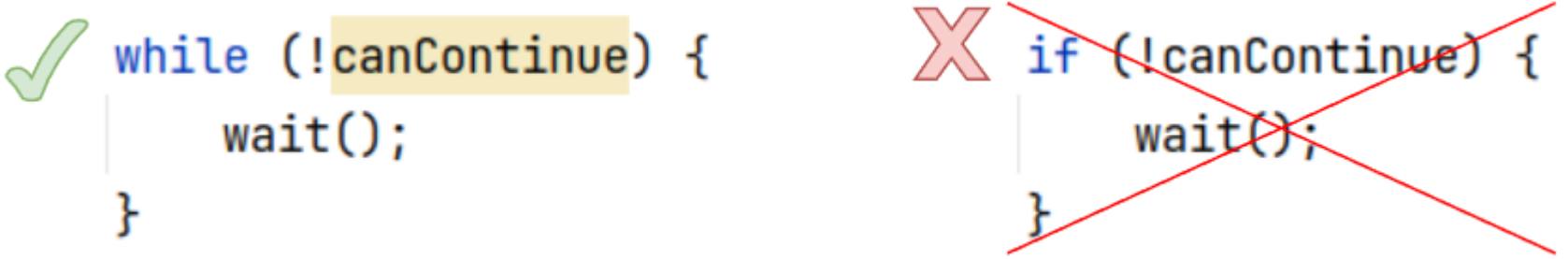
Методы вызываются для объекта, у которого был захвачен монитор !

Методы вызываются только внутри блока synchronized !

Spurious Wakeups (Ложные пробуждения)

Существует баг «Spurious Wakeups», из-за которого потоки, для которых ранее был вызван метод **wait()**, могут «**проснуться**», даже если **notify()** и **notifyAll() не были вызваны**.

Чтобы защититься от ложных пробуждений, необходимо **ВСЕГДА** метод **wait()** окружать **циклом while**, в котором будет проверяться условие возможности продолжения работы потока.



Ключевое слово **volatile**

```
public class Account {

    public volatile long balance;

}
```

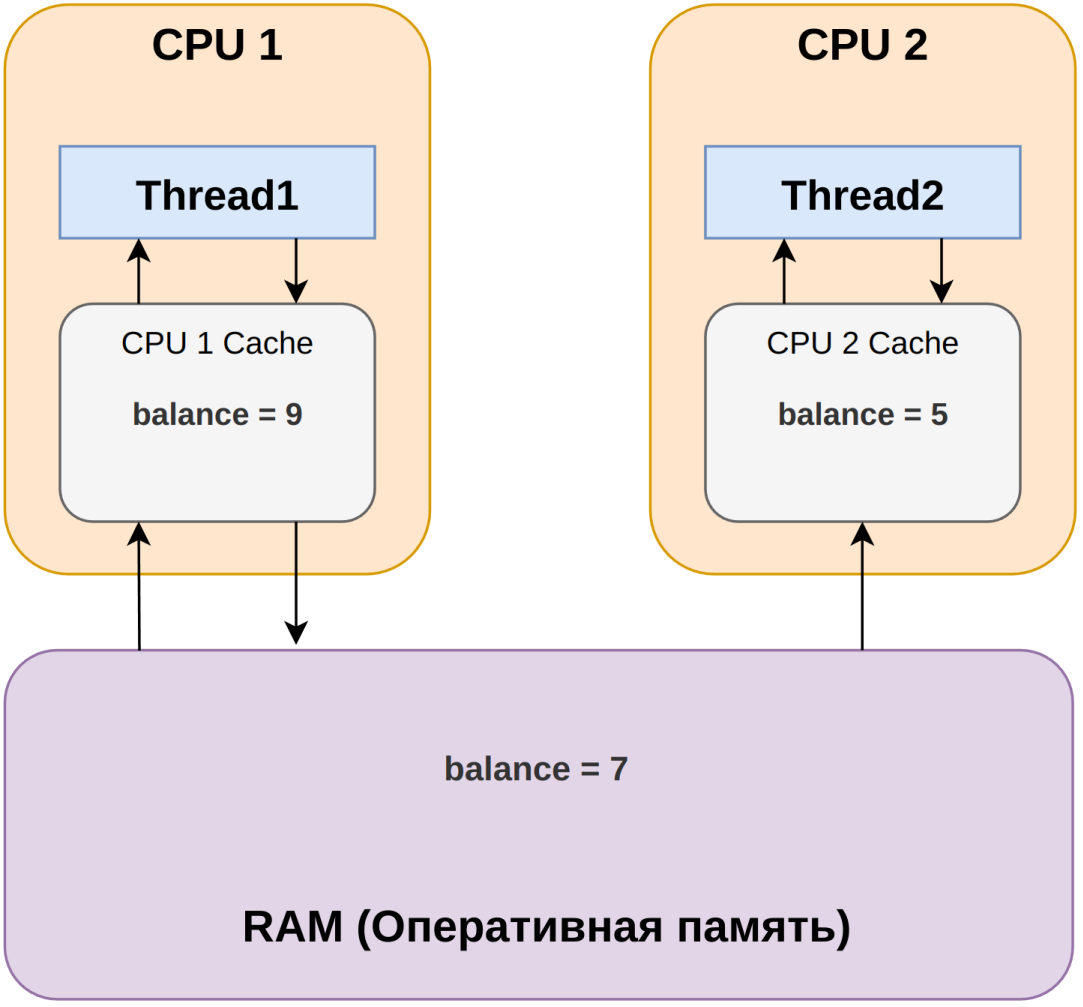
Ключевое слово **volatile** может быть указано для **полей** класса.

При указании у поля **volatile гарантируется**, что:

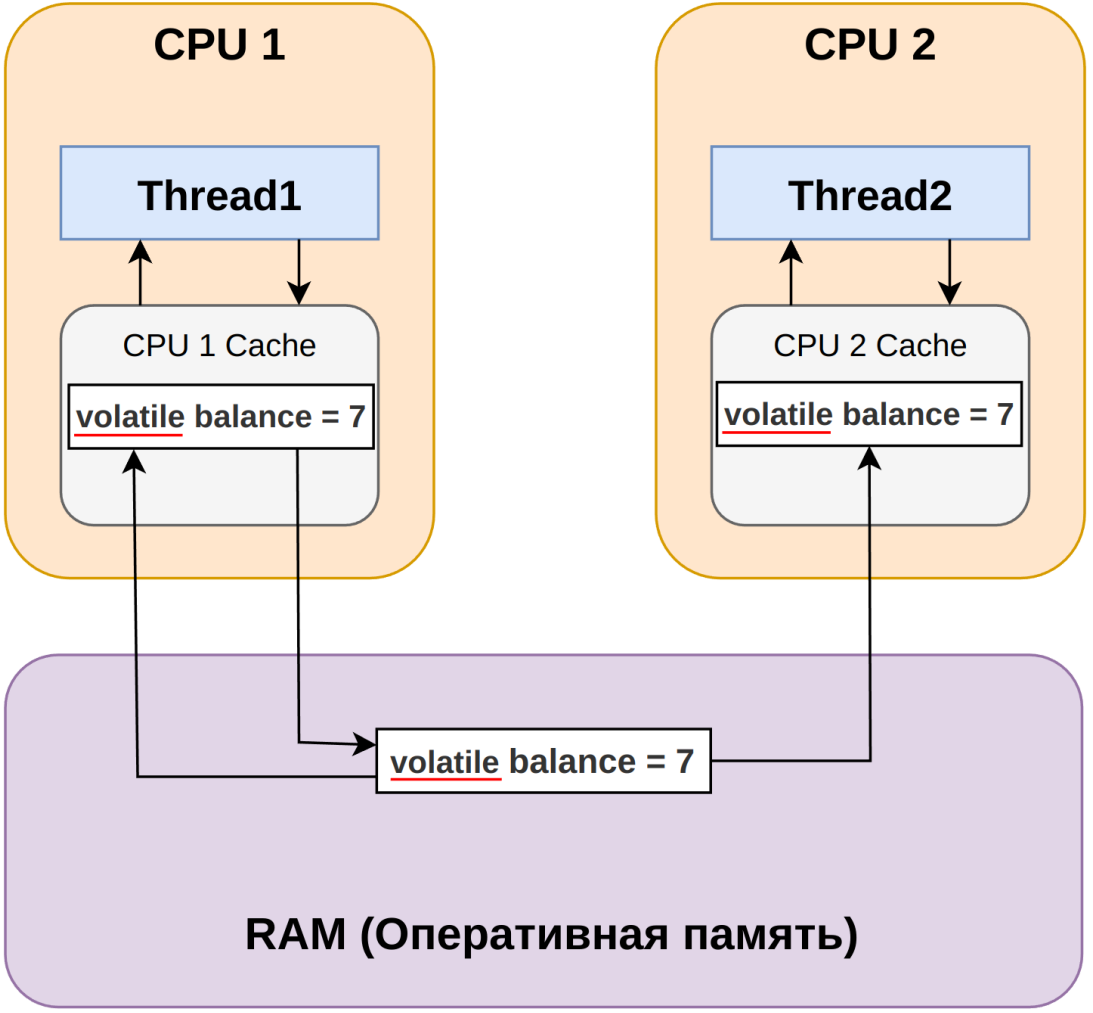
1. Операция **чтения** и **записи** volatile переменной будет **атомарной** (т.е. будет выполняться за «один шаг»). Этот пункт **важен** только для переменных **long** и **double!**

2. **Все потоки** программы всегда «видят» (читают) **акутальное значение** volatile переменной.

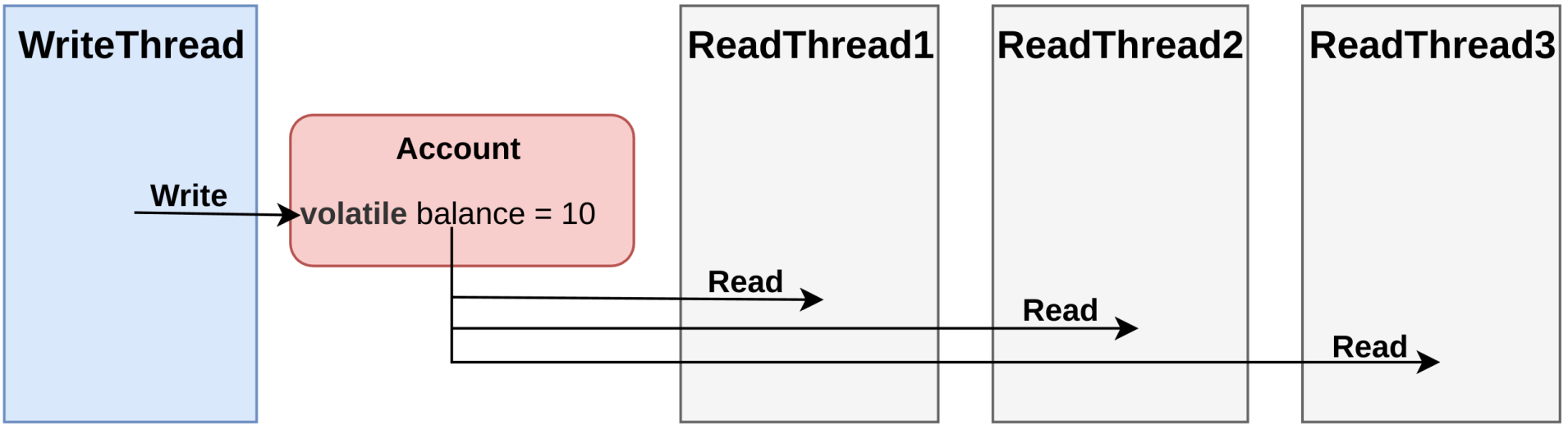
БЕЗ использования volatile



Применение **volatile**



Пример, когда может быть использовано **volatile**



Только **ОДИН** поток может изменять значение **volatile** переменной! Читать значение volatile переменной может любое количество потоков.

Важные моменты, которые не нужно забывать

1. Поток запускается методом **start().** Вызывать метод start() можно только **один раз**!

2. **Порядок** выполнения потоков **не гарантируется**!

3. По возможности нужно проектировать программы так, чтобы **синхронизация** между потоками **не требовалась** (модель параллелизма «Раздельное состояние» (Separate State)).

4. Каждый объект в Java имеет свой **монитор**. Монитор представляет собой инструмент для управления доступа к объекту. Когда выполнение кода доходит до оператора **synchronized**, **монитор** объекта класса **«блокируется»**.

5. Методы **wait(), notify(), notifyAll()** вызываются для **объекта**, у которого был **захвачен монитор** (synchronized(object))! Методы **wait(), notify(), notifyAll()** вызываются только **внутри** блока **synchronized!**

6. Баг «**Ложные пробуждения**» (Spurious Wakeups). Метод **wait()** нужно всегда «окружать» **циклом while**.

7. Ключевое слово **volatile** гарантирует, что **все потоки** программы всегда «видят» (читают) **акутальное** значение volatile переменной (читают из основной памяти, а не кешей ядер процессора).