MTC True Tech

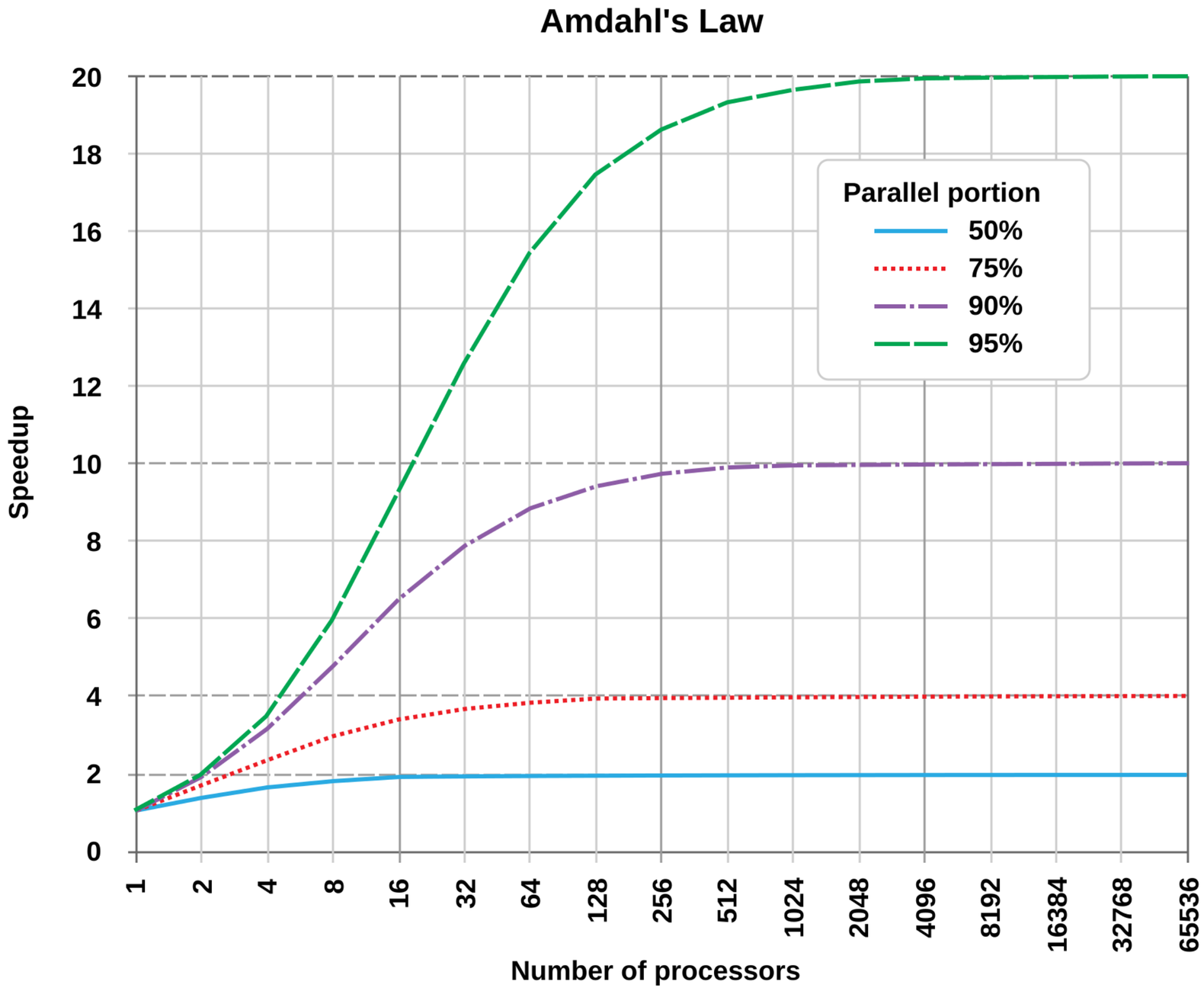
Java Multithreading

Калайчьян Сурен
Backend developer

Предыстория

Закон Амдала

«В случае, когда задача разделяется на несколько частей, суммарное время её выполнения на параллельной системе не может быть меньше времени выполнения самого медленного фрагмента».



$$S_p = \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

Universal Scalability Law (USL)

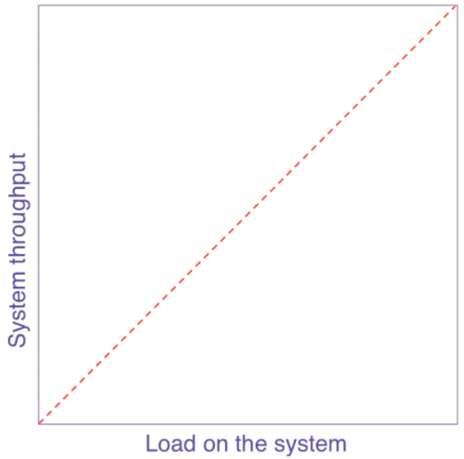
In distributed systems, you can use Universal Scalability Law (USL) to model and to optimize scalability of your system.

→ **Contention** - refers to delay due to waiting or queueing for shared resources

→ **Coherency** - refers to delay for data to become consistent

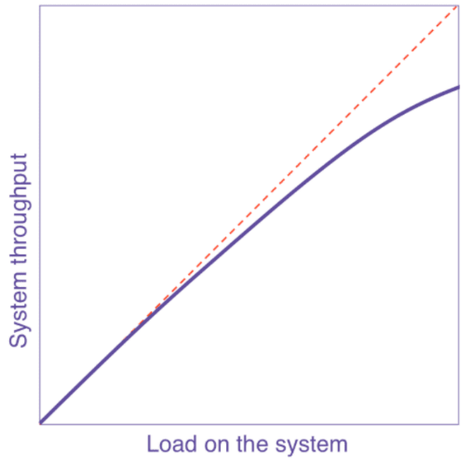
Universal Scalability Law (USL)

A. Equal bang for the buck



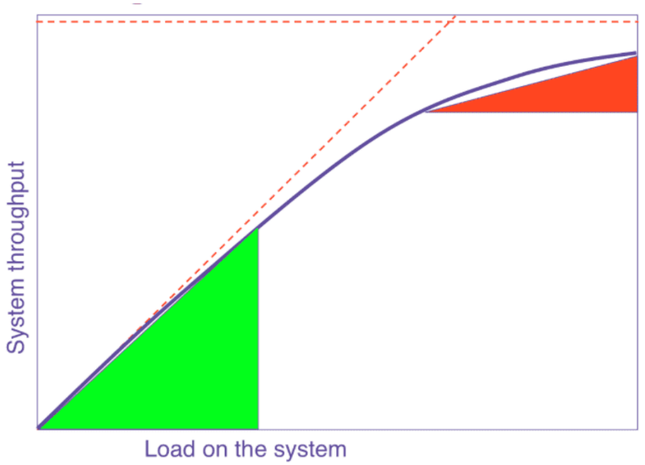
$\alpha = 0, \beta = 0$

B. Cost of sharing resources



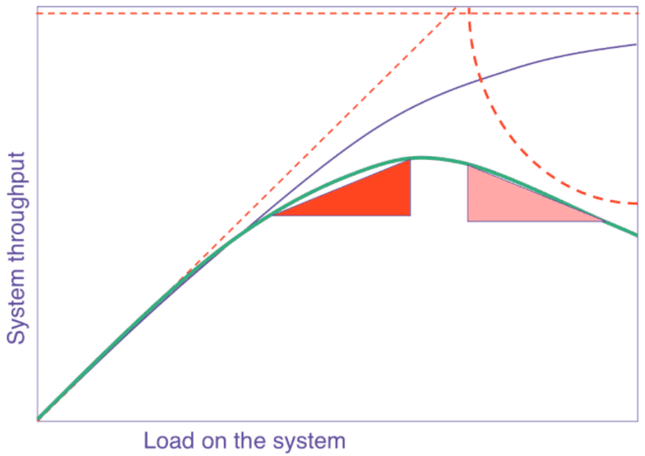
$\alpha > 0, \beta = 0$

C. Diminishing returns from contention



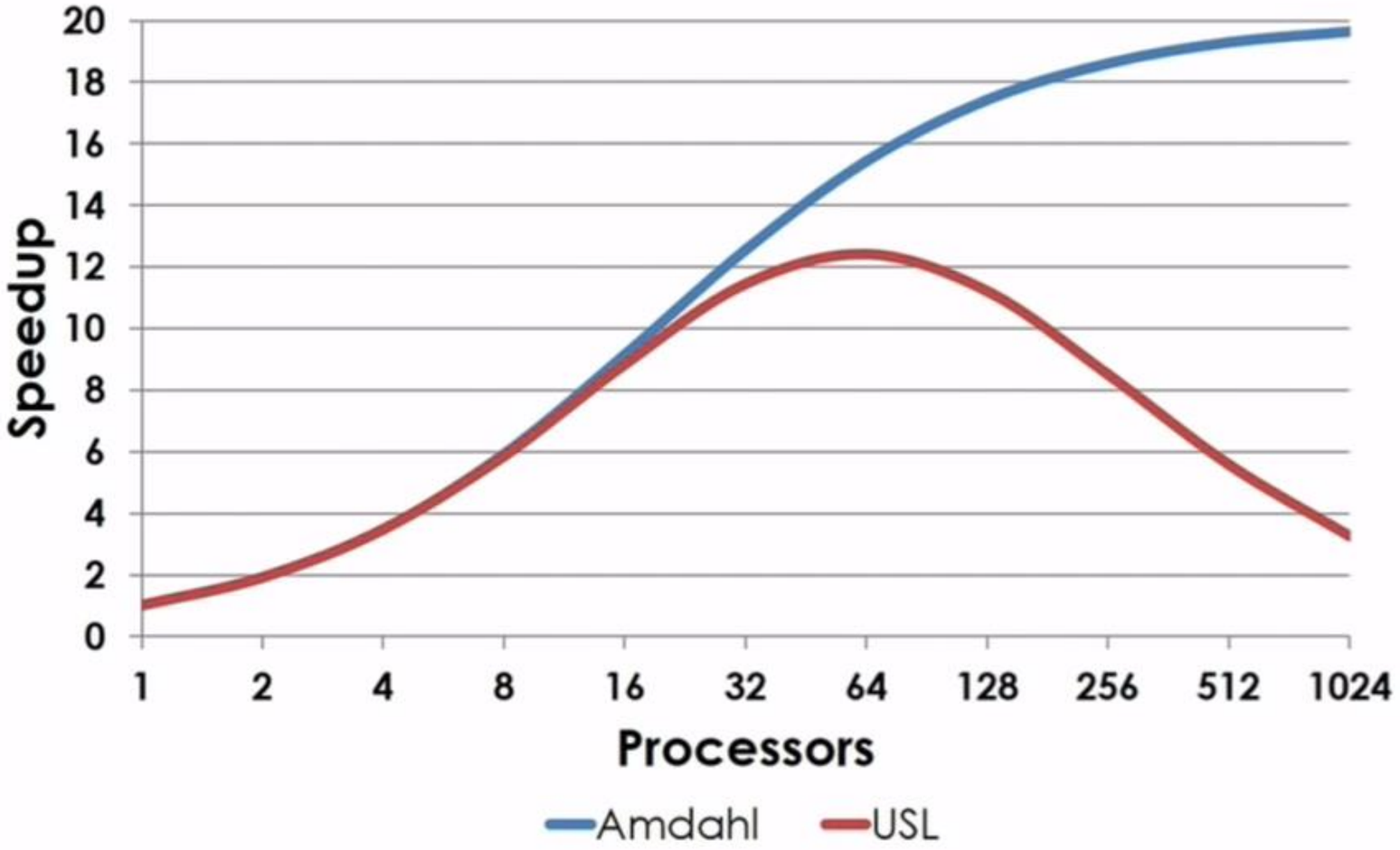
$\alpha >> 0, \beta = 0$

D. Negative returns from incoherency

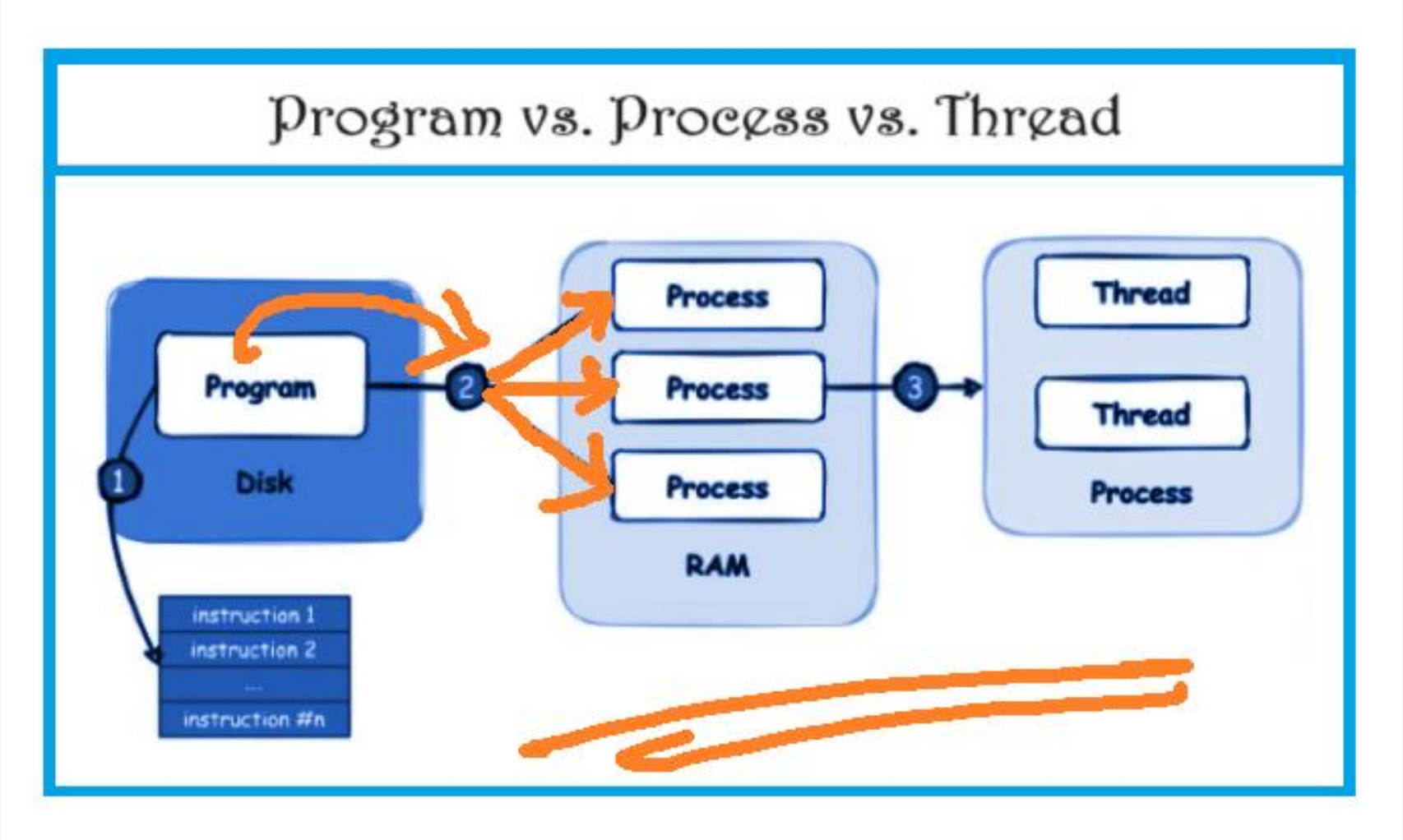


$\alpha >> 0, \beta > 0$

Universal Scalability Law



Программа, процесс, поток



Основы

Thread

Это независимый путь выполнения в рамках одного процесса, который позволяет программе выполнять несколько задач параллельно, что повышает её производительность.

Сам по себе класс java.lang.Thread является API для управления потоками, а не самим потоком, который представляет собой наименьшую единицу выполнения инструкций, управляемую операционной системой

Thread

Это независимый путь выполнения в рамках одного процесса, который позволяет программе выполнять несколько задач параллельно, что повышает её производительность.

Сам по себе класс java.lang.Thread является API для управления потоками, а не самим потоком, который представляет собой наименьшую единицу выполнения инструкций, управляемую операционной системой

- **run()**: Содержит код, который будет выполнен потоком. Этот метод необходимо переопределить, если вы создаёте свой класс, наследуясь от Thread.
- **start()**: Запускает поток, вызывая метод run() в новом потоке выполнения.
- **join()**: Позволяет текущему потоку ждать завершения выполнения другого потока, прежде чем продолжить свою работу.

Thread

```
public class MyThread extends Thread {

    @Override
    public void run() {
        System.out.println("Hello MyThread run");
    }
}
```

Thread

```
public class MyThread extends Thread {

    @Override
    public void run() {
        System.out.println("Hello MyThread run");
        try {
            MyThread.sleep(1000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Join

```
MyThread myThread = new MyThread();
myThread.start();

try {
    myThread.join();
} catch (InterruptedException e) {
    throw new RuntimeException(e);
}
```

Thread + Runnable

```
public class ThreadRunnable {

    public void executeThread() {
        Thread thread = new Thread(() ->  System.out.println("Hello ThreadRunnable 1"));
        thread.start();
    }
}
```

Thread + Runnable

```
public class ThreadRunnable {

    public void executeThread() {
        Thread thread = new Thread(() -> {
            System.out.println("Hello ThreadRunnable 1");
            try {
                sleep(1000);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            System.out.println("Hello ThreadRunnable 2");
        });
        thread.start();
    }
}
```

Thread

Типы потоков:

- **Non-daemon**: Основные потоки программы. Программа завершается только после завершения всех не-демон потоков.
- **Daemon**: Фоновые потоки, которые завершаются при завершении всех не-демон потоков, даже если они ещё выполняются

Thread + Runnable

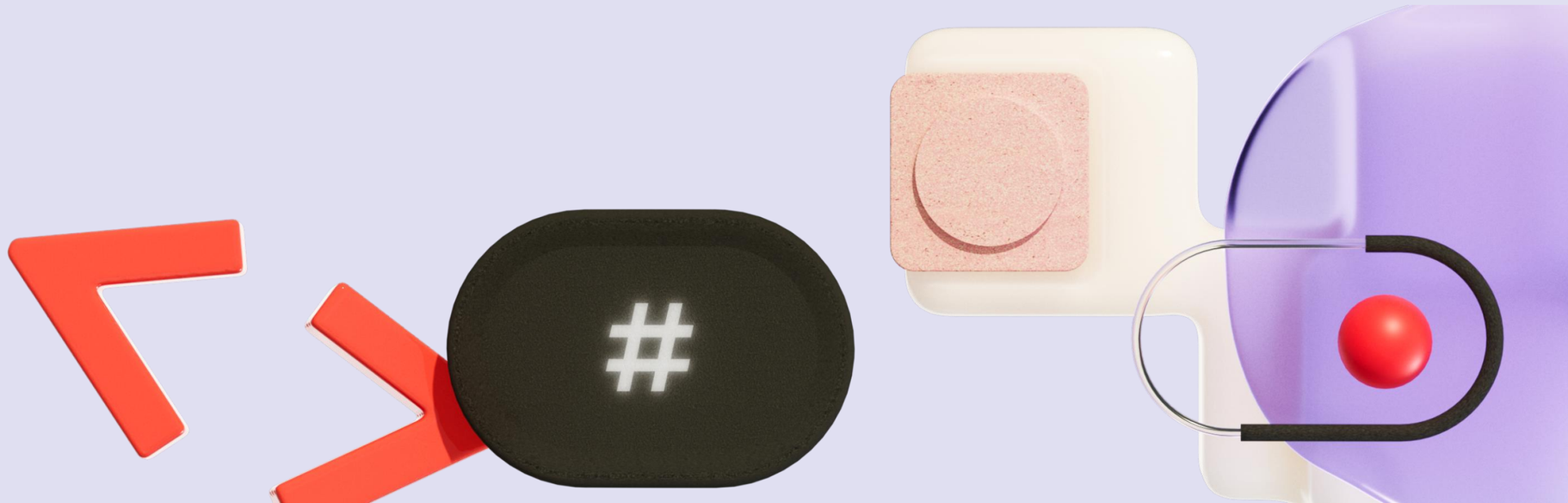
```
thread.isDaemon();
```

Thread + Runnable

```
thread.isDaemon();
```

```
throw new RuntimeException();
```

Проблемы потокобезопасности



Counter

```
public class Counter {

    int count = 0;

    public void increment() {
        count++;
    }

    public void decrement() {
        count--;
    }

    public int getCount() {
        return count;
    }
}
```

Что такое потокобезопасность

Класс является потокобезопасным, если он ведет себя корректно при доступе из нескольких потоков, независимо от чередования выполнения этих потоков и без дополнительной синхронизации или другой координации со стороны вызывающего кода.

Классы без состояний априори потокобезопасны.

Примитивы синхронизации

Доступ к разделяемому ресурсу

Для того чтобы класс Counter вёл себя корректно при многопоточной работе, необходим способ предотвратить использование поля count другими потоками, пока мы находимся в процессе его изменения. Этого можно добиться разными способами:

Mutex (mutual exclusion) (взаимное исключение)

→Synchronized

→Locks

Lock free:

→ Atomic (CAS)

Wait free:

→CAS + фиксированное число шагов

Synchronized

В Java есть механизм блокировок - блок synchronized. Данный блок состоит из двух частей: кода, который должен выполняться атомарно, и объекта, который будет являться блокировкой.

В качестве блокировки может выступать любой Java-объект.

Код, расположенный внутри блока synchronized и защищённый одним и тем же объектом, в один момент времени может исполняться только одним потоком.
Иными словами, только один поток может "войти" внутрь synchronized. Остальные потоки вынуждены ждать. Однако, если блок synchronized будет защищён разными объектами, то возможно одновременное выполнение.

Доступ к разделяемому ресурсу

Для того чтобы класс Counter вёл себя корректно при многопоточной работе, необходим способ предотвратить использование поля count другими потоками, пока мы находимся в процессе его изменения. Этого можно добиться разными способами:

Mutex (mutual exclusion) (взаимное исключение)

→Synchronized

→Locks

Lock free:

→ Atomic (CAS)

Wait free:

→CAS + фиксированное число шагов

Synchronized

```
public synchronized void do()            // this

public static synchronized void do()  // .class


Object monitor = new Object();

public void do()  {
   synchronized(monitor) { }
}
```

Counter

```
public class SynchronizedCounter {

    private int count = 0;

    public synchronized void increment() {
        count++;
    }

    public void decrement() {
        synchronized (this) {
            count--;
        }
    }

    public synchronized int getCount() {
        return count;
    }
}
```

Deadlock

Это ситуация, когда два или более потока **навечно** блокируют друг друга, ожидая освобождения ресурсов, которыми владеет другой поток в этом же цикле.

Это приводит к тому, что ни один из потоков не может продолжить выполнение, и вся программа останавливается или зависает.

Deadlock

Это ситуация, когда два или более потока **навечно** блокируют друг друга, ожидая освобождения ресурсов, которыми владеет другой поток в этом же цикле.

Это приводит к тому, что ни один из потоков не может продолжить выполнение, и вся программа останавливается или зависает.

Как возникает deadlock:

- **Конфликт за ресурсы:** Два или более потока пытаются получить доступ к общим ресурсам (например, объектам с synchronized блоками), но делают это в разном порядке.
- **Циклическая зависимость:** У каждого потока возникает циклическая зависимость от ресурсов другого потока.

Deadlock

Пример:

- Поток A захватывает монополию над объектом X и ждет объект Y.
- Поток B захватывает монополию над объектом Y и ждет объект X.
- Поскольку ни один поток не может освободить свой ресурс, чтобы получить доступ к другому, оба потока остаются в состоянии ожидания

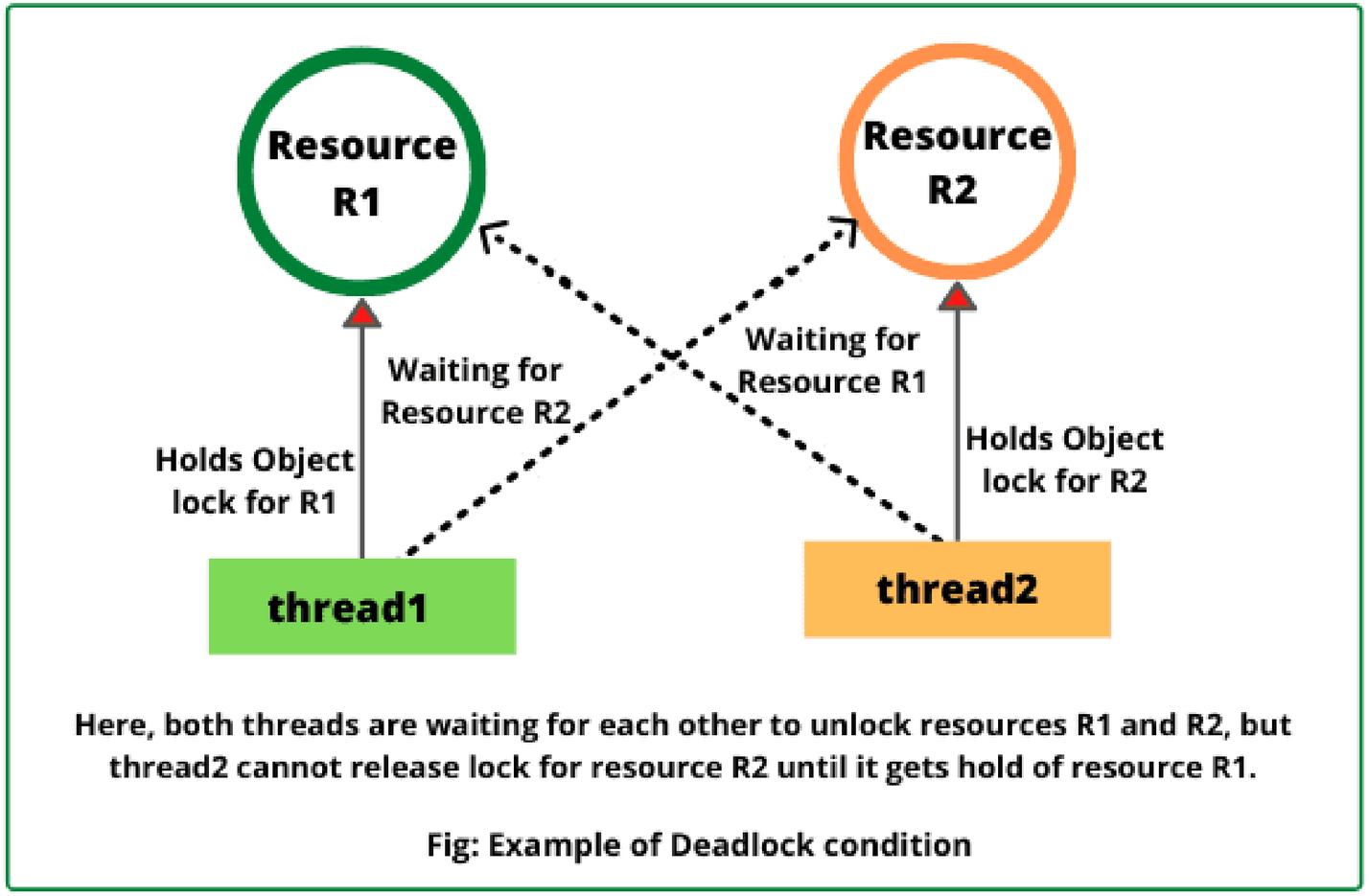
Deadlock

```
public class Account {

    public void doTransfer(Object account1, Object account2) {

        synchronized (account1) {
            synchronized (account2) {
                System.out.println("Perevod sredstv enterprise logic");
            }
        }
    }
}
```

Deadlock



Here, both threads are waiting for each other to unlock resources R1 and R2, but thread2 cannot release lock for resource R2 until it gets hold of resource R1.

Fig: Example of Deadlock condition

MTC True Tech

Вопросы?

QR