



УНИВЕРСИТЕТ ИТМО

## Программирование. 2 семестр

# Stream API

ITMO It's more than a  
UNIVERSITY

- ☑ Конвейерная обработка данных
- ☑ Поток — последовательность элементов
- ☑ Поток может быть последовательным или параллельным
- ☑ Конвейер — последовательность операций

## ☑ Отличия конвейера от коллекции

- Элементы не хранятся
- Неявная итерация
- Функциональный стиль — операции не меняют источник
- Большинство операций работают с  $\lambda$ -выражениями
- Ленивое выполнение
- Возможность неограниченного числа элементов

☑ Конвейер =

- Источник
- Промежуточные операции (0 или больше)
- Завершающая операция (одна)

```
List<String> words
```

```
long count = 0;
for (String s : words) {
    if (s.length() > 5) {
        count++;
    }
}
```

```
List<String> words
```

```
long count = 0;
for (String s : words) {
    if (s.length() > 5) {
        count++;
    }
}
```

```
long count =
    words.stream()
        .filter(s -> s.length() > 5)
        .count();
```

- ✓ `Collection.stream()`
- ✓ `Arrays.stream(Object[])`
- ✓ `Stream.of(Object[])`
- ✓ `IntStream.range(int, int)`
- ✓ `Files.lines(Path)`
- ✓ `Stream.empty()`
- ✓ `Stream.generate(Supplier<T> s)`
- ✓ `Stream.iterate(T seed, UnaryOperator<T> f)`

- ☑ Возвращают поток
- ☑ Выполняются "лениво"
  - выполнение операции происходит, когда вызывается завершающая операция
- ☑ Делятся на:
  - Не хранящие состояние (stateless)
    - ◆ выполнение не зависит от других элементов
  - Хранящие состояние (stateful)
    - ◆ выполнение зависит от других элементов (сортировка)

- ✓ Возвращают результат
- ✓ Выполняют побочное действие
- ✓ Поток прекращает существование.

## ☑ интерфейс BaseStream

- void close()
- S parallel()
- S sequential()
- S unordered()
- Iterator iterator()
- Spliterator spliterator()

## ☑ Интерфейс Stream<T>

## ☑ Интерфейсы IntStream, LongStream, DoubleStream

## ☑ Параллельный Iterator

- `Spliterator trySplit()` — возвращает часть элементов как отдельный сплитератор
- `boolean tryAdvance(Consumer action)` — выполнить операцию для очередного элемента
- `void forEachRemaining(Consumer action)` — выполнить операцию для всех оставшихся элементов

- ☑ Методы для промежуточных операций (stateless)
- ☑ **`Stream<T> filter (Predicate<T> p)`**
  - возвращает поток из элементов, соответствующих условию
- ☑ **`Stream<R> map(Function<T,R> mapper)`**
  - преобразует поток элементов T в поток элементов R
- ☑ **`Stream<R> flatMap(Function <T, Stream<R>> mapper)`**
  - преобразует каждый элемент потока T в поток элементов R
- ☑ **`Stream<T> peek(Consumer<T> action)`**
  - выполняет действие для каждого элемента потока T

- ✓ Методы для промежуточных операций (stateful)
- ✓ `Stream<T> distinct()`
  - возвращает поток неповторяющихся элементов
- ✓ `Stream<T> sorted(Comparator<T> comp)`
  - возвращает отсортированный поток
- ✓ `Stream<T> limit(long size)`
  - возвращает усеченный поток из size элементов
- ✓ `Stream<T> skip(long n)`
  - возвращает поток, пропустив n элементов

- ✓ `void forEach(Consumer<T> action)`
- ✓ `void forEachOrdered(Consumer<T> action)`
  - выполняет действие для каждого элемента потока
  - второй вариант гарантирует сохранение порядка элементов
- ✓ `Optional<T> min(), Optional<T> max()`
  - возвращают минимальный и максимальный элементы,
- ✓ `long count(), int (long, double) sum()`
  - возвращают количество и сумму элементов
- ✓ `OptionalInt, OptionalLong, OptionalDouble`
  - `int getAsInt(), long getAsLong(), double getAsDouble()`

- ✓ Оболочка, которая содержит или не содержит значение
- ✓ `boolean isPresent()` - true, если значение есть
- ✓ `T get()` - возвращает значение
- ✓ `Optional<T> of(T value)` — возвращает оболочку
- ✓ `T orElse(T other)` — возвращает значение или other
- ✓ `Optional<U> map(Function<T, U> mapper)` - применить функцию, если есть значение

- ☑ `reduce(BinaryOperator accumulator)`  
    `.stream()`  
    `.reduce((a, b) -> a * b) // подсчет произведения`
- ☑ `collect(Collector collector)`  
    `.stream()`  
    `.collect(Collectors.toList());`

## ☑ класс `Collectors`

- `toCollection(Supplier factory), toList(), toSet()`
- `toMap(Function k, Function v, BinaryOperator merge, Supplier factory)`
- `joining(String delimiter, String prefix, String suffix)`
- `mapping(Function<T,U> mapper, Collector<U> s)`
- `minBy(Comparator), maxBy(Comparator)`
- `counting(), summingDouble(), averagingDouble()`
- `reducing(identity, Function<T,U> mapper, BinaryOperator op)`
- `groupingBy(Function<T,K> classifier)`
- `partitioningBy(Predicate<T> predicate)`

## ☑ boolean anyMatch(Predicate<T> p)

- истина, если условие выполняется хотя бы для одного элемента
- При нахождении первого совпадения прекращает проверку

## ☑ boolean allMatch(Predicate<T> p)

- истина, если условие выполняется для всех элементов.
- При нахождении первого несовпадения прекращает проверку

## ☑ boolean noneMatch(Predicate<T> p)

- истина, если условие не выполняется ни для одного элемента.
- При нахождении первого совпадения прекращает проверку

```
public static void main(String[] args) {  
    Arrays.asList(args).stream()  
        .filter(s -> s.length() < 5)  
        .map(String::toUpperCase)  
        .sorted()  
        .forEachOrdered(System.out::println);  
}
```



УНИВЕРСИТЕТ ИТМО

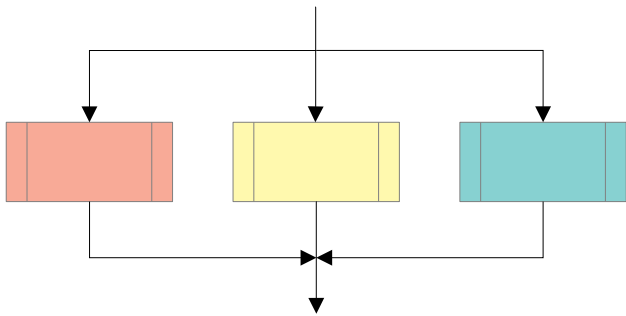
Программирование. 2 семестр

# Многопоточность. Теория



## ☑ Thread

- нить, резьба
- ПОТОК
  - ◆ поток выполнения
  - ◆ параллельно



## ☑ Stream

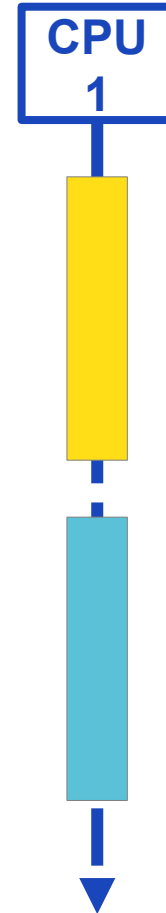
- поток, ручей
- ПОТОК
  - ◆ поток данных
  - ◆ последовательно



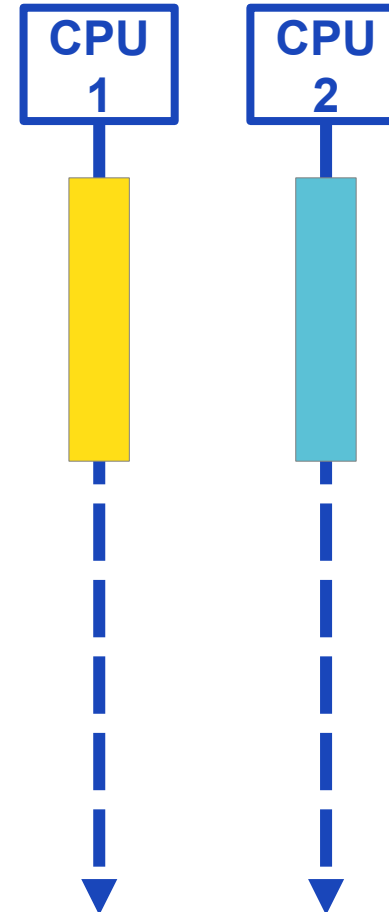
- Один процессор
- Одна задача



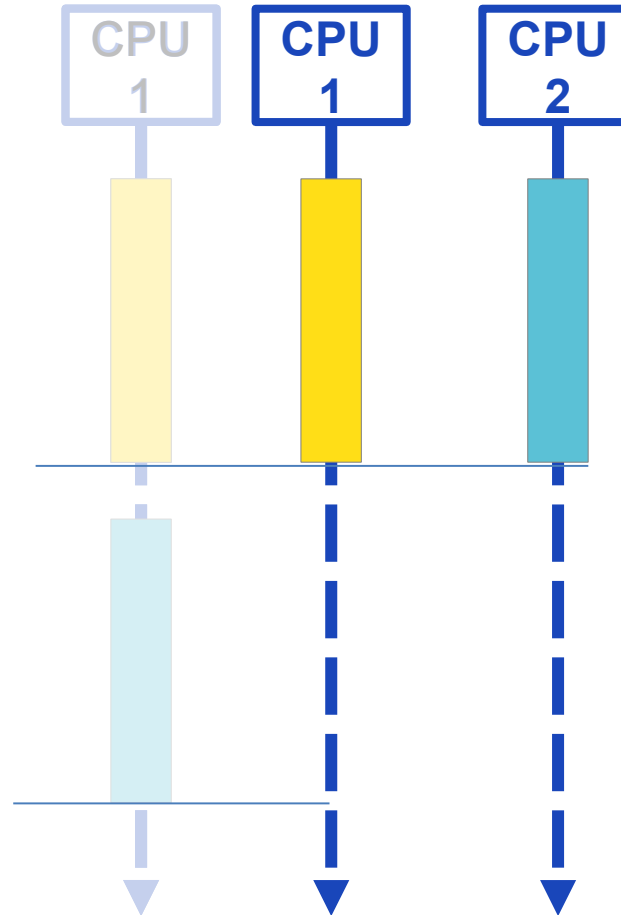
- Один процессор
- Две задачи
- Последовательное исполнение
- Простое управление



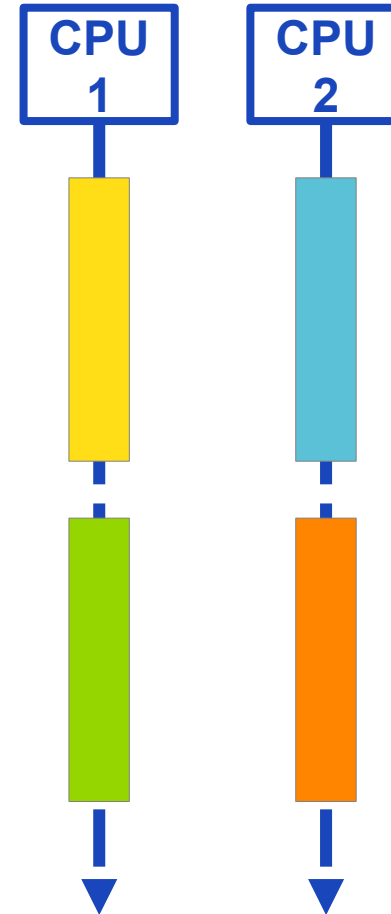
- Два процессора
- Две задачи
- Параллельное исполнение
- Более сложное управление



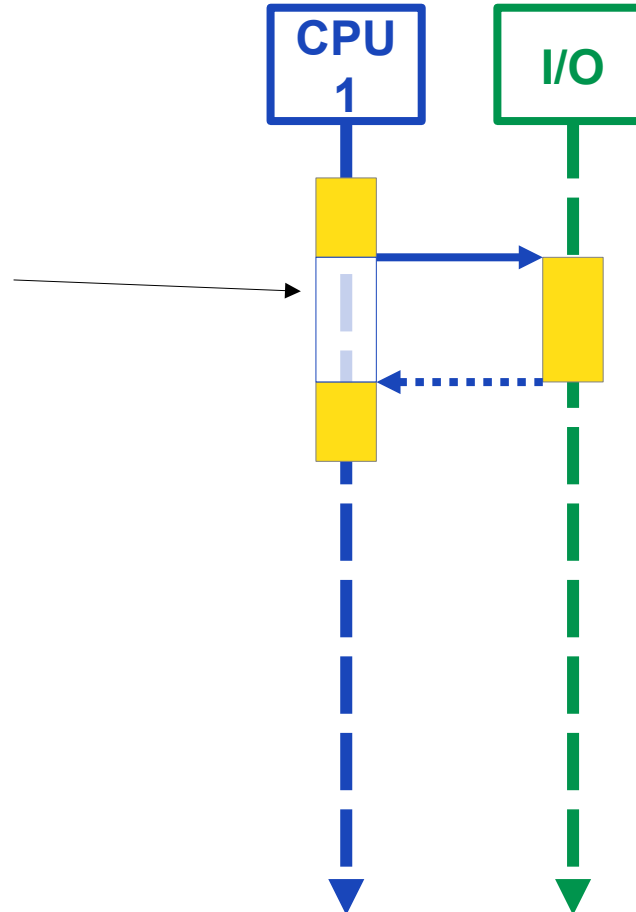
- Два процессора
- Две задачи
- Параллельное исполнение
- Более сложное управление



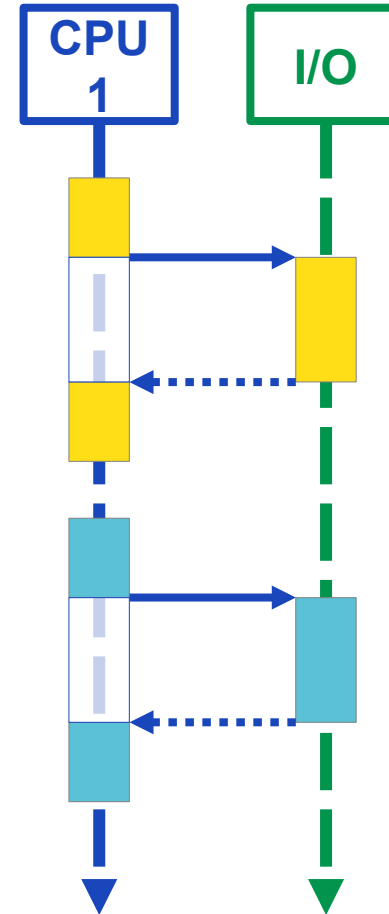
- Много процессоров
- Много задач
- Параллельное исполнение
- Высокая производительность



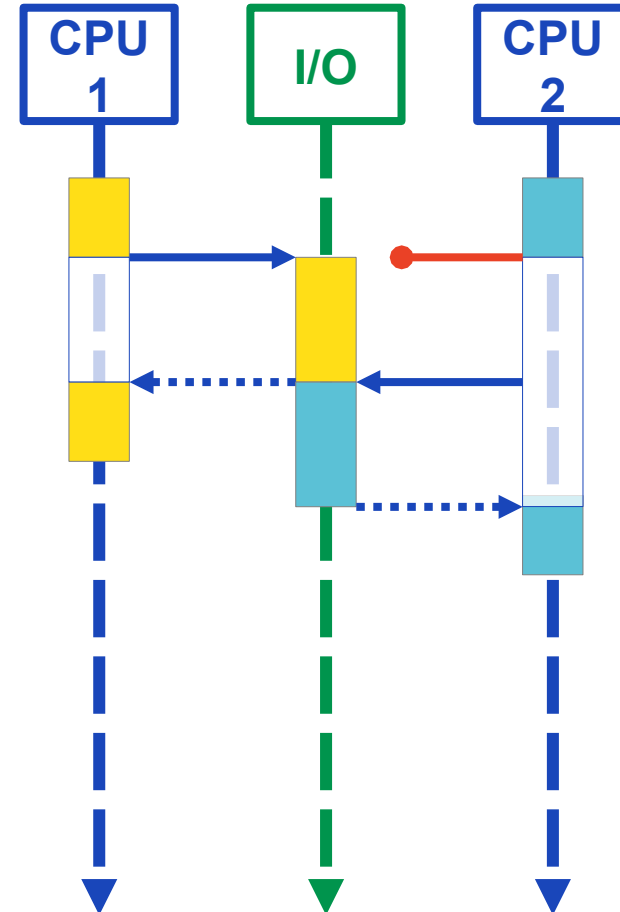
- Один процессор
- Одна задача
- Ввод-вывод
- Процессор простаивает



- 

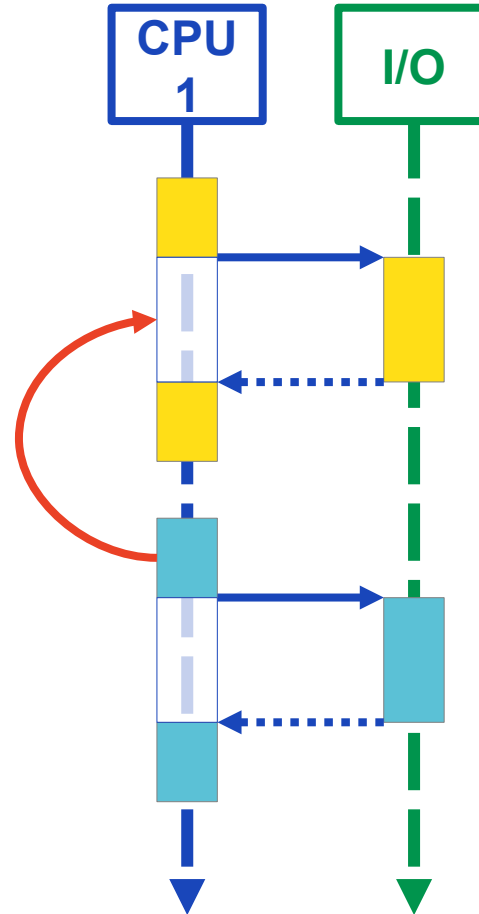


- Два процессора
- Две задачи
- Ввод-вывод
- Процессор простаивает
- Занятость В/У

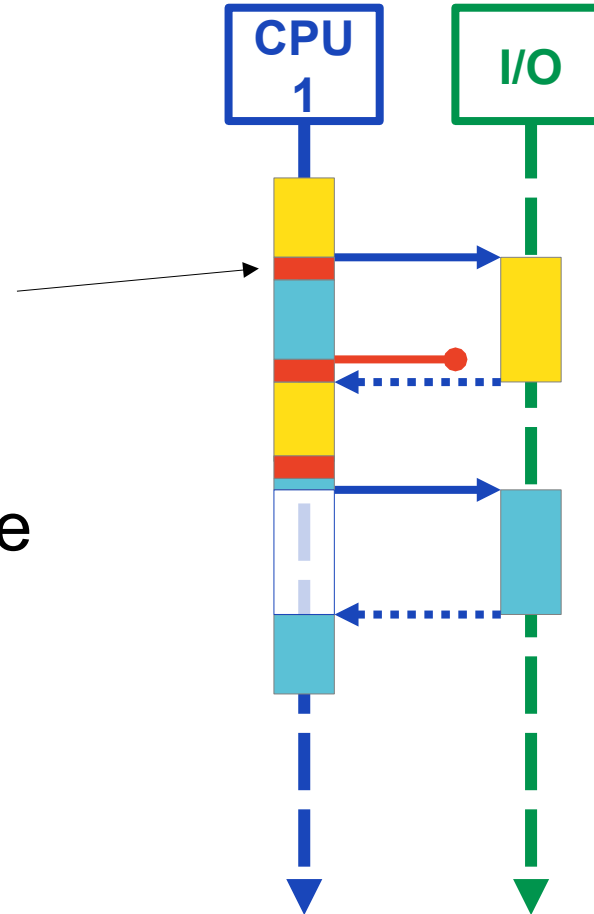


# Управление задачами

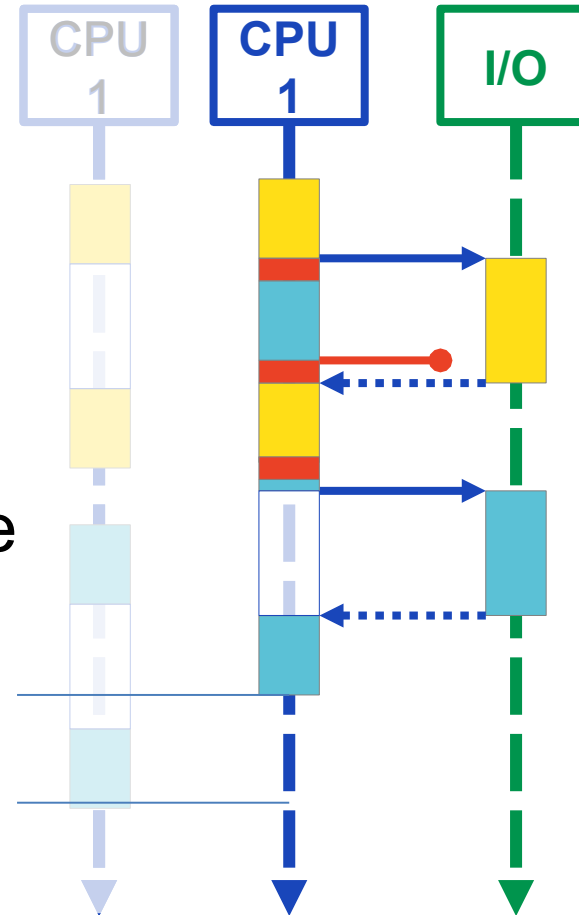
- Один процессор
- Две задачи
- Ввод-вывод
- Процессор простаивает!



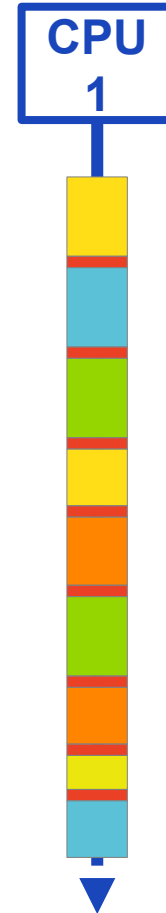
- Один процессор
- Две задачи
- Ввод-вывод
- Переключение контекста
- Конкурентное исполнение
- Более сложное управление



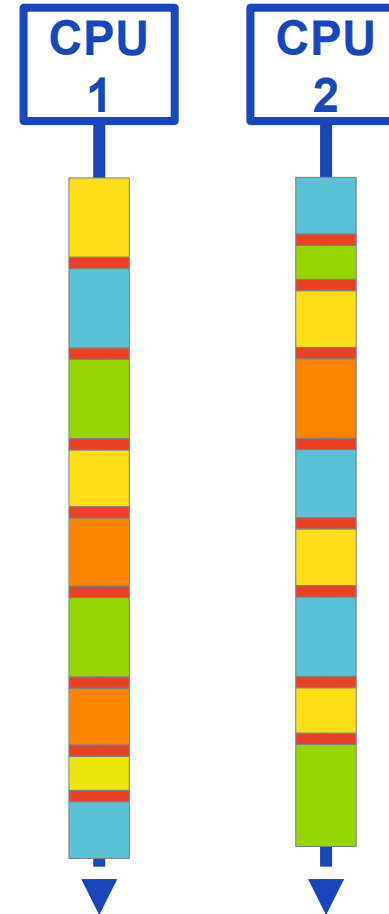
- Один процессор
- Две задачи
- Ввод-вывод
- Переключение контекста
- Конкурентное исполнение
- Более сложное управление



- Один процессор
- Много задач
- Многозадачность
- Конкурентное исполнение



- Много процессоров
- Много задач
- Многозадачность
- Конкурентное исполнение



## ☑ Кооперативная многозадачность

- Добровольное переключение в удобный момент
- Если задача не делится ресурсами — другие страдают

## ☑ Вытесняющая многозадачность

- Задачи переключает диспетчер
- Переключение в произвольные моменты
- Необходимо сохранения состояния
- Необходимо согласование доступа

## ☑ Процесс

- отдельное приложение
- свои ресурсы и память
- долгое переключение контекста

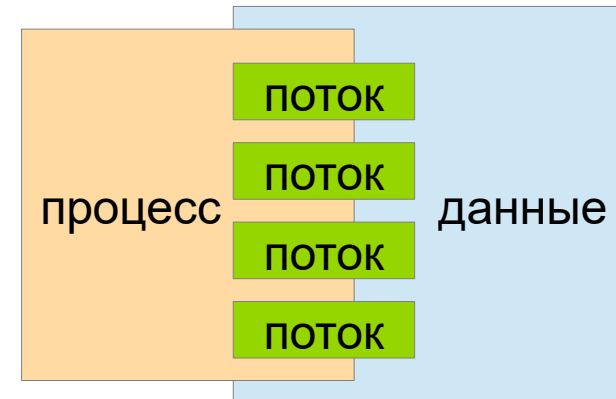
## ☑ Многозадачность



## ☑ Поток (thread)

- работает внутри процесса
- общие ресурсы и память
- быстрое переключение контекста

## ☑ Многопоточность



## ☑ Аппаратные

- Многопроцессорность
- Многоядерность
- Многопоточность — переключение контекста на уровне процессора
  - ◆ Временная (Temporal) — в один момент времени один поток
  - ◆ Одновременная (Simultaneous) — в один момент времени несколько потоков

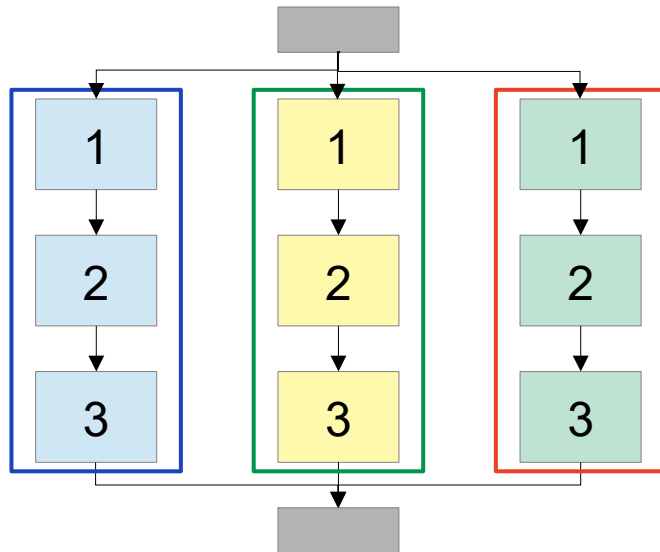
## ☑ Программные

- Многозадачность
- Многопоточность на уровне ядра ОС
- Многопоточность на пользовательском уровне
  - ◆ Green Threads — потоки, управляемые виртуальной машиной

# Варианты многопоточной обработки

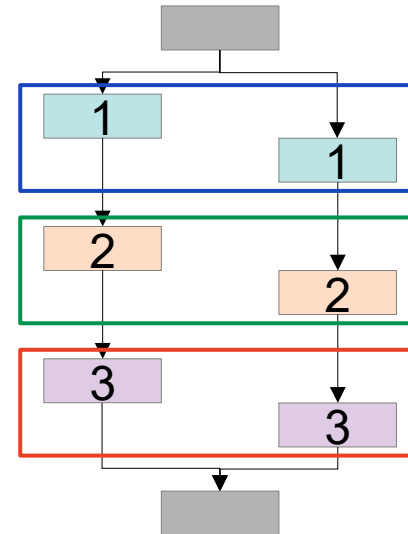
## ✓ распараллеливание

- обработчик выполняет все этапы одной задачи



## ✓ конвейерная обработка

- обработчик выполняет один этап всех задач





УНИВЕРСИТЕТ ИТМО

Программирование. 2 семестр

# Многопоточность. Java

ITMO More than a  
UNIVERSITY

## ☑ Потоки в JVM

- системные потоки:
  - ◆ основной поток виртуальной машины
  - ◆ сборщик мусора
  - ◆ поток периодических задач
  - ◆ поток динамической компиляции
- прикладные потоки
  - ◆ основной поток (main)
  - ◆ создаваемые программным путем

- ☑ интерфейс Runnable — выполняемая задача
  - метод run()
  - завершается метод — завершается поток
- ☑ класс Thread implements Runnable — поток
  - метод start()
  - метод не ждет завершения потока
- ☑ Что будет, если вызвать метод run() напрямую?

# Класс Thread и интерфейс Runnable

```
class A extends Thread {  
    public void run() {  
        /* тело потока */  
    }  
}  
  
new A().start();
```

# Класс Thread и интерфейс Runnable

```
class A extends Thread {  
    public void run() {  
        /* тело потока */  
    }  
}  
  
new A().start();
```

```
class A implements Runnable {  
    public void run() {  
        /* тело потока */  
    }  
}  
  
new Thread(new A()).start();
```

# Класс Thread и интерфейс Runnable

```
class A extends Thread {  
    public void run() {  
        /* тело потока */  
    }  
}  
  
new A().start();
```

```
class A implements Runnable {  
    public void run() {  
        /* тело потока */  
    }  
}  
  
new Thread(new A()).start();
```

```
new Thread(() -> {  
    /* тело потока */  
}).start();
```

- ☑ `Thread.sleep(1000);` // спать 1000 мс
- ☑ `t.join();` // ожидать завершения `t`
- ☑ и другие полезные методы
  - `Thread.currentThread()`
  - `getId()`
  - `getName()`
  - `isAlive()`
  - `getPriority()`
  - `getThreadGroup()`

## ☑ метод `interrupt()`

- прерывает выполнение некоторых методов (`sleep`, `join`, `wait`) прерыванием `InterruptedException`
  - ◆ прерывание можно поймать и завершить поток (выйти из `run()`)
  - ◆ прерывание можно пробросить
- устанавливает флаг прерывания потока
  - ◆ флаг можно проверить и завершить поток
  - ◆ флаг можно игнорировать

## ☑ проверка флага

- `Thread.interrupted()` - сбрасывает флаг
- `isInterrupted()` - не сбрасывает флаг

```
public class ThreadTest {  
    public static void main(String[] args) {  
        Runnable r = () -> {  
            String name = Thread.currentThread().getName();  
            System.out.println(name + " started");  
            try {  
                Thread.sleep(500 + (long)(100 * Math.random()));  
            } catch (InterruptedException e) { return; }  
            System.out.println(name + " finished");  
        };  
  
        for (int i = 0; i < 10; i++) {  
            (new Thread(r)).start();  
        }  
    }  
}
```

```
Thread-0 started  
Thread-2 started  
Thread-9 started  
Thread-8 started  
Thread-6 started  
Thread-7 started  
Thread-5 started  
Thread-1 started  
Thread-4 started  
Thread-3 started
```

```
Thread-0 started  
Thread-2 started  
Thread-9 started  
Thread-8 started  
Thread-6 started  
Thread-7 started  
Thread-5 started  
Thread-1 started  
Thread-4 started  
Thread-3 started
```

```
Thread-8 finished  
Thread-7 finished  
Thread-9 finished  
Thread-5 finished  
Thread-0 finished  
Thread-1 finished  
Thread-2 finished  
Thread-6 finished  
Thread-4 finished  
Thread-3 finished
```

# Гонки (race condition)

```
class Shared {  
    int counter = 1;  
    void up()    { counter++; }  
    void down() { counter--; }  
}
```

# Гонки (race condition)

```
class Shared {  
    int counter = 1;  
    void up()    { counter++; }  
    void down() { counter--; }  
}
```

```
Shared sh = new Shared();(  
new Thread(sh::down)).start();  
new Thread(sh::up)).start();
```

# Гонки (race condition)

```
class Shared {  
    int counter = 1;  
    void up()    { counter++; }  
    void down() { counter--; }  
}
```

```
Shared sh = new Shared();(  
new Thread(sh::down)).start();  
new Thread(sh::up)).start();
```

++    --

- 1) load counter
- 2) add 1 (sub 1)
- 3) store counter

# Гонки (race condition)

```
class Shared {  
    int counter = 1;  
    void up()    { counter++; }  
    void down() { counter--; }  
}
```

```
Shared sh = new Shared();  
new Thread(sh::down).start();  
new Thread(sh::up).start();
```

| поток 1           | counter | поток 2           |
|-------------------|---------|-------------------|
| load counter (1)  | 1       |                   |
| sub 1 (0)         | 0       |                   |
| store counter (0) | 0       |                   |
|                   | 0       | load counter (0)  |
|                   | 1       | add 1 (1)         |
|                   | 1       | store counter (1) |

++    --

```
1) load counter  
2) add 1 (sub 1)  
3) store counter
```

# Гонки (race condition)

```
class Shared {  
    int counter = 1;  
    void up()    { counter++; }  
    void down() { counter--; }  
}
```

```
Shared sh = new Shared();  
new Thread(sh::down).start();  
new Thread(sh::up).start();
```

| поток 1           | counter | поток 2           |
|-------------------|---------|-------------------|
| load counter (1)  | 1       |                   |
|                   | 1       | load counter (1)  |
|                   | 1       | add 1 (2)         |
|                   | 2       | store counter (2) |
| sub 1 (0)         | 2       |                   |
| store counter (0) | 0       |                   |

|    |               |    |
|----|---------------|----|
|    | ++            | -- |
| 1) | load counter  |    |
| 2) | add 1 (sub 1) |    |
| 3) | store counter |    |

# Гонки (race condition)

```
class Shared {  
    int counter = 1;  
    void up()    { counter++; }  
    void down() { counter--; }  
}
```

**КРИТИЧЕСКАЯ СЕКЦИЯ**

```
Shared sh = new Shared();  
new Thread(sh::down).start();  
new Thread(sh::up).start();
```

| поток 1           | counter | поток 2           |
|-------------------|---------|-------------------|
| load counter (1)  | 1       |                   |
|                   | 1       | load counter (1)  |
|                   | 1       | add 1 (2)         |
|                   | 2       | store counter (2) |
| sub 1 (0)         | 2       |                   |
| store counter (0) | 0       |                   |

++    --  
1) load counter  
2) add 1 (sub 1)  
3) store counter

# Гонки (race condition)

```
class Shared {  
    int counter = 1;  
    synchronized void up()    { counter++; }  
    synchronized void down() { counter--; }  
}  
sh = new Shared();  
new Thread(sh::down).start();  
new Thread(sh::up).start();
```

# Гонки (race condition)

```
class Shared {  
    int counter = 1;  
    synchronized void up()    { counter++; }  
    synchronized void down() { counter--; }  
}  
sh = new Shared();  
new Thread(sh::down).start();  
new Thread(sh::up).start();
```

| поток 1           | counter | поток 2           |
|-------------------|---------|-------------------|
| load counter (1)  | 1       |                   |
| sub 1 (0)         | 0       |                   |
| store counter (0) | 0       |                   |
|                   | 0       | load counter (0)  |
|                   | 1       | add 1 (1)         |
|                   | 1       | store counter (1) |

- ✓ Защита критической секции от выполнения двумя потоками
- ✓ Любой объект имеет встроенную блокировку (intrinsic lock)
- ✓ При входе в критическую секцию поток захватывает блокировку
- ✓ При выходе из критической секции поток освобождает блокировку
- ✓ Блокировка реентерабельна — не блокирует себя
- ✓ Остальные потоки ждут в очереди

- ✓ Защита критической секции от выполнения двумя потоками
- ✓ Любой объект имеет встроенную блокировку (intrinsic lock)
- ✓ При входе в критическую секцию поток захватывает блокировку
- ✓ При выходе из критической секции поток освобождает блокировку
- ✓ Блокировка реентерабельна — не блокирует себя
- ✓ Остальные потоки ждут в очереди
- ✓ **synchronized** — защита метода или блока
  - обычный метод — по объекту, у которого вызван метод
  - статический метод — по объекту класса Class
  - по явно указанному объекту для synchronized блока

```
public class MyClass {  
    Object lock = new Object(),  
    public synchronized void add() { }  
    public synchronized void rem() { }  
    public static synchronized int min() { }  
    public static synchronized int max() { }  
    public void x() { ... synchronized (lock) { } ... }  
    public void y() { ... synchronized (this) { } ... }  
}  
MyClass m = new MyClass();  
m.add(); // один поток начал выполнять m.add()  
// другие потоки не могут вызвать m.add(), m.rem()  
// и войти в синхронизированный блок внутри метода m.y()
```

```
public class MyClass {  
    Object lock = new Object(),  
    public synchronized void add() { }  
    public synchronized void rem() { }  
    public static synchronized int min() { }  
    public static synchronized int max() { }  
    public void x() { ... synchronized (lock) { } ... }  
    public void y() { ... synchronized (this) { } ... }  
}  
MyClass m = new MyClass();  
MyClass.min(); // один поток начал выполнять min()  
// другие потоки не могут вызвать min(), max()
```

```
public class MyClass {
    Object lock = new Object(),
    public synchronized void add() { }
    public synchronized void rem() { }
    public static synchronized int min() { }
    public static synchronized int max() { }
    public void x() { ... synchronized (lock) { } ... }
    public void y() { ... synchronized (this) { } ... }
}
MyClass m = new MyClass();
m.x(); // один поток вошел в блок внутри метода x()
// другие потоки не могут войти в любой блок,
// синхронизированный по объекту lock
```

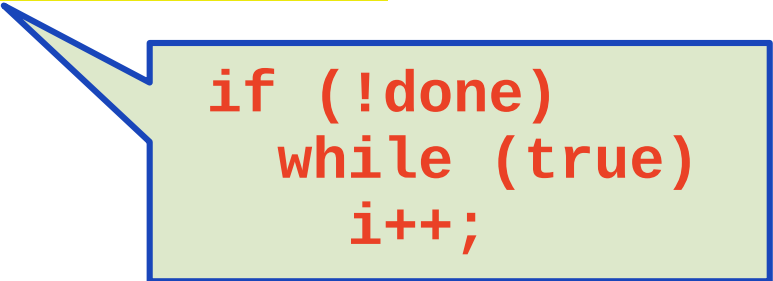
- ☑ Процессор может сохранять значения переменных в локальном кэше для повышения производительности
- ☑ Разные потоки могут видеть разные значения переменных

- ☑ Компилятор и JVM могут оптимизировать порядок выполнения инструкций

```
(new Thread(() -> { while (!done) i++; })).start();  
Thread.sleep(1000);  
done = true;    // первый поток остановится через 1 с
```

- ☑ Компилятор и JVM могут оптимизировать порядок выполнения инструкций

```
(new Thread(() -> { while (!done) i++; })).start();  
Thread.sleep(1000);  
done = true;
```



```
if (!done)  
while (true)  
    i++;
```

- ☑ Компилятор и JVM могут оптимизировать порядок выполнения инструкций

```
a = 0, b = 0;  
x = 0, y = 0;  
m1() { b = 1; x = a; }  
m2() { a = 2; y = b; }
```

- ☑ Компилятор и JVM могут оптимизировать порядок выполнения инструкций

```
a = 0, b = 0;  
x = 0, y = 0;  
m1() { b = 1; x = a; }  
m2() { a = 2; y = b; }
```

```
// x = 2; y = 0;
```

- ☑ Компилятор и JVM могут оптимизировать порядок выполнения инструкций

```
a = 0, b = 0;  
x = 0, y = 0;  
m1() { b = 1; x = a; }  
m2() { a = 2; y = b; }
```

```
// x = 2; y = 0;  
// x = 0; y = 1;
```

- ☑ Компилятор и JVM могут оптимизировать порядок выполнения инструкций

```
a = 0, b = 0;  
x = 0, y = 0;  
m1() { b = 1; x = a; }  
m2() { a = 2; y = b; }
```

```
// x = 2; y = 0;  
// x = 0; y = 1;  
// x = 2; y = 1;
```

- ☑ Компилятор и JVM могут оптимизировать порядок выполнения инструкций

```
a = 0, b = 0;
```

```
x = 0, y = 0;
```

```
m1() { b = 1; x = a; }
```

```
m2() { a = 2; y = b; }
```

**x = a; b = 1;**

**y = b; a = 2;**

```
// x = 2; y = 0;
```

```
// x = 0; y = 1;
```

```
// x = 2; y = 1;
```

- ☑ Компилятор и JVM могут оптимизировать порядок выполнения инструкций

```
a = 0, b = 0;
```

```
x = 0, y = 0;
```

```
m1() { b = 1; x = a; }
```

```
m2() { a = 2; y = b; }
```

**x = a; b = 1;**

**y = b; a = 2;**

```
// x = 2; y = 0;
```

```
// x = 0; y = 1;
```

```
// x = 2; y = 1;
```

```
// x = 0; y = 0;
```

- ☑ Модификатор `volatile` — переменная может быть изменена не в текущем потоке
- ☑ **Операции чтения-записи** переменной с модификатором `volatile` должны выполняться **без использования кэша**

- ☑ Модификатор volatile — переменная может быть изменена не в текущем потоке
- ☑ **Операции чтения-записи** переменной с модификатором volatile должны выполняться **без использования кэша**
- ☑ **Порядок операций** чтения-записи переменной с модификатором volatile **не должен меняться** — должно соблюдаться ограничение «**happens-before**»
  - **Запись** volatile переменной должна выполняться **ПОСЛЕ** предшествующих операций чтения и записи других переменных
  - **Чтение** volatile переменной должно выполняться **ДО** последующих операций чтения и записи других переменных

- ☑ У каждого потока есть стек, где хранятся локальные переменные и параметры методов
- ☑ Примитивные локальные переменные потокобезопасны
- ☑ Переменные класса — общие для всех потоков
- ☑ ThreadLocal — потокобезопасные переменные

```
private ThreadLocal<String> tl = new ThreadLocal();  
tl.set("local");  
String s = tl.get();
```

## ☑ Вариант 1 — общая переменная и флаг

```
class Block {  
    volatile boolean ready;  
    int value;  
  
    void put(int i) {  
        while (ready);  
        synchronized(this) {  
            value = i;  
            ready = true;  
        }  
    }  
    int get() {  
        while (!ready);  
        synchronized(this) {  
            ready = false;  
            return value;  
        }  
    }  
}
```

активное  
ожидание

```
Block g = new Block();
```

```
Thread t1 = new Thread(() -> {  
    g.put(100);  
});
```

```
Thread t2 = new Thread(() -> {  
    System.out.println(g.get());  
});
```

```
t1.start();  
t2.start();
```

☑ Методы `wait()`, `notify()`, `notifyAll()` вызываются **только после захвата блокировки**

☑ `wait()`

- поток помещается в очередь ожидания (wait set) объекта
- поток освобождает блокировку и ждет:
  - ◆ сигнал `notify`
  - ◆ прерывание
  - ◆ окончание времени ожидания
- поток получает блокировку и завершает метод `wait()`

☑ `notify()` - выводит из очереди ожидания один из потоков.



☑ `notifyAll()` - выводит из очереди ожидания все потоки.

## ☑ Вариант 1 — wait / notify

```
class Block {  
    volatile boolean ready;  
    int value;  
  
    synchronized void put(int i) {  
        while (ready) wait();  
        value = i;  
        ready = true;  
        notifyAll();  
    }  
    synchronized int get() {  
        while (!ready) wait();  
        ready = false;  
        notifyall();  
        return value;  
    }  
}
```

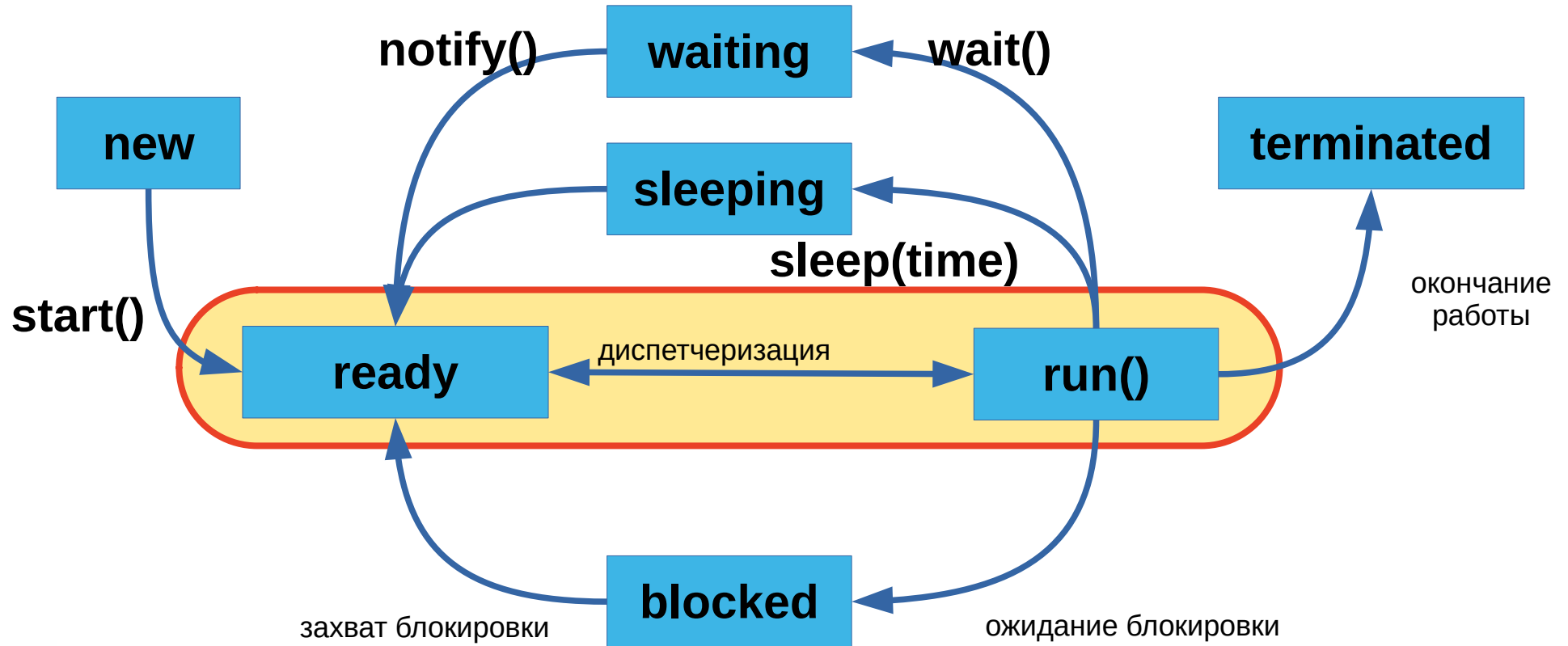
ожидание  
в очереди

```
Block g = new Block();
```

```
Thread t1 = new Thread(() -> {  
    g.put(100);  
});
```

```
Thread t2 = new Thread(() -> {  
    System.out.println(g.get());  
});
```

```
t1.start();  
t2.start();
```





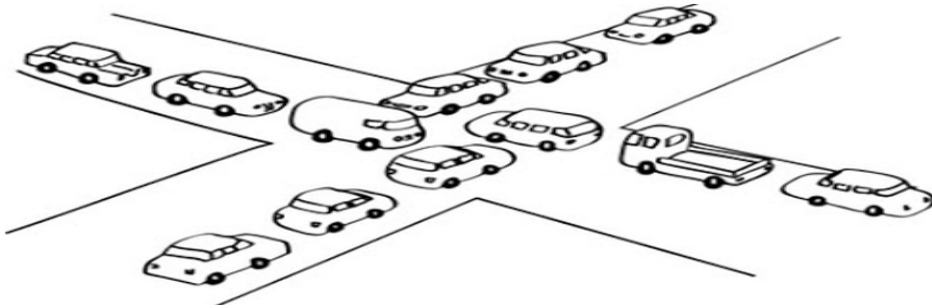
УНИВЕРСИТЕТ ИТМО

Программирование. 2 семестр

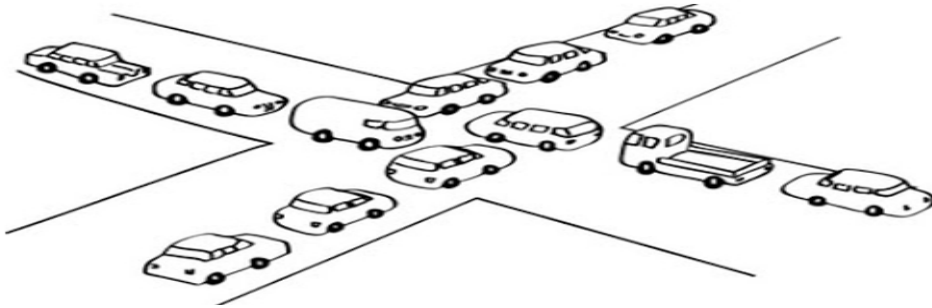
Многопоточность.  
java.concurrent.\*



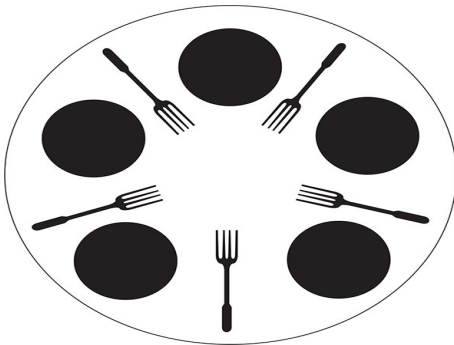
## ☑ Взаимная блокировка (deadlock)



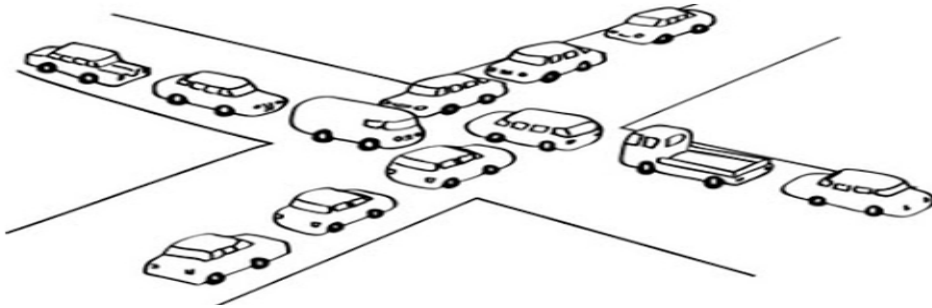
- ✓ Взаимная блокировка (deadlock)



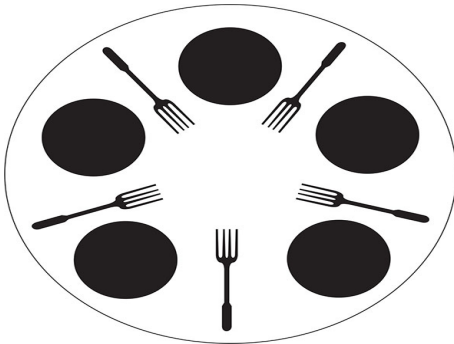
- ✓ Обедающие философы



✓ Взаимная блокировка (deadlock)



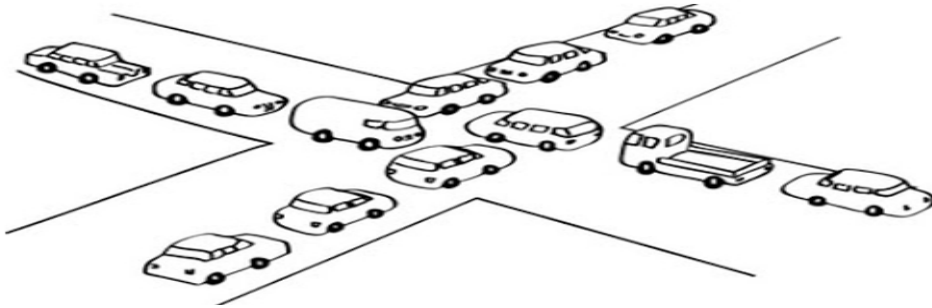
✓ Обедающие философы



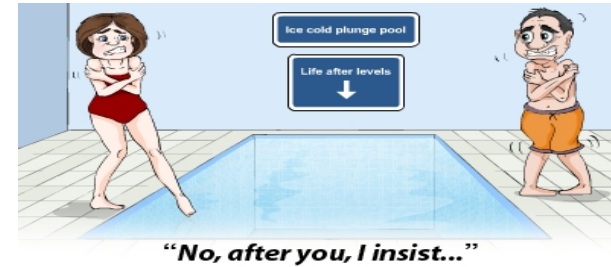
✓ Голодание (starvation)



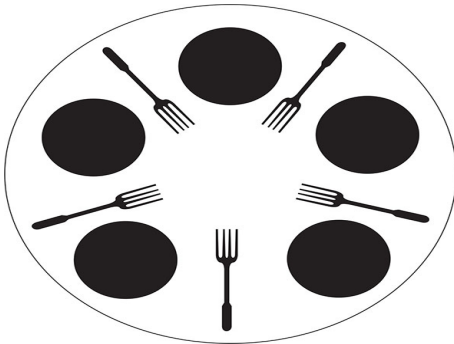
✓ Взаимная блокировка (deadlock)



✓ Зацикливание (livelock)



✓ Обедающие философы



✓ Голодание (starvation)



- ☑ Неизменяемый объект — нет проблем многопоточности
  - Убрать сеттеры
  - Все поля — `private final`
  - Все методы — `final`
  - Не сохранять ссылки на изменяемые объекты — сохранять копии объектов

## ☑ java.util.concurrent

- интерфейсы Executor, Callable, Future
- классы ThreadPoolExecutor, ForkJoinPool
- классы-синхронизаторы
- интерфейсы BlockingQueue, TransferQueue
- коллекции ConcurrentX и CopyOnWriteArrayX

## ☑ java.util.concurrent.locks

- интерфейсы Lock, Condition

## ☑ java.util.concurrent.atomic

- AtomicInteger, AtomicLong, AtomicReference

- ✓ interface **Executor**
- ✓ Thread — абстракция потока
- ✓ Executor — абстракция исполнителя
- ✓ void execute(**Runnable** task)
  - выполнить задачу в какой-то момент времени в будущем
    - ◆ возможно сразу
    - ◆ возможно в новом потоке

```
Executor executor = ...;  
executor.execute(task1);  
executor.execute(task2);
```

```
(new Thread(task1)).start();  
(new Thread(task2)).start();
```

- ☑ interface **ExecutorService** extends **Executor**
  - **Future<T>** submit(**Callable<T>** task)
  - List<Future<T>> invokeAll(Collection<Callable<T>> tasks)
  - void shutdown()
- ☑ interface **Callable<T>**
  - T call()
- ☑ interface **Future<T>**
  - T get()
  - boolean isDone()
  - boolean cancel()
- ☑ interface **ScheduledExecutorService** extends **ExecutorService**
  - **ScheduledFuture** schedule(task, delay, unit)
  - **ScheduledFuture** scheduleAtFixedRate(task, initial, period, unit)
  - **ScheduledFuture** scheduleAtFixedDelay(task, initial, delay, unit)

```
String search(String f, String text) { }  
s = "find";  
text = "very long text";  
ExecutorService service = ...;  
Callable<String> task = () -> search(s, text);  
Future<String> future = service.submit(task);  
// while(!future.isDone()) { ... }  
String searchResult = future.get();
```

- ☑ Создание потока требует ресурсов и времени
- ☑ Потоки в пуле повторно используются по мере освобождения
- ☑ Постепенная деградация при увеличении нагрузки

- ☑ class ThreadPoolExecutor implements ExecutorService
- ☑ class ScheduledThreadPoolExecutor implements ScheduledThreadPoolExecutor
- ☑ класс Executors — статические методы
  - ExecutorService new**SingleThread**Executor()
  - ScheduledExecutorService newSingleThreadScheduledExecutor()
  - ExecutorService new**FixedThreadPool**()
  - ExecutorService new**CachedThreadPool**()
  - ScheduledExecutorService newScheduledThreadPool()

- ✓ Реализация параллельного программирования
- ✓ Стратегия «разделяй и властвуй» (divide and conquer)
- ✓ Алгоритм «перехват работы» (work stealing)

```
if (size(task) = small) {  
    return result  
} else {  
    foreach (subtask[i] : task) {  
        result[i] = execute(subtask[i])  
    }  
    return merge(result)
```

## ☑ class ForkJoinPool

- ForkJoinPool.commonPool()
- ForkJoinTask<V> fork()

## ☑ class ForkJoinTask

- V join()
- V invoke(ForkJoinTask<V>)
- invokeAll(ForkJoinTask... tasks)
- class RecursiveAction extends ForkJoinTask
  - ◆ abstract void compute()
- class RecursiveTask extends ForkJoinTask
  - abstract V compute()

```
public class DoubleTask {
    final int[] array;
    final int lo, hi;
    final static int SMALL_SIZE = 10;

    DoubleTask(int[] array, int lo, int hi) {
        this.array = array; this.lo = lo, this.hi = hi;
    }

    protected void compute() {
        if ((hi - lo) < SMALL_SIZE) {
            for (int i = lo; i < hi; i++) array[i] *= 2;
        } else {
            int mid = (lo + hi) / 2;
            DoubleTask dt1 = new DoubleTask(array, lo, mid);
            DoubleTask dt2 = new DoubleTask(array, mid, hi);
            invokeAll(dt1, dt2);
        }
    }
}
```

```
int[] arr = {0, ... , 33554431};  
DoubleTask dt = new DoubleTask(arr, 0, arr.length - 1);  
ForkJoinPool pool = ForkJoinPool.commonPool();  
pool.invoke(dt);
```

☑ interface Lock — аналог synchronized

- lock()
- unlock()
- tryLock()
- tryLock(long time)
- lockInterruptibly()
- Condition newCondition()

☑ interface Condition — аналог wait-notify

- await()
- signal()
- signalAll()

☑ class ReentrantLock implements Lock

```
int value;  
Lock lock = new ReentrantLock();
```

```
void put(int i) {  
    lock.lock();  
    try { value = i; }  
    finally { lock.unlock(); }  
}
```

```
int get() {  
    lock.lock();  
    try { return value; }  
    finally { lock.unlock(); }
```

☑ class ReentrantLock implements Lock

```
boolean right = rightFork.tryLock();
try {
    if (right) {
        boolean left = leftFork.tryLock();
        try {
            if (left) {
                eat();
            }
        } finally { leftFork.unlock(); }
    }
} finally { rightFork.unlock(); }
```

## ☑ interface ReadWriteLock

- Lock readLock()
  - ◆ возвращает Lock для операций чтения (множественный доступ)
- Lock writeLock()
  - ◆ возвращает Lock для операций записи (блокирующий доступ)

## ☑ class ReentrantReadWriteLock

# Интерфейс Condition

```
Lock lock = new ReentrantLock();
Condition notFull = lock.newCondition();
Condition notEmpty = lock.newCondition();
int[] values = new int[100]; int count;

public void put(int i) {
    lock.lock();
    try {
        while(count == values.length) { notFull.await(); }
        values[count++] = i;
        notEmpty.signal();
    } finally { lock.unlock(); }
}

public int get() {
    lock.lock();
    try {
        while(count == 0) { notEmpty.await(); }
        notFull.signal();
        return values[--count];
    } finally { lock.unlock(); }
```

- ☑ Semaphore — синхронизатор со счетчиком разрешений. Мьютекс — простейший бинарный семафор.
  - методы `acquire()`, `release()`
- ☑ CountdownLatch — синхронизатор с обратным счетчиком ожидает определенное количество событий
  - методы — `await()`, `countDown()`
- ☑ CyclicBarrier — синхронизатор, который синхронизирует определенное количество потоков
  - метод `await()`, `reset()`
- ☑ Phaser — синхронизатор, ожидающий завершения действий, состоящих из этапов или фаз
  - методы `register()`, `arrive()`, `arriveAndAwaitAdvance()`, `arriveAndDeregister()`
- ☑ Exchanger<V> — синхронизатор обмена данными между двумя потоками
  - метод `V exchange(V data)`

- ☑ Атомарная операция — операция, выполняющаяся без промежуточных состояний.
- ☑ Атомарные операции не вызывают состояние гонок
- ☑ операции чтения-записи ссылок и примитивных типов (кроме long и double) — атомарные
- ☑ Пакет java.util.concurrent.atomic — атомарные типы

```
class A {  
    AtomicInteger counter = new AtomicInteger(0);  
    public void up() { counter.incrementAndGet(); }  
    public void down() { counter.decrementAndGet(); }  
}
```

## ☑ AtomicInteger

- get – аналог чтения volatile переменной
- set – аналог записи volatile переменной
- CAS = compareAndSwap(int expected, int updated)

```
int value;
synchronized int get() { return value; }
synchronized int compareAndSwap(int expected, int updated) {
    int old = value;
    if (old == expected) {
        value = updated;
    }
    return old;
}
synchronized boolean compareAndSet(int expected, int updated) {
    return (expected == compareAndSwap(expected, updated));
}
```

## ☑ AtomicInteger

- get – аналог чтения volatile переменной
- set – аналог записи volatile переменной
- compareAndSet(int expect, int update)

```
int operation(int arg) { }  
int operationAndGet(AtomicInteger x) {  
    int prev, next;  
    do {  
        prev = x.get();  
        next = operation(prev);  
    } while (!x.compareAndSet(prev, next));  
    return next;  
}
```

## ☑ BlockingQueue / BlockingDeque

- put(E) — поместить элемент в очередь, ожидая, пока появится свободное место
- E take() — получить элемент из очереди, ожидая, пока в ней появятся элементы
- для Deque — putFirst, putLast, takeFirst, takeLast
- ArrayBlockingQueue, LinkedBlockingQueue, PriorityBlockingQueue
- DelayQueue — элементы доступны после задержки
- SynchronousQueue — синхронное добавление-получение

## ☑ TransferQueue extends BlockingQueue

- transfer(E) — дождаться получения элемента
- LinkedTransferQueue

- ✓ Синхронизированные коллекции — используют блокировки
- ✓ Конкурентные коллекции — оптимизированные алгоритмы для многопоточной работы
- ✓ ConcurrentMap / ConcurrentNavigableMap
  - атомарные операции — методы putIfAbsent, remove, replace
  - ConcurrentHashMap,
  - ConcurrentSkipListMap, ConcurrentSkipListSet
- ✓ ConcurrentLinkedQueue
  - потокобезопасная очередь
- ✓ CopyOnWriteArrayList / CopyOnWriteArraySet
  - операции, изменяющие коллекцию, создают новую копию.
  - операции чтения, а также итераторы продолжают работать со старой копией.

- ☑ java.nio — асинхронные каналы — Future<V>
- ☑ java.util.stream — Spliterator, parallelStream()
- ☑ лабы
  - 7 - многопоточный сервер
    - ◆ каждый клиент - отдельный поток на сервере
  - 8 - анимация в GUI