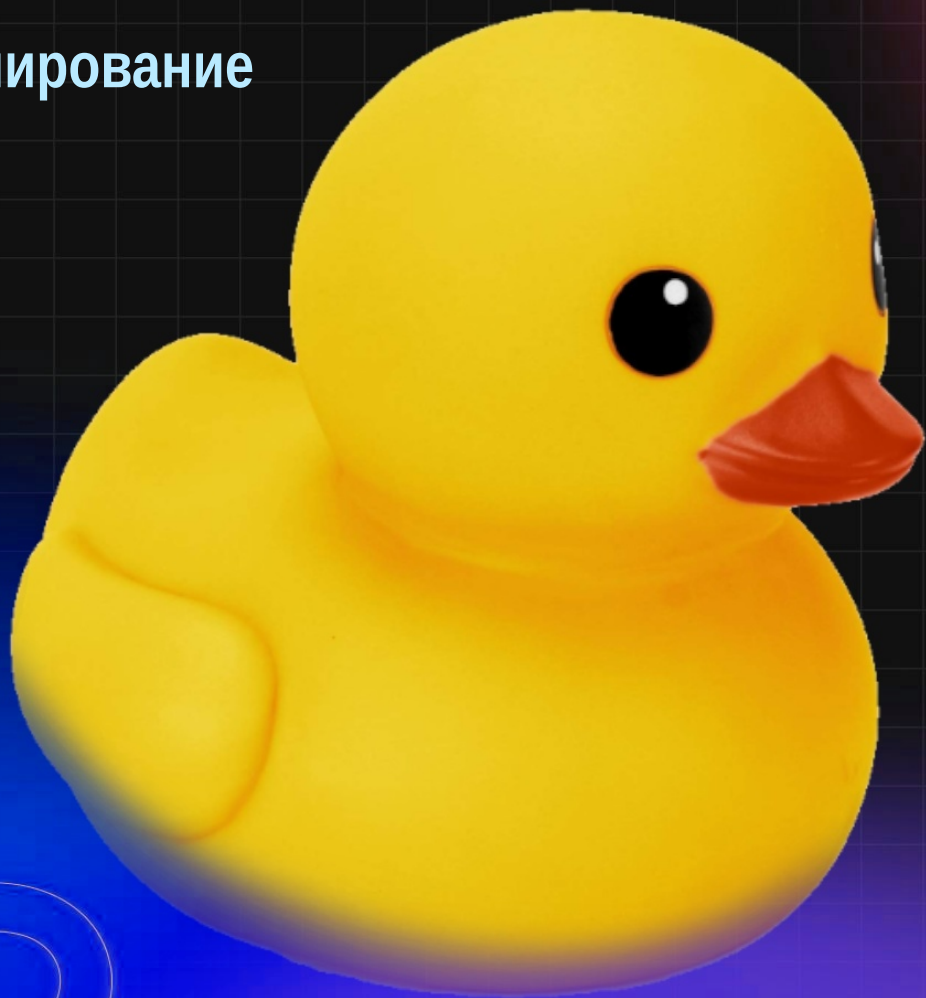


Программирование
2 семестр
2025



ІТМО

Дата и время
Date/Time API

- Дата и время — 2 варианта представления:
 - ❖ Человеческое время — часы, минуты, дни, недели, месяцы
 - ❖ Машинное время — миллисекунды от точки отсчета
 - EPOCH - 1 января 1970 года, 00:00:00 (Java, UNIX)
 - Windows - 1 января 1601 года
 - ❖ Y2K problem (2000)
 - ❖ Y 2038 problem
 - 19-01-2038 03:14:07 + 1 sec = 13-12-1901 20:45:52

- Date 1.0

- ❖ все действия с датой
- ❖ человеческое и машинное представление
- ❖ форматирование даты

- Конструкторы

- ❖ Date
- ❖ Date(long)

- Date 1.1

- ❖ только момент времени
- ❖ почти все методы - deprecated

- Методы

- ❖ long getTime()
- ❖ boolean after(Date)
- ❖ boolean before(Date)

- Временная зона — смещение от стандартного:
- до 1972 года - Гринвич (GMT)
- после 1972 — UTC — всемирное координированное
- Методы
 - ❖ `getDefault()`
 - ❖ `getAvailableIDs()`
 - ❖ `getRawOffset()` - смещение без учета летнего времени
 - ❖ `getOffset(long date)` — с учетом летнего времени
- Класс `SimpleTimeZone` — реализованный потомок

- Абстрактный класс — машинное ↔ человеческое
 - ❖ `Calendar getInstance()`
 - ❖ `add(int field, int amount);`
 - ❖ `roll(int field, int amount);`
 - ❖ `set(int field, int value);`
 - ❖ `Date getTime()`
 - ❖ `setTime(Date)`
- класс `GregorianCalendar`
 - ❖ сочетает 2 календаря (григорианский и юлианский)

- `java.time` - дата, время, периоды
 - ❖ `Instant`, `Duration`, `Period`, `LocalDate`, `LocalTime`, `LocalDateTime`, `OffsetTime`, `OffsetDateTime`, `ZonedDateTime`
- `java.time.chrono` - календарные системы
- `java.time.format` - форматирование даты и времени
- `java.time.temporal` - единицы измерения и отдельные поля
- `java.time.zone` - временные зоны и правила

- enum DayOfWeek (1 (MONDAY) — 7 (SUNDAY))
- enum Month (1 (JANUARY) — 12 (DECEMBER))
- метод `getDisplayName(style, locale)`
- стиль — FULL, NARROW, SHORT / STANDALONE

- Year
- YearMonth
- MonthDay
- LocalDate
- LocalTime
- LocalDateTime

- Статические

- ❖ of — создает экземпляр на основе входных параметров
 - `LocalDate.of(year, month, day)`, `ofYearDay(year, dayOfYear)`
- ❖ from — конвертирует экземпляр из другого типа
 - `LocalDate.from(LocalDateTime)`
- ❖ parse — создает экземпляр из строкового представления
 - `LocalDate.parse("2022-02-22")`

- Методы экземпляра

- ❖ `format` — форматирует объект в строку
- ❖ `get` — возвращает поля объекта // `getHours()`
- ❖ `with` — возвращает копию с изменением // `withYear(2021)`
- ❖ `plus` — возвращает копию с добавлением // `plusDays(2)`
- ❖ `minus` — возвращает копию с убавлением // `minusWeeks(3)`
- ❖ `to` — преобразует объект в другой тип // `toLocalTime()`
- ❖ `at` — комбинирует объект с другим // `date.atTime(time)`

- ZoneId — идентификатор зоны
 - ❖ Europe/Moscow
- ZoneOffset — разница со стандартным временем
 - ❖ UTC+01:00, GMT-2
- OffsetTime = LocalTime + ZoneOffset
- OffsetDateTime = LocalDateTime + ZoneOffset
- ZonedDateTime = LocalDateTime + ZoneId
 - ❖ использует `java.time.zone.ZoneRules`

- YearMonth - срок действия банковской карты
- MonthDay - праздники
- LocalDateTime - местное время
- OffsetDateTime - зональное время (веб, базы данных)
- ZonedDateTime - зональное время с учетом коррекций

- Класс Instant (timestamp)

- ❖ now()
- ❖ plusNanos()
- ❖ plusMillis()
- ❖ plusSeconds()
- ❖ minusNanos()
- ❖ minusMillis()
- ❖ minusSeconds()

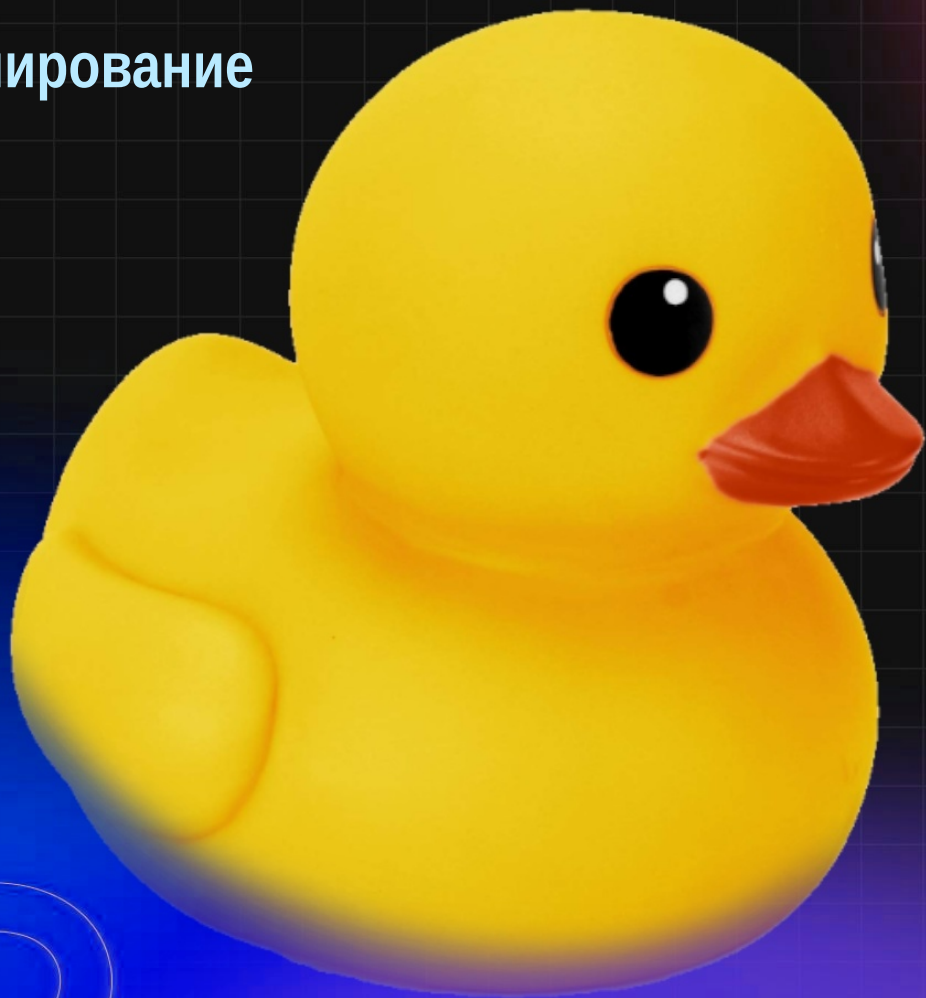
- `java.time.format.DateTimeFormatter`
 - ❖ константы формата:
 - `BASIC_ISO_DATE` ('20240229')
 - `ISO_LOCAL_DATE` ('2024-02-29')
 - `ISO_ZONED_DATETIME` ('2024-02-29T18:15:30+03:00[Europe/Moscow]')
 - `RFC_1123_DATE_TIME` ('Thu, 29 Feb 2024 18:15:30 GMT+3')
 - ❖ локализованные форматы
 - `ofLocalizedDateTime(dateStyle, timeStyle)`
 - стиль (FULL, LONG, MEDIUM, SHORT)
 - ❖ шаблон
 - `ofPattern()`
 - ❖ методы `format()` и `parse()`

- Duration — продолжительность в часах и менее
 - ❖ toNanos(), toMillis(), toSeconds(), toMinutes(), toHours(), toDays()
- Period — период в днях и более
 - ❖ getDays(), getMonths(), getYears()
- .between()
- .plus
- .minus

- **Соответствия:**
 - ❖ `java.util.Date` — `java.time.Instant`
 - ❖ `java.util.GregorianCalendar` — `java.time.ZonedDateTime`
 - ❖ `java.util.TimeZone` — `ZoneId`, `ZoneOffset`
- **Методы:**
 - ❖ `Calendar.toInstant()`
 - ❖ `GregorianCalendar.toZonedDateTime()`
 - ❖ `GregorianCalendar.fromZonedDateTime()`
 - ❖ `Date.fromInstant()`
 - ❖ `Date.toInstant()`
 - ❖ `TimeZone.toZoneId()`

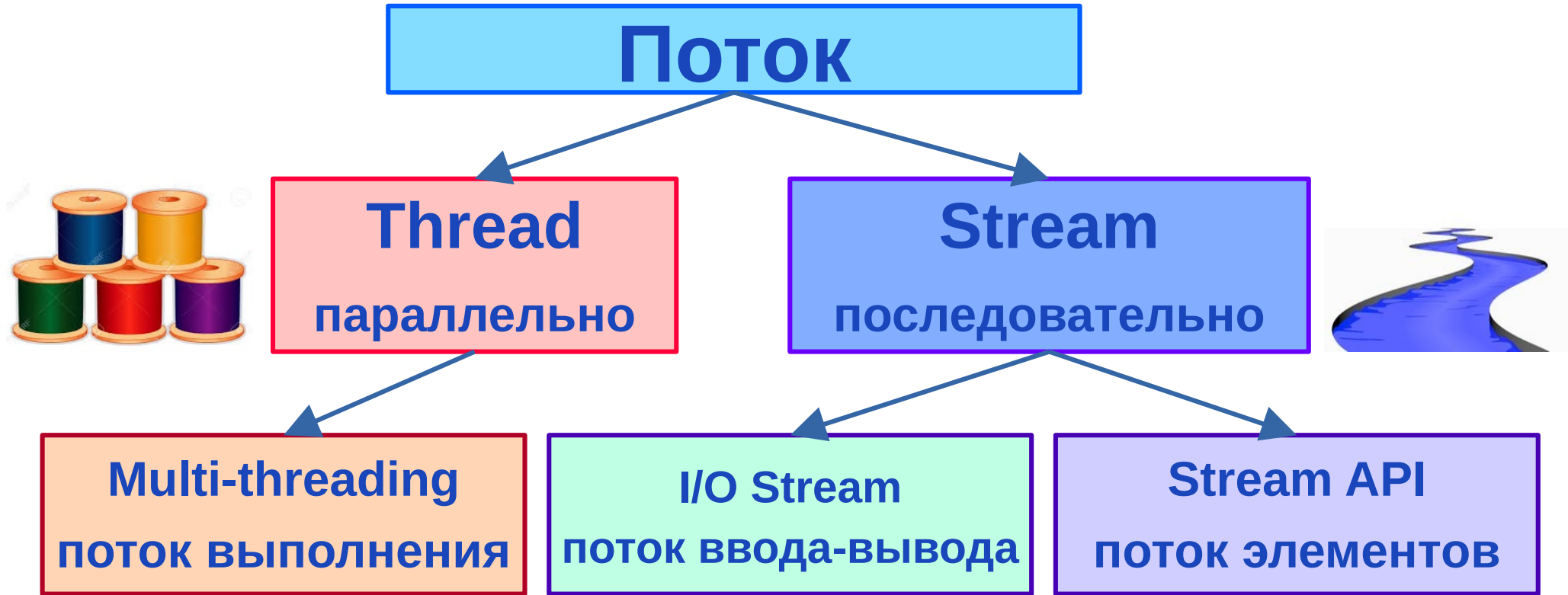

```
var date1 = LocalDate.of(2024, 03, 27);  
var date2 = LocalDate.parse("2024-04-27");  
var date3 = LocalDate.now()  
                .with(TemporalAdjusters.firstDayOfMonth());  
  
var dateTime1 = LocalDateTime.now();  
var dateTime2 = dateTime1.minusHours(2).plusMinutes(30);  
  
var zoneId = ZoneId.of("Europe/Moscow");  
var zonedDateTime = ZonedDateTime.of(dateTime, zoneId);  
  
var period = Period.between(date1, date2);  
var duration = Duration.between(dateTime1, dateTime2);
```

Программирование
2 семестр
2025



ІТМО

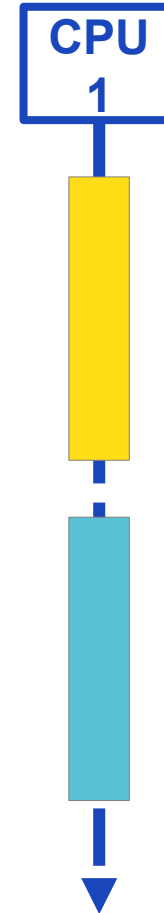
Многопоточність
Теорія



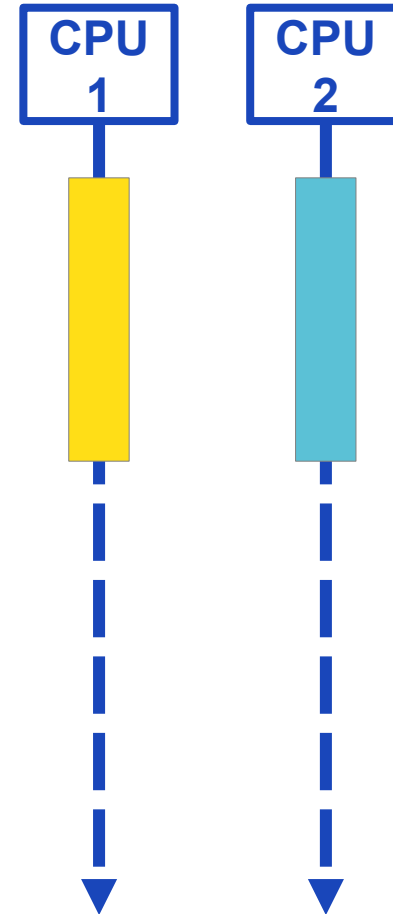
- Один процессор
- Одна задача



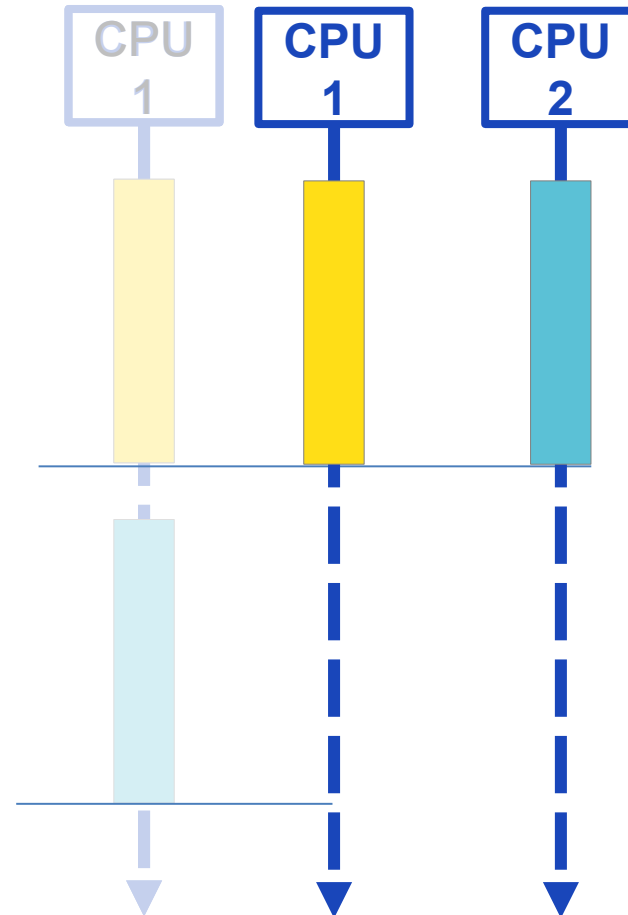
- Один процессор
- Две задачи
- Последовательное исполнение
- Простое управление



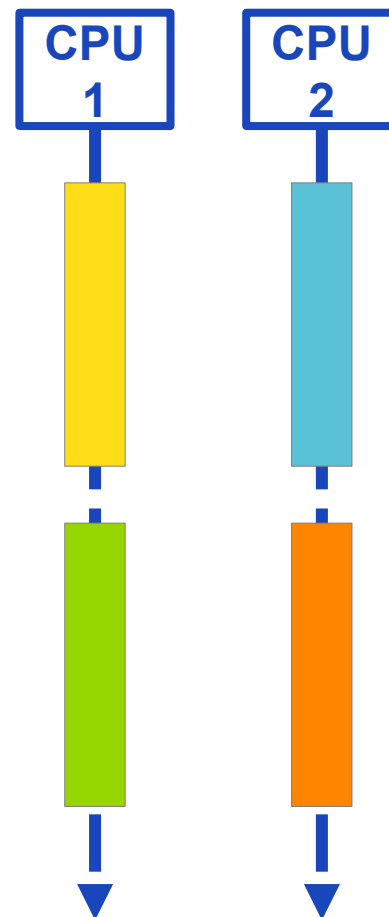
- Два процессора
- Две задачи
- Параллельное исполнение
- Более сложное управление



- Два процессора
- Две задачи
- Параллельное исполнение
- Более сложное управление
- Меньше затраты времени

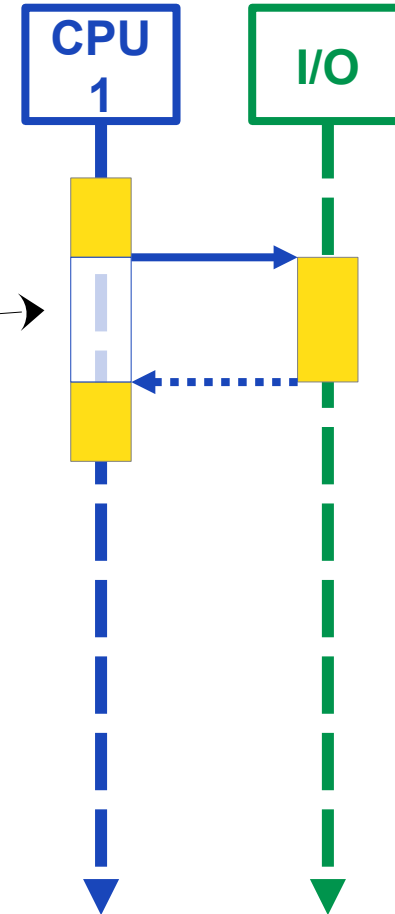


- Много процессоров
- Много задач
- Параллельное исполнение
- Высокая производительность

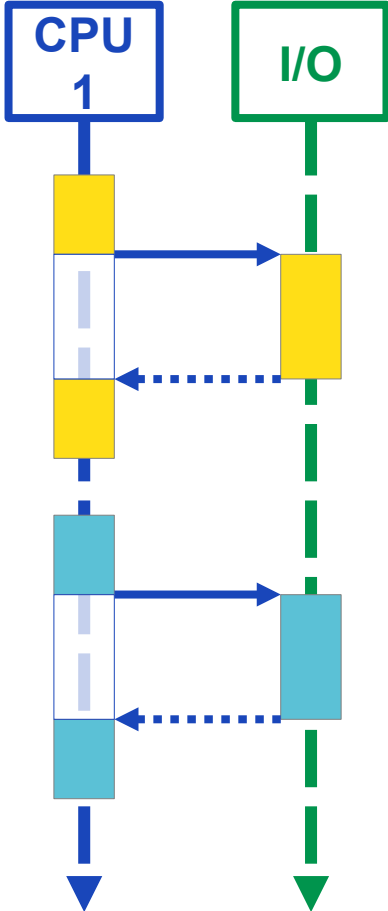


Управление задачами

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

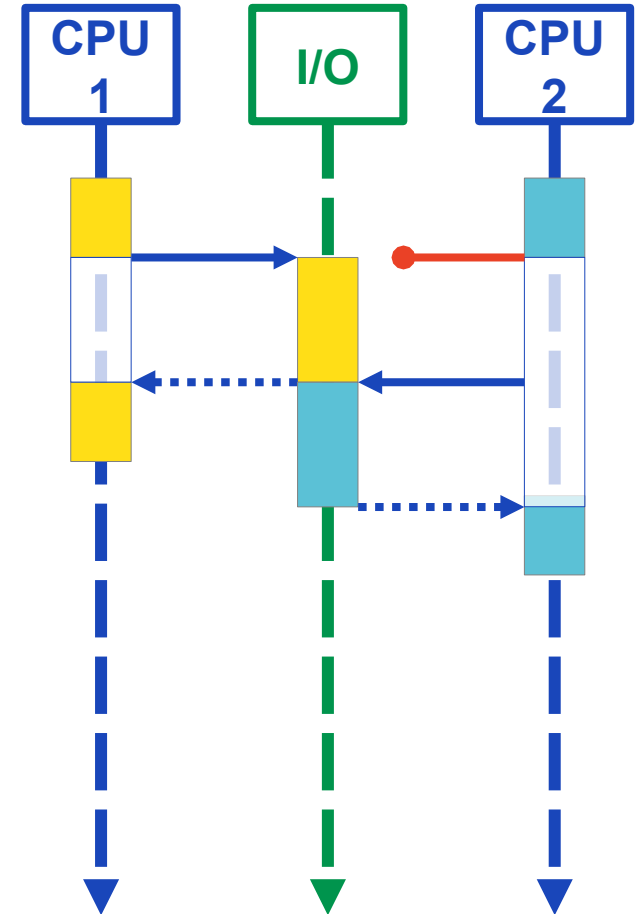


- Один процессор
- Две задачи
- Ввод-вывод
- Процессор простаивает

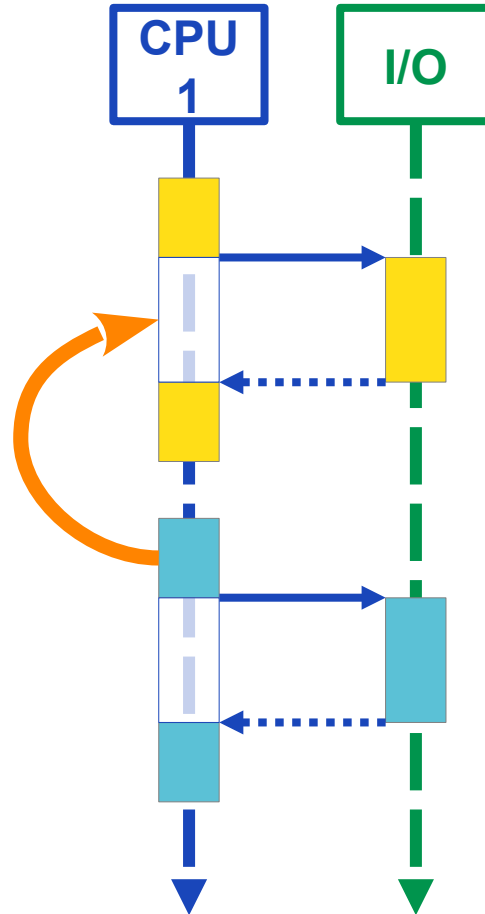


Управление задачами

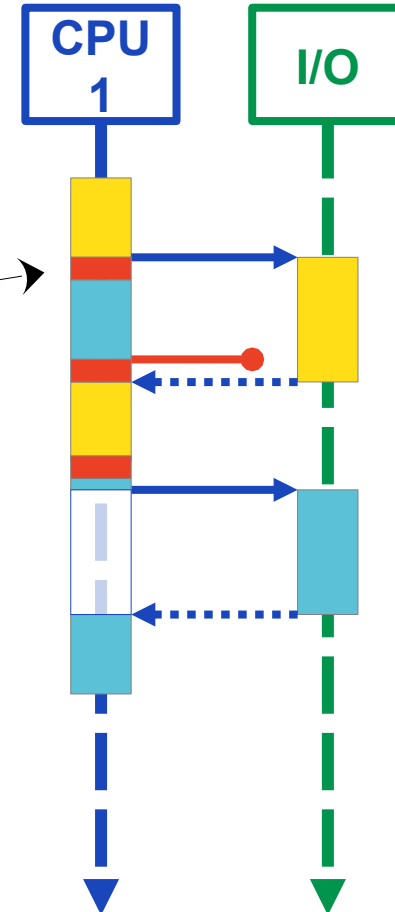
- Два процессора
- Две задачи
- Ввод-вывод
- Процессор простаивает
- Занятость ВУ



- Один процессор
- Две задачи
- Ввод-вывод
- Процессор простаивает
и это можно использовать

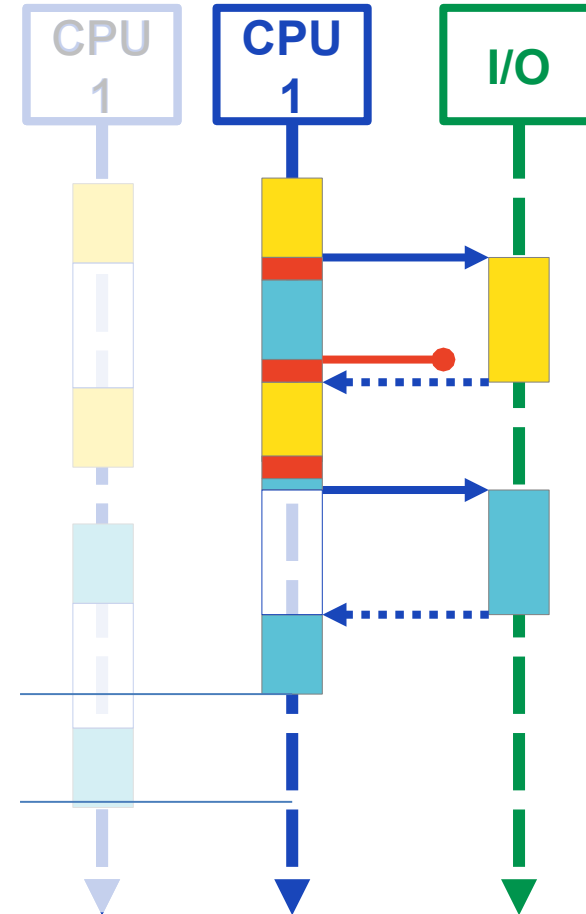


- Один процессор
- Две задачи
- Ввод-вывод
- Переключение контекста
- Конкурентное исполнение
- Более сложное управление

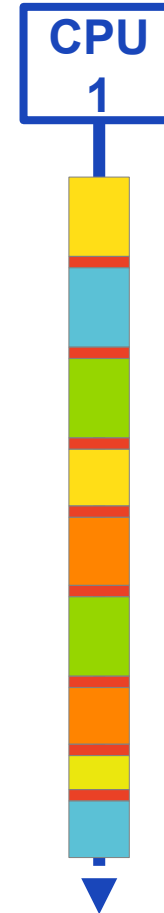


Управление задачами

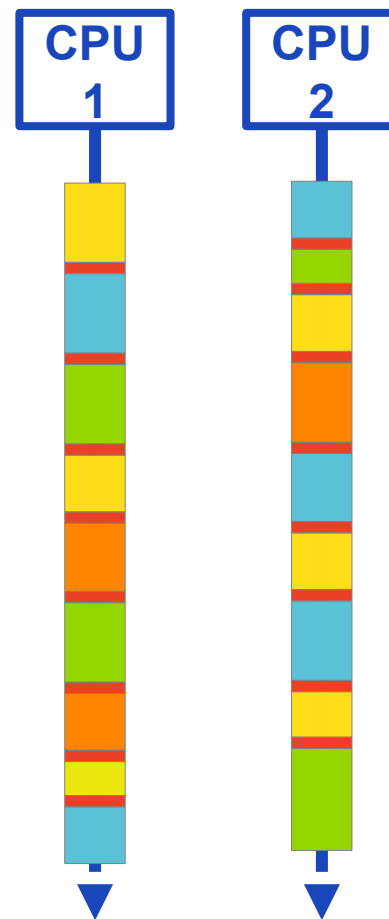
- Один процессор
- Две задачи
- Ввод-вывод
- Переключение контекста
- Конкурентное исполнение
- Более сложное управление
- Меньше затраты времени



- Один процессор
- Много задач
- Многозадачность
- Конкурентное исполнение



- Много процессоров
- Много задач
- Многозадачность
- Конкурентное исполнение
- Универсальная модель



- 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

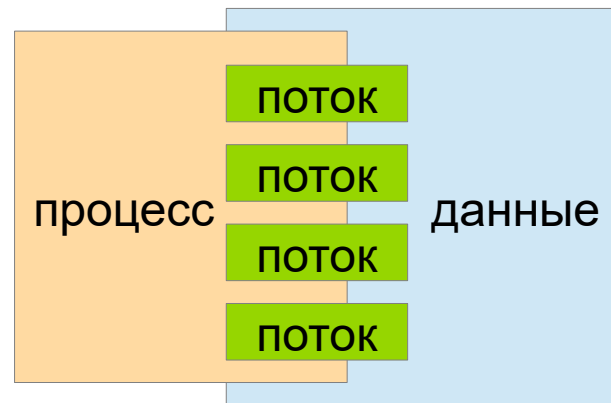
- **Многозадачность**



- **Поток (thread)**

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

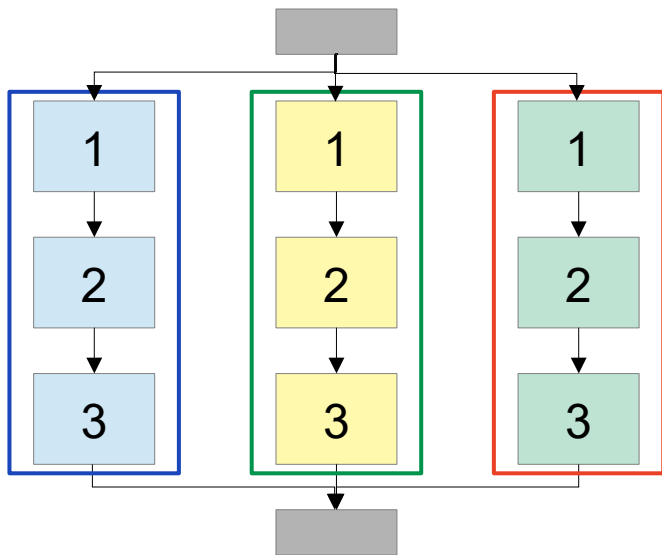
- **Многопоточность**



- Аппаратные
 - ❖ Многопроцессорность
 - ❖ Многоядерность
 - ❖ Многопоточность на уровне процессора
- Программные
 - ❖ Многозадачность
 - ❖ Многопоточность на уровне ядра ОС
 - ❖ Многопоточность на пользовательском уровне

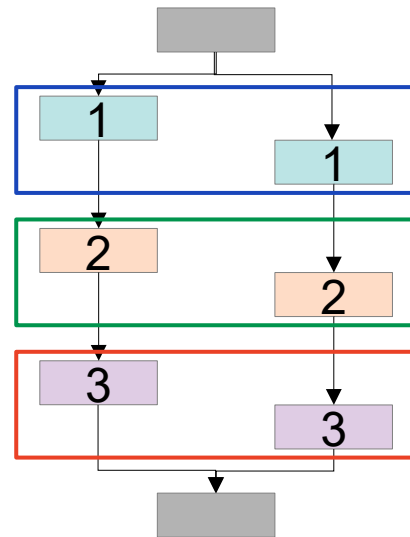
- распараллеливание

- ❖ обработчик выполняет все этапы одной задачи



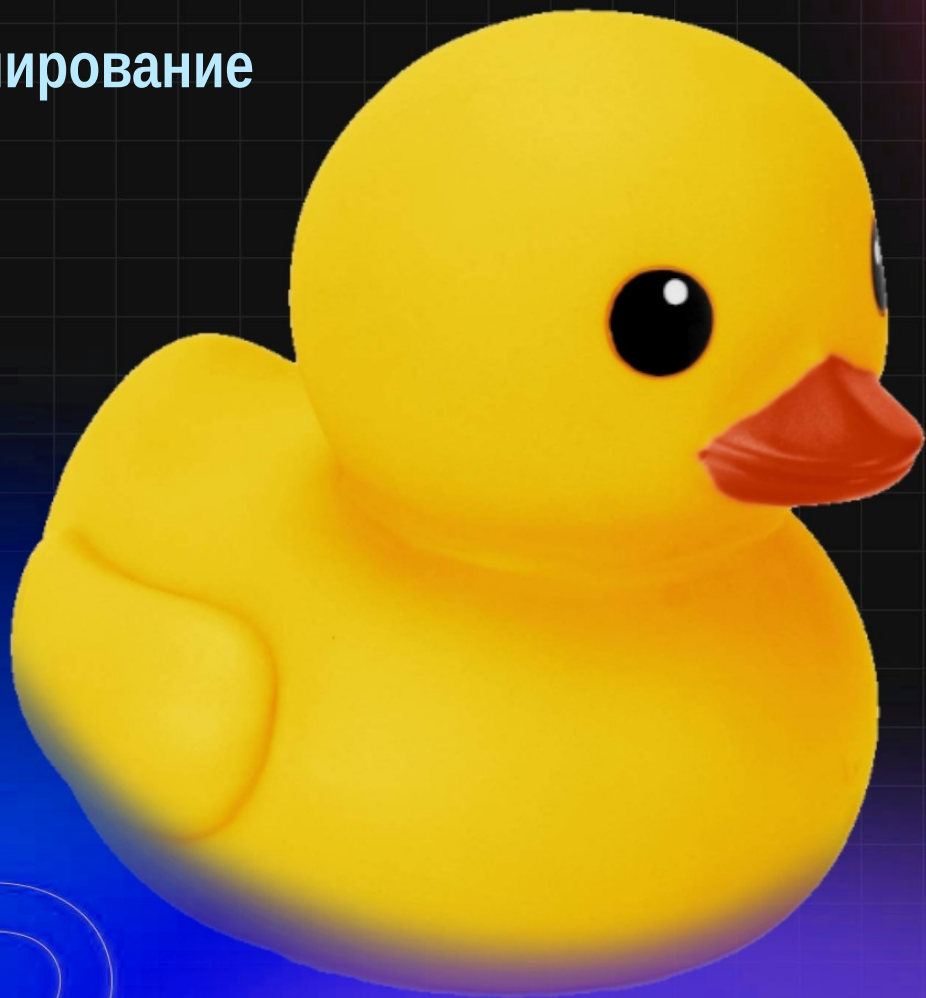
- конвейерная обработка

- ❖ обработчик выполняет один этап всех задач



Программирование
2 семестр
2025

ІТМО



Многопоточность
Java

- системные

- ❖ основной поток JVM
- ❖ сборщик мусора
- ❖ периодические задачи
- ❖ поток JIT-компиляции

- прикладные

- ❖ **основной поток (main)**
- ❖ созданные программно

- потоки Java : потоки ОС
- 1.0 - Green Threads - **N:1**
- 1.3 + - Platform (Native) Threads - **1:1**
- 21+ - Virtual Threads - **N:M**

- интерфейс **Runnable** — задача
 - ❖ **run()**
 - ❖ завершается run() — завершается задача

- интерфейс Runnable — задача
 - ❖ **run()** - код задачи
 - ❖ завершается run() — завершается задача
- класс **Thread** — исполнитель
 - ❖ **start()** - запуск задачи в отдельном потоке
 - ❖ возврат в основной поток без ожидания
 - ❖ собственный стек вызовов

- интерфейс Runnable — задача
 - ❖ **run()** - код задачи
 - ❖ завершается run() — завершается задача
- класс **Thread implements Runnable** — исполнитель
 - ❖ **start()** - запуск задачи в отдельном потоке
 - ❖ возврат в основной поток без ожидания
 - ❖ собственный стек вызовов

- интерфейс Runnable — задача
 - ❖ run() - код задачи
 - ❖ завершается run() — завершается задача
- класс Thread implements Runnable — исполнитель
 - ❖ start() - запуск задачи в отдельном потоке
 - ❖ возврат в основной поток без ожидания
 - ❖ собственный стек вызовов
- А если вызвать Thread.run() ?

```
class Task implements Runnable {  
  
}
```

- задача - Runnable

```
class Task implements Runnable {  
    public void run() {  
        /* тело потока */  
    }  
}
```

- задача - Runnable
- код задачи - run()

```
class Task implements Runnable {  
    public void run() {  
        /* тело потока */  
    }  
}  
  
new Thread(new Task())
```

- задача - Runnable
- код задачи - run()
- поток - Thread

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

- задача - Runnable
- код задачи - run()
- поток - Thread
- запуск - start()

Класс Thread и интерфейс Runnable

```
class Task implements Runnable {  
  
    public void run() {  
        /* тело потока */  
    }  
}
```

```
new Thread(new Task()).start();
```

```
class Task extends Thread {  
  
    public void run() {  
        /* тело потока */  
    }  
}
```

```
new Task().start();
```


Класс Thread и интерфейс Runnable

```
class Task implements Runnable {  
  
    public void run() {  
        /* тело потока */  
    }  
}
```

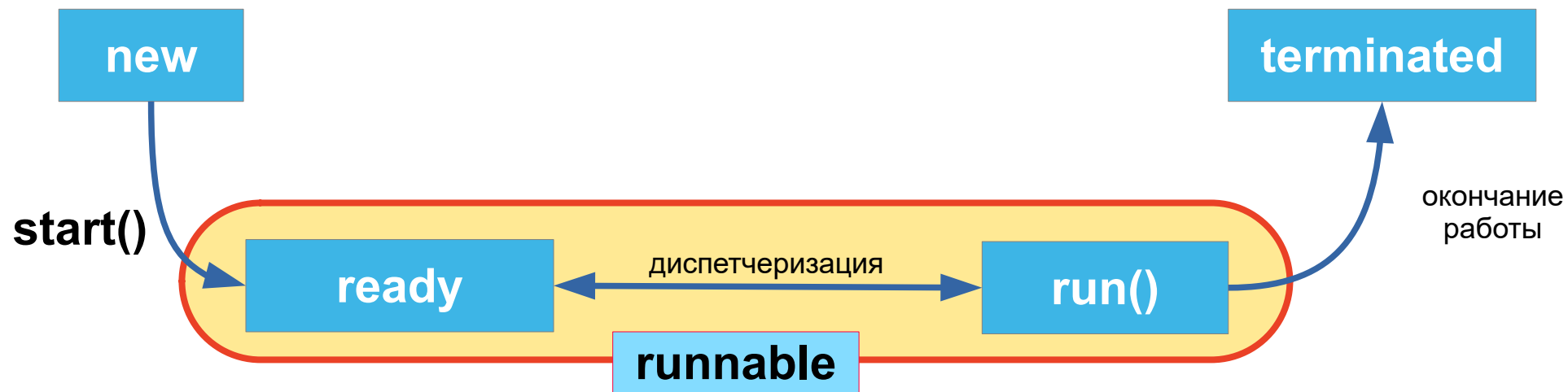
```
new Thread(new Task()).start();
```

```
class Task extends Thread {  
  
    public void run() {  
        /* тело потока */  
    }  
}
```

```
new Task().start();
```

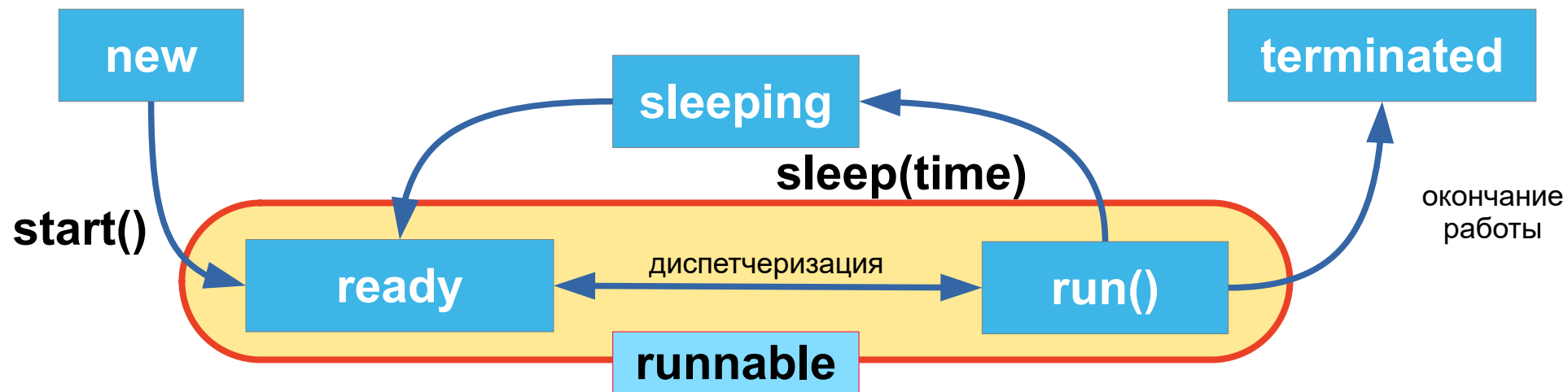
```
new Thread( () -> { /* тело потока */ } ).start();
```





- `static Thread.currentThread()`
- `getID()`
- `getName() / setName()`
- `getPriority / setPriority()`
- `getState()`
- `isAlive()`
- `isDaemon() / setDaemon()` - до вызова `start()`

- `Thread.sleep(long millis)` // спать
- `t.join()` // ждать завершения `t` и продолжить работу
- `yield()` // дать выполниться другим потокам



- `t.interrupt()` - устанавливает флаг прерывания потока `t`
 - ❖ проверка флага
 - `Thread.interrupted()` - со сбросом флага
 - `isInterrupted()` - без сброса флага
 - ❖ флаг можно игнорировать
- методы `sleep`, `join`, `wait` бросают `InterruptedException`
 - ❖ можно обработать в блоке `catch`
 - ❖ можно пробросить

- **завершение метода run()**
- прерывание с помощью interrupt() и завершение run()
- ~~методы Thread.stop() / suspend() / resume()~~
- выключение JVM (System.exit() / Ctrl-C)
 - ❖ хуки - Runtime.getRuntime().addShutdownHook (Thread hook)
 - ❖ демоны - setDaemon(true); start()
 - ❖ Object.finalize()


```
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
```

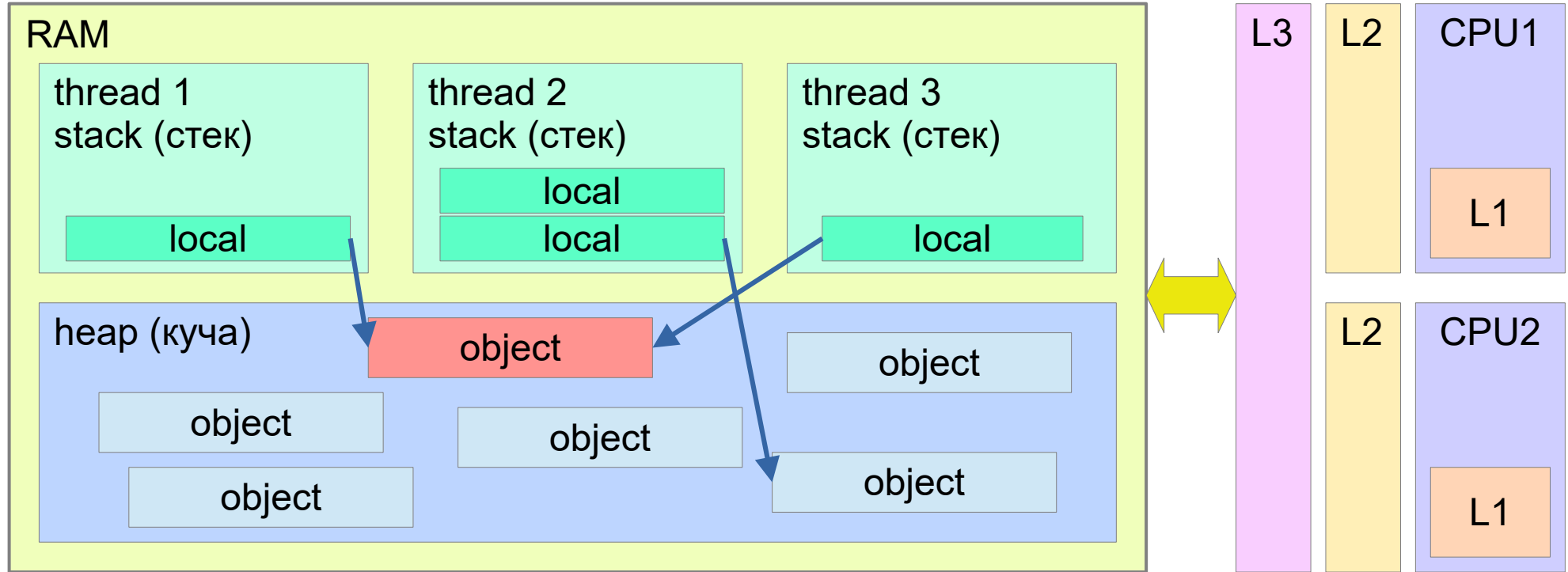
Результат всегда разный

```
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
```

● Параллельное выполнение ➡ Недетерминированность

JMM - Java Memory Model



Гонки (race condition)

```
class Shared {  
    int counter = 0;  
    void up()    { counter++; }  
    void down() { counter--; }  
}
```

Гонки (race condition)

```
class Shared {  
    int counter = 0;  
    void up()    { counter++; }  
    void down() { counter--; }  
}
```

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

Гонки (race condition)

```
class Shared {  
    int counter = 0;  
    void up()    { counter++; }  
    void down() { counter--; }  
}
```

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

++ --

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

Гонки (race condition)

```
class Shared {  
    int counter = 0;  
    void up()    { counter++; }  
    void down() { counter--; }  
}
```

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

**++ и --
не атомарные**

++ --

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

Гонки (race condition)

```
class Shared {  
    int counter = 0;  
    void up()    { counter++; }  
    void down() { counter--; }  
}
```

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

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

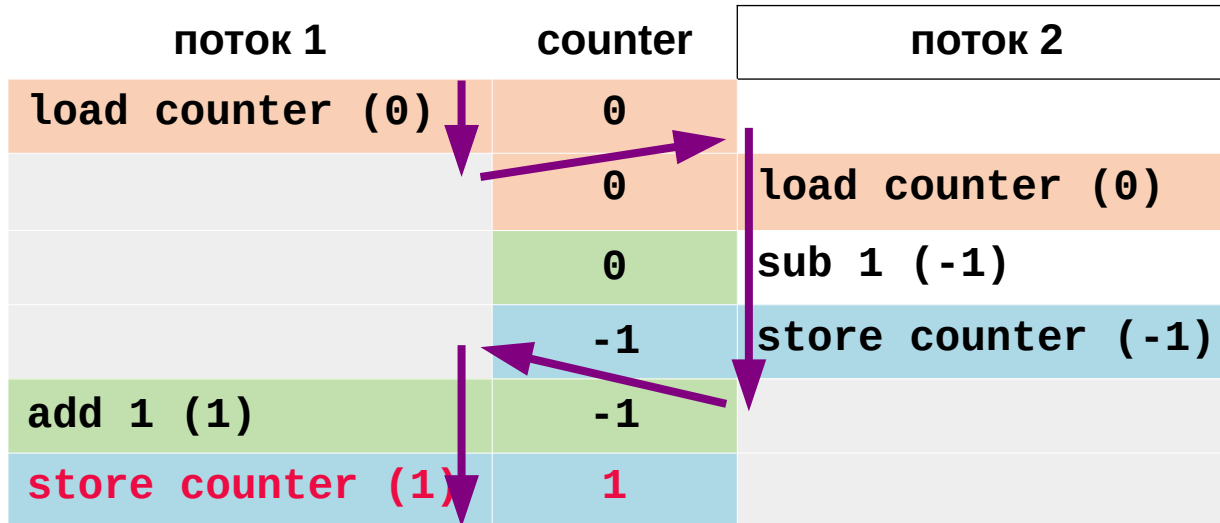
++ --

```
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--; }  
}
```

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



++ --

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

- Совместные изменяемые данные (shared mutable data)
 - ❖ Более одного потока имеют доступ к общей переменной
 - ❖ По крайней мере один поток выполняет запись

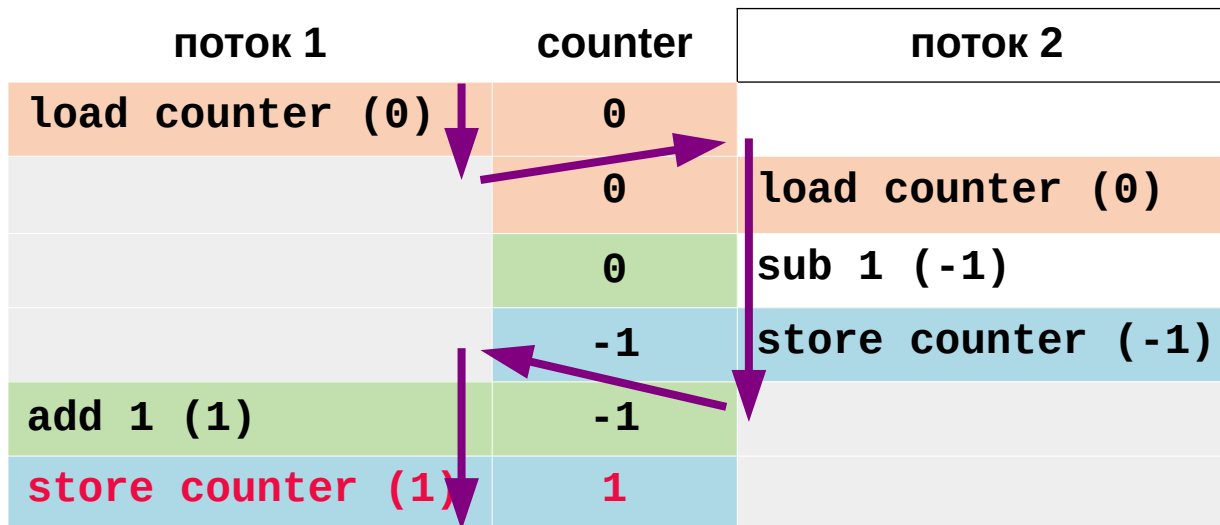
- Совместные изменяемые данные (shared mutable data)
 - ❖ Более одного потока имеют доступ к общей переменной
 - ❖ По крайней мере один поток выполняет запись
- **Решения проблемы**
 - ❖ **отсутствие совместных данных** - локальные данные
 - ❖ **неизменяемость данных** - immutability
 - ❖ **синхронизация**

- Совместные изменяемые данные (shared mutable data)
 - ❖ Более одного потока имеют доступ к общей переменной
 - ❖ По крайней мере один поток выполняет запись
- **Решения проблемы**
 - ❖ **отсутствие совместных данных** - локальные данные
 - ❖ **неизменяемость данных** - immutability
 - ❖ **синхронизация**
 - ❖ **однопоточность**

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

КРИТИЧЕСКАЯ СЕКЦИЯ

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



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

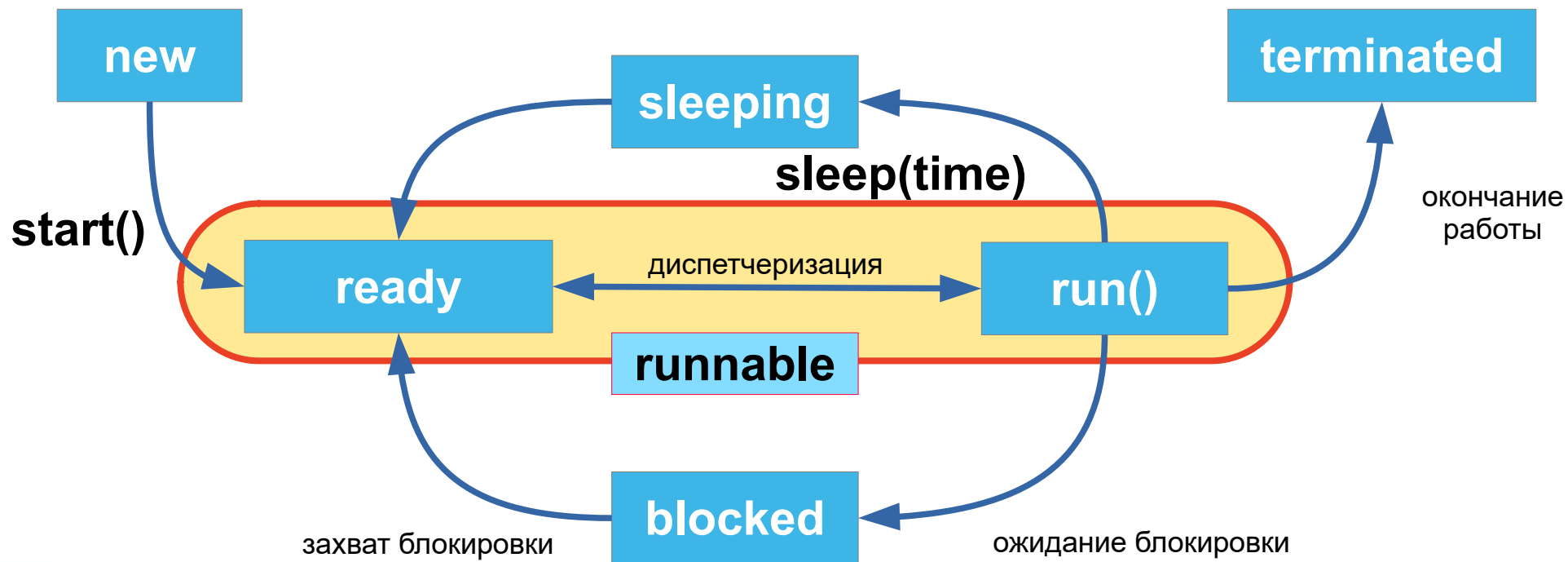
```
class Shared {  
    int counter = 1;  
    synchronized void up()    { counter++; }  
    synchronized void down() { counter--; }  
}  
sh = new Shared();  
new Thread(sh::up).start();  
new Thread(sh::down).start();
```

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

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

- Защита критической секции (КС)
 - ❖ Любой объект имеет встроенную блокировку (intrinsic lock)
 - ❖ КС доступна **только** потоку **с блокировкой**
 - ❖ При **входе** в КС поток забирает блокировку
 - ❖ При **выходе** из КС поток отдает блокировку
 - ❖ Блокировка реентерабельна — не блокирует себя
 - ❖ Остальные потоки **ждут** в очереди





synchronized метод

```
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()
```

по объекту, у которого вызван метод

synchronized блок

```
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
```

по параметру блока synchronized (любой объект)

static synchronized метод

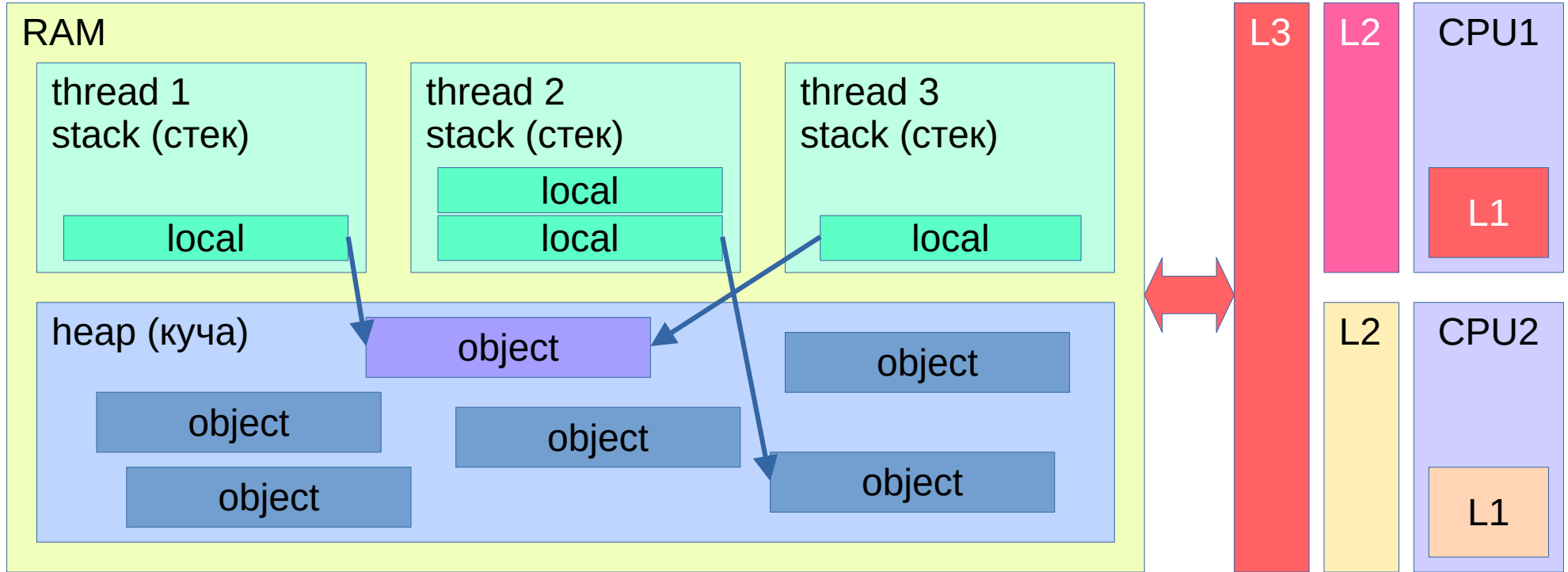
```
public class MyClass {  
    Object lock = new Object(),  
    public synchronized void add() { }  
    public synchronized void rem() { }  
    public static synchronized int min() { }  
    public static synchronized int max() { }  
    public void x() { ... synchronized (lock) { } ... }  
   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

по объекту класса Class класса, у которого вызван метод

```
Class<? extends MyClass> lock = m.getClass()  
MyClass.class
```

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

JMM - Java Memory Model



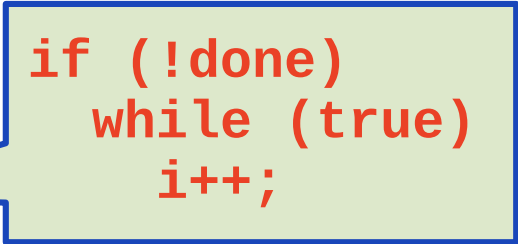
- Компилятор и JVM могут оптимизировать инструкции

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


- Компилятор и JVM могут оптимизировать инструкции

```
(new Thread(() -> {
```

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



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

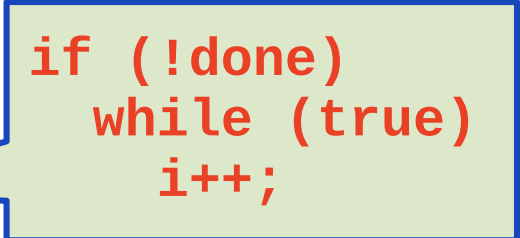
```
}).start();    // запустился первый поток  
              // main продолжает работать
```

```
Thread.sleep(1000);
```

```
done = true;    // первый поток остановится через 1 с?
```

- Компилятор и JVM могут оптимизировать инструкции

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



```
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; }
```

```
// x = ?; y = ?;
```

```
(new Thread() -> m1()).start();  
(new Thread() -> m2()).start();
```

- Компилятор и JVM могут оптимизировать порядок выполнения инструкций

```
a = 0, b = 0, x = 0, y = 0;
```

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

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

```
(new Thread() -> m1()).start();  
(new Thread() -> m2()).start();
```

- Компилятор и JVM могут оптимизировать порядок выполнения инструкций

```
a = 0, b = 0, x = 0, y = 0;
```

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

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

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

```
(new Thread() -> m1()).start();  
(new Thread() -> m2()).start();
```

- Компилятор и JVM могут оптимизировать порядок выполнения инструкций

`a = 0, b = 0, x = 0, y = 0;`

`m1() { b = 1; x = a; } // 2`

`m2() { a = 2; y = b; } // 1 3`

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

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

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

```
(new Thread() -> m1()).start();  
(new Thread() -> m2()).start();
```

Эквивалентный код?

```
b = 1; x = a;
```

=

```
x = a; b = 1;
```

```
a = 2; y = b;
```

=

```
y = b; a = 2;
```


Эквивалентный код?

```
b = 1; x = a;
```

=

```
x = a; b = 1;
```

```
a = 2; y = b;
```

=

```
y = b; a = 2;
```

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

```
(new Thread() -> m1()).start();  
(new Thread() -> m2()).start();
```

- Компилятор и JVM могут оптимизировать порядок выполнения инструкций

```
a = 0, b = 0, x = 0, y = 0;
```

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

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

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

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

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

```
(new Thread() -> m1()).start();  
(new Thread() -> m2()).start();
```

- Компилятор и JVM могут оптимизировать порядок выполнения инструкций

a = 0, **b** = 0, **x** = 0, **y** = 0;

```
m1() { x = a; b = 1; } // 1 3
m2() { y = b; a = 2; } // 2
```

```
(new Thread() -> m1()).start();
(new Thread() -> m2()).start();
```

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

- Компилятор и JVM могут оптимизировать порядок выполнения инструкций

a = 0, **b** = 0, **x** = 0, **y** = 0;

```
m1() { x = a; b = 1; } // 2
m2() { y = b; a = 2; } // 1 3
```

```
(new Thread() -> m1()).start();
(new Thread() -> m2()).start();
```

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

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



- Модификатор **`volatile`** — переменная может измениться 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 отношение «**`happens-before`**»
 - ❖ Значения переменных, видимые в потоке 1 до записи значения в `volatile` переменную должна быть видны в потоке 2 после чтения `volatile` переменной

- Когда действия одного потока **при отсутствии гонок** будут **гарантированно видимы** другому потоку
 - ❖ $X ; Y \Rightarrow X \text{ happens-before } Y$
 - ❖ `unlock(obj)` **happens-before** next `lock(obj)`
 - ❖ volatile write **happens-before** next volatile read
 - ❖ `start()` **happens-before** \forall action **happens-before** `TERMINATED`
 - ❖ `interrupt()` **happens-before** `isInterrupted() == true`
 - ❖ $X \text{ happens-before } Y \ \& \ Y \text{ happens-before } Z \Rightarrow X \text{ happens-before } Z$

- synchronized гарантирует видимость и атомарность
- volatile гарантирует видимость, но не гарантирует атомарность
- запись в переменные long и double - не атомарная

- 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();
```

- Вариант 2 — 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

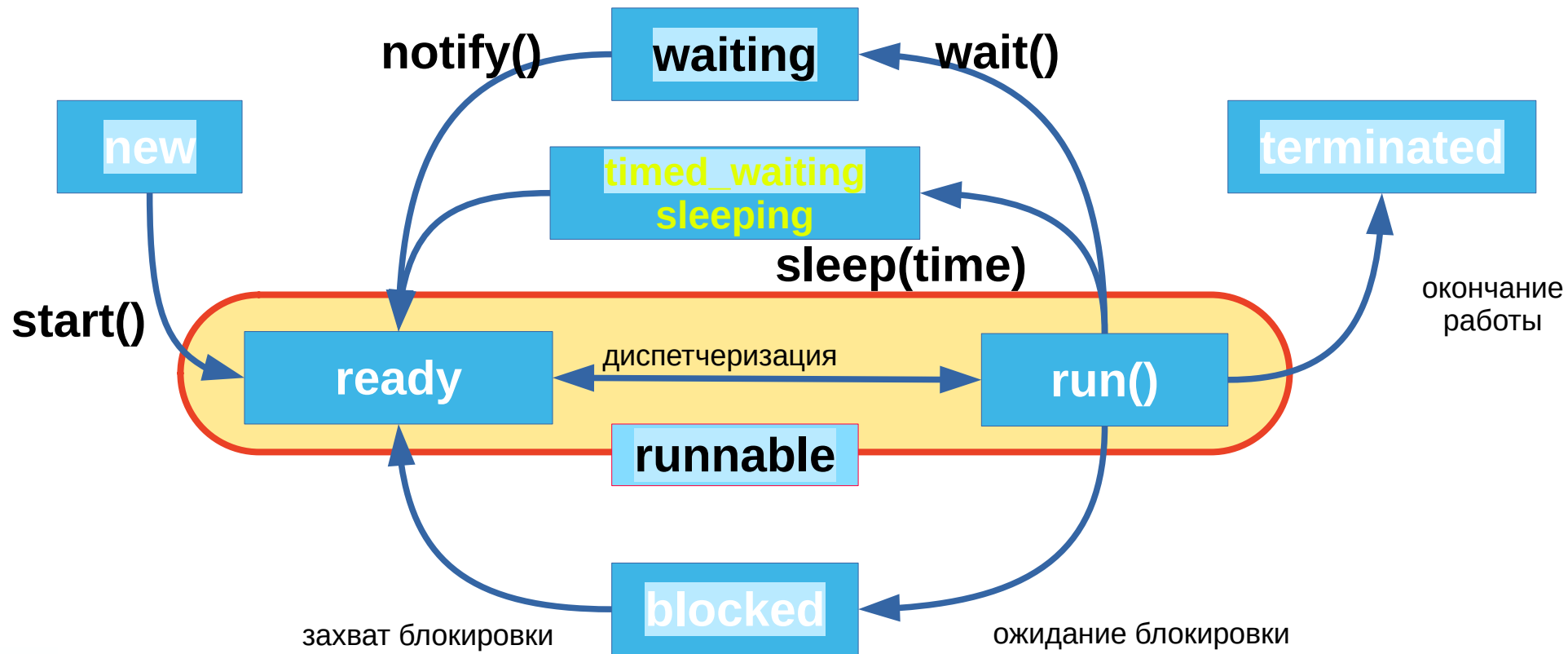
```
Block g = new Block();
```

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

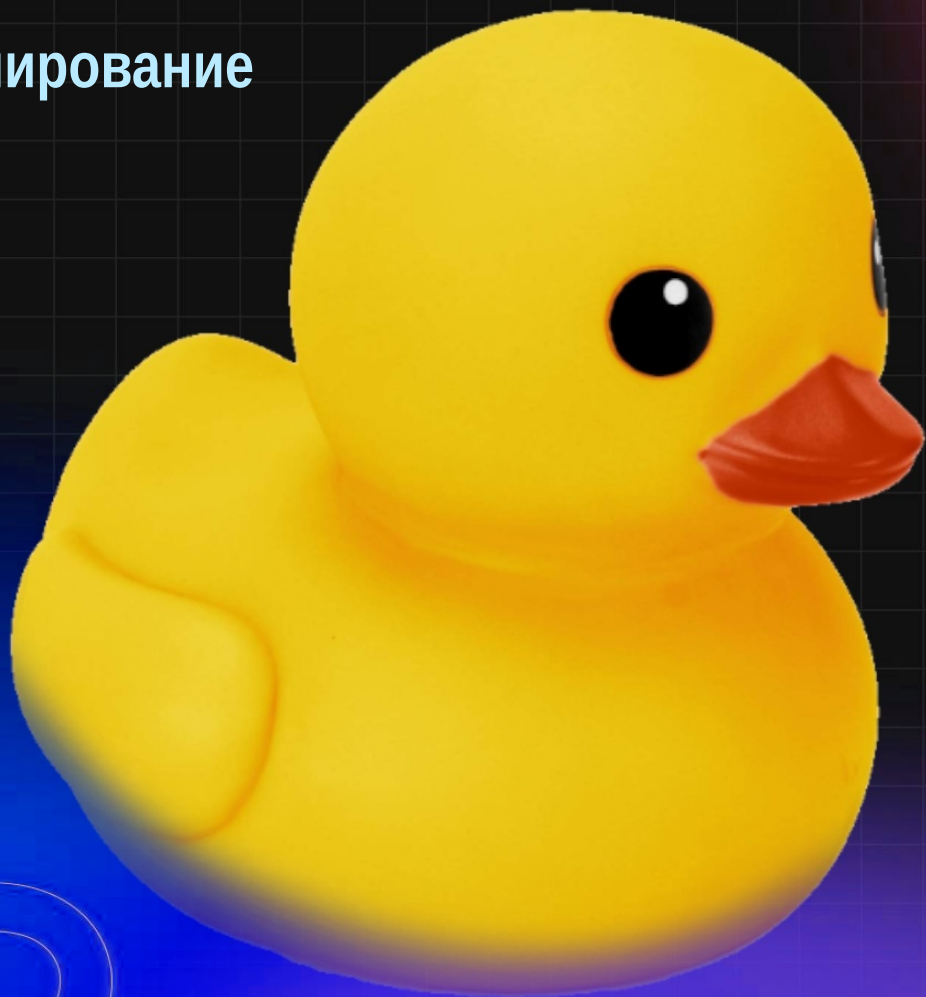
```
Thread t2 = new Thread(() -> {  
    System.out.println(g.get());  
});
```

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

- 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.



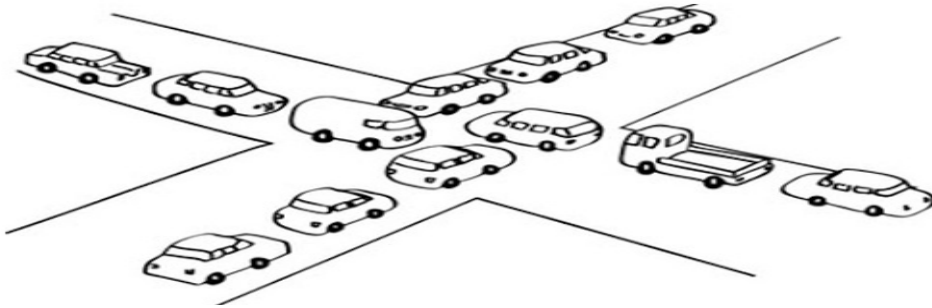
Программирование
2 семестр
2025



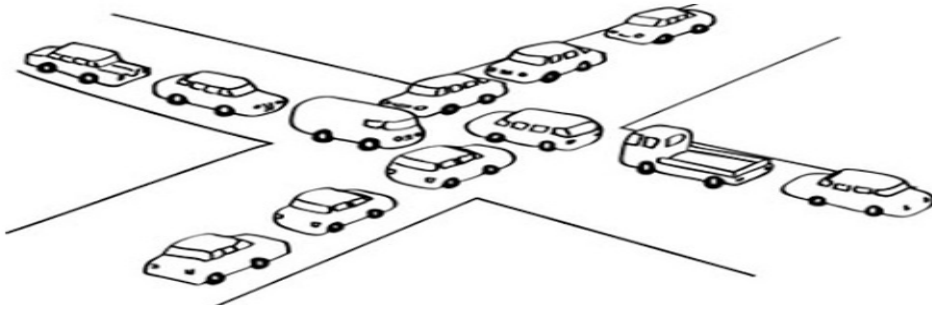
ІТМО

Многопоточность
`java.concurrent.*`

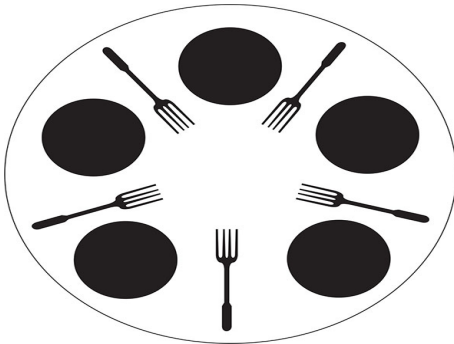
- Взаимная блокировка (deadlock)



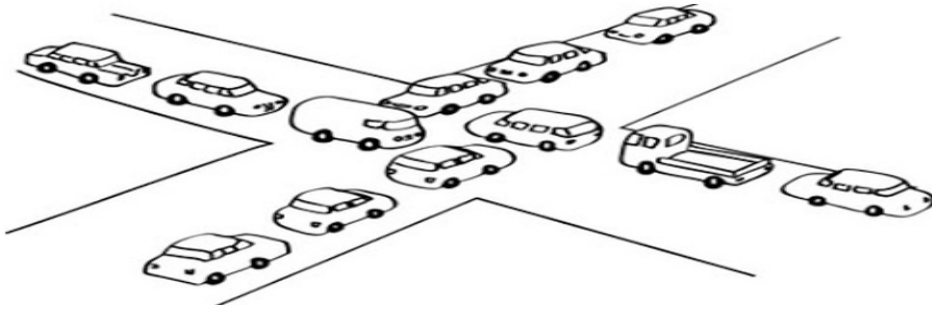
- Взаимная блокировка (deadlock)



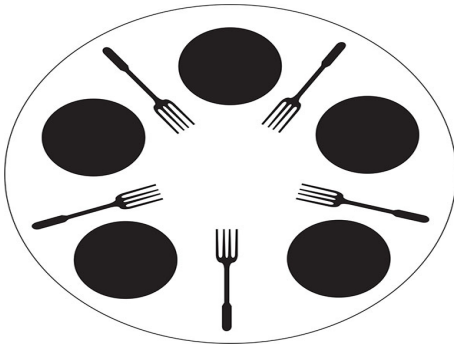
- Обедающие философы



- Взаимная блокировка (deadlock)



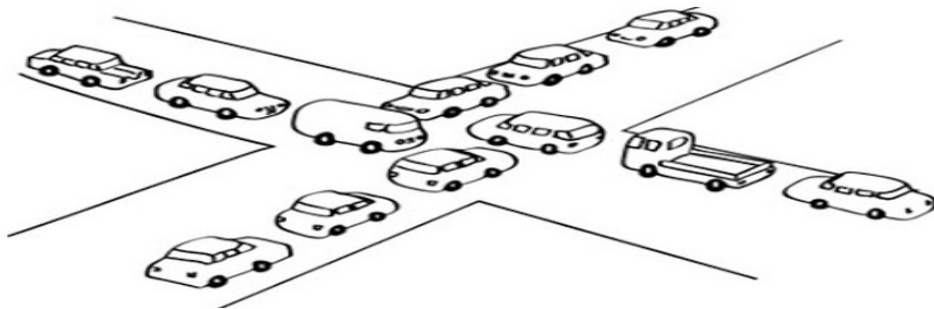
- Обедаящие философы



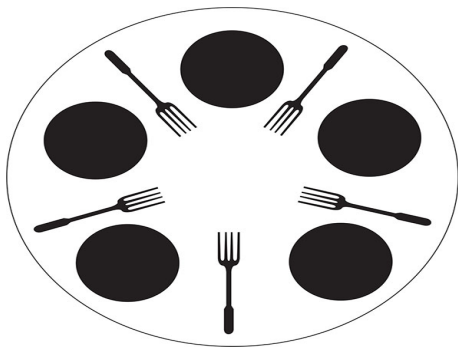
- Голодание (starvation)



- 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 `Concurrent` и `CopyOnWrite`
- `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

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

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

- interface **ExecutorService** extends **Executor**
 - ❖ **Future<T> submit(Callable<T> task)**
 - ❖ **void shutdown() List<Runnable> shutdownNow()**
 - ❖ **List<Future<T>> invokeAll(Collection<Callable<T>> tasks)**
- interface **Callable<T>**
 - ❖ **T call()**
- interface **Future<T>**
 - ❖ **T get()**
 - ❖ **boolean isDone()**
 - ❖ **boolean cancel()**

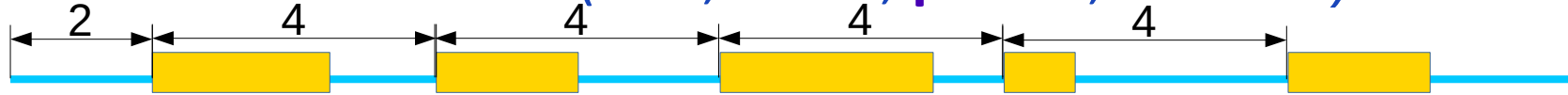
```
var s = "toFind";  
var text = ""very very very ...  
          very long text ..."";  
String search(s, text) {}
```

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

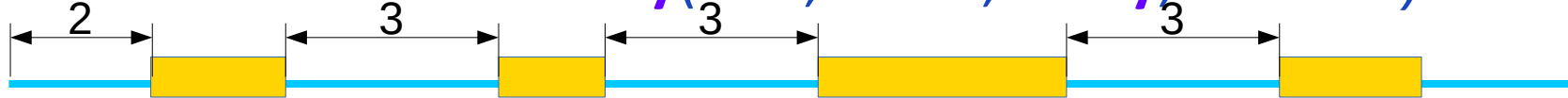
- interface **ScheduledExecutorService** extends **ExecutorService**

- ❖ **ScheduledFuture** **schedule**(task, delay, timeunit)

- ❖ **scheduleAtFixedRate**(task, initial, period, timeunit)



- ❖ **scheduleWithFixedDelay**(task, initial, delay, timeunit)



- **ScheduledFuture** extends **DelayedFuture**

- ❖ long **getDelay**(timeunit)

- Создание потока требует ресурсов и времени
- Пул позволяет повторно использовать потоки
- Постепенная деградация при увеличении нагрузки

- Параметры:
 - ❖ `corePoolSize` - основной размер пула
 - ❖ `maximumPoolSize` - максимальный размер пула
 - ❖ `keepAliveTime` - время жизни дополнительных потоков

- Действия для поступающих задач:
 - ❖ `threads < core` - новый поток или в очередь
 - ❖ `core < threads < max` - в очередь или новый поток
 - ❖ `threads = max` - в очередь или отклонить
- Очередь:
 - ❖ Синхронная - размер 0
 - ❖ Ограниченная (bounded)
 - ❖ Неограниченная (unbounded)

- класс Executors — статические методы
 - ❖ ExecutorService new**SingleThreadExecutor**()
 - ❖ ExecutorService new**FixedThreadPool**(size)
 - core = max = size; keepAliveTime = 0;
 - ❖ ExecutorService new**CachedThreadPool**()
 - core = 0; max = max int; keepAliveTime = 60 c
 - ❖ ExecutorService new**WorkStealingPool**()
 - ForkJoinPool
 - ❖ ExecutorService new**VirtualThreadPerTaskExecutor**()

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

```
если (задача небольшая) {  
    делаем сами  
} иначе {  
    делим на подзадачи и раздаем другим (fork)  
    ждем результаты (можем помочь - work stealing)  
    объединяем полученные результаты (join)  
}  
возвращаем итоговый результат
```

- class **ForkJoinPool** implements **ExecutorService**
 - ❖ `ForkJoinPool.commonPool()`
 - ❖ `ForkJoinTask<V> submit(ForkJoinTask<V>)`
 - ❖ `V invoke(ForkJoinTask<V>)`
- class **ForkJoinTask** implements **Future**
 - ❖ `ForkJoinTask<V> fork()`
 - ❖ `V join()`
- class **RecursiveAction** extends **ForkJoinTask**
 - abstract void `compute()`
- class **RecursiveTask** extends **ForkJoinTask**
 - abstract `V compute()`

```
public class Task extends RecursiveAction {
```

```
    final int[] array;  
    final int lo, hi;  
    final static int SIZE = 10;
```

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

```
}
```

```
public class Task extends RecursiveAction {  
    protected void compute() {  
        if ((hi - lo) < SIZE) {  
            for (int i = lo; i < hi; i++) {  
                array[i] *= 2;  
            }  
        } else {  
            int mid = (lo + hi) / 2;  
            var task1 = new Task(array, lo, mid);  
            var task2 = new Task(array, mid, hi);  
            task1.fork();  
            task2.fork();  
            task2.join();  
            task1.join();  
        }  
    }  
    final int[] array;  
    final int lo, hi;  
    final static int SIZE = 10;  
    Task(int[] array, int lo, int hi) {  
        this.array = array;  
        this.lo = lo, this.hi = hi;  
    }  
}
```



```
public class Task extends RecursiveAction {
```

```
    protected void compute() {
```

```
        if ((hi - lo) < SIZE) {
```

```
            for (int i = lo; i < hi; i++) {
```

```
                array[i] *= 2;
```

```
            }
```

```
        } else {
```

```
            int mid = (lo + hi) / 2;
```

```
            var task1 = new Task(array, lo, mid);
```

```
            var task2 = new Task(array, mid, hi);
```

```
            task1.fork();
```

```
            task2.fork();
```

```
            task2.join();
```

```
            task1.join();
```

```
        }
```

```
    }
```

```
int[] array = {0, ... ,33554431};
```

```
var task = new Task(array, 0, array.length);
```

```
var FJPool = ForkJoinPool.commonPool();
```

```
FJPool.invoke(task);
```

```
final int[] array;
```

```
final int lo, hi;
```

```
final static int SIZE = 10;
```

```
Task(int[] array, int lo, int hi) {
```

```
    this.array = array;
```

```
    this.lo = lo, this.hi = hi;
```

```
}
```

```
}
```

- CompletableFuture implements Future, CompletionStage
 - ❖ Класс для асинхронного выполнения задач
 - ❖ Контроль и комбинирование результатов
 - ❖ 2 вида методов: `doSmth()`, `doSmthAsync()`
- `runAsync(Runnable)`, `supplyAsync(Supplier)`
- `thenApply(Function)`
- `thenAccept(Consumer)`
- `thenCombine(Function)`
- `T get()`

```
CompletableFuture
    .supplyAsync( () -> getResult() )
    .thenApply( String::toUpperCase )
    .thenAccept( System.out::println );
```

- Stream
 - ❖ `.parallel()` - преобразовать в параллельный вариант
 - ❖ `.parallelStream()` - запустить сразу параллельно
- Внутри параллельные стримы используют `ForkJoinPool`
- Заранее известный размер и произвольный доступ
 - ❖ `Stream.range()`, `Arrays.stream()`, `ArrayList.stream()`
- `map()`, `filter()`, `unordered()`, `findAny()`, `forEach()`

- Итератор для конвейеров
 - ❖ boolean `tryAdvance(Consumer action)` - обработать элемент
 - false, если элементы закончились
 - ❖ `Spliterator trySplit()` - отдать часть новому сплитератору
 - null, если нечего отдавать
 - ❖ long `estimateSize()`
 - ❖ default void `forEachRemaining()` - обработать остаток

- 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

```
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 **неблокирующий** доступ)
 - ❖ Lock `writeLock()`
 - возвращает Lock для операций **записи** (**блокирующий** доступ)
- class ReentrantReadWriteLock

```
Lock lock = new ReentrantLock();
Condition notFull = lock.newCondition();
Condition notEmpty = lock.newCondition();
int[] values = new int[100];
int count = 0;
```

```
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(); }
}
```


- Управляет несколькими разрешениями на доступ
- **Semaphore**(int permits, boolean fair)
- Каждый поток:
 - ❖ получает разрешение - semaphore.**acquire()**
 - while (permits == 0) wait; permits--;
 - ❖ возвращает разрешение - semaphore.**release()**
 - permits++;
- **Semaphore(1)** - бинарный семафор (mutex)

- Открывает доступ после обратного отсчета
- **CountDownLatch(int count)**
- Каждый поток:
 - ❖ извещает о событии - latch.**countDown()**
 - count--;
 - ❖ ждет разрешения - latch.**await()** :: void
 - while (count > 0) wait;

- Синхронизация группы потоков
- **CyclicBarrier**(int parties, Runnable task)
- Каждый поток:
 - ❖ ждет остальных - barrier.**await**() :: int // --parties
 - if (parties > 0) wait;
 - ❖ последний поток открывает барьер - **notifyAll**
 - перед открытием выполняет задачу - task.**run**()
 - ❖ сброс барьера - barrier.**reset**()
 - ❖ барьер может сломаться - BrokenBarrierException

- Универсальный барьер-защелка
- **Phaser**(Phaser parent, int parties)
 - ❖ **phase** = 0 (номер фазы, возвращается методами)
- Действия потоков:
 - ❖ **register()** - регистрация
 - ❖ **arrive()** - прибытие
 - **arriveAndDeregister()** - и отмена регистрации
 - **arriveAndAwaitAdvance()** - и ожидание остальных
 - ❖ все прибыли - **phase++** и поехали дальше

- 2 потока синхронно меняются объектами
- **Exchanger()**
- Потоки:
 - ❖ обмен по готовности - V **exchange**(V obj)

- интерфейсы BlockingQueue / BlockingDeque
- Операции
 - ❖ стандартные: add(e)/remove/element ; offer(e)/poll/peek
 - ❖ блокирующие: put(e) / take
 - ❖ с таймаутом: offer (e, time, unit) / poll (time, unit)
 - ❖ для BlockingDeque: putFirst, putLast, takeFirst, takeLast
- Применение:
 - ❖ Producer-Consumer
 - ❖ Обмен сообщениями
 - ❖ Пулы потоков

- Реализации

- ❖ `ArrayBlockingQueue` — ограниченная очередь
- ❖ `LinkedBlockingQueue` — опционально ограниченная очередь
- ❖ `LinkedBlockingDeque` — опционально ограниченный дек
- ❖ `PriorityBlockingQueue` — неограниченная + приоритет
- ❖ `DelayQueue<E extends Delayed>` — выдача с задержкой
- ❖ `SynchronousQueue` — синхронное добавление-получение

- interface `TransferQueue` extends `BlockingQueue`
 - ❖ `transfer(E)` — дождаться получения элемента
- реализация `LinkedTransferQueue`

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

- List
- Set
- SortedSet
- Queue
- Deque
- Map
- SortedMap
- ArrayList
- HashSet
- TreeSet
- LinkedList
- ArrayDeque
- HashMap
- TreeMap
- CopyOnWriteArrayList
- CopyOnWriteArraySet
- ConcurrentSkipListSet
- ConcurrentLinkedQueue
- ConcurrentLinkedDeque
- ConcurrentHashMap
- ConcurrentSkipListMap

- Map с гарантией потокобезопасности и атомарности
- + действия в потоке перед помещением объекта в Map "happen-before" действий после получения этого объекта в другом потоке.
- Методы
 - ❖ `getOrDefault(key, defaultValue)`
 - ❖ `forEach(BiConsumer<K, V> action)`
 - ❖ `compute(key, BiFunction<K, V, V> function)`

- Атомарная операция — операция, выполняющаяся без промежуточных состояний.
- Атомарные операции не вызывают состояние гонок
- В JVM операции чтения-записи ссылок и примитивных типов (кроме long и double) — атомарные
- Реализованы на основе оптимистичного подхода с использованием алгоритма CAS (Compare-and-Swap)

```
class fakeCAS { // симуляция инструкции процессора CMPXCHG
    int value;
    atomic int cmpxchg(int expected, int updated) {
        int old = value;
        if (old == expected) { value = updated; }
        return old;
    }
    atomic int get() {
        return value;
    }
}
```

```
var fcas = new fakeCAS();
int increment() {
    int v;
    do {
        v = fcas.get();
    } while (v != fcas.cmpxchg(v, v+1));
    return v+1;
}
```

- Пакет java.util.concurrent.atomic
 - ❖ AtomicInteger, AtomicLong,
 - ❖ AtomicBoolean, AtomicReference
 - ❖ AtomicIntegerArray
 - ❖ LongAccumulator, DoubleAccumulator
 - ❖ LongAdder, DoubleAdder

- AtomicInteger

- ❖ get — аналог чтения volatile переменной
- ❖ set — аналог записи volatile переменной
- ❖ incrementAndGet()
- ❖ getAndIncrement()
- ❖ addAndGet(int delta)
- ❖ getAndAdd(int delta)
- ❖ getAndSet(int newValue)

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

Счетчик ConcurrentHashMap + LongAdder

```
List<String> keys; // список строк для подсчета
ConcurrentHashMap<String, LongAdder> counter; // счетчик
counter = new ConcurrentHashMap<>();
for (String key : keys) {
    counter.computeIfAbsent(key, LongAdder::new).increment();
}
```


- `java.nio` — асинхронные каналы — `Future<V>`
- `java.util.stream` — `Splitterator`, `parallelStream()`
- лабы
 - ❖ 7 - обработка запросов на сервере
 - ❖ 8 - анимация в GUI