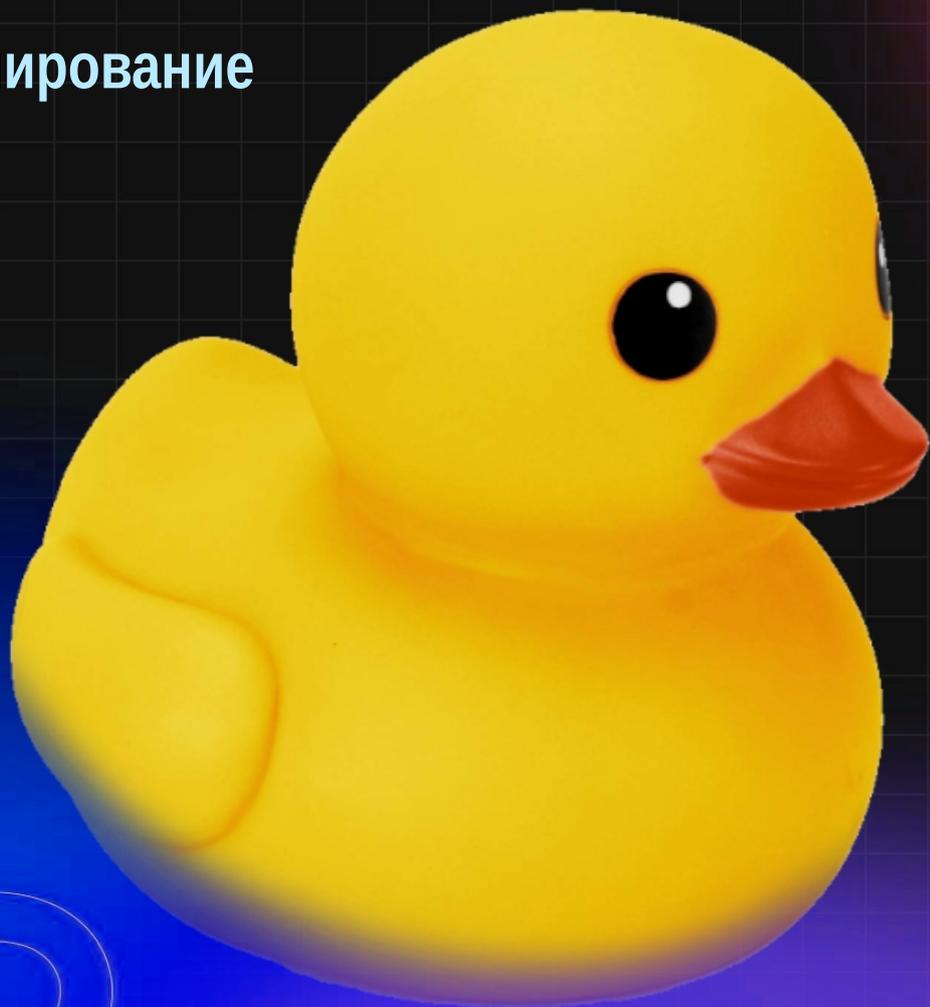
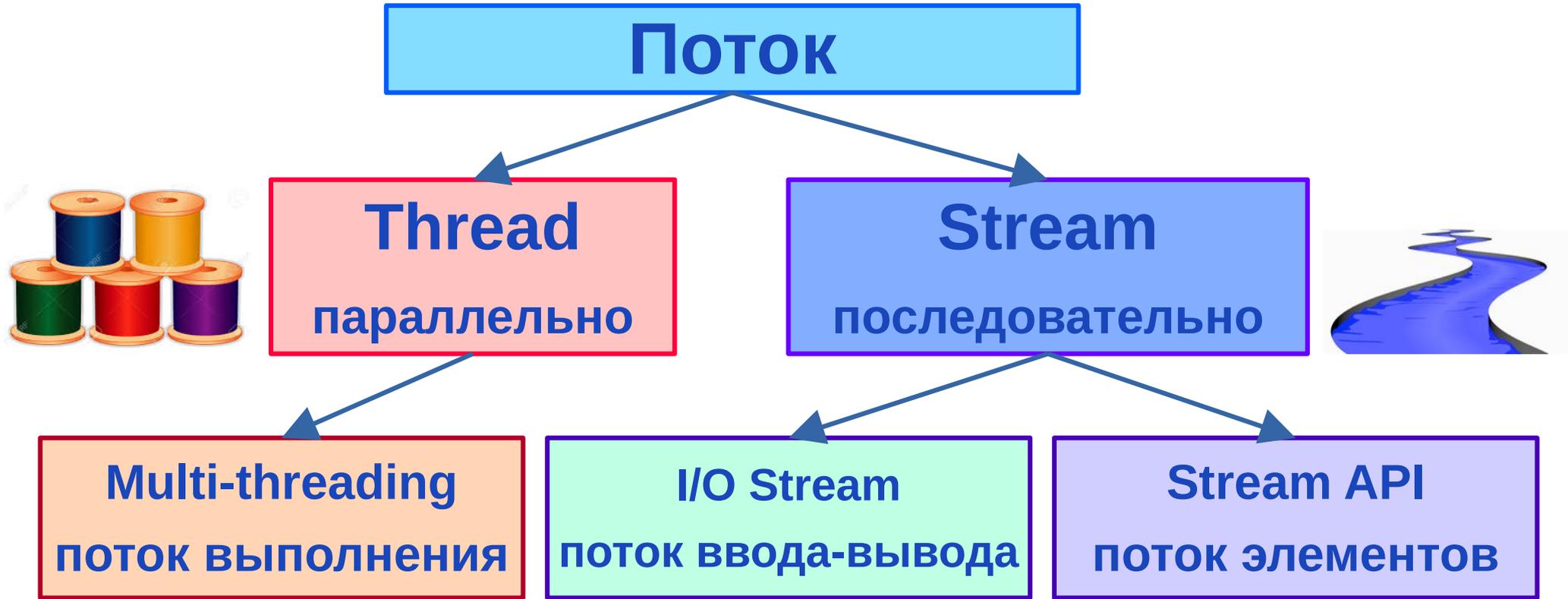


Программирование
2 семестр
2024



ІТМО

Многопоточность
Теория

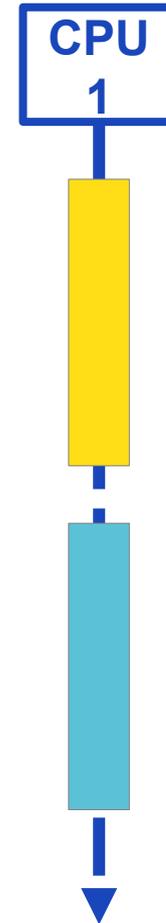


Управление задачами

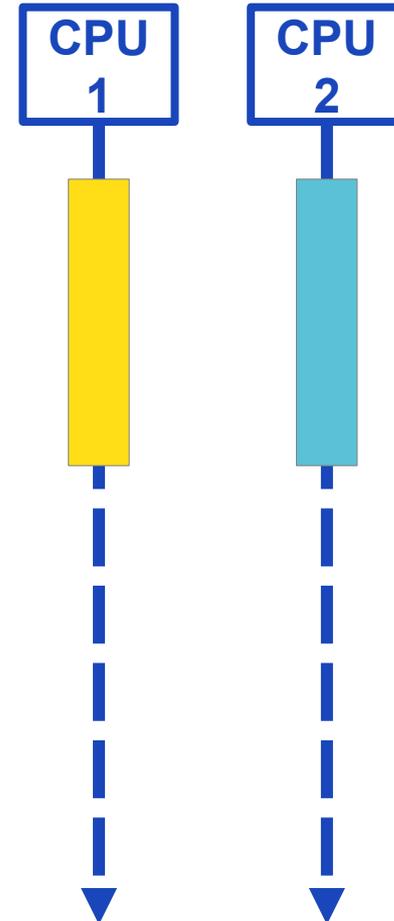
- Один процессор
- Одна задача



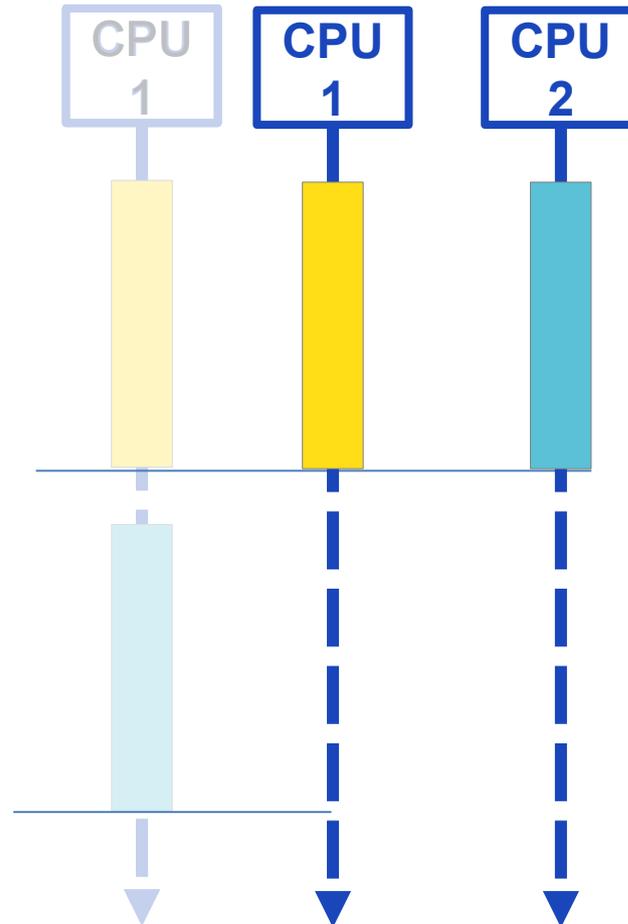
- Один процессор
- Две задачи
- Последовательное исполнение
- Простое управление



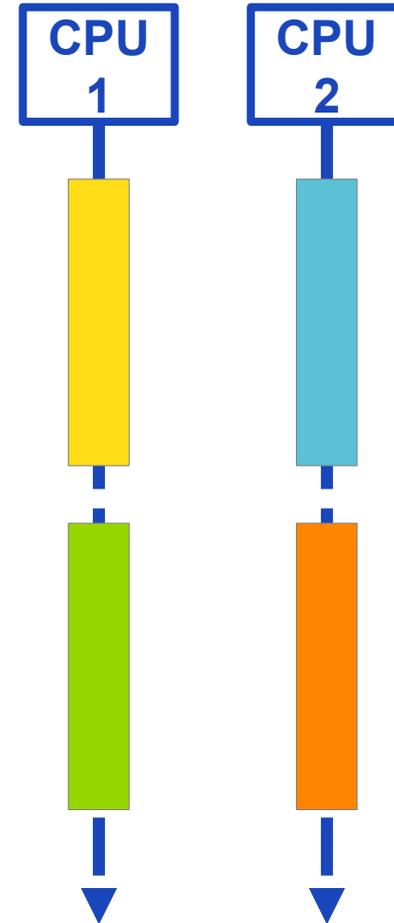
- Два процессора
- Две задачи
- Параллельное исполнение
- Более сложное управление



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

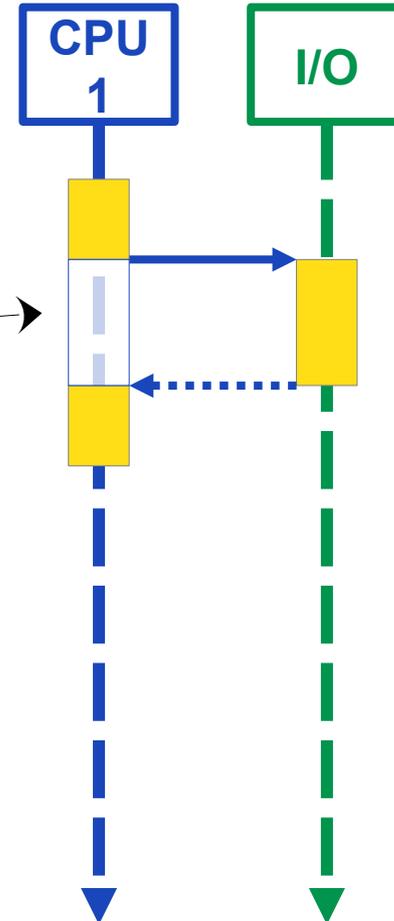


- Много процессоров
- Много задач
- Параллельное исполнение
- Высокая производительность



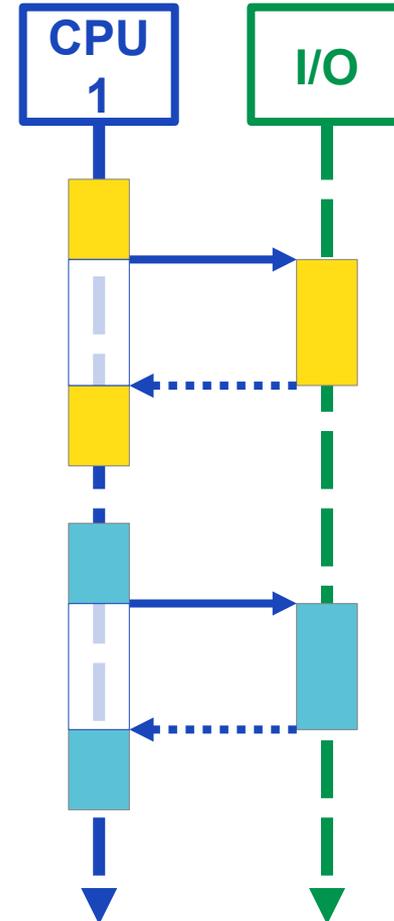
Управление задачами

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



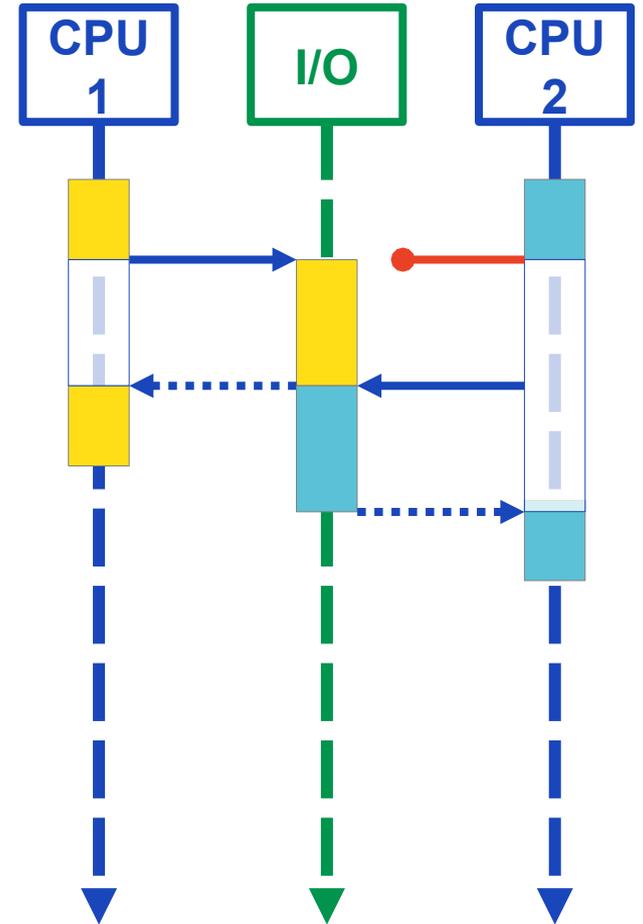
Управление задачами

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

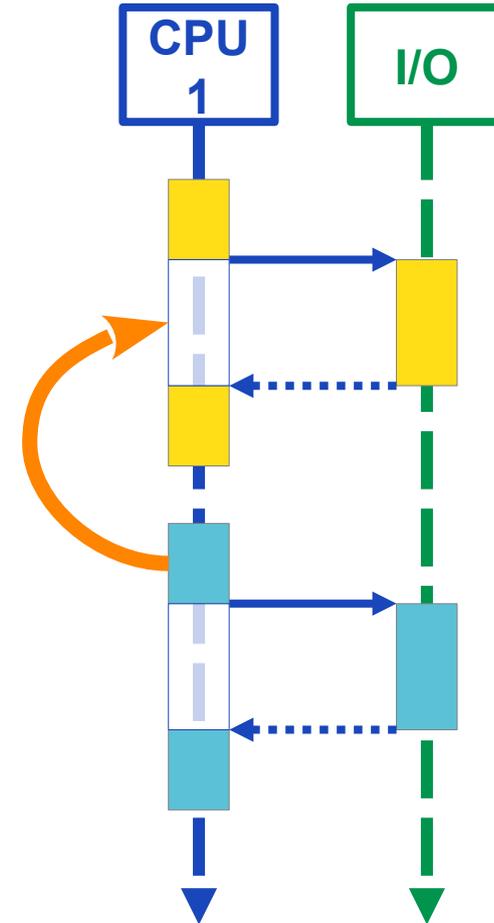


Управление задачами

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

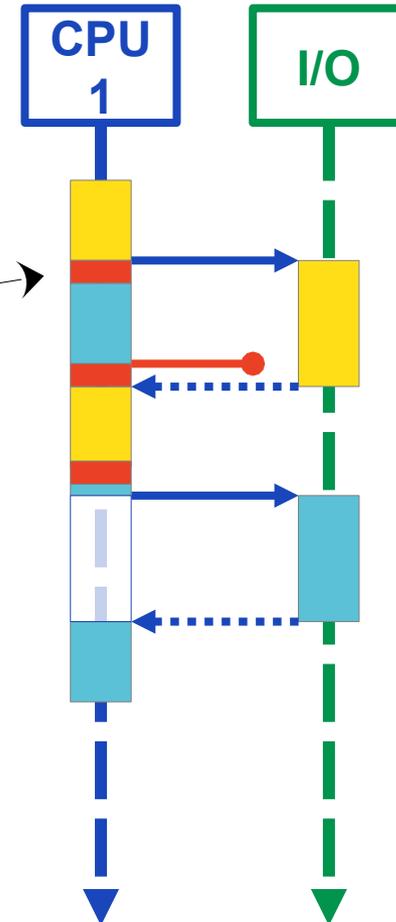


- Один процессор
- Две задачи
- Ввод-вывод
- Процессор простаивает
и это можно использовать

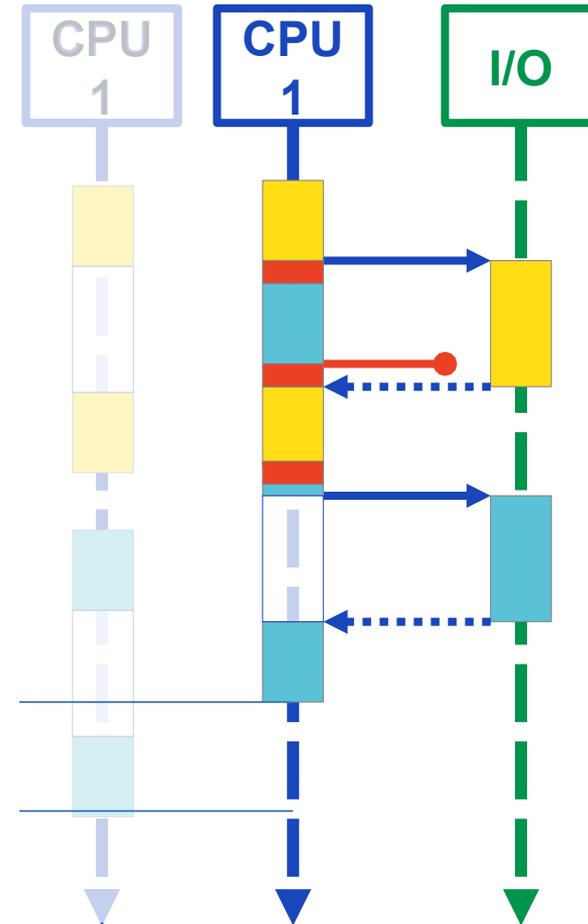


Управление задачами

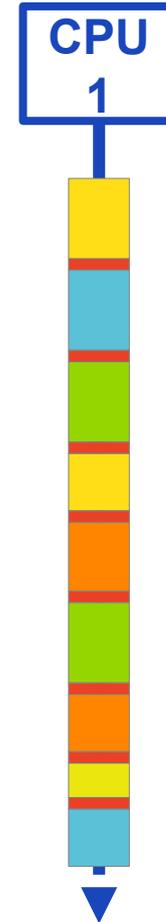
- Один процессор
- Две задачи
- Ввод-вывод
- Переключение контекста
- Конкурентное исполнение
- Более сложное управление



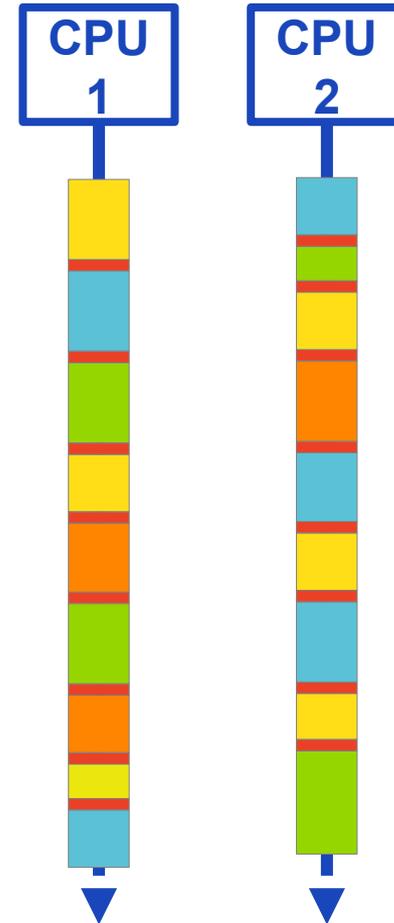
- Один процессор
- Две задачи
- Ввод-вывод
- **Переключение контекста**
- Конкурентное исполнение
- Более сложное управление
- Меньше затраты времени



- 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

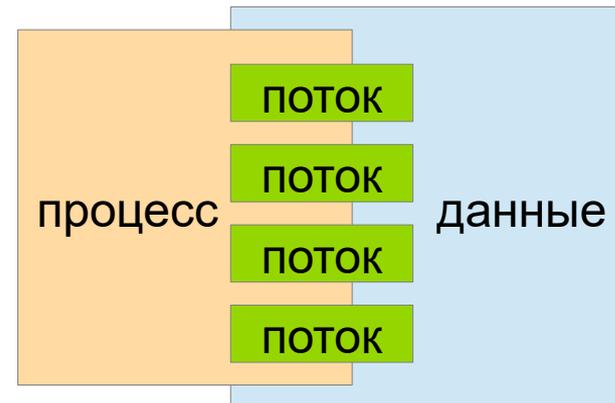
- **Многозадачность**



- **Поток (thread)**

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

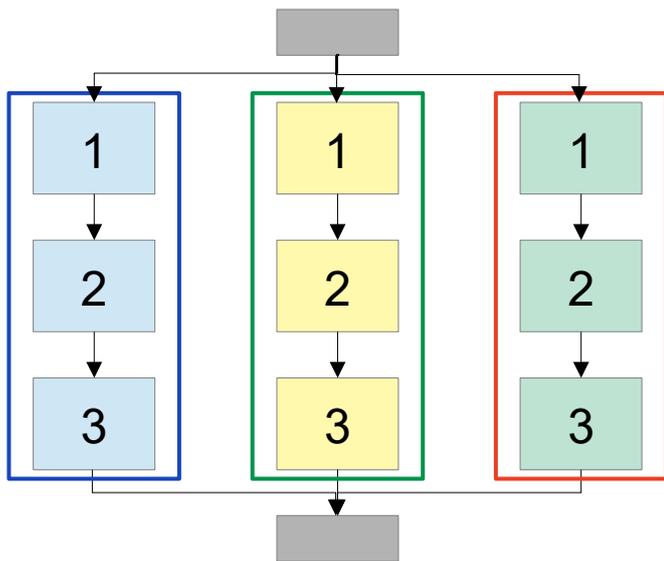
- **Многопоточность**



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

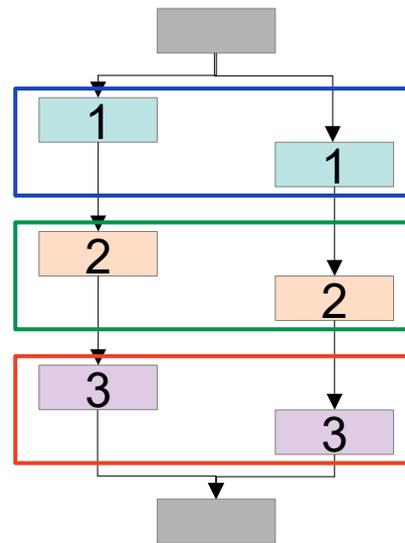
- распараллеливание

- ❖ обработчик выполняет все этапы одной задачи

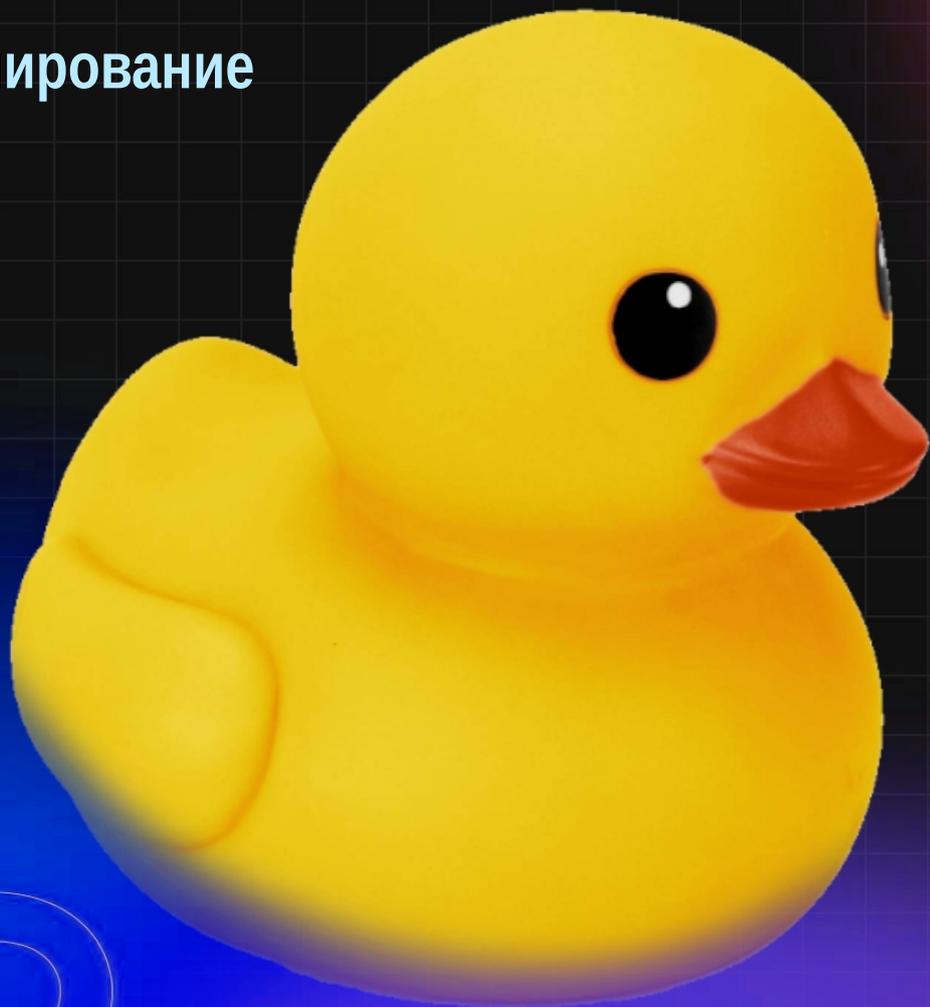


- конвейерная обработка

- ❖ обработчик выполняет один этап всех задач



Программирование
2 семестр
2024



ІТМО

МНОГОПОТОЧНОСТЬ
Java

- системные

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

- прикладные

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

- Первые версии - Green Threads - N:1
- 1.3+ - Platform (Native) Threads - 1:1
- 21+ - Virtual Threads - N:M

- интерфейс Runnable — выполняемая 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 {  
    public void run() {  
        /* тело потока */  
    }  
}  
  
new Thread(new Task()).start();
```

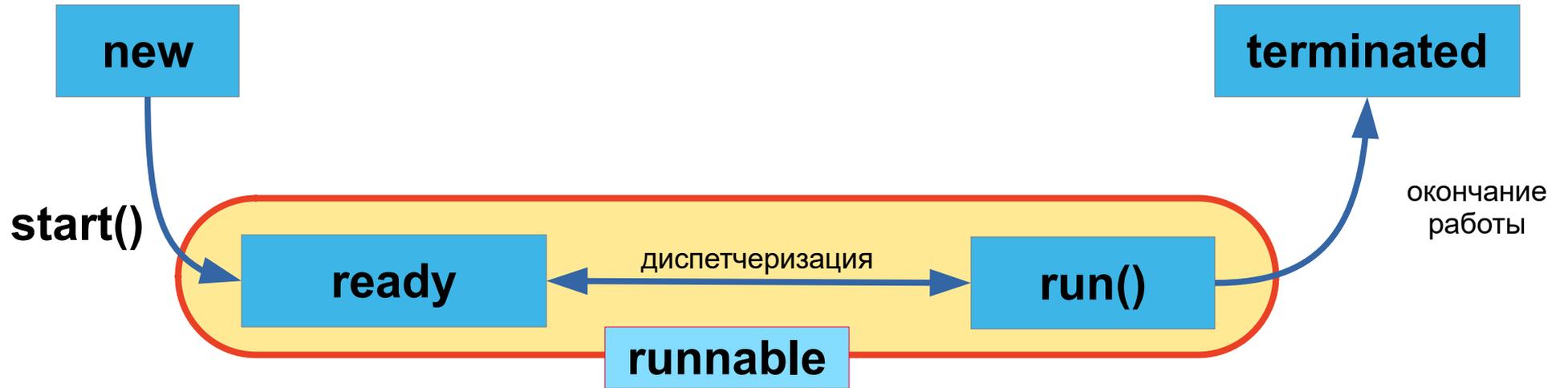
- задача - Runnable
- код задачи - run()
- поток - Thread
- запуск - start()

Класс Thread и интерфейс Runnable

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

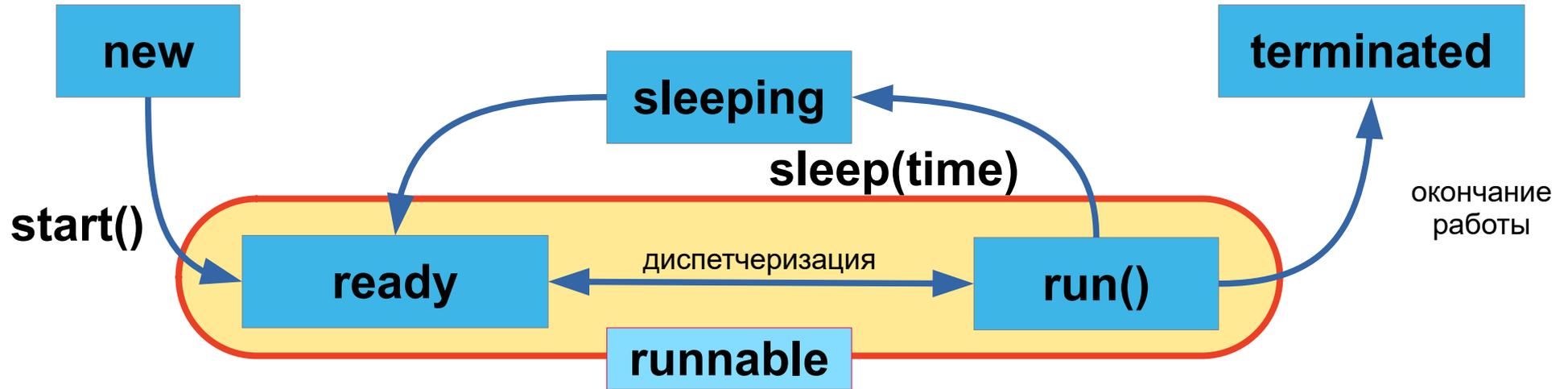
```
class Task extends Thread {  
    public void run() {  
        /* тело потока */  
    }  
}  
  
new Task().start();
```

```
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();
        }
    }
}
```

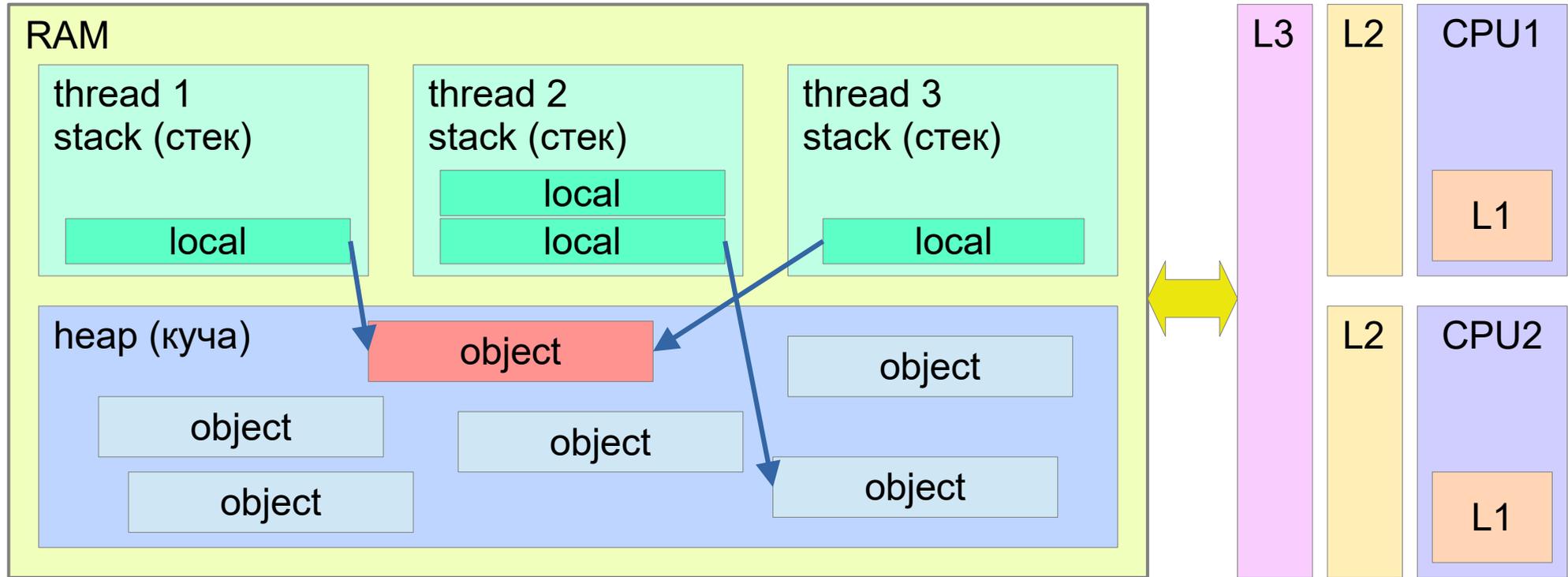
Результат всегда разный

```
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--; }  
}
```

```
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();
```

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

++ --
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
- **Решения проблемы**
 - ❖ **отсутствие совместных данных** - локальные данные
 - ❖ **неизменяемость данных** - immutability
 - ❖ **синхронизация**
 - ❖ **однопоточность**

КРИТИЧЕСКАЯ СЕКЦИЯ

```
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();
```

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

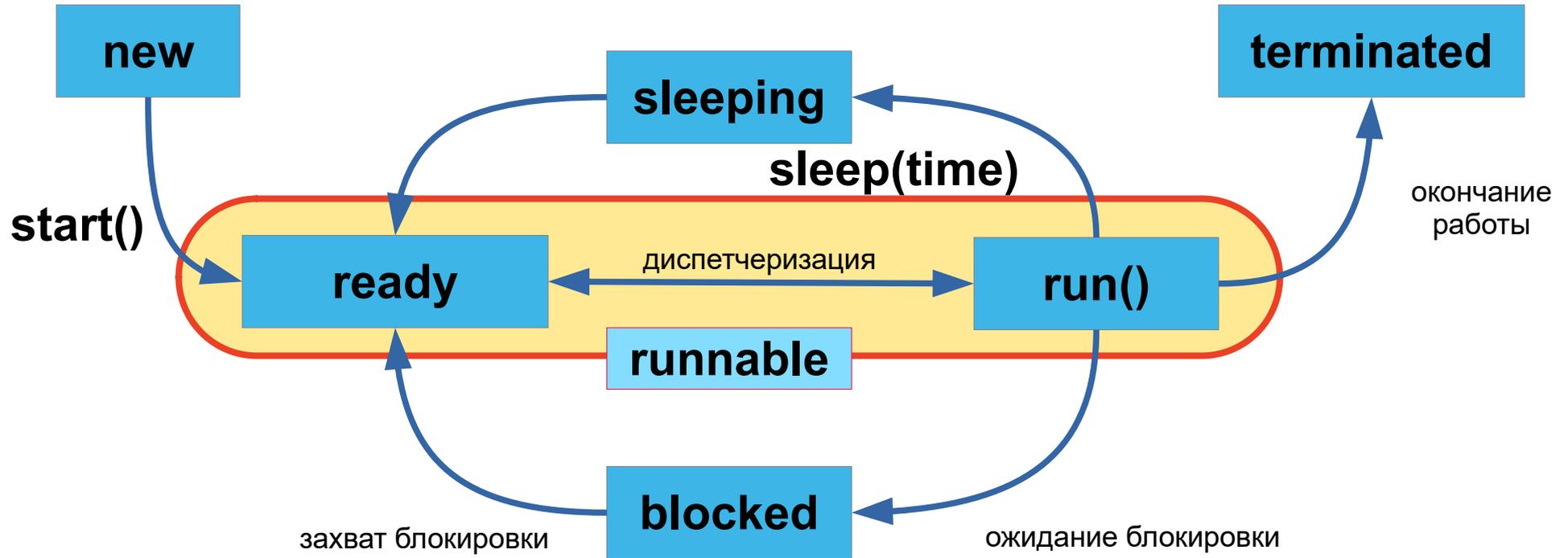
- C++** **C--**
- 1) load counter
 - 2) add 1 (sub 1)
 - 3) store counter

```
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();
```

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

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
```

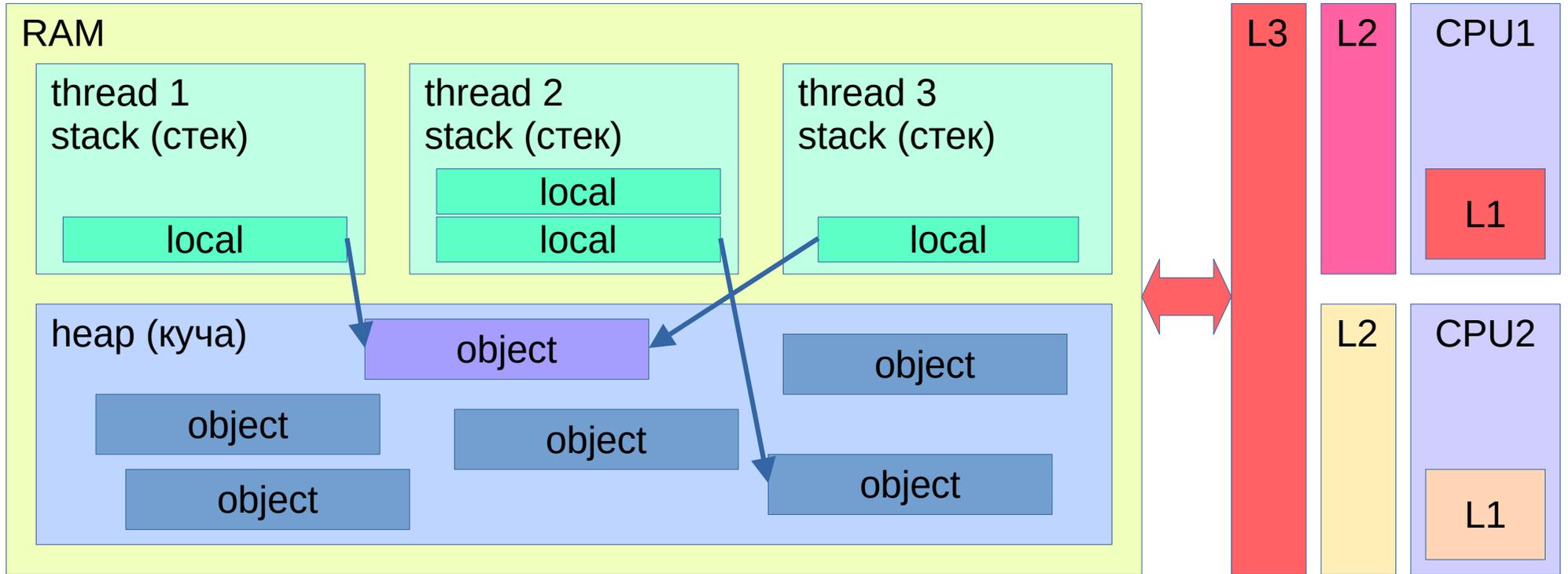
synchronized блок

```
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)

- 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?
```

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

- 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();
```

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

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

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

- 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
 - ❖ $X ; Y; \Rightarrow X \text{ happens-before } Y$
 - ❖ $\text{unlock(obj)} \text{ happens-before next } \text{lock(obj)}$
 - ❖ $\text{volatile write happens-before next volatile read}$
 - ❖ $\text{start()} \text{ happens-before } \forall \text{ action happens-before TERMINATED}$
 - ❖ $\text{interrupt()} \text{ happens-before } \text{isInterrupted()} == \text{true}$
 - ❖ $X \text{ happens-before } Y \ \& \ Y \text{ happens-before } Z \Rightarrow X \text{ happens-before } Z$

- 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;  
    }  
}
```

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

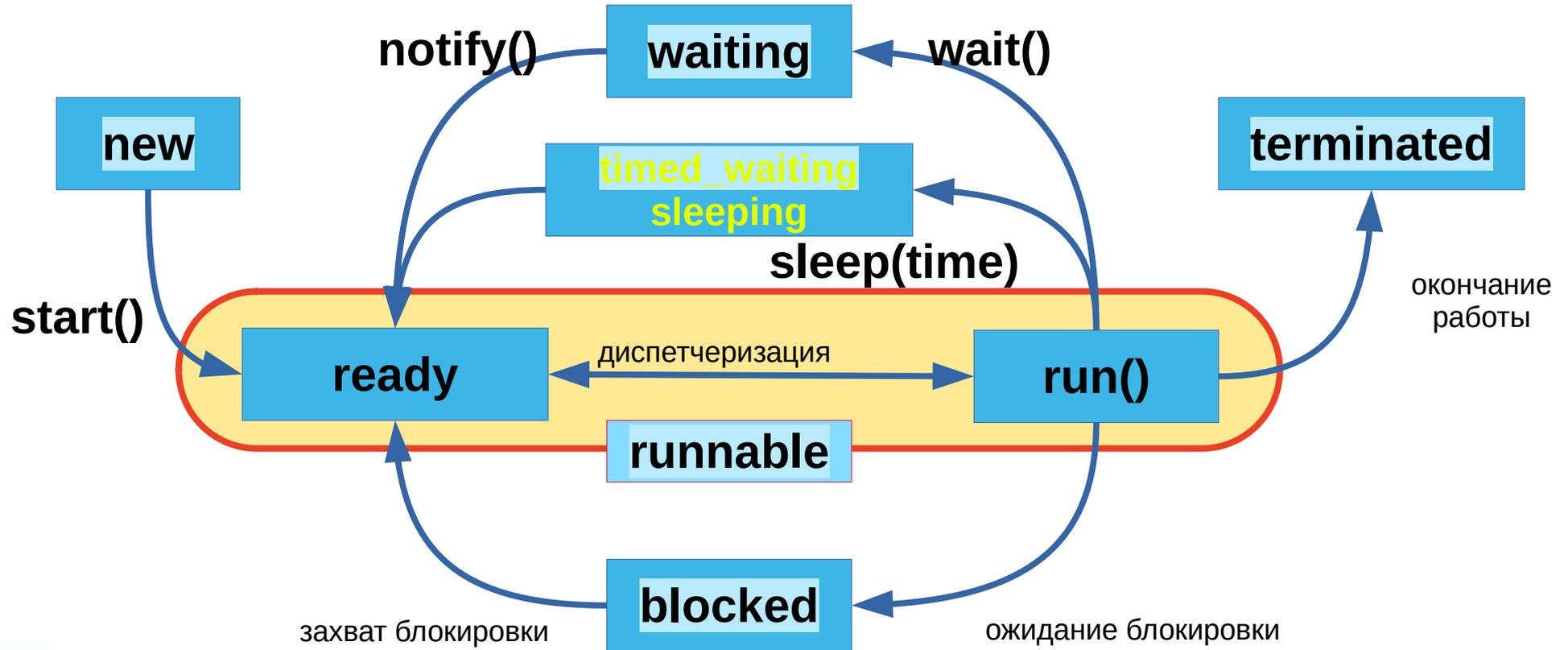
```
Thread t1 = new Thread(() -> {  
    g.put(100);  
});
```

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

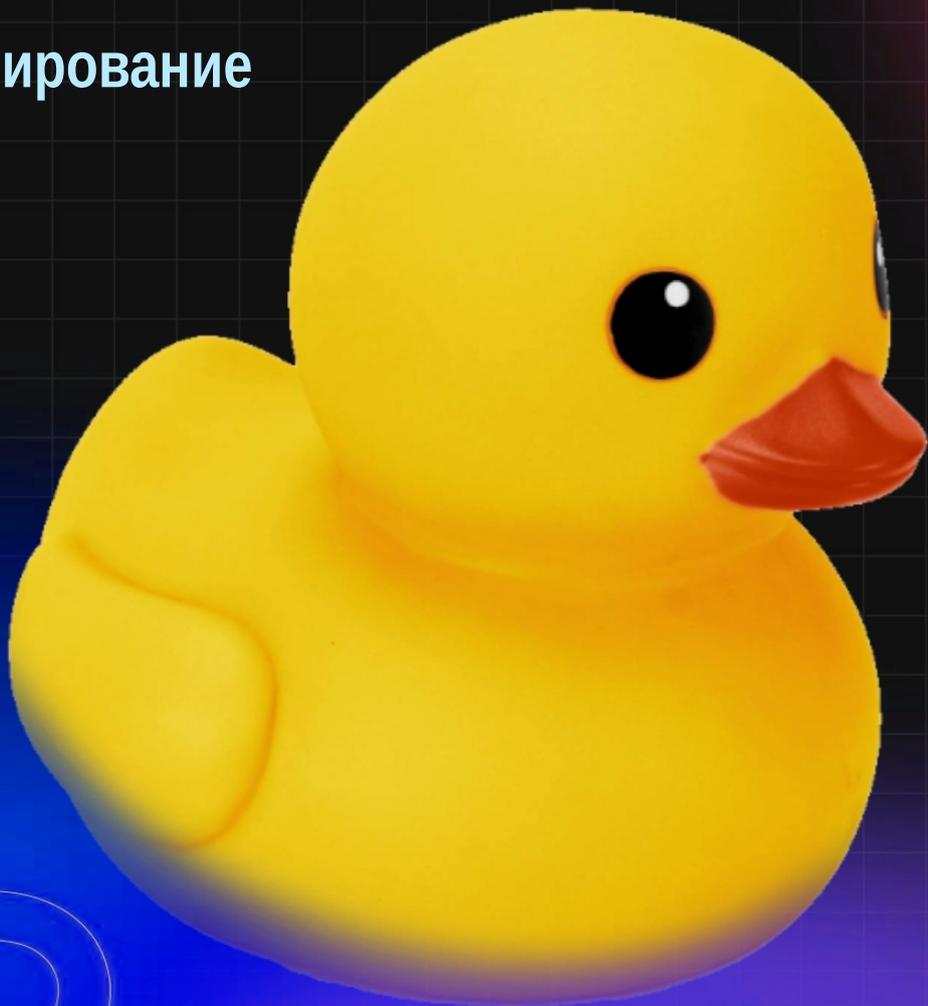
```
t1.start();  
t2.start();
```

ожидание
в очереди

- 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.



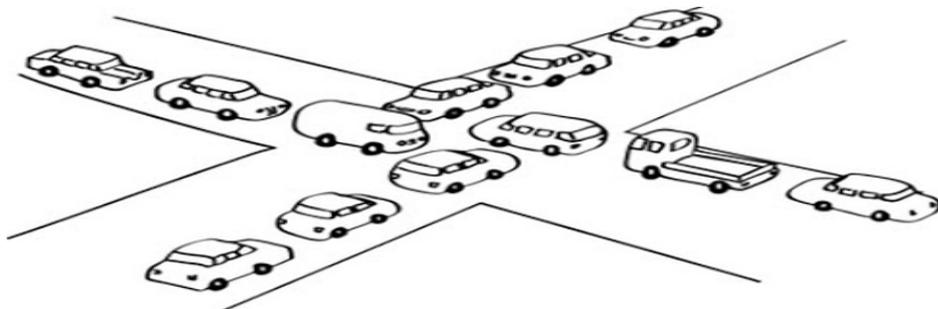
Программирование
2 семестр
2024



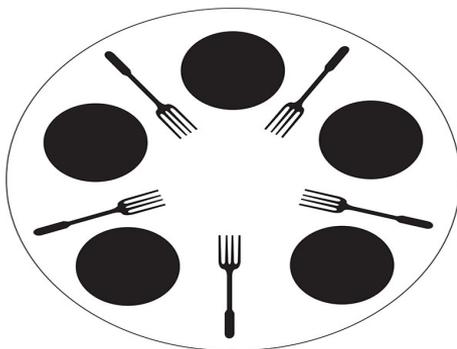
ІТМО

Многопоточность
`java.concurrent.*`

- 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)`

- Future - результат в будущем
- CompletionStage - законченный этап
- CompletableFuture implements Future, CompletionStage
 - ❖ Класс для получения результата асинхронной задачи
 - ❖ С возможностью контролировать и комбинировать выполнение задач
 - thenApply, thenAccept, thenRun, thenCombine, thenCompose
 - acceptEither, ..., runAfterBoth, ..., whenComplete, ...

- 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**VirtualThread**PerTaskExecutor()

- 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
 - ❖ 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 SIZE = 10;

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

    protected void compute() {
        if ((hi - lo) < 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);
```

- Итератор для конвейеров
- `Splitter trySplit()` - отдать часть элементов другому
- `boolean tryAdvance()` - обработать очередной элемент
- `void forEachRemaining()` - обработать то, что осталось

- Stream
 - ❖ `.parallel()` - преобразовать в параллельный вариант
 - ❖ `.parallelStream()` - запустить сразу параллельно
- Внутри параллельные стримы используют `ForkJoinPool`

- 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 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)`

- 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
 - ❖ AtomicInteger, AtomicLong,
 - ❖ AtomicBoolean, AtomicReference
 - ❖ AtomicIntegerArray
 - ❖ LongAccumulator, DoubleAccumulator
 - ❖ LongAdder, DoubleAdder

```
class Count {  
    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
 - ❖ `incrementAndGet()`
 - ❖ `getAndIncrement()`
 - ❖ `addAndGet(int delta)`
 - ❖ `getAndAdd(int delta)`
 - ❖ `getAndSet(int newValue)`



CAS = Compare-And-Swap

Compare-and-Swap

```
class CAS {
    int value;
    synchronized int get() { return value; }
    synchronized int cas(int expected, int updated) {
        int old = value;
        if (old == expected) {
            value = updated;
        }
        return old;
    }
}
```

```
CAS x = new CAS();
int increment() {
    int v = x.get();
    while (x.cas(v, v+1) != v) {
        v = x.get();
    }
    return v+1;
}
```

- 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 ограниченная очередь
- ❖ `PriorityBlockingQueue` - неограниченная с приоритетом
- ❖ `DelayQueue<E extends Delayed>` — доступ с задержкой
- ❖ `SynchronousQueue` — синхронное добавление-получение

- 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`,
 - ❖ `ConcurrentSkipListMap`, `ConcurrentSkipListSet`
- **ConcurrentLinkedQueue**
 - ❖ потокобезопасная очередь
- **CopyOnWriteArrayList** / **CopyOnWriteArraySet**
 - ❖ операции, изменяющие коллекцию, создают новую копию.
 - ❖ операции чтения, а также итераторы продолжают работать со старой копией.

- 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)`

Счетчик на основе ConcurrentHashMap

```
var counter = ConcurrentHashMap<String, LongAdder>();  
counter.computeIfAbsent(key, k -> new  
LongAdder()).increment();
```

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