

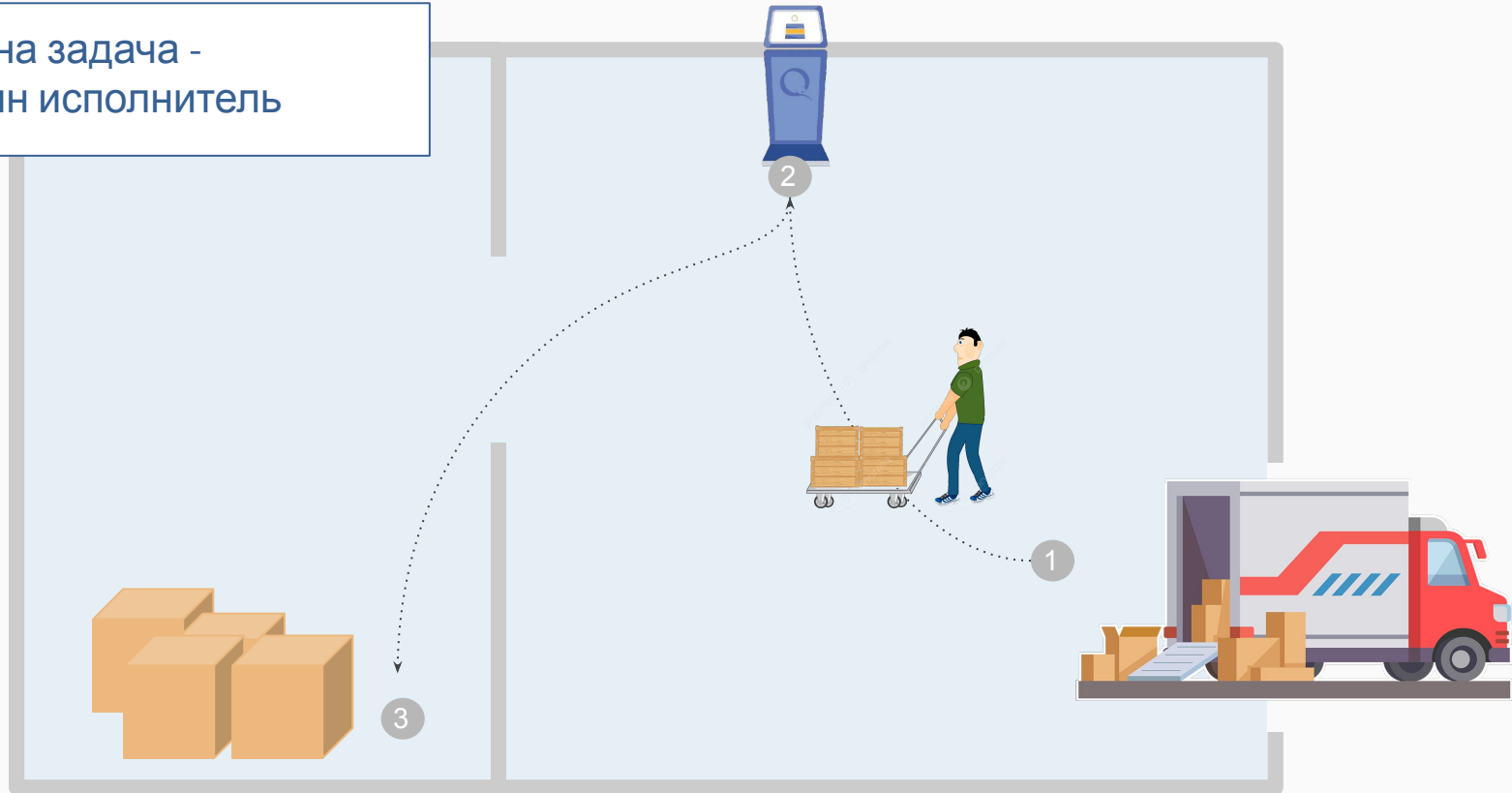
Java BackEnd

Multithreading



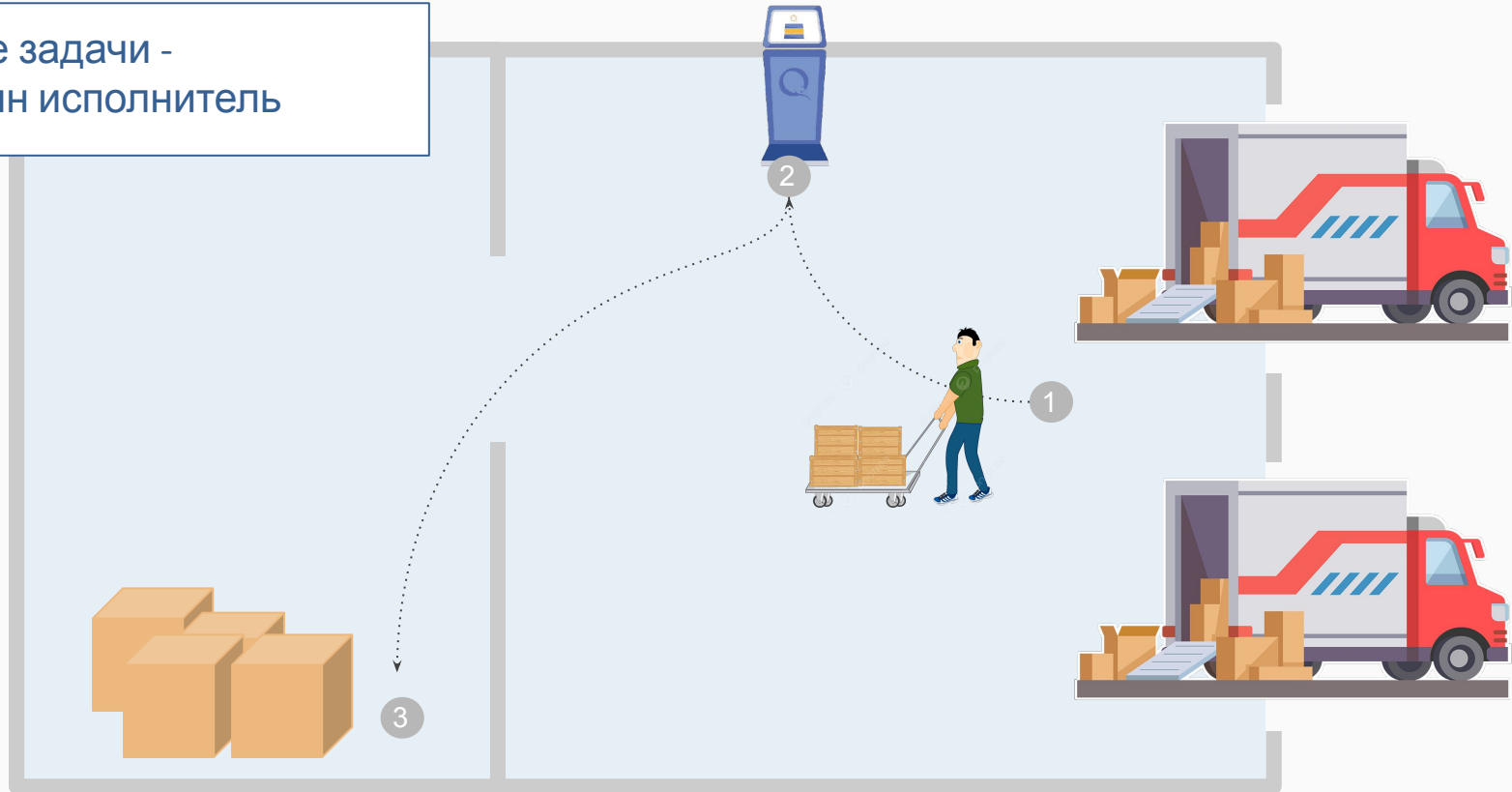
Multithreading

Одна задача -
один исполнитель



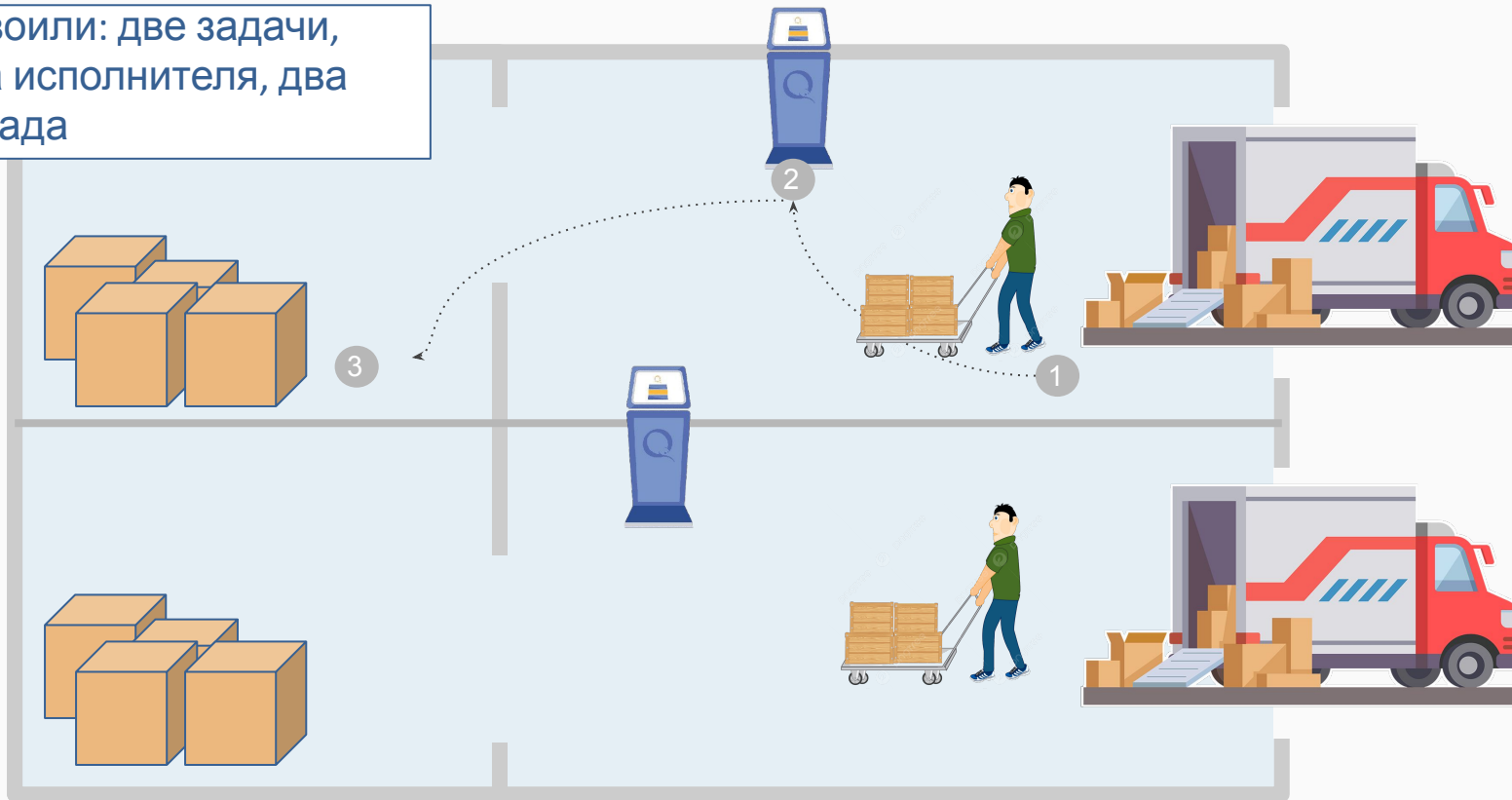
Multithreading

Две задачи -
один исполнитель



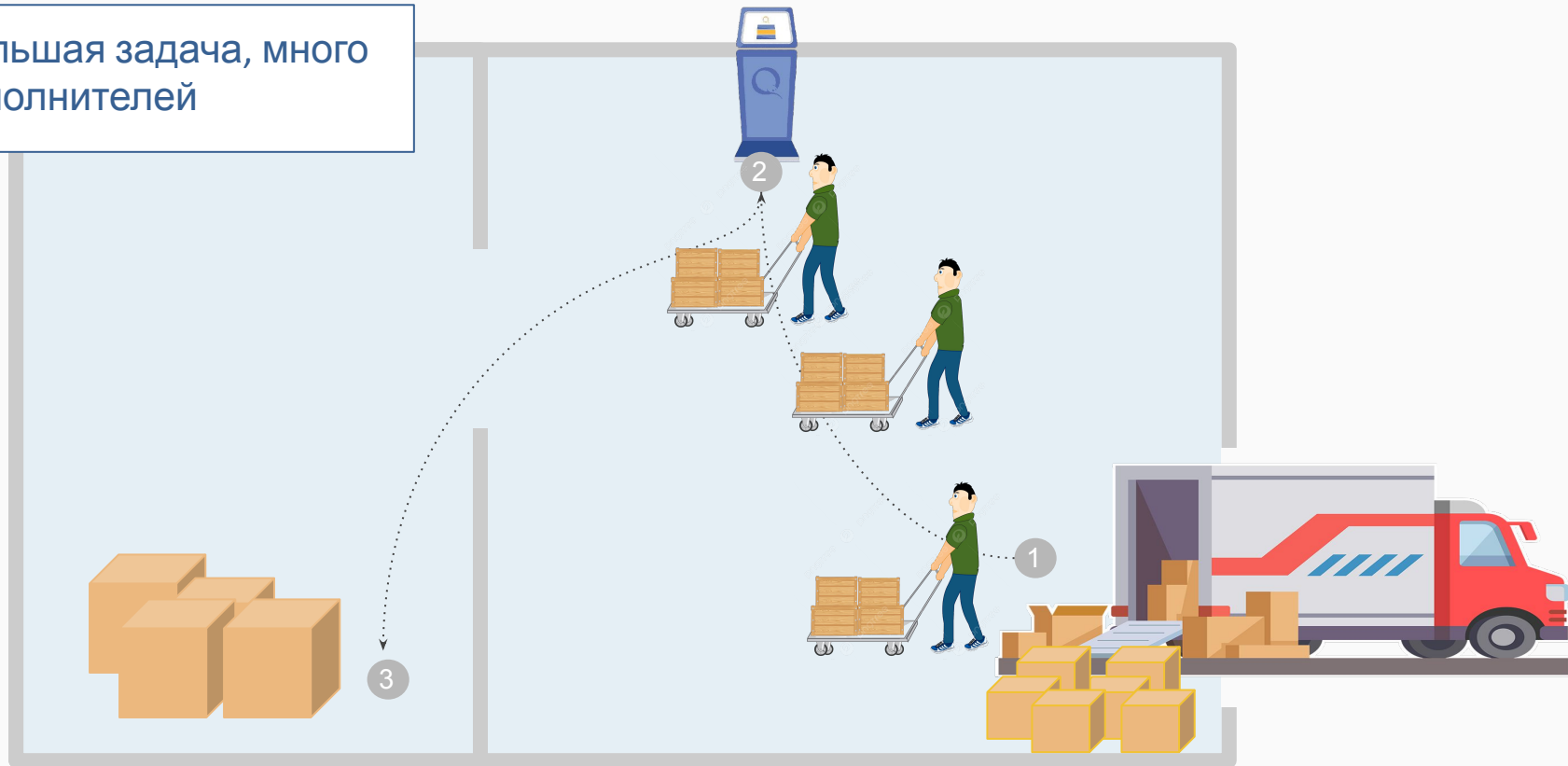
Multithreading

Удвоили: две задачи,
два исполнителя, два
склада



Multithreading

Большая задача, много исполнителей



Multithreading

Много исполнителей



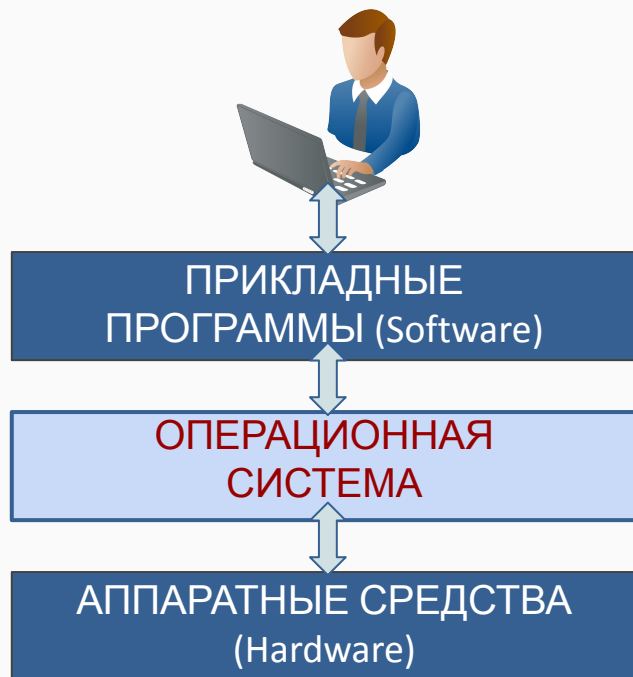
Конкурентность (Concurrency)

КОНКУРЕНТНОСТЬ (CONCURRENCY) — это подход, предполагающий управление несколькими задачами в одно и то же время, даже если они не выполняются в буквальном смысле одновременно.



- **Параллелизм (Parallelism)** — реальное одновременное выполнение задач (при наличии нескольких CPU)
- **Многозадачность (Multitasking)** — способность операционной системы запускать несколько приложений одновременно.
- **Многопоточность (Multithreading)** — возможность одного процесса выполнять несколько потоков одновременно
- **Процесс (process)** — программа в состоянии выполнения
- **Поток (Thread)** — легковесный процесс, отдельная линия кода в рамках процесса. Несколько потоков могут работать в рамках одного процесса и делить между собой память (heap).

Операционная система



ОПЕРАЦИОННАЯ СИСТЕМА (operating system, OS) — комплекс программ, который действует как интерфейс между приложениями и пользователями с одной стороны, и аппаратурой компьютера с другой стороны.

ОС как виртуальная машина :

Операционная система скрывает аппаратное обеспечение компьютера от прикладных программ пользователей. И пользователь, и его программы взаимодействуют с компьютером через интерфейсы операционной системы.

Основные задачи операционной системы:

- **управление аппаратными ресурсами компьютера**

- распределение загрузки процессоров
- распределение памяти
- IO (файловая система, диски, сеть, GUI и т.д.)
- взаимодействие между компонентами системы

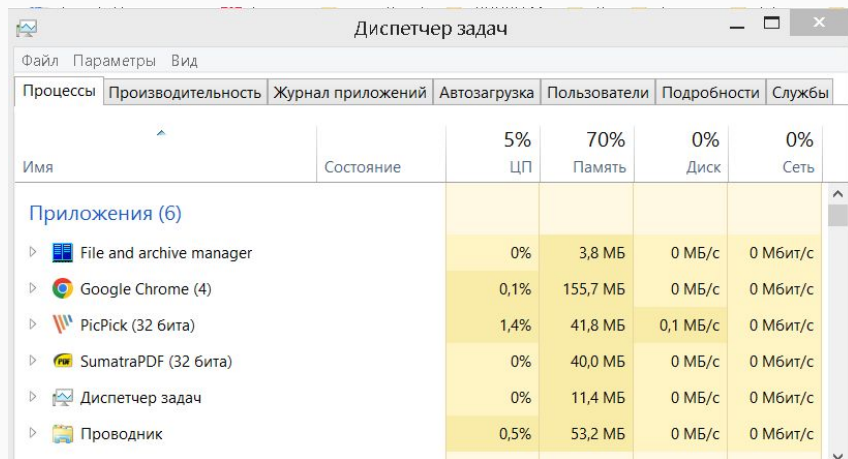
- **управление выполнением программ**

- предоставление аппаратных ресурсов
- обеспечение средств коммуникации и синхронизации выполняемых процессов

- **организация взаимодействия программной и аппаратной частей системы**



МНОГОЗАДАЧНОСТЬ (*multitasking*)
— способность операционной
системы запускать несколько
приложений (процессов)
одновременно



Диспетчер задач

Файл Параметры Вид

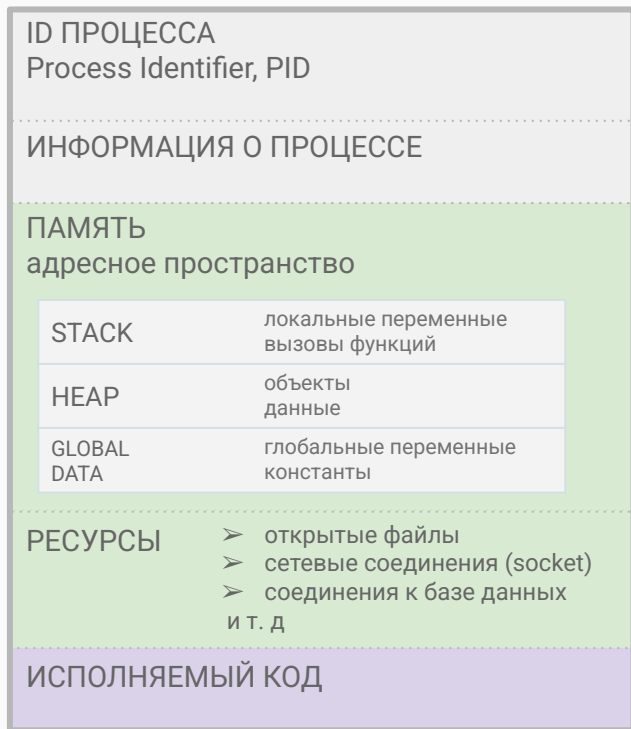
Процессы Производительность Журнал приложений Автозагрузка Пользователи Подробности Службы

Имя	Состояние	5% ЦП	70% Память	0% Диск	0% Сеть
Приложения (6)					
File and archive manager		0%	3,8 МБ	0 МБ/с	0 Мбит/с
Google Chrome (4)		0,1%	155,7 МБ	0 МБ/с	0 Мбит/с
PicPick (32 бита)		1,4%	41,8 МБ	0,1 МБ/с	0 Мбит/с
SumatraPDF (32 бита)		0%	40,0 МБ	0 МБ/с	0 Мбит/с
Диспетчер задач		0%	11,4 МБ	0 МБ/с	0 Мбит/с
Проводник		0,5%	53,2 МБ	0 МБ/с	0 Мбит/с

Вытесняющая многозадачность
(preemptive multitasking)

- Операционная система сама передает управление от одной выполняемой программы другой.
- Распределение процессорного времени осуществляется планировщиком процессов.
- Операционная система может прервать (вытеснить) текущий процесс в любой момент времени и передать управление другому процессу.

Процессы и потоки



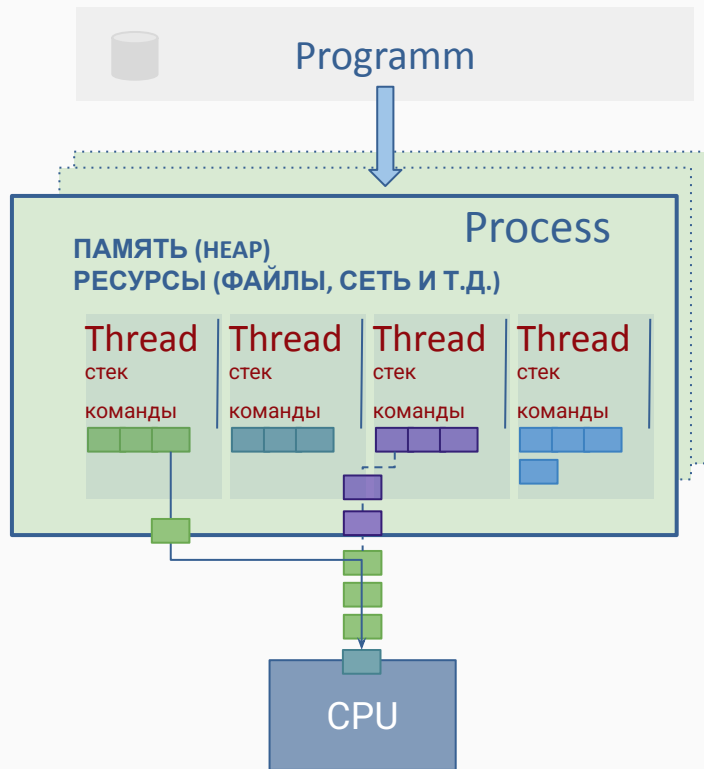
ПРОЦЕСС (process) — работающее приложение*, плюс ресурсы, выделенные для его выполнения.

Каждый процесс обладает **собственным набором ресурсов** таких как оперативная память (адресное пространство), файлы, сетевые соединения и т.д.

* Как правило, одна выполняемая программа соответствует одному процессу, но бывают и исключения (например, Chrome создает отдельный процесс для каждой вкладки)

Операционная система создают иллюзию того, что каждый процесс полностью распоряжается всей компьютерной системой, хотя обычно несколько процессов работают в конкурентном режиме

Современные ОС, как правило, ограничивают возможности процессов по управлению другими процессами. Процессы могут взаимодействовать опосредованно, используя механизмы IPC (Inter Process Communication), через pipes, сокеты, сигналы, сообщения, CORBA, разделяемые файлы или память



ПОТОК (*thread*) — подпроцесс, отдельная линия кода в рамках процесса, выполнением которой управляет операционная система.

- Thread — легковесный процесс (*Light-weight process*)
- Правильный перевод Thread – «нить», но общепринятым является термин «поток».
- обслуживание потока (создание, уничтожение, переключение) не требует значительных ресурсов (в сравнение с процессом)
- Потоки одного процесса могут работать параллельно, используя используя разные CPU
- Потоки одного процесса имеют доступ к ресурсам процесса.
Общая куча (Heap) — можно легко делиться объектами.
- Каждый поток имеет собственный стек и контекст выполнения, что позволяет им работать независимо.

Единица распределения процессорного времени — поток.

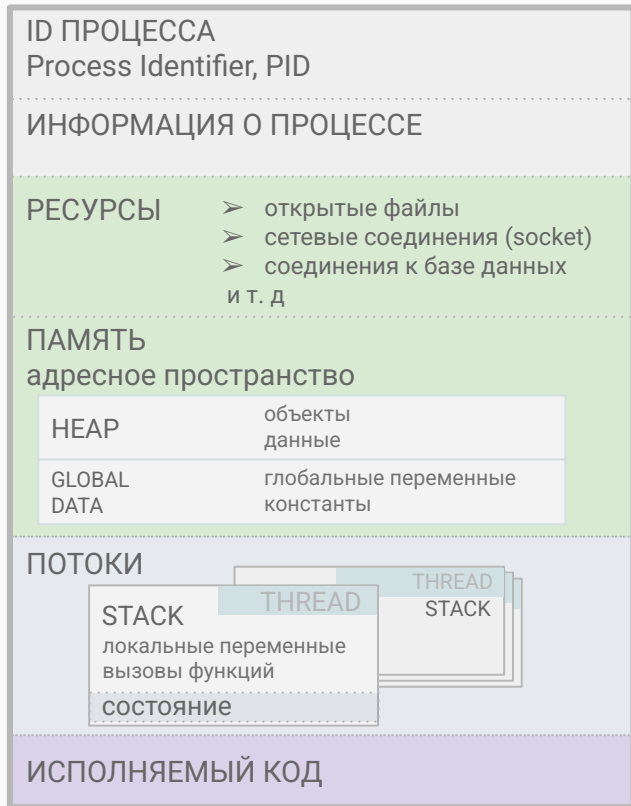


МНОГОПОТОЧНОСТЬ (multithreading) — это способность программы выполнять несколько потоков выполнения (threads), каждый из которых представляет собой отдельную линию исполнения кода.

Как правило, процесс порождает довольно много потоков.

Например, веб-сервер обрабатывает 1000 клиентов — каждому создается отдельный поток (thread-by-request) с собственным стеком, но общими кэширующими структурами в куче.

Process



И процесс и поток — абстракции операционной системы, которые позволяют изолировать задачи и назначить аппаратные ресурсы для выполнения этих задач.

ПРОЦЕСС — ЕДИНИЦА РЕСУРСОВ

ПОТОК — ЕДИНИЦА ИСПОЛНЕНИЯ

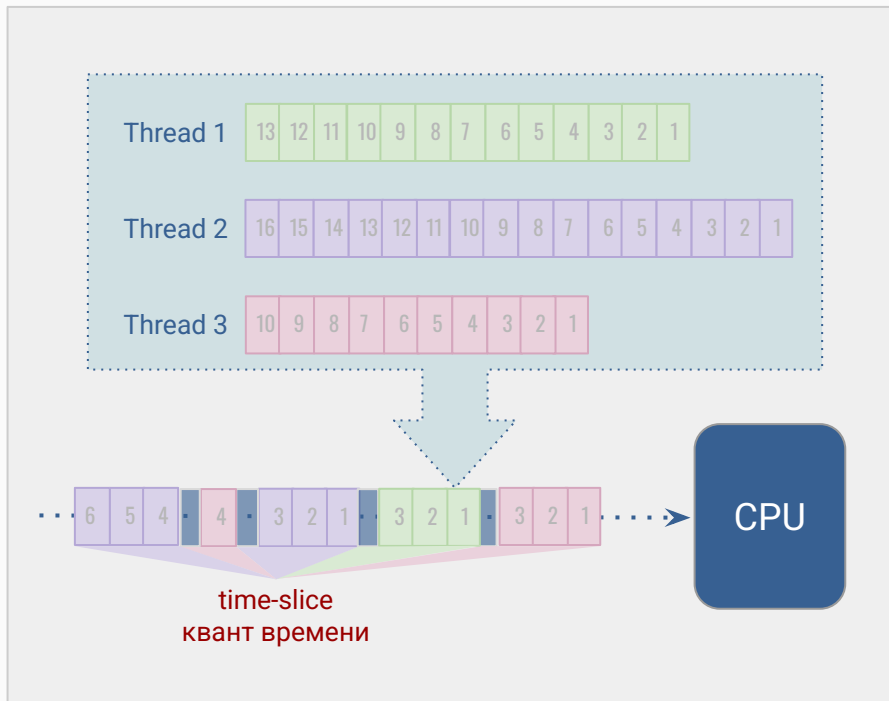
На уровне машинных команд не существует понятия процесса или потока!

* В большинстве современных операционных систем процессорное время распределяется на уровне потоков (threads)

Поток - виртуальный процессор?

Process vs Thread

PROCESS (ПРОЦЕСС)	THREAD (ПОТОК)
Позволяет запустить несколько программ в рамках ОС	Позволяет запустить несколько задач в рамках процесса
Создание и уничтожение процесса очень дорогостоящие операции.	Создание и уничтожение потока в десятки раз быстрее создания и уничтожения процесса.
Процессы имеют независимые друг от друга адресные пространства и ресурсы.	Несколько потоков одного процесса напрямую делят память и ресурсы
Обмен данными между процессами достаточно сложен и требует использования определенных механизмов IPC (Inter Processing Communication)	Общая память позволяет легко делиться данными с другими потоками в рамках одного процесса.
Процесс всегда создает хотя бы один поток, но как правило, включает в себя множество потоков	Поток является частью некоторого процесса
Однопоточный процесс одновременно использует только один CPU.	Многопоточный процесс может выполнять параллельно несколько потоков используя разные CPU
Дорогой Switch Context	Switch Context обходится значительно дешевле



Квантование времени (Time-slicing)

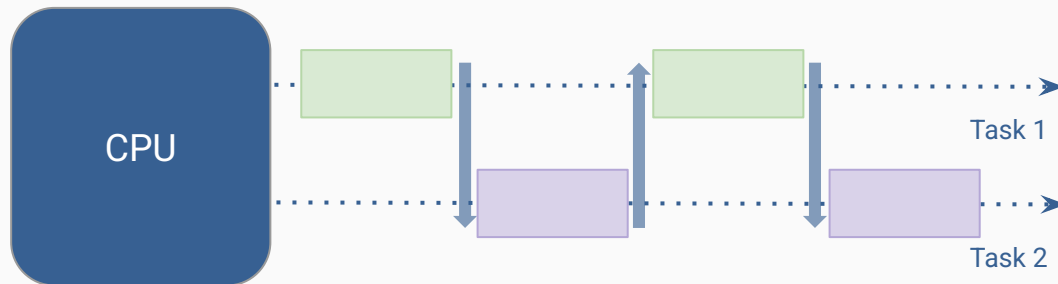
- Время разделяется на интервалы - *кванты времени (time slice)*
- Во время одного кванта обрабатывается один поток
- Решение о выборе потока принимается до начала интервала
- Переключение между потоками с высокой частотой
- Выбор потока не обязательно определяется его приоритетом или очередностью

Иллюзия параллельности даже в
однопроцессорной системе

КОНТЕКСТ ПРОЦЕССА
(Process Context) — это всё то, что необходимо ОС, чтобы "поставить процесс на паузу и потом возобновить его с того же места".

Process control block (PSB) — это структура данных, используемая операционной системой для хранения всей информации о процессе.

- Process ID (PID)
- Состояние процесса
- Регистры CPU
- Program Counter
- Указатели на Стек и Heap
- Открытые файлы
- Приоритет
- Использование CPU и I/O



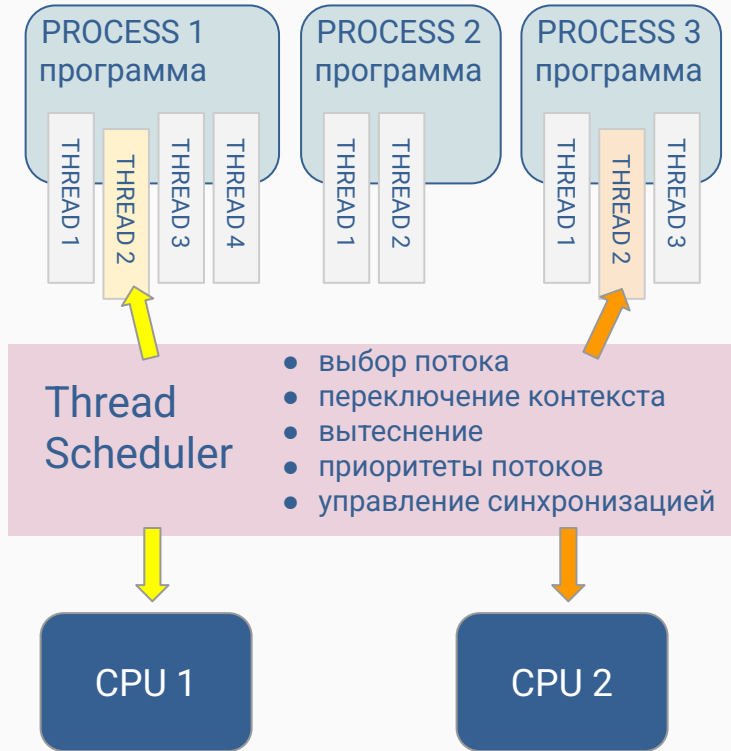
Переключение контекста (Context Switch)

Переключение контекста — момент передачи управления от одной задачи к другой. Операционная система:

1. Сохраняет контекст текущего процесса (PCB — Process Control Block или "Дескриптор процесса")
2. Загружает контекст другого процесса
3. CPU продолжает выполнять уже другой процесс, как будто ничего не произошло

Это позволяет одному ядру обслуживать много процессов по очереди.

Thread Scheduler



ПЛАНИРОВЩИК ПОТОКОВ — это компонент операционной системы, отвечающий за управление выполнением потоков. Он решает, какой поток будет запущен в каждый момент времени, обеспечивая эффективное использование ресурсов ЦП.

Основные задачи планировщика включают:

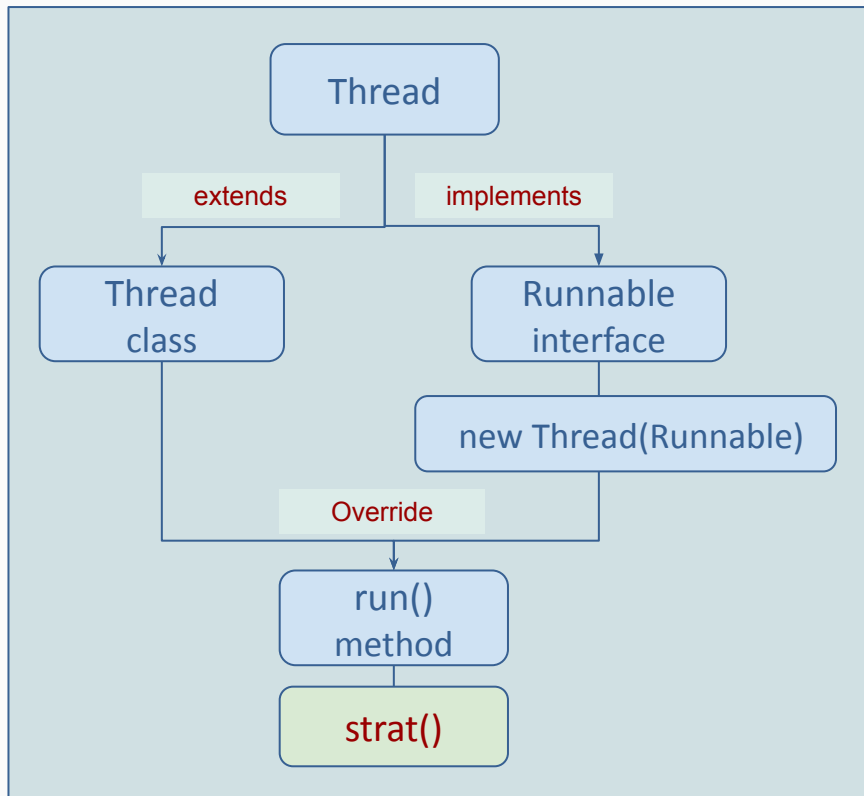
- **Приоритезация потоков:** на основе таких факторов, как приоритет, время выполнения или справедливость.
- **Переключение контекста:** перемещение между потоками для обеспечения многозадачности.
- **Вытеснение:** временная остановка потока для выполнения другого потока.
- **Обработка синхронизации:** управление доступом потоков к общим ресурсам без конфликтов.

... распределить M потоков на N ядер CPU

Многопоточность в Java



- *Каждый экземпляр JVM выполняющий вашу программу — процесс.*
- *Java программа создает как минимум один поток в котором выполняется метод `main()`*
- *С точки зрения Java поток — объект класса `java.lang.Thread` (или его наследника).*
- *Поток в Java-программе не всегда соответствует потоку в ОС, но как правило это так.*



Создать Thread можно двумя способами:

1. Создать объект- наследник класса Thread
2. Реализовать Runnable. Создать объект Thread передав экземпляр-реализацию Runnable в конструкторе

Как в первом, так и во втором случае необходимо переопределить метод `run()`, в котором определить “работу потока”.

Для старта потока необходимо использовать метод `start()`

Единственным способом создать поток является создание экземпляра класса Thread. При этом потоку необходимо передать код для исполнения

Создание Thread

Создание потока через наследование от Thread

```
class MyThread extends Thread{

    @Override
    public void run() {
        // код метода run() будет
        // выполнен в отдельном потоке
        System.out.println("Hello from thread");
    }
}
```

старт потока на выполнения

```
/* создаем объект класс MyThread */
Thread thread = new MyThread();

/* Стартуем выполнение потока */
thread.start(); // !!! НЕ run() !!!
```

... или наследуем от Thread через анонимный класс

```
var thread = new Thread(){
    @Override
    public void run() {
        System.out.println("Hello from anonymous Thread!");
    }
};
thread.start();
```

Создание потока через реализацию Runnable

```
class MyRunnableTask implements Runnable{

    @Override
    public void run() {
        // код метода run() будет
        // выполнен в отдельном потоке
        System.out.println("Hello from runnable thread");
    }
}
```

старт потока на выполнения

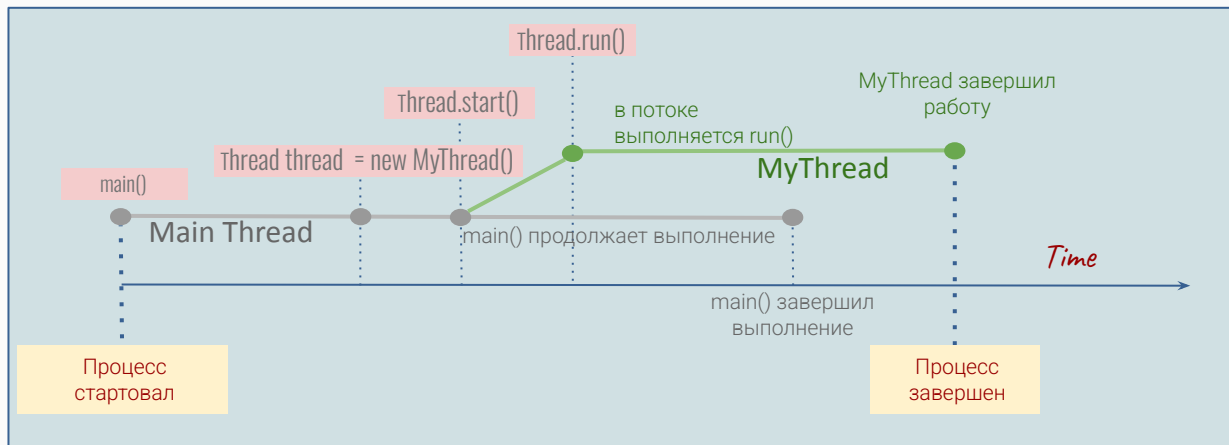
```
/* создаем объект Thread в конструкторе передаем нашу задачу */
Thread thread = new Thread( new MyRunnableTask() );

/* Стартуем выполнение потока */
thread.start(); // !!! НЕ run() !!!
```

... или имплементируем Runnable лямбдой

```
/* создаем Thread имплементируя Runnable лямбдой и стартуем его */
new Thread(
    ()-> System.out.println("I am weeny thread")
).start();
```

thread

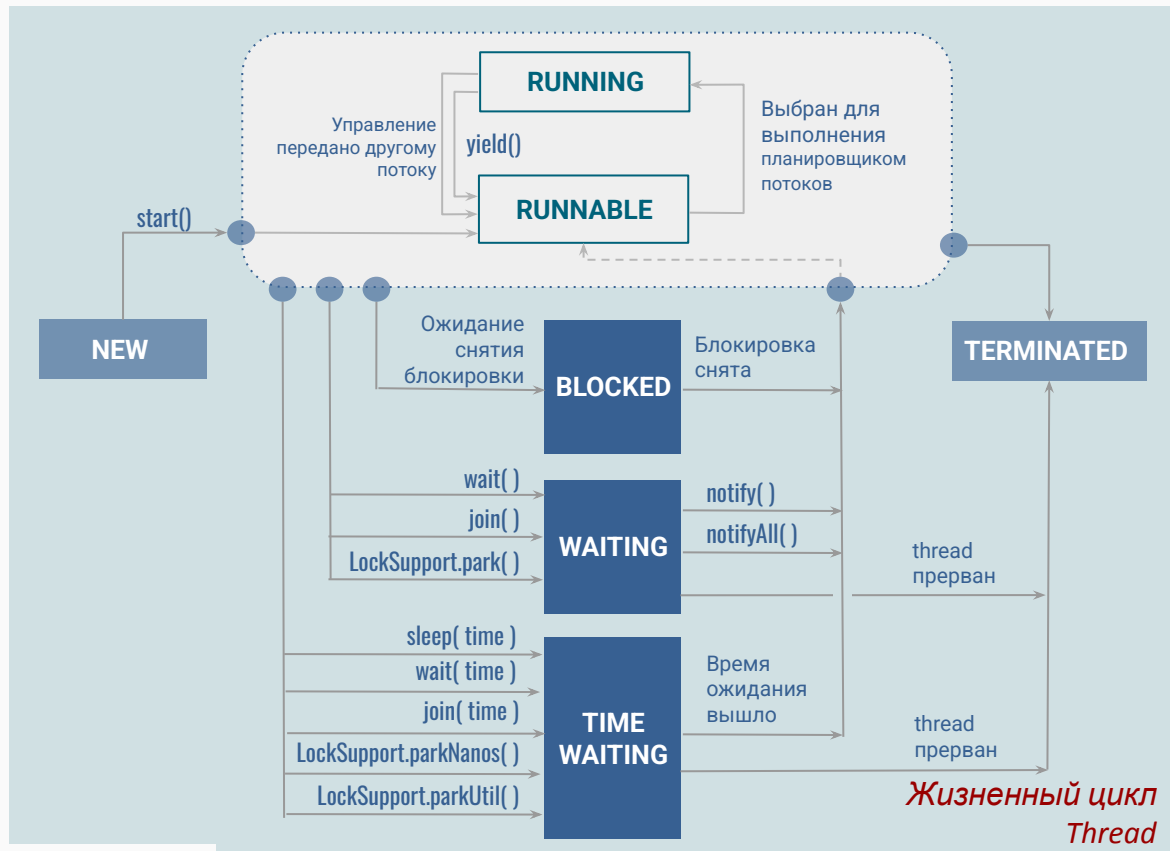


После вызова метода `start()`, потоки переходят в состояние готовности к выполнению (`RUNNABLE`), но их фактический запуск и выполнение зависят от планировщика потоков операционной системы.

start(): Запускает поток. Именно вызов `start()` заставляет JVM создать новый поток выполнения и вызывать метод `run()` в этом потоке.

run() — это метод, содержащий код, который будет выполнен в потоке. Если вызвать `run()` напрямую, без вызова `start()`, код будет выполнен в текущем потоке, как обычный метод, и новый поток не будет создан.

Жизненный цикл потока



NEW Поток создан, но метода `start()` ещё вызывн, поток не запущен. В этом состоянии поток не потребляет системные ресурсы.

RUNNABLE После вызова метода `start()` поток переходит в состояние **RUNNABLE**. Поток может фактически выполняться (**RUNNING**) или ожидать своей очереди на выполнение.

BLOCKED Поток переходит в это состояние, если он пытается войти в `synchronized` блок или метод, доступ к которому удерживается другим потоком. Как только монитор освобождается, поток возвращается в состояние **READY**.

WAITING Состояние ожидания без указания времени. Поток остаётся в этом состоянии, пока не получит уведомление или пока другой поток не завершится (в случае `join()`).

TIMED_WAITING Похожее на состояние **WAITING**, но с указанным временем ожидания. По истечении времени ожидания поток автоматически возвращается в состояние **READY**.

TERMINATED Поток переходит в это состояние, когда метод `run()` завершает своё выполнение (штатно или из-за `exception`). В этом состоянии поток больше не может быть перезапущен. Попытка вызвать `start()` повторно приведет к исключению `IllegalThreadStateException`

Методы Thread

- **start()** – запускает поток, вызывая его метод `run()`.
- **run()** – содержит код, который выполняется в потоке (обычно переопределяется).
- **join()** – заставляет текущий поток ждать завершения другого потока.
- **join(long millis)** – ждет указанное количество миллисекунд.
- **sleep(long millis)** – приостанавливает выполнение потока на указанное время.
- **yield()** – даёт шанс другим потокам выполниться, но не гарантирует переключение.
- **interrupt()** – прерывает поток, устанавливая флаг `interrupted`.
- **isInterrupted()** – проверяет, был ли поток прерван.
- **interrupted()** (статический) – проверяет и сбрасывает флаг прерывания.
- **setPriority(int newPriority)** – устанавливает приоритет (от `Thread.MIN_PRIORITY = 1` до `Thread.MAX_PRIORITY = 10`).
- **getPriority()** – возвращает приоритет потока.
- **setName(String name)** – задаёт имя потока.
- **getName()** – возвращает имя потока.
- **getId()** – получает уникальный идентификатор потока.
- **currentThread()** – возвращает текущий выполняющийся поток.

join()

thread.join() — блокирует текущий поток до завершения потока *thread*.

Поток в котором вызван метод join() (текущий поток) ждет завершения потока, у которого вызван метод join().

thread.join(long millis) — блокирует текущий поток до завершения потока *thread*, но не более заданного времени



Допустим, в main() есть два потока:

1. thread1: “спит” три секунды и выводит “поток 1”
2. thread2: “спит” одну секунду и выводит “поток 2”

Сравните:

```
thread1.start();
thread2.start();
System.out.println("main(): END");
```

main(): END
Поток 2
Поток 1

```
thread1.start();
thread2.start();
thread1.join();
thread2.join();
System.out.println("main(): END");
```

Поток 2
Поток 1
main(): END

```
thread1.start();
thread1.join();
System.out.println("main(): 1");
thread2.start();
thread2.join();
System.out.println("main(): 2");
System.out.println("main(): END");
```

Поток 1
main(): 1
Поток 2
main(): 2
main(): END



Поток-демон автоматически прерывается JVM, даже если его метод run() не завершен!

ПОТОК-ДЕМОН (daemon thread) — это фоновый поток, который работает пока существуют обычные (user) потоки. Как только все пользовательские потоки завершаются, виртуальная машина Java (JVM) автоматически завершает все потоки-демоны, даже если они еще выполняются.

Потоки-демоны:

- Не мешает завершению программы – если в приложении остались только потоки-демоны, JVM автоматически завершает их и завершает работу.
- Используется для вспомогательных, фоновых задач. Например: мониторинг системы, логирование, сборка мусора и т. д.

Как сделать поток демоном?

Сделать поток демоном можно с помощью метода `setDaemon(true)`. Метод `isDaemon()` – проверяет, является ли поток демоном.

ВАЖНО: сделать поток демоном можно только до старта потока, т.е. до вызова `start()`

Приоритеты потоков



Каждый поток в Java имеет **приоритет**, который определяет, какой поток с большей вероятностью получит процессорное время при планировании задач.

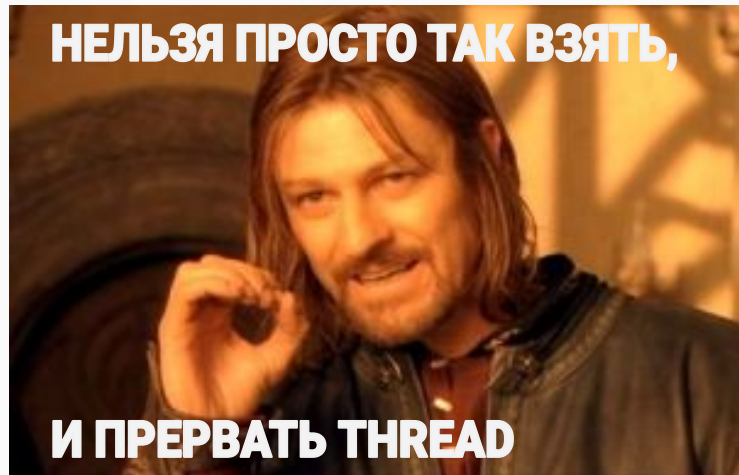
- Приоритет — это всего лишь рекомендация для JVM, а не гарантия.
- В современных JVM приоритеты могут мало влиять, так как ОС сама управляет потоками.
- **НЕ ИСПОЛЬЗУЙТЕ приоритеты для управления логикой программы!**

Приоритет потока — это целое число в диапазоне **от 1 до 10**:

- **Thread.MIN_PRIORITY = 1**
(самый низкий)
- **Thread.NORM_PRIORITY = 5**
(по умолчанию)
- **Thread.MAX_PRIORITY = 10**
(самый высокий)

Приоритет потока можно установить с помощью метода `setPriority(int priority)`. Этот метод должен быть вызван после создания потока, но до его запуска методом `start()`.

Thread Interrupt



Один поток не может остановить другой поток в принудительном порядке!

Метод `interrupt()` не прерывает поток, его задача "попросить" поток корректно завершиться.

`interrupt()`: Отправляет сигнал прерывания потоку, устанавливая флаг прерывания.

`isInterrupted()`: Проверяет, был ли поток прерван, не сбрасывая флаг прерывания.

`interrupted()`: Статический метод, который проверяет флаг прерывания текущего потока и сбрасывает его.

Метод `interrupt()` устанавливает флаг прерывания (`interrupted`) для потока, сигнализируя, что поток был прерван. Поток должен сам проверить этот флаг и принять решение о завершении работы.

Прерывание потока во время ожидания (`sleep`, `wait`, `join`)

Если вызов `interrupt()` попал на момент, когда поток выполняет `sleep()`, `wait()` или `join()`, то:

- выбрасывается исключение `InterruptedException`
- автоматически сбрасывается флаг `interrupted`.

В этом случае, задача программиста корректно обработать `exception`

Thread Interrupt

Пример обработки прерывания:

```
class MyThread extends Thread {
    public void run() {
        while (!isInterrupted()) { // Проверяем флаг прерывания
            System.out.println("Поток работает...");

            try {
                Thread.sleep(1000); // Поток засыпает
            } catch (InterruptedException e) {
                System.out.println("Поток прерван во время сна!");
                break; // Выход из цикла
            }
        }
        System.out.println("Поток завершен.");
    }
}
```

```
public class Main {
    public static void main(String[] args) throws InterruptedException
    {
        MyThread thread = new MyThread();
        thread.start();

        Thread.sleep(3000);
        thread.interrupt(); // Отправляем запрос на прерывание
    }
}
```

- Поток работает в цикле, пока не прерван (`isInterrupted()`).
- При вызове `interrupt()` поток не завершается мгновенно, а получает флаг `interrupted`.
- Если поток спит (`sleep()`), он выбросит исключение `InterruptedException`, и мы сможем завершить выполнение.