

1.Еднонишково и многонишково програмиране.

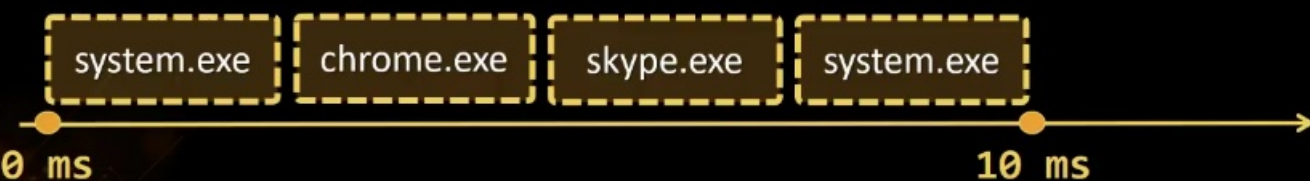
- Time Slicing - Модерните операционни системи могат да изпълняват много задачи едновременно примерно да се пусне скайп, spotify, някой филм и др. и всичко това изглежда, че работи едновременно. Само че когато говорим за едноядрен процесор, Parallelism се случва чрез т.нар. системен scheduler(нещо, което определя графика на процесите).

Java Advanced - Asynchronous Programming - януари 2017 - Петър Пенев

Time Slicing

SOFTWARE UNIVERSITY FOUNDATION

- A computer can run **many processes** (applications) at once
 - But single core CPU can execute one instruction at a time
 - **Parallelism** is achieved by the operating system's **scheduler**
 - Grants each **process** a small interval of time to run



0 ms 10 ms


- Multitasking - Всеки един процес ако го видим по-отблизо ще забележим, че той в себе си също може да изпълнява различни процеси и по абсолютно същия начин да дава на някой процес определено време да си изпълнява функциите и по този начин да изглежда, че се вършат няколко неща едновременно. И тези процеси, които процесорът стартира и дава да се обработват се наричат thread-ове или нишки.

Java Advanced - Asynchronous Programming - януари 2017 - Петър Пенев

Multi-Threading

SOFTWARE UNIVERSITY FOUNDATION

- Processes have **threads** (at least a main thread)
- Similar to OS Multi-Tasking
- By **switching between threads**, a process can **do multiple tasks** "at the same time"



0 ms 10 ms

- Thread - той изпълнява някакви задачи, които са под формата на код. Всеки един Thread(нишка) може изпълнявайки се да пуска и други thread-ове(нишки).

Threads

- A **thread** executes a task
- A thread can **start other threads**



Multiple Threads
"At the same time"

Tasks

- A task is a **block of code** that is **executed by a Thread**
- A **Task in Java** is represented by the **Runnable** class

```
Runnable task = () -> {
    for (int i = 0; i < 10; i++) {
        System.out.printf("[%s] ", i);
    }
};
```

Сега тази задача, за да я пуснем паралелно да се изпълнява в нашата програма трябва да я подадем на отделен thread (нишка).

```
Runnable task = () -> {
    for (int i = 0; i < 10; i++) {
        System.out.printf("[%s] ", i);
    }
};
```

```
Thread thread = new Thread(task);
```

Трябва да се има предвид, че Main thread пускайки нов thread(нишка) не е задължен да го чака, тоест може Main threada като приключи и цялата програма да спре и да не се изчакаат пуснатите паралелно tread-ove да приключат. И за да контролираме това нещо, можем да използваме:

- Join thread-ове - Извикващият thread , ще бъде блокиран докато извиканият не приключи работата си.

- **Join** == waiting for a thread to finish

```
Thread thread = new Thread(() -> {  
    while (true) { }  
});  
  
thread.start();  
System.out.println("Executes.");  
thread.join();  
System.out.println("Can't be reached.");
```

Thread.yield() -> позволява на scheduler-а да превключи на друг task.

Thread Interruption -Ако в даден момент сме пуснали thread и имаме да вършим някаква много тежка задача и в даден момент искаме да я прекъснем. Чрез interrupt можем да спрем thread-а безопасно. Когато е удобен момента scheduler-а ще спре нишката.

- **interrupt()** – notifies the thread to interrupt its execution

```
Thread thread = new Thread(task);  
thread.start();  
thread.interrupt();
```

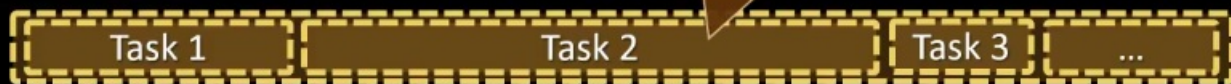
```
Runnable task = () -> {  
    if (Thread.currentThread().isInterrupted())  
        // Safely break the task  
}
```

Основни предимства в използването на многонишков код:

- Потребителският интерфейс е responsive.
- Дава възможност за по-добро използване на процесора.

Multi-Threaded CPU Utilization

Single-Threaded



Multi-Threaded



В Java има класове, които ни дават малко по-голяма абстракция на многонишковите програми. Един от тези класове е `ExecutorService`. Представява клас, но който можем да му кажем в какъв вид да си запазва thread-вете. Управлява ги по какъв начин да се изпълняват. Малко взема ролята на scheduler-а и определя някакъв по-добър ред, който нас ни интересува.

High Level Threading

- **ExecutorService** class provides easier thread management

```
ExecutorService es =  
    Executors.newFixedThreadPool(2);  
  
Runnable task = () -> isPrime(number);
```

```
ExecutorService es =
    Executors.newFixedThreadPool(2);
```



Runnable не може да връща резултат. Той е void. Callable обаче може.

- **Future<T>** - defines a result from a **Callable**:

```
ExecutorService es =
    Executors.newFixedThreadPool(4);

Future<Boolean> future =
    es.submit(() -> isPrime(number));

if (future.isDone())
    System.out.println(future.get());
```

При по-бърз алгоритъм, когато се стартират повече thread-ве, изпълнението на програмата може да е по-бавно, защото когато става дума за някакви задачи(tasks), които се изпълняват бързо, тогава създаването и поддържането на нови thread-ве добавя някакъв overhead към програмата.

От това че работим с повече от 1 нишка едновременно, могат да възникнат много проблеми. Първо - Какво става ако работим с повече от 1 нишка ? Многото нишки могат да работят с едни и същи ресурси. Примерно ако имаме някаква променлива, която в даден момент от време трябва да се презапише от двете нишки. Какво се случва ? Коя от двете нишки печели ? .. Печели тази, която презапише последна.

Atomicity - В Java това са всички операции, които се извършват като едно цяло. Atomicity операции са четенето и писането

Java Advanced - Asynchronous Programming - януари 2017 - Петър Пенев


Atomicity

SOFTWARE UNIVERSITY FOUNDATION

- Atomic action is one that happens all at once
- Java – reads and writes on primitives (except double and long)

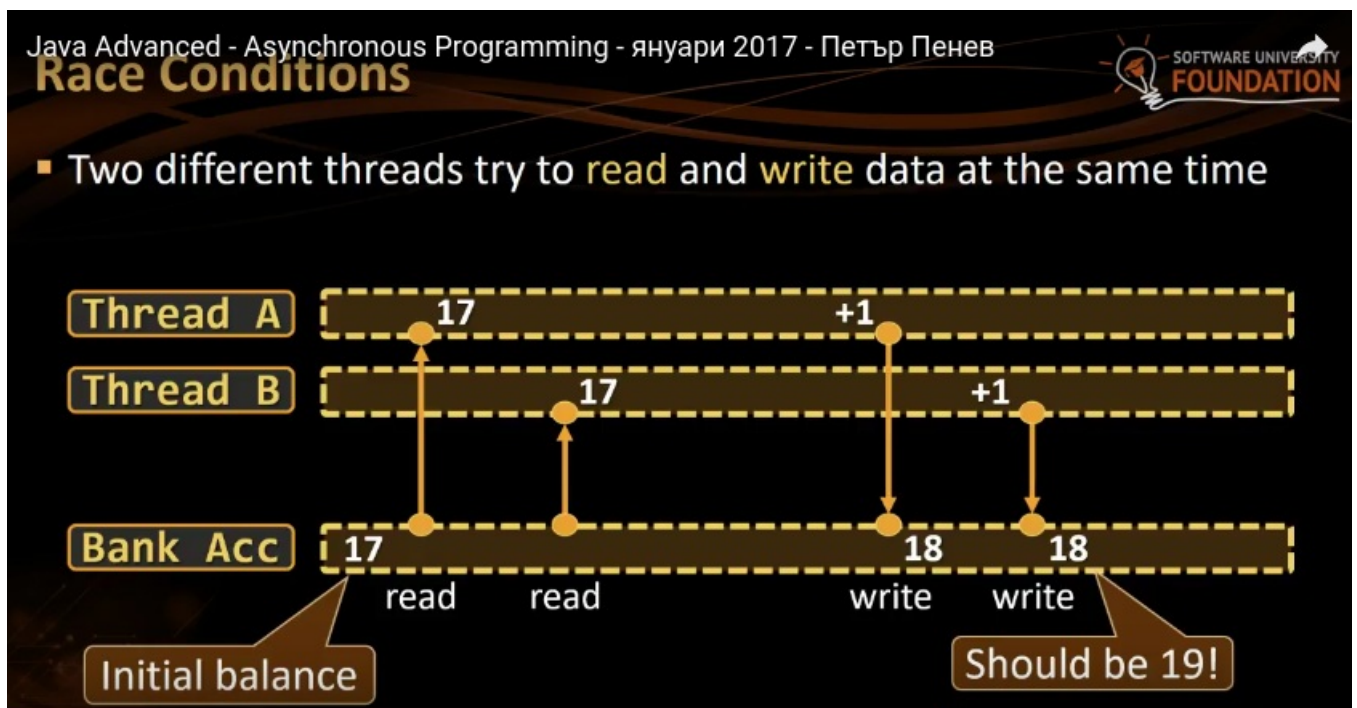
```
int a = 5; // atomic
int b = 6;

a++; // non-atomic
a = a + b;
a = b;
```



Атомарна операция е такава, която не може да бъде прекъсната по средата.

Race Conditions - Когато две или повече нишки се опитват да четат и пишат по файл едновременно. Както се вижда от следната картинка в случая с банката едното плащане се губи. Крайната сума трябва да е 19, а не 18. Това се случва, защото първият thread A започва операциите по четене, записване и пресмятане на новата стойност, това е атомарна операция и не се извършва като едно цяло. И в промеждутъка от време докато thread A изпълнява операциите, се намира thread B, който взема още не инкрементираната стойност на thread A - 17. Тоест взема старата стойност 17 вместо новата 18 и съответно почва да работи с нея. И оттогава идва проблемът където се губи едното плащане.



Този проблем се оправя с ключовата дума synchronized. Чрез тази команда никой друг thread не може да се вмъкне в промеждутъка от време докато се изпълнява атомарната функция.. Тоест в горния пример thread B няма да може да се вмъкне и да вземе стойността на thread A по този начин. Първо thread A ще завърши изпълнението си и тогава thread B ще вземе стойността на thread A, която този път вече ще е инкрементирана и ще започне да работи върху новата стойност.

Как работи синхронизирането ?

Synchronized - Locks

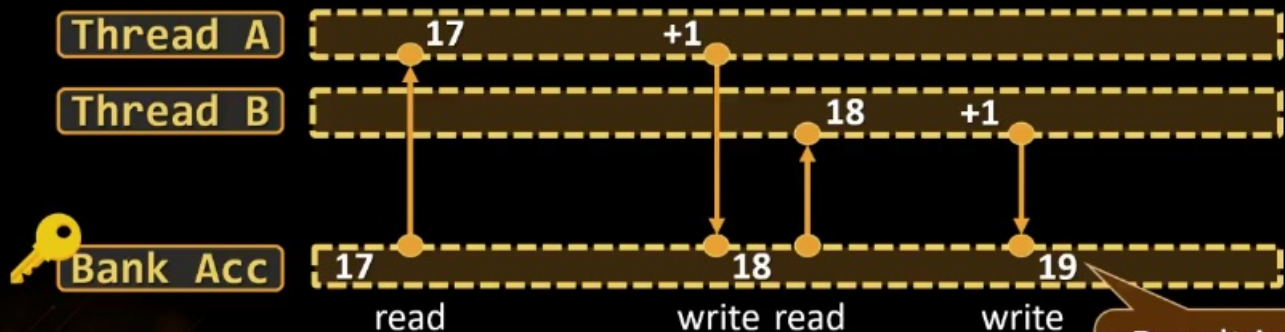
- Synchronized works by taking an object's Key



Синхронизирането работи с ключове. Един thread може да вземе ключа на даден обект. Всеки такъв в Java си има ключ. А в Java всичко е наследник на обект по някаква линия, тоест абсолютно всичко в Java притежава ключ.

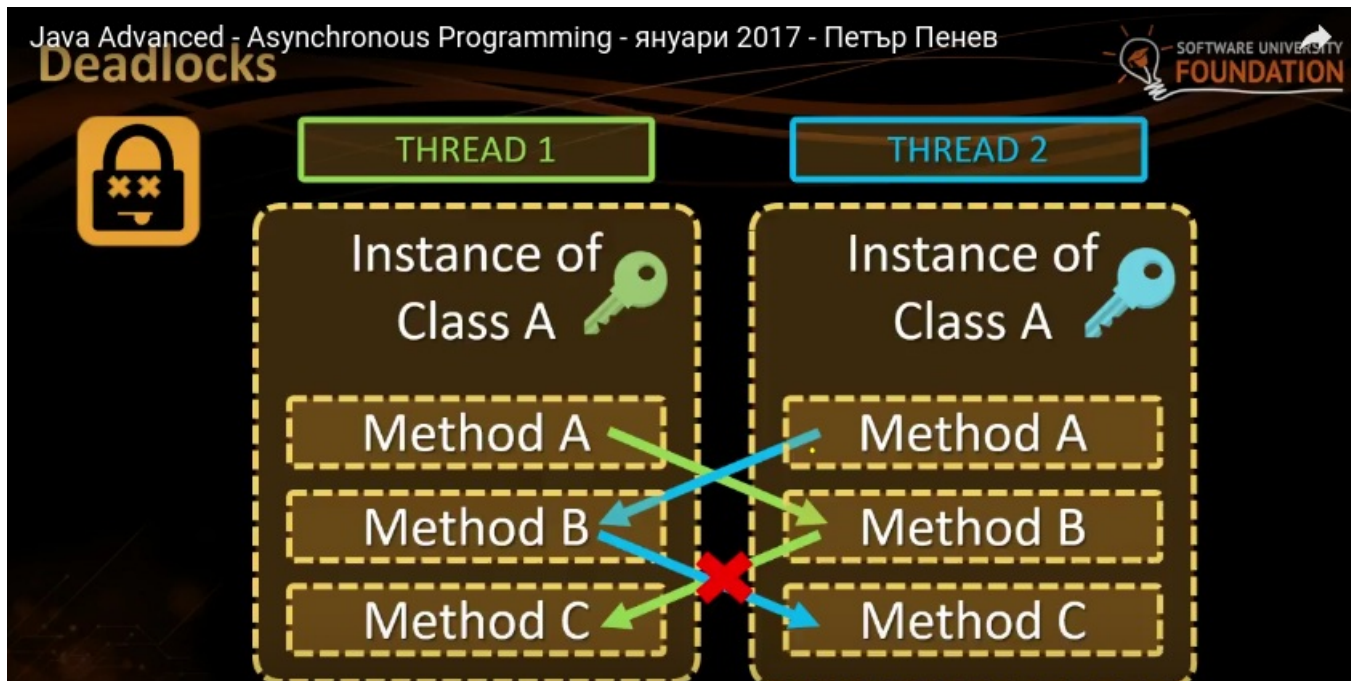
Synchronized - Locks

- Synchronized works by taking an object's Key

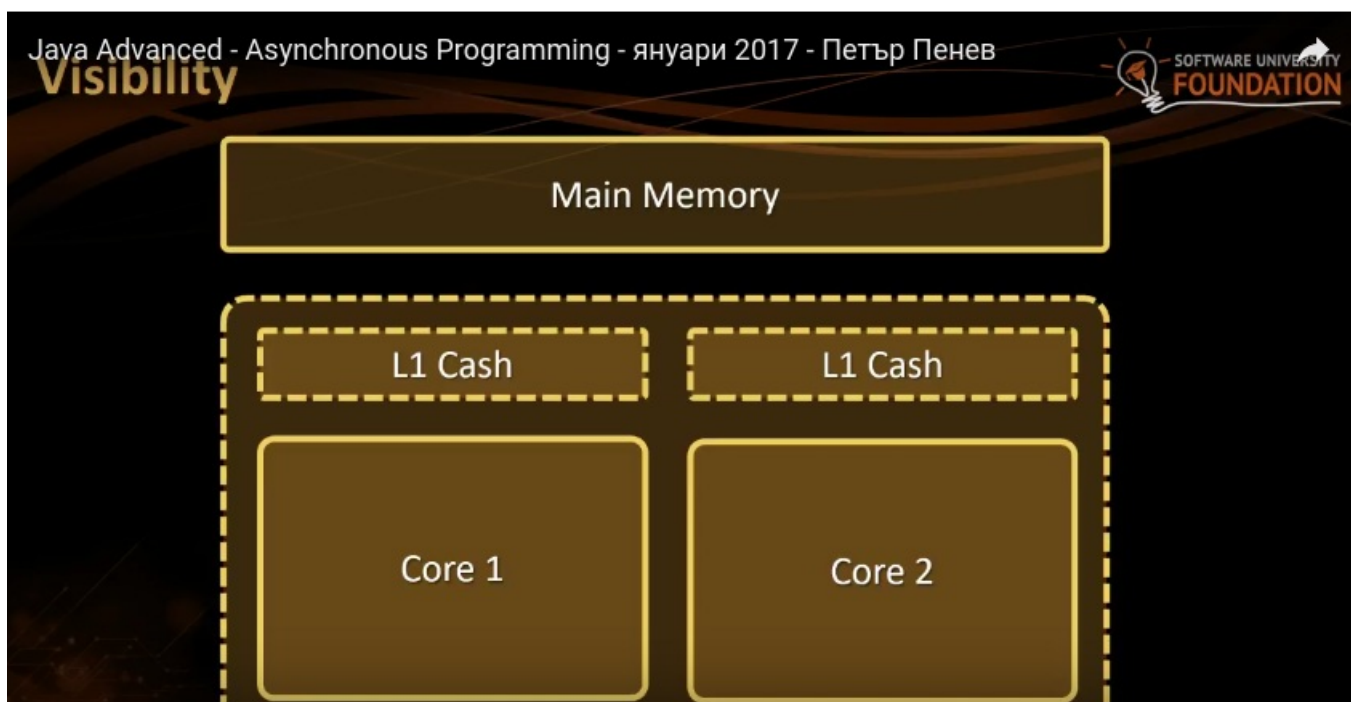


Result is correct!

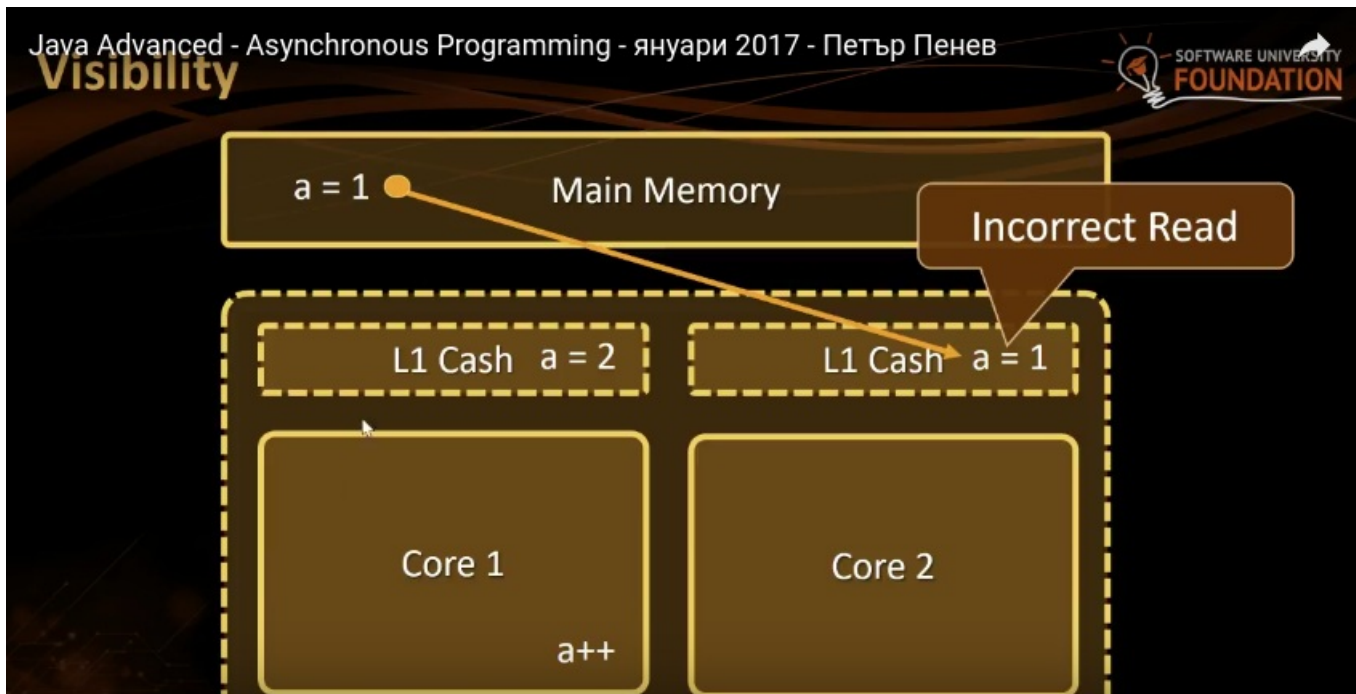
Deadlocks- Когато два различни threada искат да изпълнят едно и също нещо и единият thread е взел ключа на втория, а той пък е взел ключа, който трябва на първия и по този начин се блокират взаимно.



Visibility - видимост на променливите.



За примера да кажем, че разполагаме с двуюдрен или повече ядра компютър. За опростяване на примера, на картинката е двуюдрен компа. Модерните процесори работят по такъв начин, че освен основна памет - Main Memory, те си имат и кеш. Това е един друг вид памет, която е в пъти по-бърза от основната(Main Memory) и се използва, за да оптимизира работата на процесора. За да може всяко едно от ядрата да не записва всичко, което прави в основната памет(Main Memory), тъй като това е бавна операция. И затова си оптимизира работата като запазва временни променливи в кеша... и от това какъв проблем би могло да има ? .. Той не идва от thread-вете и многонишковото програмиране, а просто по такъв начин работят съвременните процесори. Проблемът е показан на следващата картинка



Проблемът показан на картинката се нарича Visibility. Как се оправя ? Този проблем се оправя с ключовата дума `volatile`. Ако се сложи тази дума пред някоя променлива , тя няма да се записва само в кеша на процесора , но и директно в основната памет - Main Memory.

- Every write inside a synchronized block is **guaranteed to be visible**
- Use `volatile` keyword

```
class Account {
    volatile int balance;
    synchronized void add (int amount) {
        balance = balance + amount;
    }
}
```

В Java има помощни класове(колекции), които са синхронизирани, тоест подходящи са за работа в многонишкова среда.

- Java **java.util.concurrent** package provides thread-safe collection classes
- Some notable concurrent collections:
 - **ConcurrentLinkedQueue**
 - **ConcurrentLinkedDeque**
 - **ConcurrentHashMap**



ТРЯБВА ДА СЕ ВНИМАВА, КОГАТО СЕ РАБОТИ С НИШКИ. ИЗПОЛЗВАНЕТО НА ПОМОЩНИТЕ КЛАСОВЕ НЕ ГАРАНТИРА , ЧЕ НЯМА ДА СЕ СЛУЧАТ ПРОБЛЕМИТЕ СПОМЕНАТИ ПО ВРЕМЕ НА ЛЕКЦИЯТА. НАПРОТИВ, ПАК СЕ СЛУЧВАТ И ТРЯБВА ДА СЕ ВНИМАВА. КОЛЕКЦИИТЕ СА ОПТИМИЗИРАНИ ДА РАБОТЯТ В МНОГОНИШКОВА СРЕДА, НО НЕ НИ РЕШАВАТ ПРОБЛЕМИТЕ. НИЕ САМИ ТРЯБВА ДА ГИ РЕШИМ.