Lecture #13

Asynchronous programming

Before we get started

Multi threading means that your program runs parallel on **multiple threads**. Nothing more, nothing less. **Async** programming means in layterms that instead of blocking **and** wait for a call (most likely IO operations) you just register a callback **and** you will be notified when the underlying operation has finished.

www.quora.com › What-is-the-difference-between-asynchronous-progr...

What is the difference between asynchronous programming and multi ...

Подробнее о выделенных описаниях... Оставить отзыв

Похожие запросы

- What is the difference between asynchronous and multithreading?
- Is multithreading synchronous or asynchronous?
- What is difference between asynchronous and synchronous?
- What's the difference between asynchronous and synchronous call and when would you use each one?

Оставить отзыв

stackoverflow.com › questions › what-is-the-... ▾ Перевести эту страницу

What is the difference between asynchronous programming and ...

2 ответа

9 янв. 2016 г. - What is the difference between **asynchronous** programming **and multithreading**? ... **Async** methods don't require **multithreading** because an **async** method doesn't run on its own thread. The method runs on the current synchronization context **and** uses time on the thread only when the method is active.

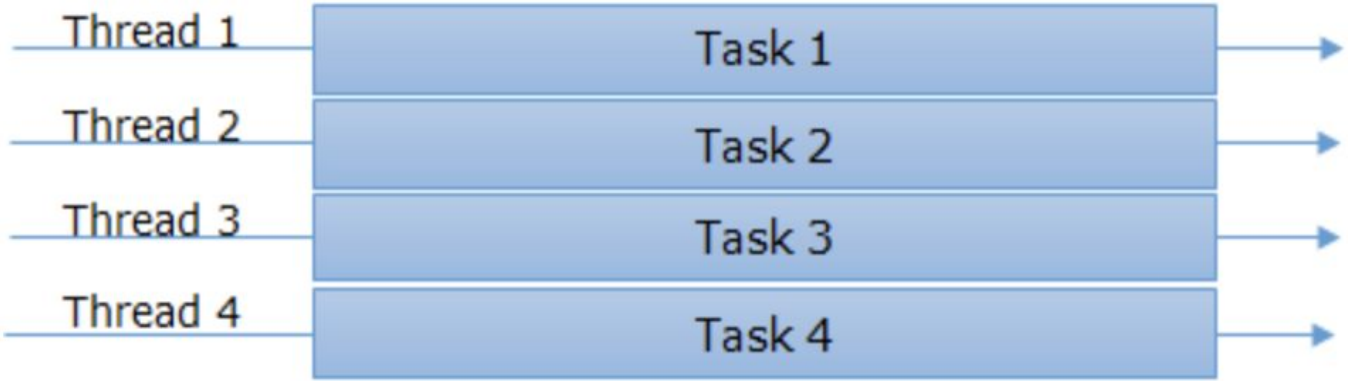
Asynchronous vs synchronous execution, what does ...	22 ответа	14 апр. 2009 г.
Asynchronous vs Multithreading - Is there a ...	10 ответов	2 мар. 2009 г.
Why is **async** considered better performing than ...	2 ответа	17 мар. 2017 г.
Asynchronous vs Multithreading	4 ответа	17 мар. 2013 г.

Другие результаты с сайта stackoverflow.com

Synchronous Programming model

Single Thread:

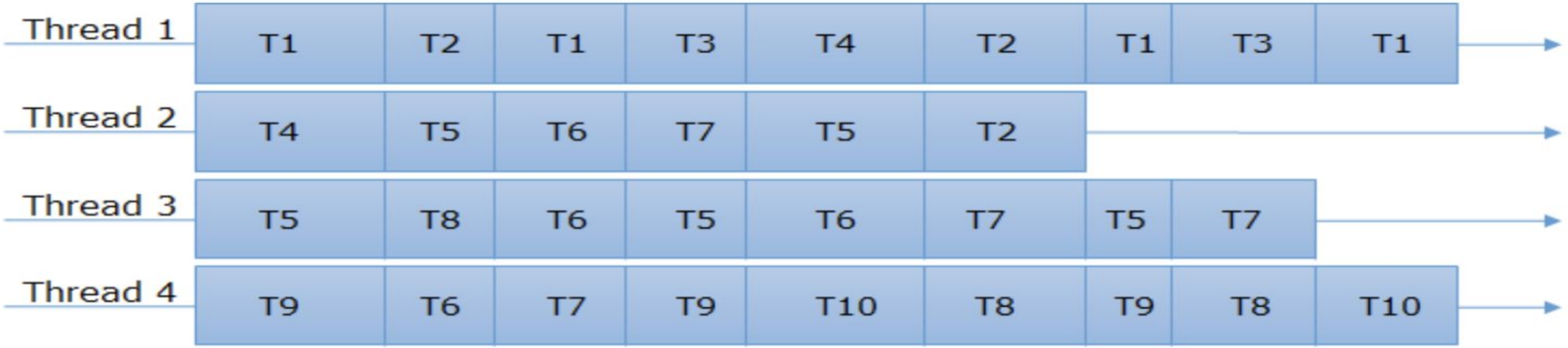
Multithreading:



Asynchronous Programming Model

Single Thread:

Multithreading:



Thread Lifecycle States

1. New
2. Runnable
3. Blocked
4. Waiting
5. Time waiting
6. Terminated

Future & Promise

Future

A `Future` is an object holding a value which may become available at some point. This value is usually the result of some other computation:

1. If the computation has not yet completed, we say that the `Future` is **not completed.**
2. If the computation has completed with a value or with an exception, we say that the `Future` is **completed**.

Completion can take one of two forms:

1. When a `Future` is completed with a value, we say that the future was **successfully completed** with that value.
2. When a `Future` is completed with an exception thrown by the computation, we say that the `Future` was **failed** with that exception.

Callbacks

We have learned a bit about the "Future" yet, but what do we do if we want to get the result of our delayed calculations and take some action with it?

The "future" have the following methods to create callbacks:

- onComplete
- Failed.foreach
- Foreach

Notes:

The order of your callbacks is not respected, they can be execute in any order.

Execution Context

ExecutionContext is a pool of streams that is responsible for how our "Future" will be executed.

There are the following implementations that you could use:

- FixedThreadPool - thread pool that reuses a fixed number of threads. (Everything is created immediately)
- CachedThreadPool - thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available
- ForkJoinPool - a pool of threading that is based on the work stealing algorithm
- Other implementations - if you have a desire, you can also implement your own

Notes:
By default the `ExecutionContext.global` sets the parallelism level of its underlying fork-join pool to the number of available processors.

Promise

While futures are defined as a type of read-only placeholder object created for a result which doesn't yet exist, a promise can be thought of as a writable, single-assignment container, which completes a future. That is, a promise can be used to successfully complete a future with a value (by "completing" the promise) using the `success` method. Conversely, a promise can also be used to complete a future with an exception, by failing the promise, using the `failure` method.

A most useful methods:

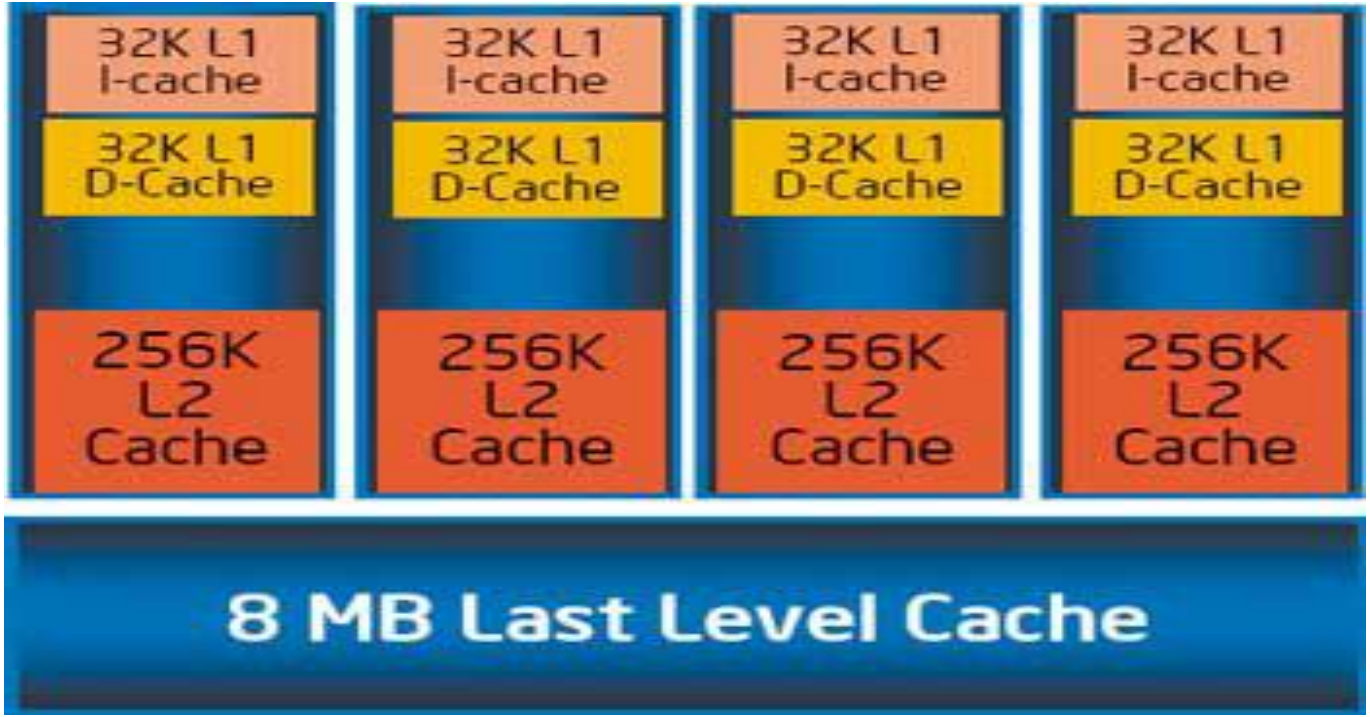
- tryCompete
- tryCompeteWith
- success
- failure

Atomic & Blocking

Volatile

In current realities the system is built in such a way that it tries to maximize the speed of execution and often uses the Cache processor (L1-L3).

In a multi-threaded application this often leads to inconsistency of data, this modifier tries to return reliability by announcing an explicit intention not to store this data in the cache, but always get it from RAM.


32K L1
I-cache
32K L1
D-Cache
256K
L2
Cache
32K L1
I-cache
32K L1
D-Cache
256K
L2
Cache
32K L1
I-cache
32K L1
D-Cache
256K
L2
Cache
32K L1
I-cache
32K L1
D-Cache
256K
L2
Cache
8 MB Last Level Cache

Synchronized Block

Primitive of synchronization of processes and threads work, which is based on the counter, over which it is possible to make two atomic operations: increase and decrease of the value by one.

In this particular case it has only one state (there are implementations that may have a several states).

That is, until the blocking element performs all necessary actions, the other elements are in the queue and wait for the unlock.

Atomic Reference & etc..

Atomic calculations are based on the compare&swap atomic instruction, which is executed in one processor operation and either replaces the value or returns false.

A most useful methods:

- addAndGet
- getAndAdd
- setOpaque - sets the value of a variable to the newValue, in program order, but with no assurance of memory ordering effects with respect to other threads
- setRelease - sets the value of a variable to the newValue, and ensures that prior loads and stores are not reordered after this access
- getAndSet

Additional resources

1. https://docs.scala-lang.org/overviews/core/futures.html