Multithreading and parallelism

Aim of this talk

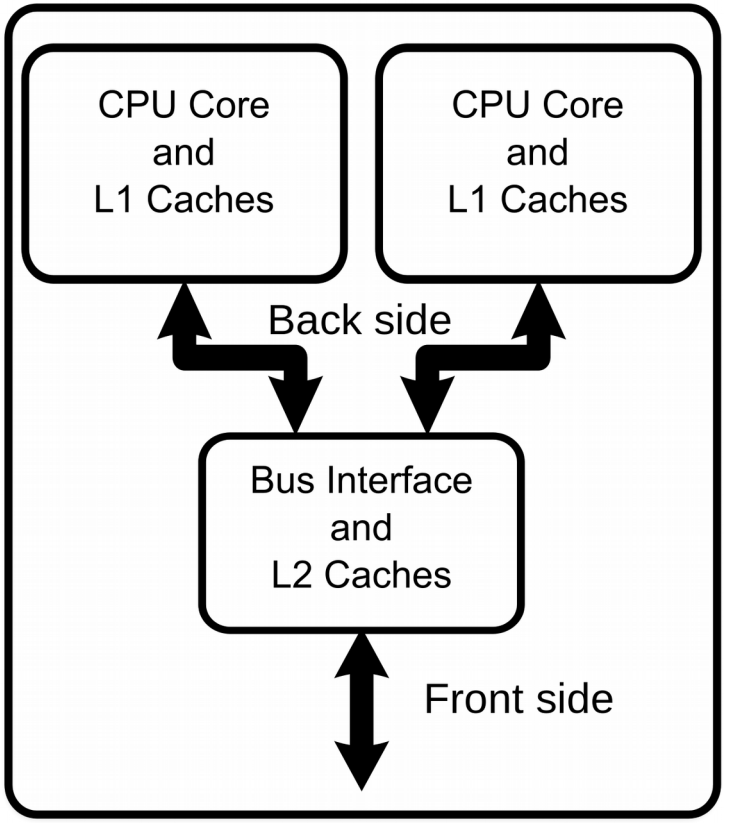
- This talk aims just to be an introduction to multithreading
- Also aims to metaphor parallelism and multitasking to real world, like for example a team of programmers solving some tasks.

NOT the aim

- To explain the GCD as a whole
- To demonstrate how to use lock-free operations on iOS

Definitions

- Core: a hardware unit
- Process: a “software” big task
- Thread: a “software” big task, but still lighter than process, and CAN SHARE DATA (EXTREMELY DANGEROUS)
- Coroutine: a lightweight thread



Definitions (cont.)

- Context switching: A core can do more than one thread (big task), when a core move from one thread to another, switching happens, where a lot amounts of memory is copied in this process. This typically happens when the thread sleeps (either, using “sleep” function, or using locks, and this is why locks are not something good)

Metaphor

- You can imagine cores as programmers
- You can imagine threads as tasks
- You may imagine coroutines as subtasks

Metaphor, handling shared resources

- What about if two programmers are doing somethings dependant on each other ?
- One of them certainly blocks the other, and when a programmer is blocked, he may choose to pick up another task, this is context switching
- When a programmer does a context switching, he takes time “putting himself in the task”, this really happens in threads by copying memory.

Spin locks and locks

- Locks is like we said, it is like there is a task blocked on another one, so the one who needs to work on it, must switch to another task until the blocker is done.
- Spin locks is not like that, it means that the blocked is actively waiting on the blocker, this is used in very small blockers, to avoid context switching. (Metaphor: Developer may wait on some blocking task because moving through git and rethinking another task would take more time if the blocker is small)

Metaphor for cache

- A dev for example is getting information from the official documentation, this is something rarely change, so when solving something similar, instead of reading the docs again, he would use the information he got before, assuming it is NOT changed (but if it is changed, this dev would use outdated info)

Memory model

- If the shared information is subject to change, you instruct both the changer and the reader NOT to use the cache, so that no outdated information would happen (this makes even using locks and even lock-free operations slower than non-shared memory). This technically happens using Java’s volatile keyword, and C/C++/Java’s atomic operations.

Reordering

- Cache is something that has a lot of performance impact, so both the compiler and the processor try to do their best to use the cache, this sometimes involve “reordering” of the operations (either on the same core by the compiler, or the shared data to be observed by the other core by the processor)
- Compiler reordering can be disabled using volatile in C/C++, and any atomic operation is volatile by nature

Metaphor for reordering

- Some times the dev try to implement the similar small subtasks after each others instead of implementing different ones, to get use of their experience quickly (instead of having to go back to the docs and read)

Metaphor in data races

- Sometimes a developer is editing an old version where there is another developer already solved another issue in the same part of the file, imagine doing this without using git
- Theoretically: Sometimes where there one wants to edit data and another in does this on the same place, the data would be hybrid (This is extremely dangerous in programming, it can lead to a random pointer)

Some memory model odds

- Memory model has a lot of odds, for example:

```
public class A {
    public final Object a = new String(“Something”);
    public int Object b = new String(“Other”);
}
A anObject;
if (anObject != null) {
     System.out.println(a); // Always “Something”
     System.out.println(b); // Sometimes “Something” and sometimes “Other” !!!
}
```

Executors in Java

- Thread is something somehow heavyweight
- More than one thread on one core = a lot of context switching which is heavy
- What if the thread is only invoking some functions in a row??

Dispatch

- The concept of executors is what is somehow present in Apple platforms in the form of Grand Central Dispatch
- DispatchQueue can have serial tasks (like Java's SingleThreadExecutor), and may have concurrent tasks (like ForkJoinPool in Java), where the somehow the same concepts of threads are applied, but using function objects instead of OS threads (so lighter).

Coroutines

- In Executor (or DispatchQueue), you find yourself writing code like that:

```
DispatchQueue.global().async {
    doSomething(withCompletion: {
        ...
    })
}
```

Coroutines

- This is why some modern compilers designed the concepts of async/await , coroutines.
- This is a “suspending function”, which something implemented in an object-oriented way, and your code is translated into this style.

  (It is just a compiler trick)

Some patterns

- Counting the operations so that you only execute something after the end of all of them is something common, this may involve an atomic counter which commits a decrement each end of operation (while loading the value before decrement), so if the number reaches zero it invokes a function.

Counting

- How do we use this in iOS ?

  Simply, DispatchGroup, and use group.enter(), group.leave() and group.wait()

  We can also use OperationQueue and make dependencies between different Operations, this is very hard

Signal

- Signal happens at the same thread, and may happen recursively.

  i.e. a function invokes a signal, this immediately starts a function, then for example, the signal function itself signaled something, it would be like recursive calls BUT BEFORE THE END OF THE FUNCTIONS, here is where we could use special things like fences to prevent compiler reordering

  (Compiler or memory ?? And WHY ?)

Signal

- How could you handle signals in more “safe” way?

  Signals should only execute very simple code, naturally just set some flags, variables, etc... so that when the normal execution restores, we would respond normally

- In iOS? DispatchSource has a signal source