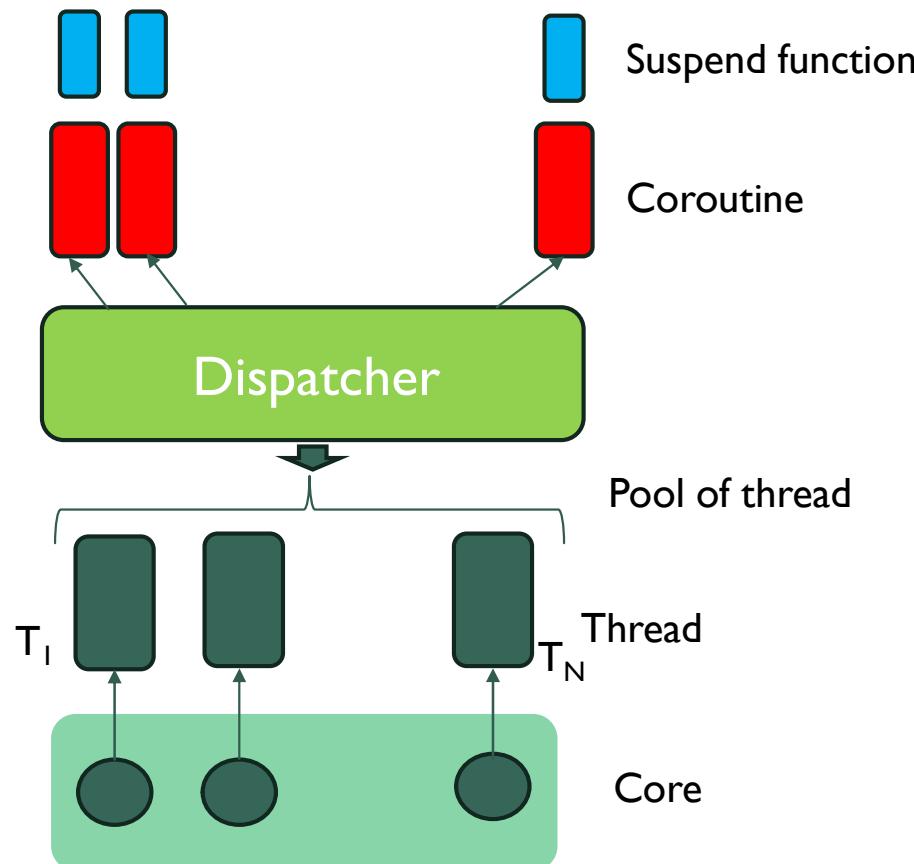




# COROUTINES

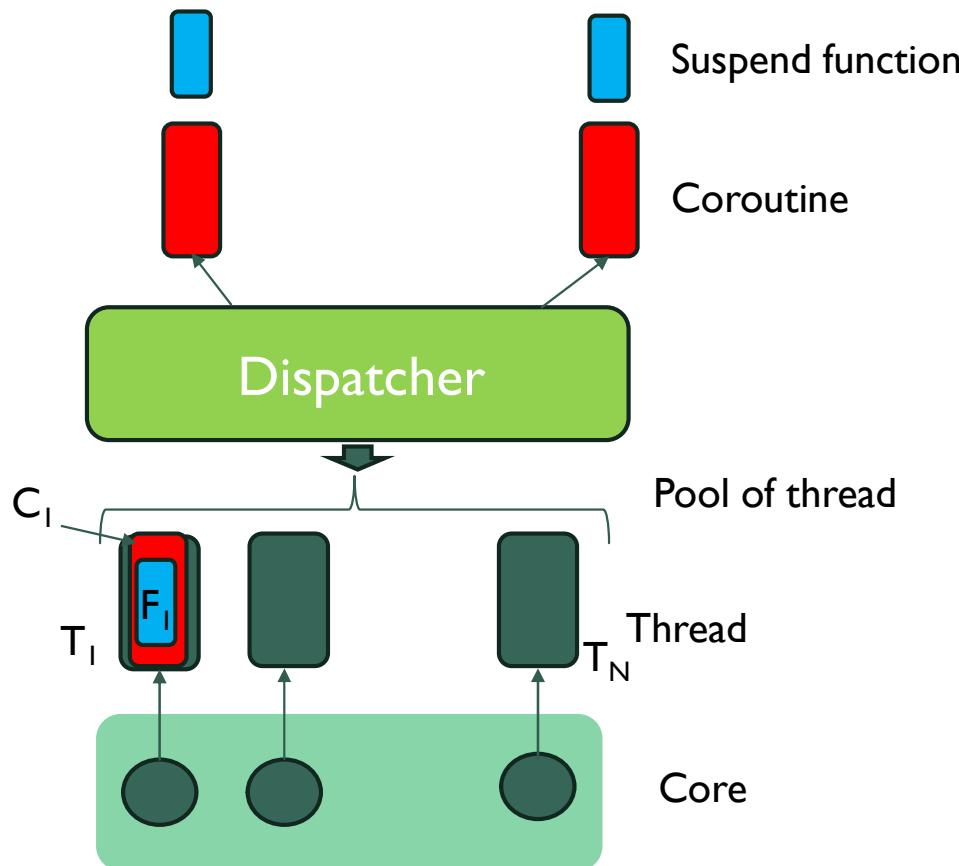


# COROUTINES



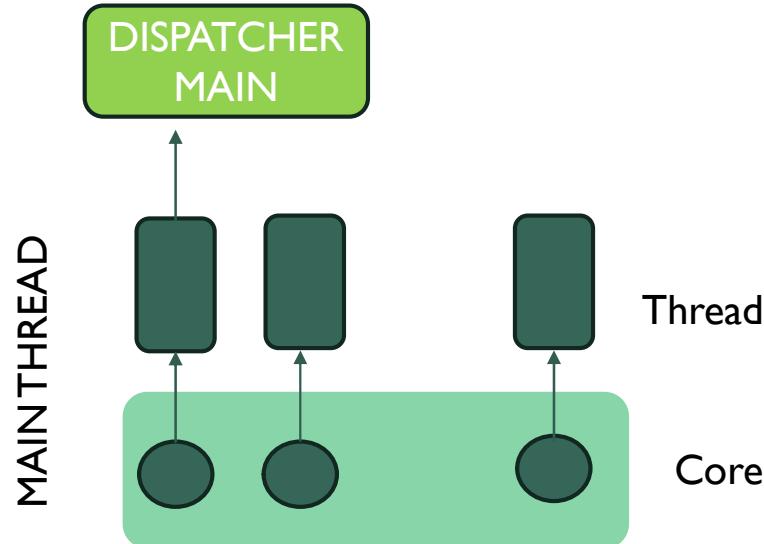
- A suspend function is a function that is executed inside a coroutine
- A coroutine is scheduled for running inside thread by a coroutine dispatcher
- A coroutine can be suspended and later moved from one thread to another to continue execution
- A coroutine can voluntarily suspend itself when it waits for some result
- The coroutine dispatcher exploits these suspension points to schedule another coroutines (cooperative scheduling)
- To simplify, the cores = hardware cores

# COROUTINES



- In this example, thread  $T_1$  runs the coroutine  $C_I$  which hosts a function  $F_1$ .
- The function  $F_1$  can call function  $F_2$  which runs inside  $C_I$
- If  $F_1$  suspends itself, the scheduler schedule another coroutine to run in  $T_1$
- The number of coroutines can be much higher than the number of threads in the pool

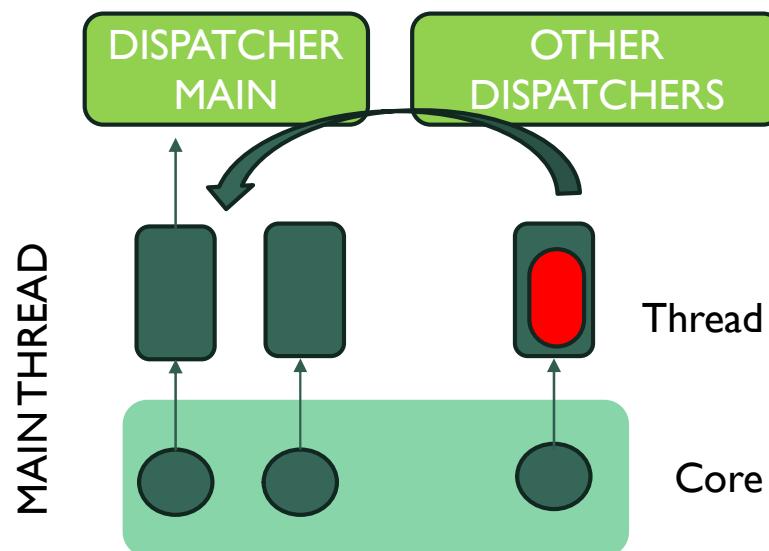
# THE MAIN THREAD



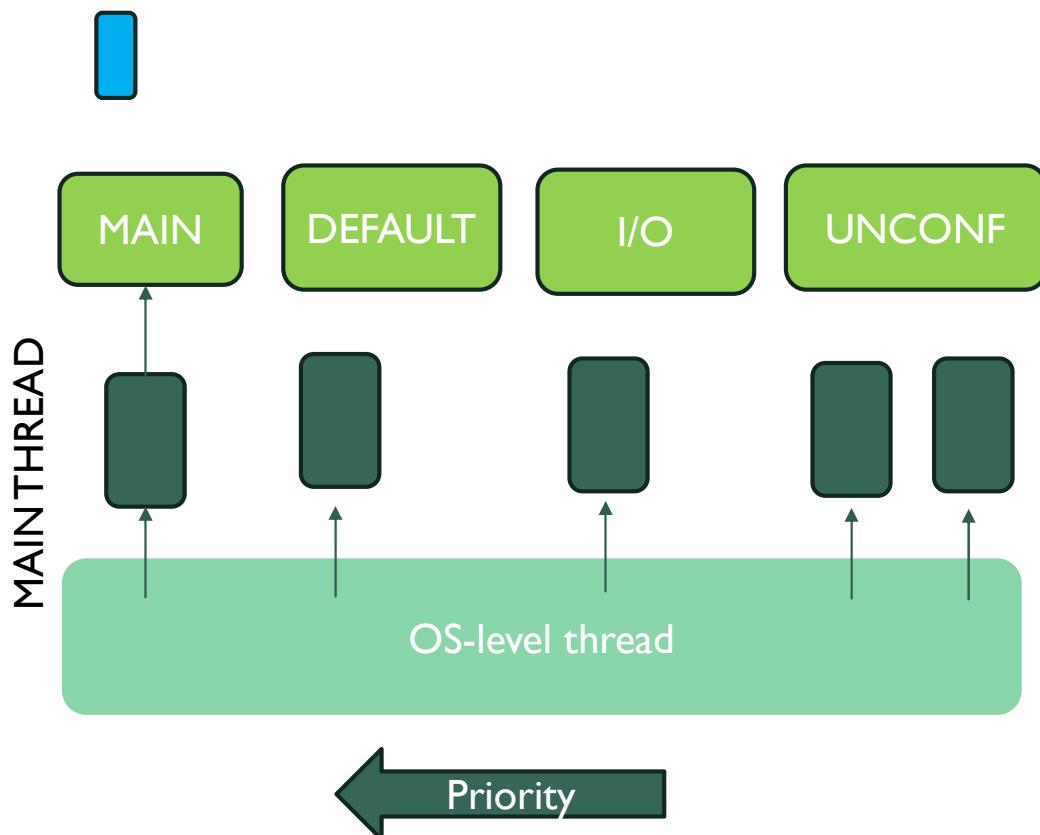
- Android uses a single thread, called the **MAIN** thread, to run the code of all the methods of an Activity
- The key goal of the **MAIN** thread is to run code that manages the UI,
- As the UI can change any 1/60 sec, the main thread must be 'light'
- Only the main thread can 'touch' the UI
- CPU intensive work will slow down the reactivity of the app, if severely the ANR message is displayed (Application Not Responding)

# THE MAIN THREAD

- A coroutine can **change the dispatcher**
- The MAIN dispatcher schedules a coroutine on the MAIN thread



# THREAD POOLS



- The thread did not correspond to HW core, rather they are managed by the OS in a preemptive way
- The main thread however gets the highest schedule priority
- The **default dispatcher** is used for CPU-intensive task and for this reason the pool of thread on which it can schedules is equal to the number of HW cores
- The **I/O dispatcher** is used for I/O operation. As they often blocks, the number of threads in this pool is higher
- Unconfined dispatcher (see documentation)

# COROUTINE BUILDER

- A builder oversees creating a coroutine
- The builder determines the scope (lifecycle) of the coroutine
- For example, all coroutines launched in the ViewModel scope are cancelled when the viewModel terminates
- The main builders are
  - `runBlocking{}` blocks the calling thread, che sospende il thread finché lo scope non termina.
  - `lunch{}`: do not return a result
  - `async{}`: the coroutine returns a result
- While the dispatcher determines where a coroutine runs, the scope limits the time (for how long) the coroutine can run

## SOME EXAMPLE

```
GlobalScope.launch { }
```

*GlobalScope* means that the coroutine leaves until the end of the app

```
GlobalScope.launch(Dispatchers.Main) {  
    Log.i(TAG, "GlobalScope: "+Thread.currentThread().name)  
}
```

*Dispatcher* allows to specify which thread pool will host the coroutine

I/info: GlobalScope: main

```
GlobalScope.launch(Dispatchers.IO) {  
    //Make a long network call  
    Log.i(TAG, "GlobalScopeIO: "+Thread.currentThread().name)  
    withContext(Dispatchers.Main){  
        Log.i(TAG, "GlobalScopeMAIN: "+Thread.currentThread().name)  
    }  
}
```

*WithContext* changes the dispatcher that will continue the work

I/info: GlobalScopeIO: DefaultDispatcher-worker-3  
I/info: GlobalScopeMAIN: main