ISIS-3510

Mobile App Development 3.0

(a.k.a., Software Engineering for Mobile Applications)

MULTI THREADING

Why should we use multi-threading in mobile apps?

Mobile apps are prone to performance bugs….

Mobile apps are prone to performance bugs….

because of hardware constraints and OS policies

PERFORMANCE BUGS IN MOBILE APPS

GUI lagging and Application Not Responding Errors (ANRs)

Memory bloats and Out of Memory Exceptions (OOMs)

Empirical Software Engineering
https://doi.org/10.1007/s10664-019-09795-6

Investigating types and survivability of performance bugs in mobile apps

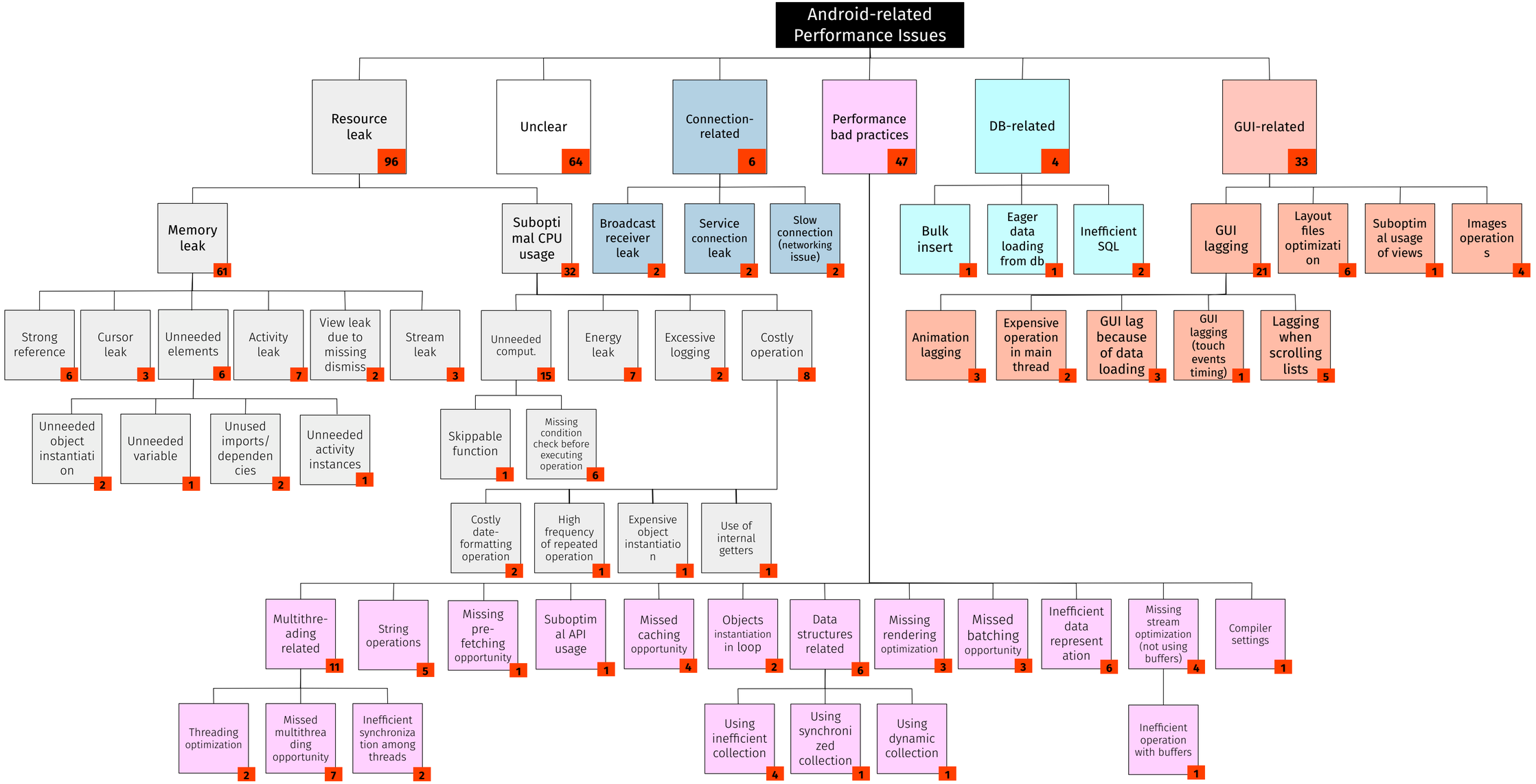
Alejandro Mazuera-Rozo[1,2] · Catia Trubiani[3] · Mario Linares-Vásquez[2] · Gabriele Bavota[1]

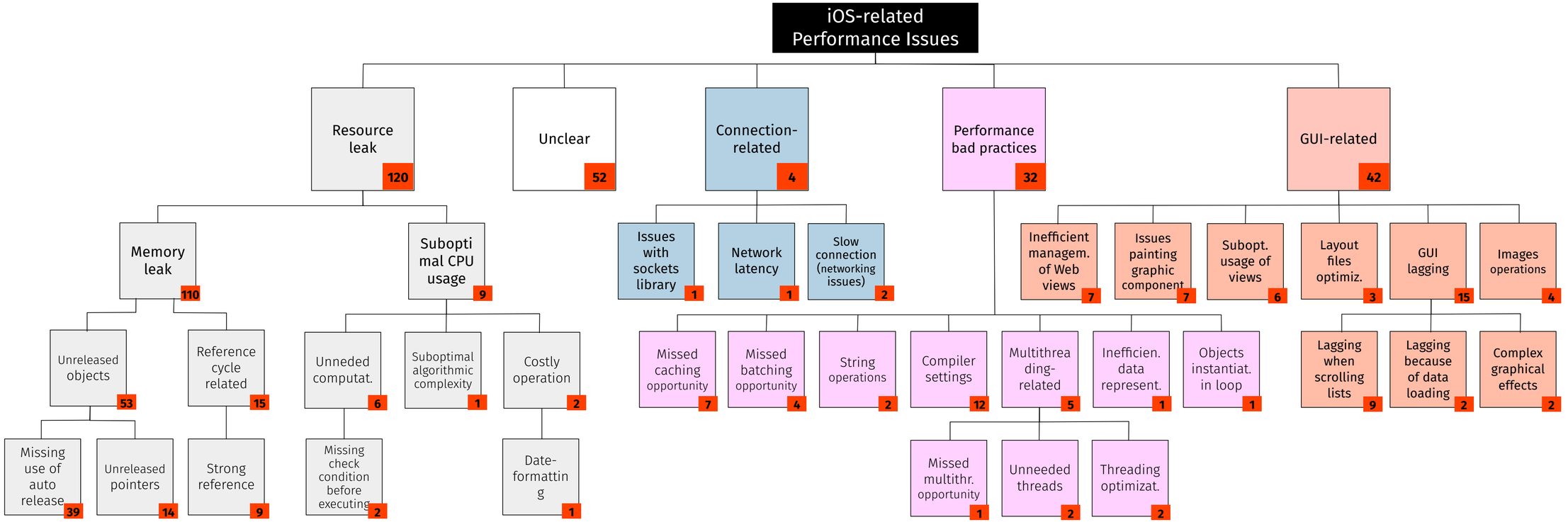
Published online: 05 March 2020

Abstract

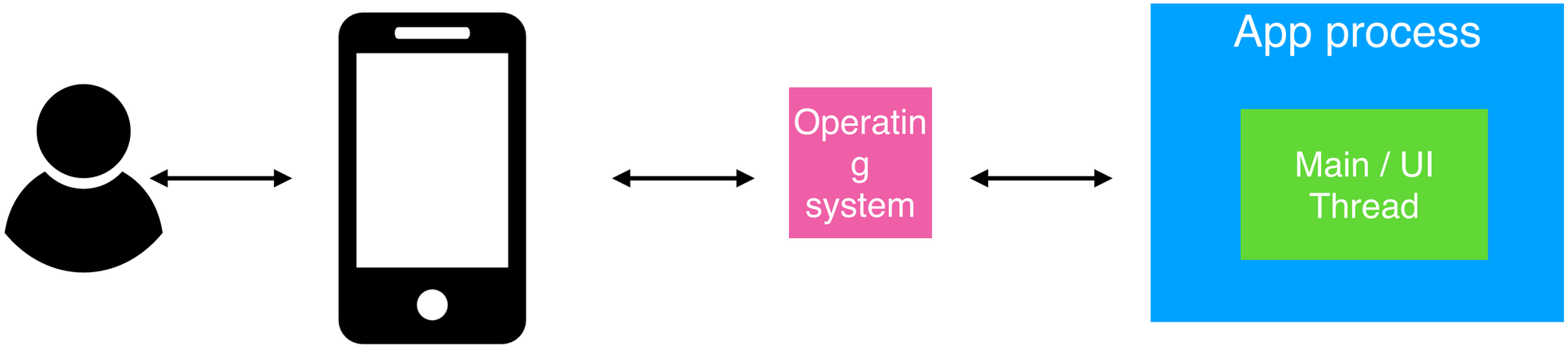
A recent research showed that mobile apps represent nowadays 75% of the whole usage of mobile devices. This means that the mobile user experience, while tied to many factors (*e.g.,* hardware device, connection speed, etc.), strongly depends on the quality of the apps being used. With "quality" here we do not simply refer to the features offered by the app, but also to its non-functional characteristics, such as security, reliability, and performance. This latter is particularly important considering the limited hardware resources (*e.g.,* memory) mobile apps can exploit. In this paper, we present the largest study at date investigating performance bugs in mobile apps. In particular, we (i) define a taxonomy of the types of performance bugs affecting Android and iOS apps; and (ii) study the survivability of performance bugs (*i.e.,* the number of days between the bug introduction and its fixing). Our findings aim to help researchers and apps developers in building performance-bugs detection tools and focusing their verification and validation activities on the most frequent types of performance bugs.

Keywords Performance bugs · Mobile apps · Empirical study

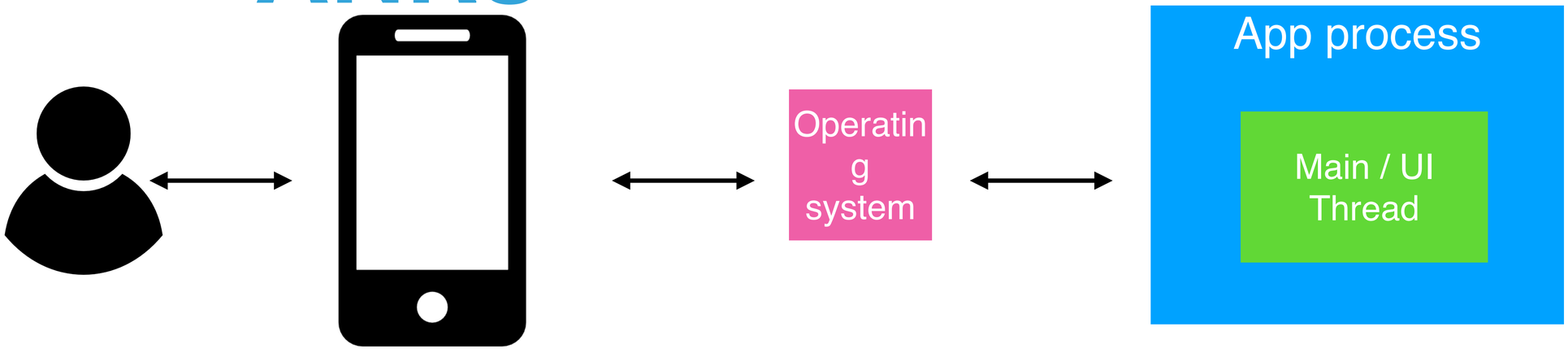

Android-related Performance Issues
Resource leak 96
Unclear 64
Connection-related 6
Performance bad practices 47
DB-related 4
GUI-related 33
Memory leak 61
Suboptimal CPU usage 32
Broadcast receiver leak 2
Service connection leak 2
Slow connection (networking issue) 2
Bulk insert 1
Eager data loading from db 1
Inefficient SQL 2
GUI lagging 21
Layout files optimization 6
Suboptimal usage of views 1
Images operations 4
Strong reference 6
Cursor leak 3
Unneeded elements 6
Activity leak 7
View leak due to missing dismiss 2
Stream leak 3
Unneeded comput. 15
Energy leak 7
Excessive logging 2
Costly operation 8
Animation lagging 3
Expensive operation in main thread 2
GUI lag because of data loading 3
GUI lagging (touch events timing) 1
Lagging when scrolling lists 5
Unneeded object instantiation 2
Unneeded variable 1
Unused imports/ dependencies 2
Unneeded activity instances 1
Skippable function 1
Missing condition check before executing operation 6
Costly date-formatting operation 2
High frequency of repeated operation 1
Expensive object instantiation 1
Use of internal getters 1
Multithre-ading related 11
String operations 5
Missing pre-fetching opportunity 1
Suboptimal API usage 1
Missed caching opportunity 4
Objects instantiation in loop 2
Data structures related 6
Missing rendering optimization 3
Missed batching opportunity 3
Inefficient data representation 6
Missing stream optimization (not using buffers) 4
Compiler settings 1
Threading optimization 2
Missed multithreading opportunity 7
Inefficient synchronization among threads 2
Using inefficient collection 4
Using synchronized collection 1
Using dynamic collection 1
Inefficient operation with buffers 1


iOS-related Performance Issues
Resource leak 120
Unclear 52
Connection-related 4
Performance bad practices 32
GUI-related 42
Memory leak 110
Suboptimal CPU usage 9
Issues with sockets library 1
Network latency 1
Slow connection (networking issues) 2
Inefficient managem. of Web views 7
Issues painting graphic component 7
Subopt. usage of views 6
Layout files optimiz. 3
GUI lagging 15
Images operations 4
Unreleased objects 53
Reference cycle related 15
Unneded computat. 6
Suboptimal algorithmic complexity 1
Costly operation 2
Missed caching opportunity 7
Missed batching opportunity 4
String operations 2
Compiler settings 12
Multithreading-related 5
Inefficien. data represent. 1
Objects instantiat. in loop 1
Lagging when scrolling lists 9
Lagging because of data loading 2
Complex graphical effects 2
Missing use of auto release 39
Unreleased pointers 14
Strong reference 9
Missing check condition before executing 2
Date-formatting 1
Missed multithr. opportunity 1
Unneeded threads 2
Threading optimizat. 2

GUI lagging and ANRs



GUI lagging and ANRs



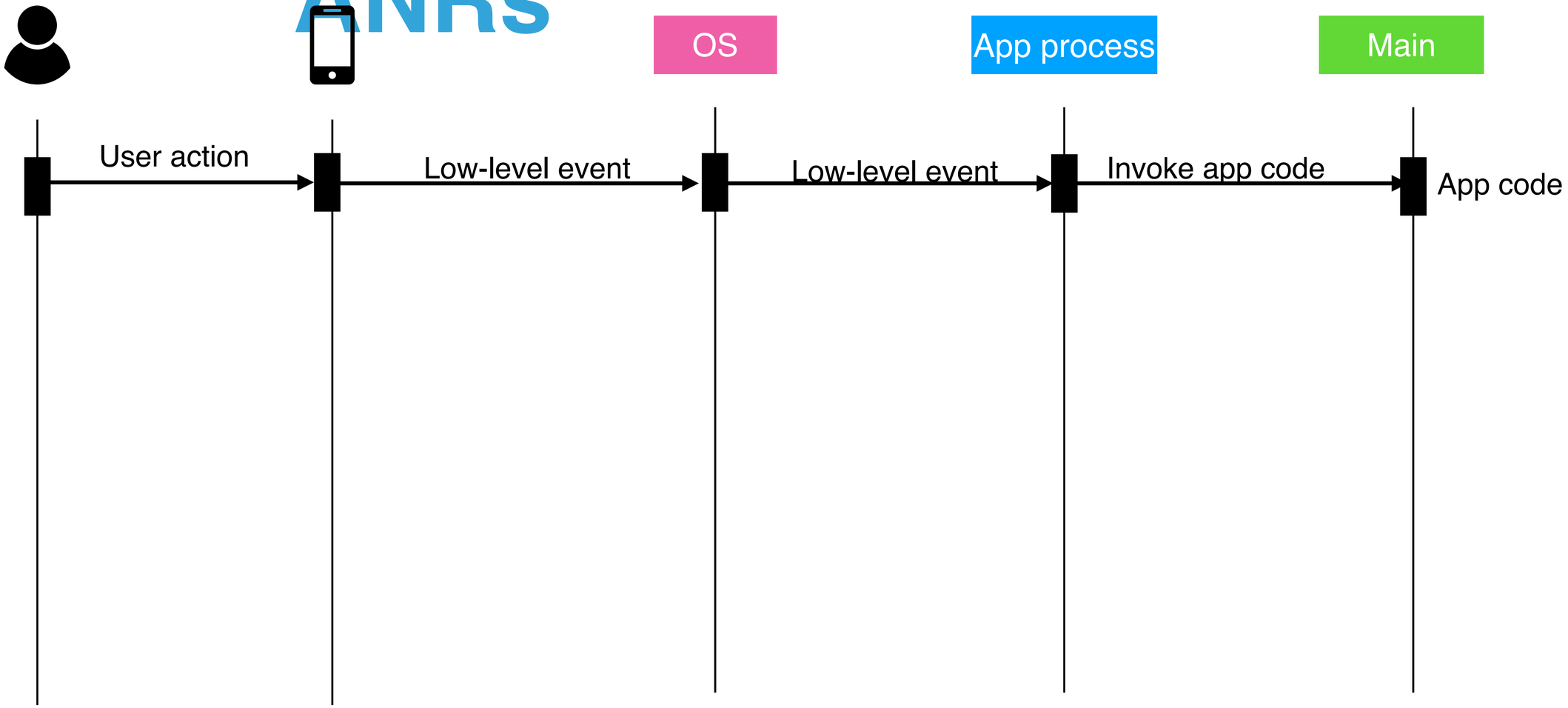
Mobile apps are single threaded by default, and in the case of Flutter you can not create more threads for being executed within the same process

What is the difference between a thread and a process?

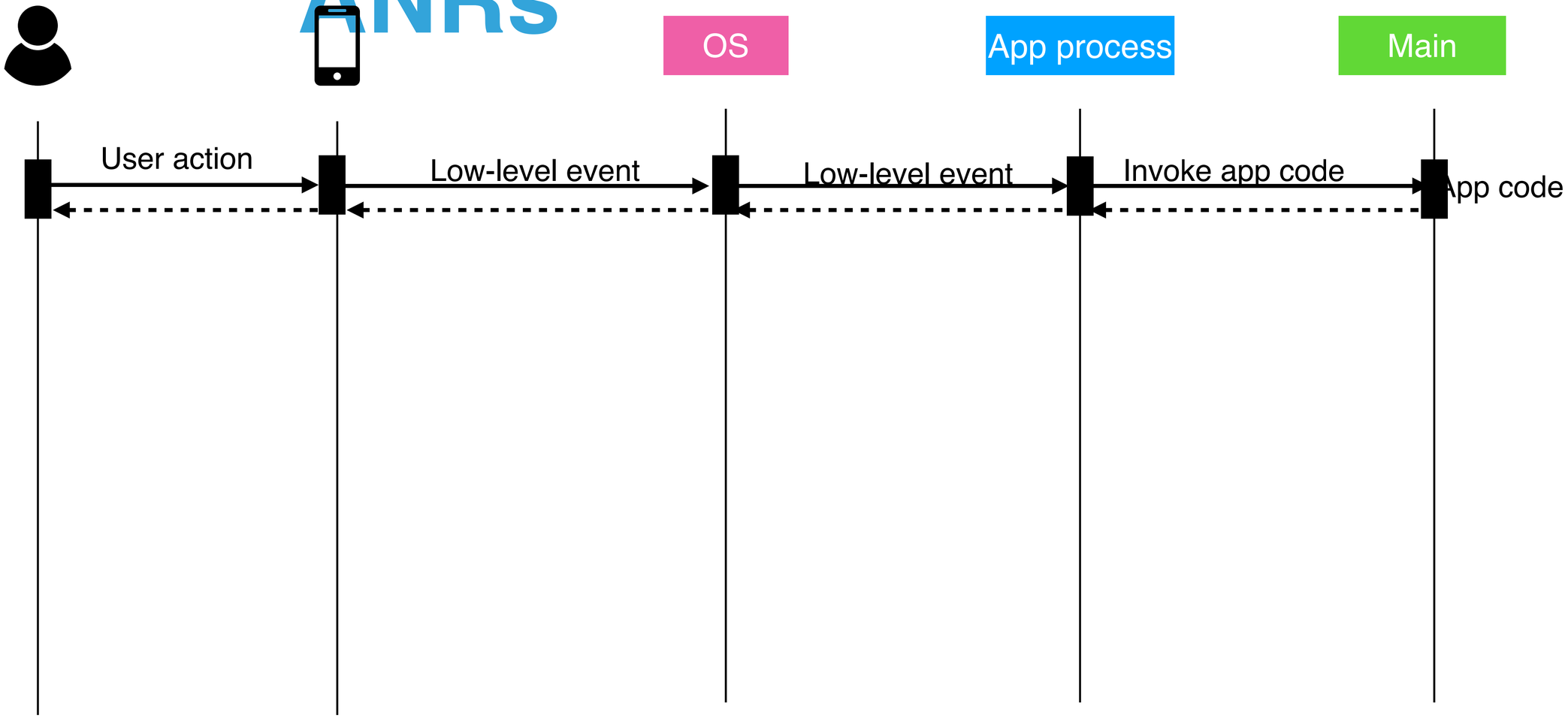
Process: it is an execution environment, that has its own memory space (like a sandbox). A process can not access to the memory space of other process. Communication is then done via IPC mechanism

Thread: a process has a main thread (like an execution line), but more threads can be started in the same process (except in Flutter). Because the threads belong to a process, then, those share the same memory space

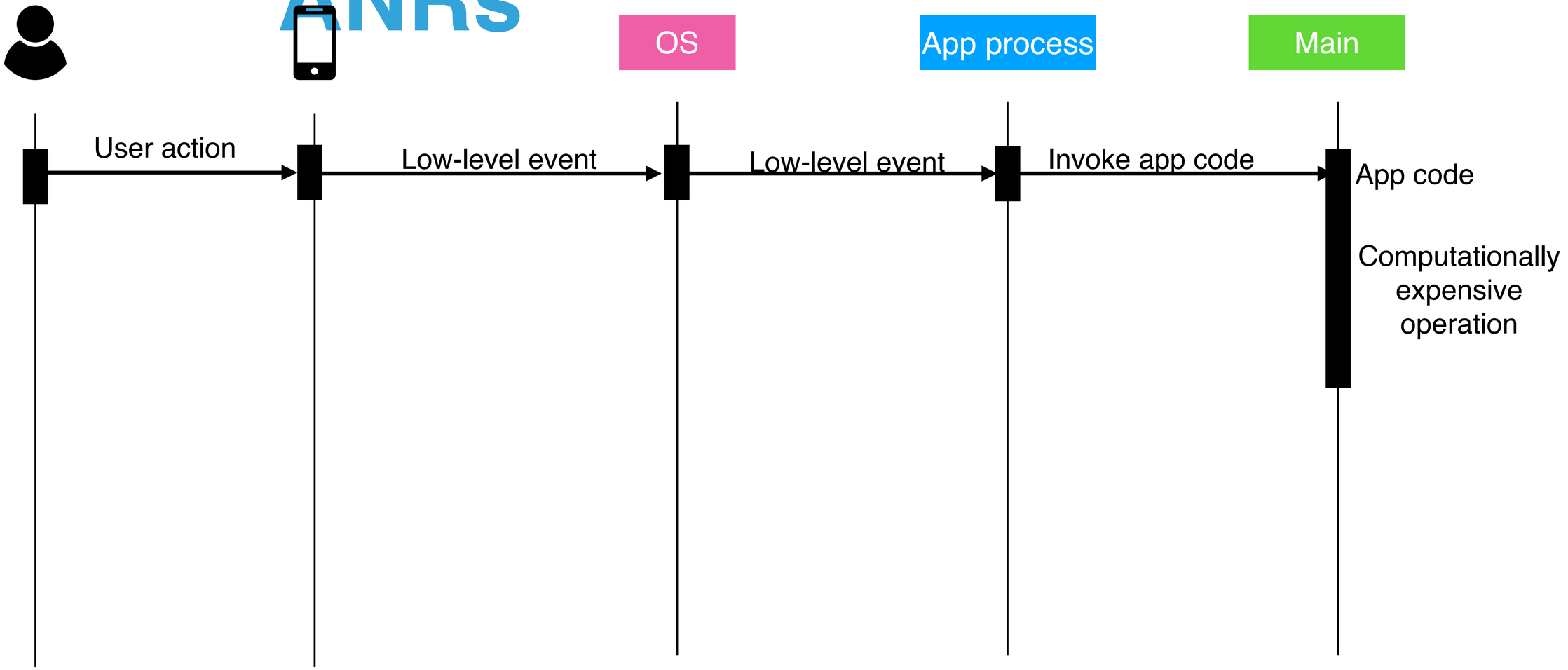
GUI lagging and ANRs



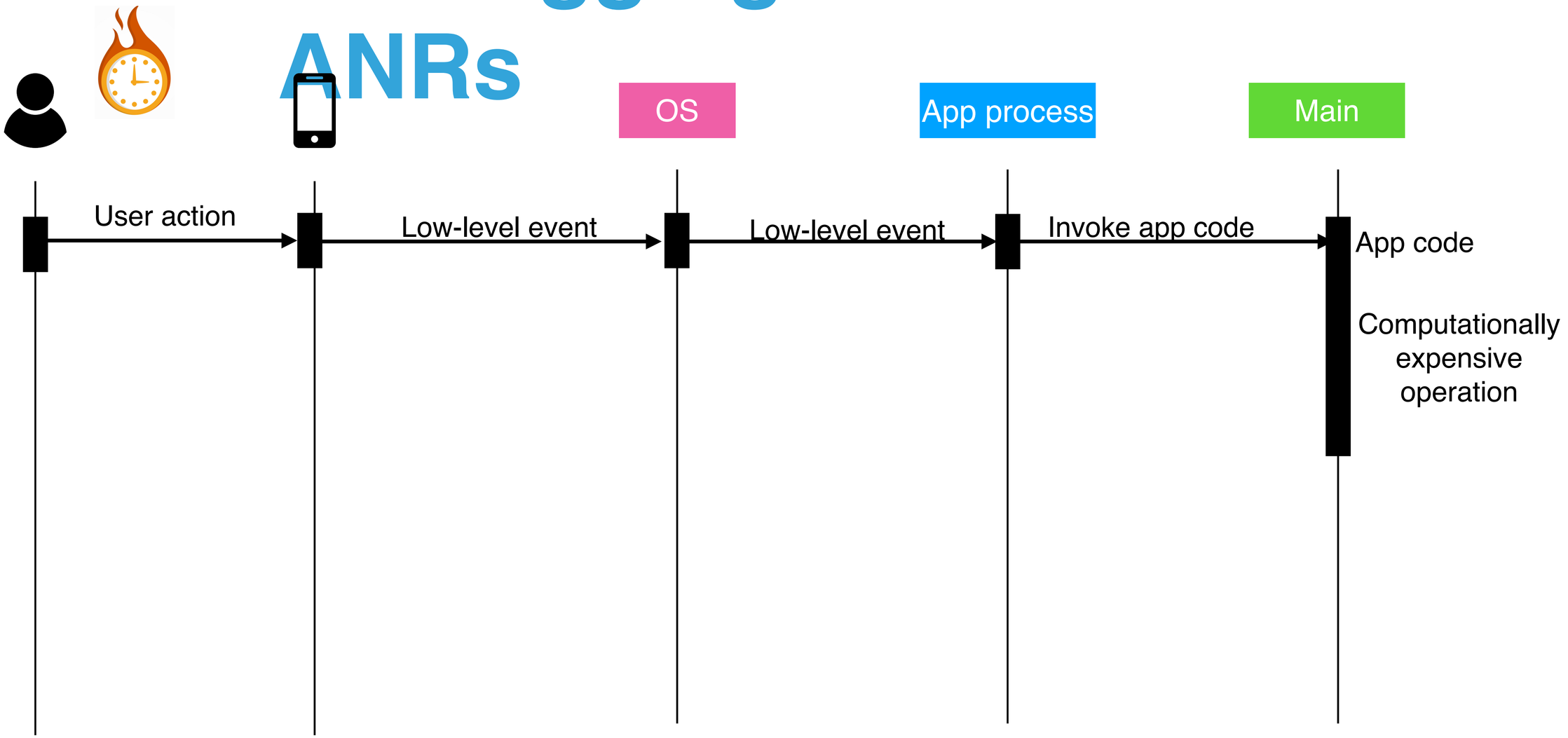
GUI lagging and ANRs



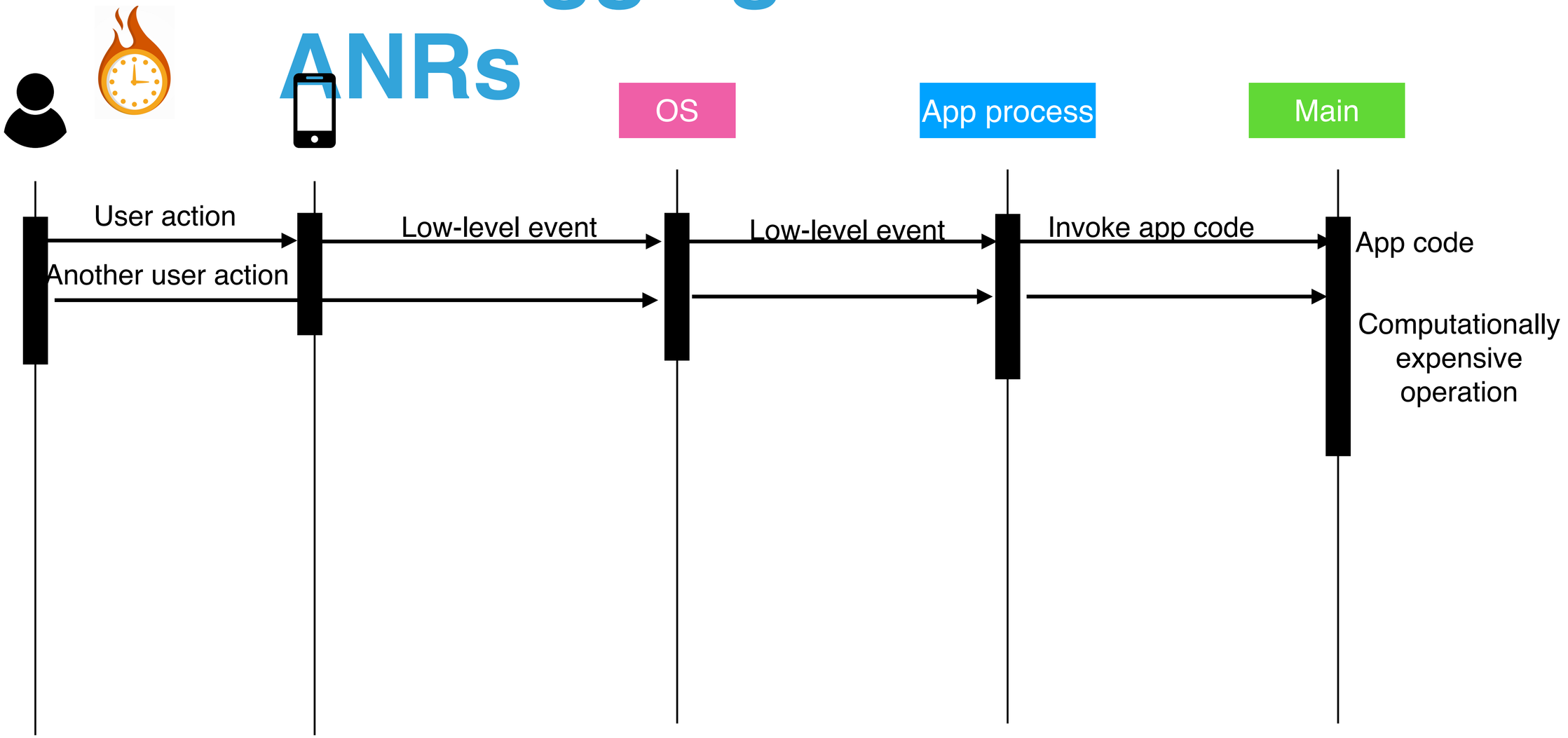
GUI lagging and ANRs



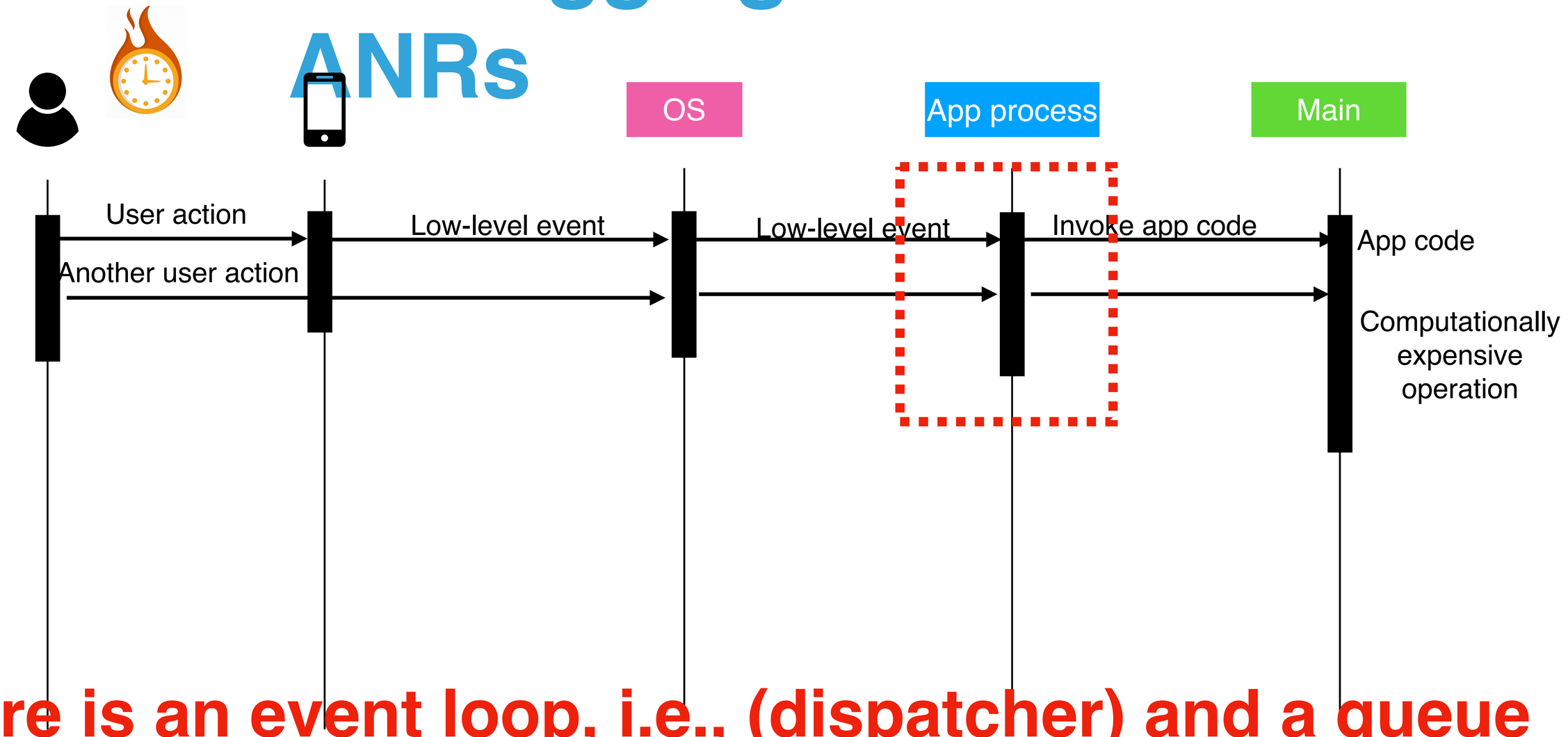
GUI lagging and ANRs



GUI lagging and ANRs

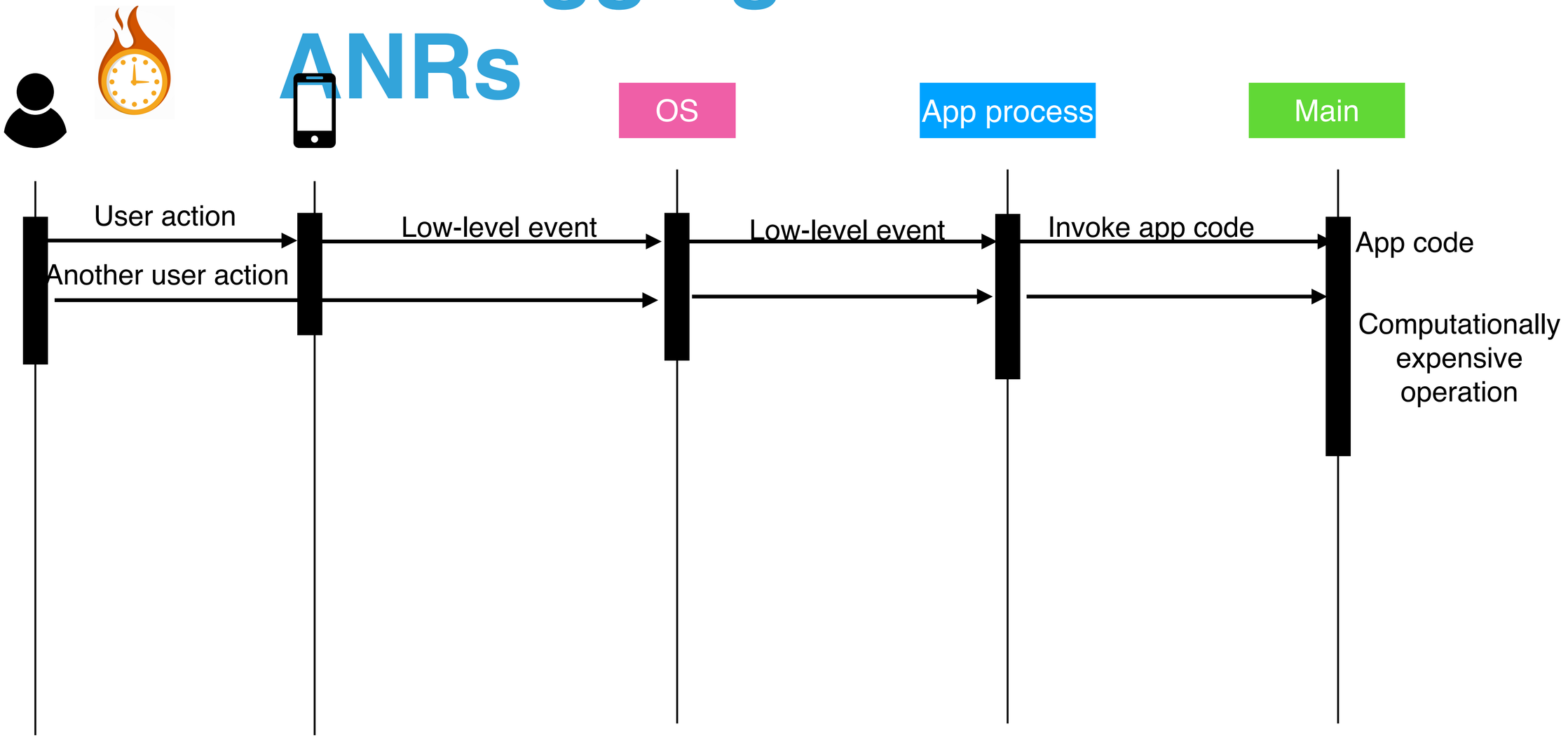


GUI lagging and ANRs

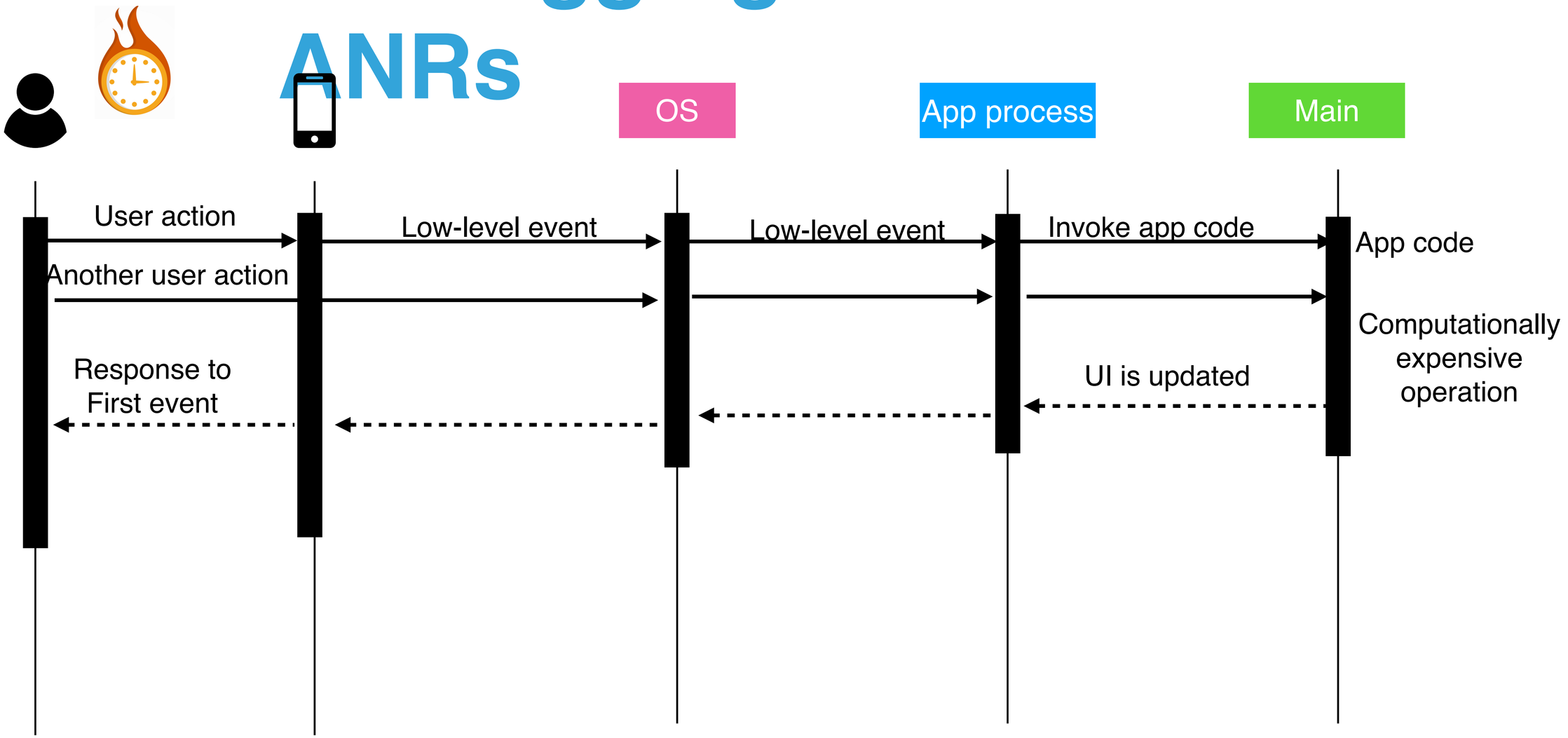


There is an event loop, i.e., (dispatcher) and a queue for events

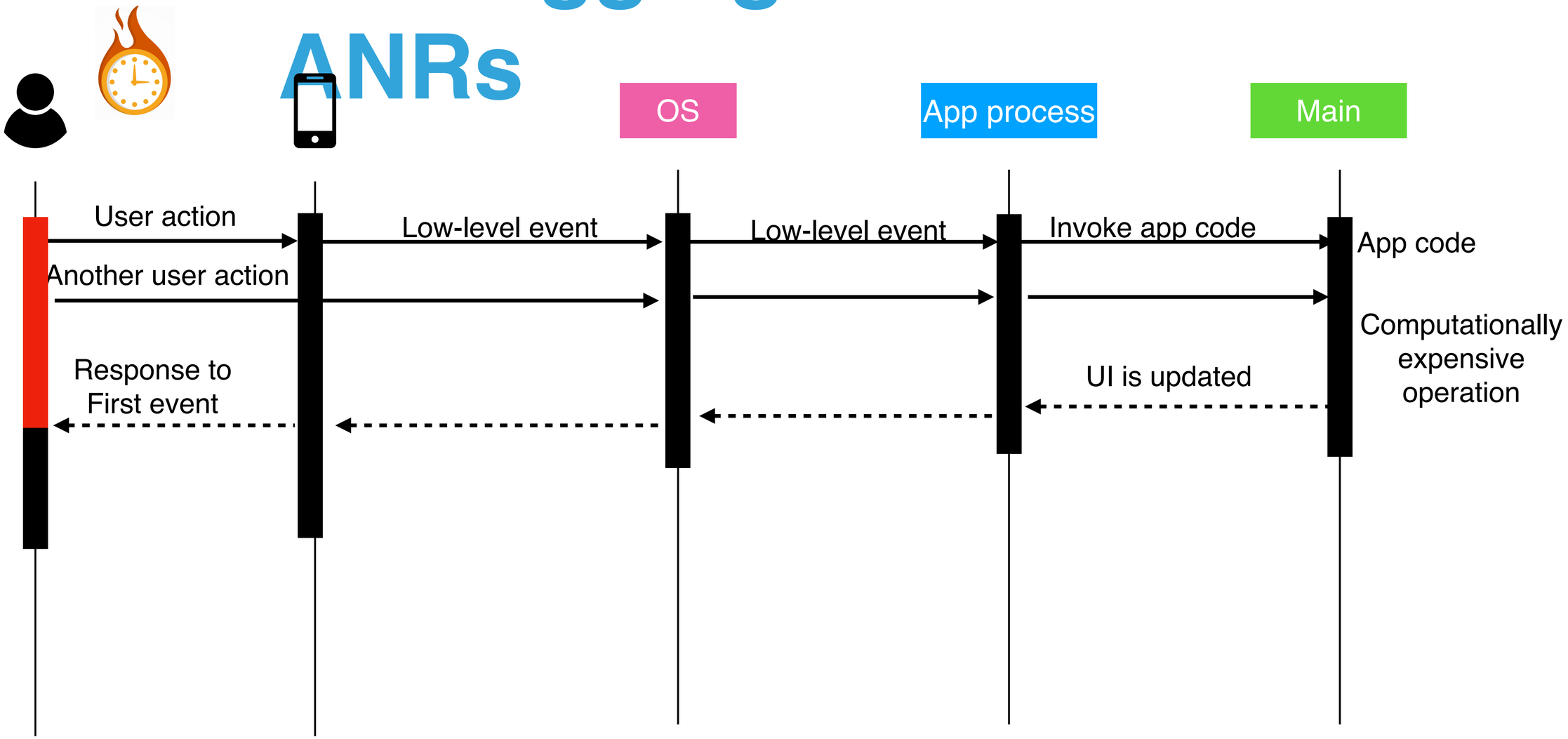
GUI lagging and ANRs



GUI lagging and ANRs

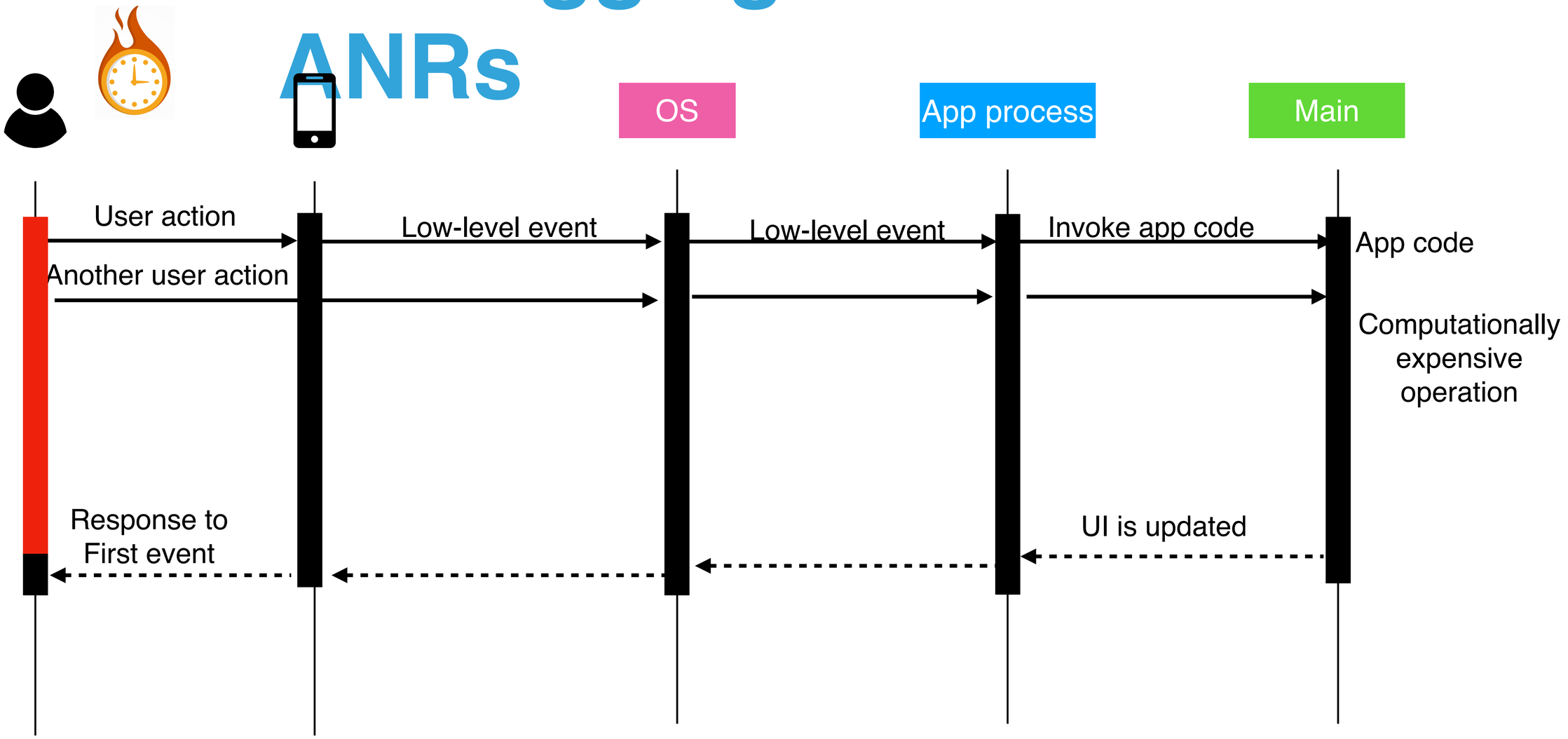


GUI lagging and ANRs



During this time the app is not responsive or there is GUI lagging

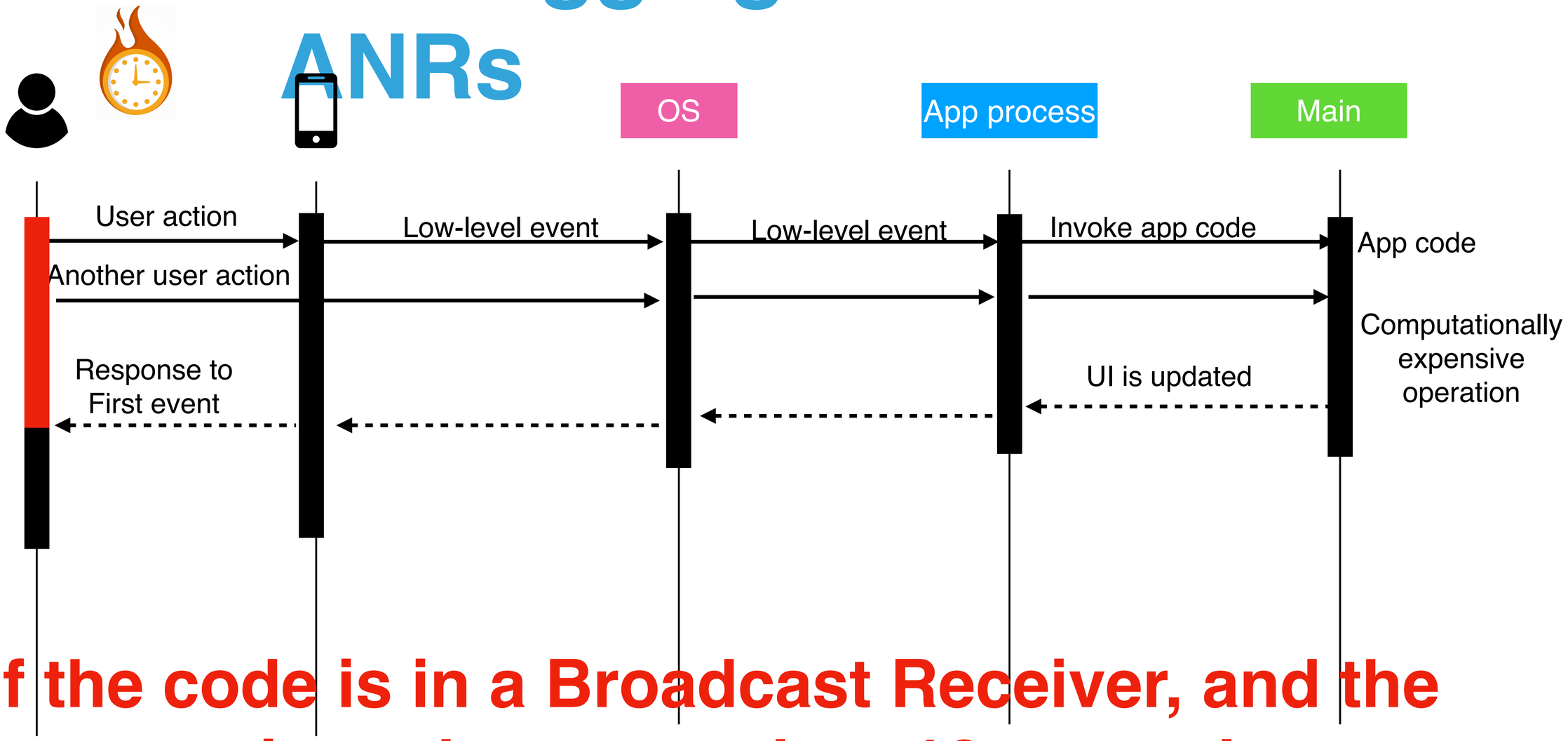
GUI lagging and ANRs



If the response time is larger than 5 seconds ANR


Hello word is not responding.
Do you want to close it?

GUI lagging and ANRs



If the code is in a Broadcast Receiver, and the execution takes more than 10 seconds….. then


Hello word is not responding.
Do you want to close it?

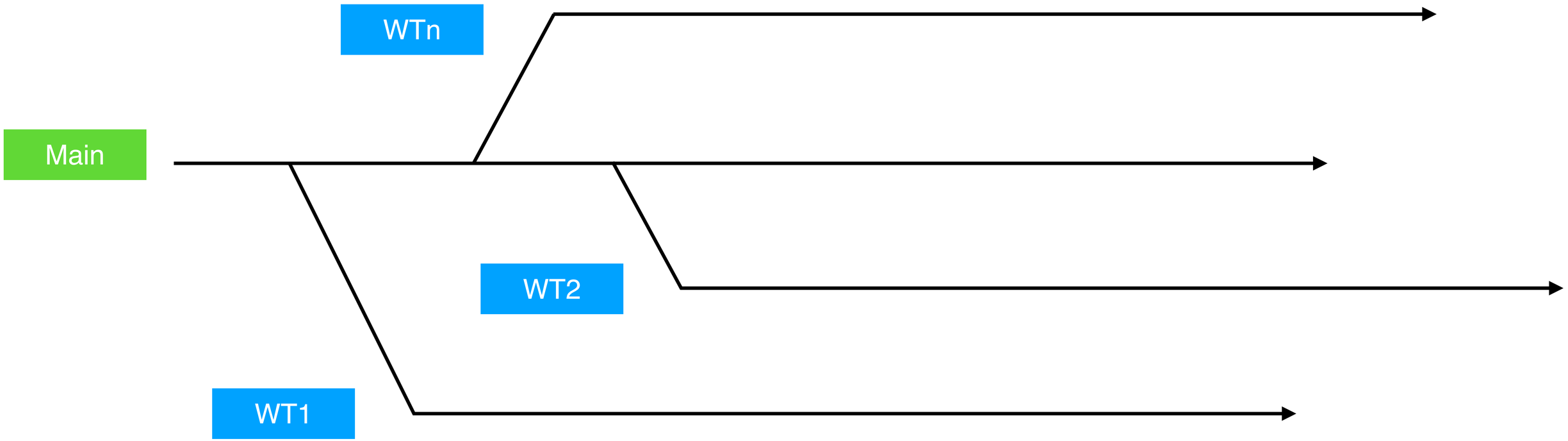
How to avoid GUI lags and ANRs?

Causes: I/O, access to remote resource, complex computation, overdrawing, excessive GC

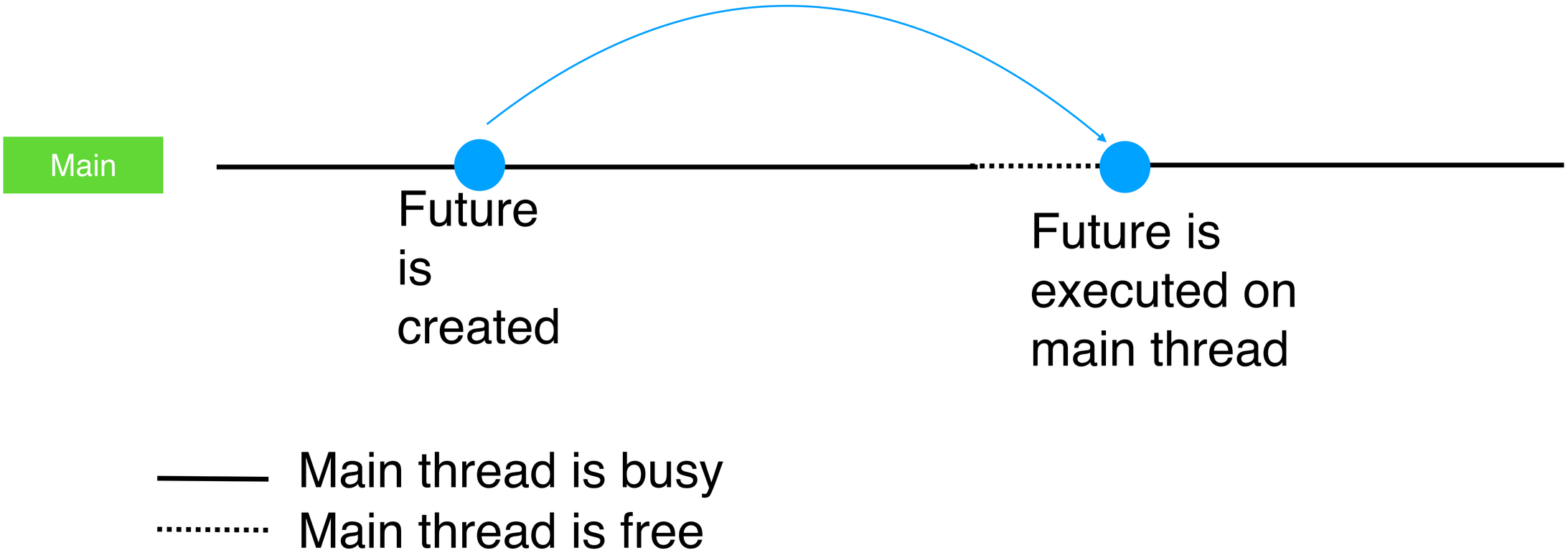
General strategy: avoid blocking the main thread with lengthy operations

Solutions: caching, remote delegation, GUI optimization, memory management, **multi-threading, suspend functions, futures, isolates, streams**

Multi-threading: use worker (background) threads for lengthy tasks



Async execution with Futures: futures are executed when the main thread is free



Async execution with Futures: futures are executed when the main thread is free

A future has three states:

- Uncompleted: the future has not started I finished
- Completed with a value
- Completed with an error

Isolates (Flutter): an isolate is a process. In Dart, your app can create different isolates, and communication between process (i.e., IPC) is done via messages

An isolated has its own memory space, event loop, and main thread

How does multi-threading work in iOS?

GRAND CENTRAL DISPATCH (GCD)

Unique API for handling multi-threading in iOS apps

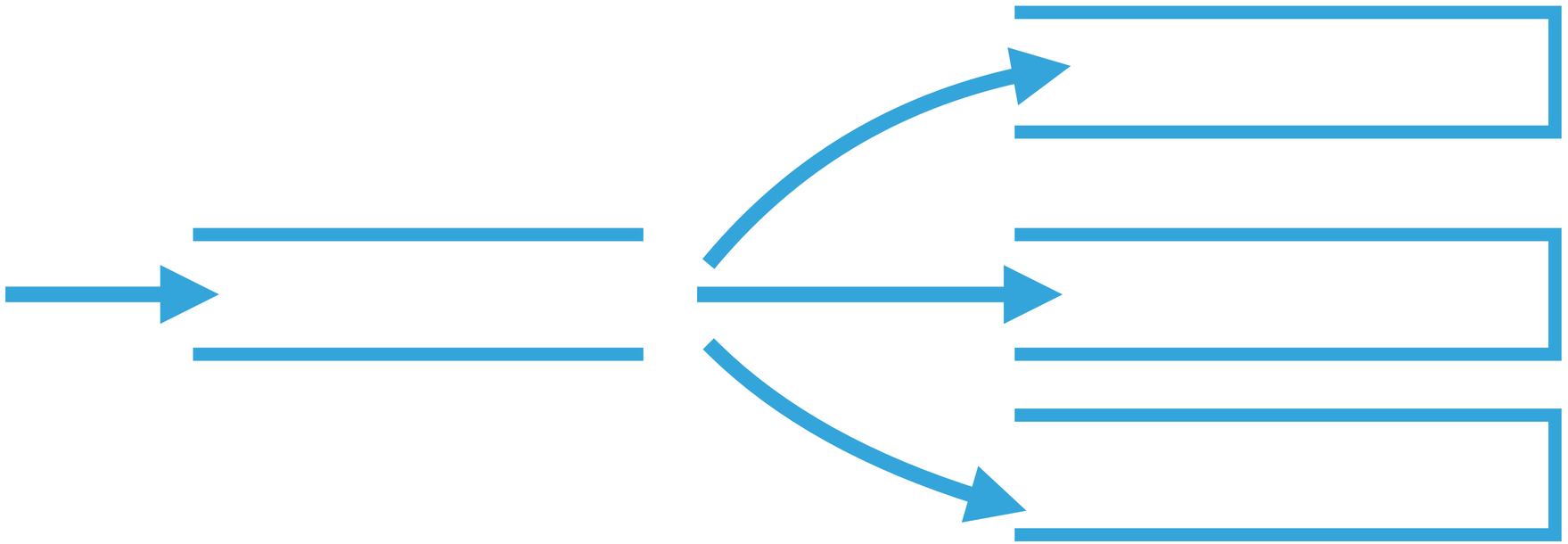
Sync and Async execution (threads)

Concurrent and Serial execution (threads)

Predefined dispatch queues for execution of tasks based on QoS attributes

SERIAL

CONCURRENT



Work submitted to dispatch queues executes on a pool of threads managed by the system

SYNCHRONOUS EXECUTION (SYNC)

The caller waits until the task finishes

ASYNCHRONOUS EXECUTION (ASYNC)

The caller **DOES NOT** wait until the taks finishes

DISPATCH QUEUES

```
DispatchQueue.main


DispatchQueue.global(qos: .userInteractive)
DispatchQueue.global(qos: .userInitiated)
DispatchQueue.global(qos: .utility)
DispatchQueue.global(qos: .background)
DispatchQueue.global(qos: .unspecified)
```

DISPATCH QUEUES

```
DispatchQueue.main
```

Main thread, serial

```
DispatchQueue.global(qos: .userInteractive)
DispatchQueue.global(qos: .userInitiated)
DispatchQueue.global(qos: .utility)
DispatchQueue.global(qos: .background)
DispatchQueue.global(qos: .unspecified)
```

Background, concurrent, shared across all apps

IOS

```
        DispatchQueue.main.async {
                // update ui here
        }



DispatchQueue.global(qos: .background).async {
    // do your job here

}
```

IOS

```
let main = DispatchQueue.main

let background = DispatchQueue.global()

let queue = DispatchQueue(label: “serial.example”)
```

IOS

```
DispatchQueue.global(qos: .background).async {

    // do your job here in background


    DispatchQueue.main.async {
        // update ui here
    }

    // more work here off the main thread

}
```

iOS

```
let delay = DispatchTime.now() + .seconds(120)


DispatchQueue.main.asyncAfter(deadline: delay) {
    // Do your stuff
}
```

IOS

```
let queue = DispatchQueue(label: “serial.example”)

queue.sync {

    // ...

}


DispatchQueue(label: “concurrent.example",
attributes: .concurrent).sync {

    // . . .

}
```

How does multi-threading work in Android (Java)?

Several options are available in Java for multi-threading, which is both pro and cons at the same time

Each of the available options need to be carefully implemented depending on the use case.

The easiest/safest way to use threads for updating the ui is with **AsyncTasks**

ANDROID

android.os.AsyncTask <Param, Progress, Resu

This is intended for background workers that update the UI, and run for short periods

It was the defacto option for multi-threading, however, it is deprecated starting on Android R

The UI can be updated in 4 methods:
onPreExecute, doInbackGround, onProgressUpdate, onPostExecute

https://developer.android.com/reference/android/os/AsyncTask.html

ANDROID

```
new DownloadFilesTask().execute(url1, url2, url3);
```

```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
    protected Long doInBackground(URL... urls) {
        int count = urls.length;
        long totalSize = 0;
        for (int i = 0; i < count; i++) {
            totalSize += Downloader.downloadFile(urls[i]);
            publishProgress((int) ((i / (float) count) * 100));
            // Escape early if cancel() is called
            if (isCancelled()) break;
        }
        return totalSize;
    }

    protected void onProgressUpdate(Integer... progress) {
        setProgressPercent(progress[0]);
    }

    protected void onPostExecute(Long result) {
        showDialog("Downloaded " + result + " bytes");
    }
}
```

https://developer.android.com/reference/android/os/AsyncTask.html

ANDROID

```
new DownloadFilesTask().execute(url1, url2, url3);
```

```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
    protected Long doInBackground(URL... urls) {
        int count = urls.length;
        long totalSize = 0;
        for (int i = 0; i < count; i++) {
            totalSize += Downloader.downloadFile(urls[i]);
            publishProgress((int) ((i / (float) count) * 100));
            // Escape early if cancel() is called
            if (isCancelled()) break;
        }
        return totalSize;
    }

    protected void onProgressUpdate(Integer... progress) {
        setProgressPercent(progress[0]);
    }

    protected void onPostExecute(Long result) {
        showDialog("Downloaded " + result + " bytes");
    }
}
```

https://developer.android.com/reference/android/os/AsyncTask.html

ANDROID

```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
    protected Long doInBackground(URL... urls) {
        int count = urls.length;
        long totalSize = 0;
        for (int i = 0; i < count; i++) {
            totalSize += Downloader.downloadFile(urls[i]);
            publishProgress((int) ((i / (float) count) * 100));
            // Escape early if cancel() is called
            if (isCancelled()) break;
        }
        return totalSize;
    }

    protected void onProgressUpdate(Integer... progress) {
        setProgressPercent(progress[0]);
    }

    protected void onPostExecute(Long result) {
        showDialog("Downloaded " + result + " bytes");
    }
}
```

AsyncTask
argument

https://developer.android.com/reference/android/os/AsyncTask.html

ANDROID

```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
    protected Long doInBackground(URL... urls) {
        int count = urls.length;
        long totalSize = 0;
        for (int i = 0; i < count; i++) {
            totalSize += Downloader.downloadFile(urls[i]);
            publishProgress((int) ((i / (float) count) * 100));
            // Escape early if cancel() is called
            if (isCancelled()) break;
        }
        return totalSize;
    }

    protected void onProgressUpdate(Integer... progress) {
        setProgressPercent(progress[0]);
    }

    protected void onPostExecute(Long result) {
        showDialog("Downloaded " + result + " bytes");
    }
}
```

https://developer.android.com/reference/android/os/AsyncTask.html

ANDROID

```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
    protected Long doInBackground(URL... urls) {
        int count = urls.length;
        long totalSize = 0;
        for (int i = 0; i < count; i++) {
            totalSize += Downloader.downloadFile(urls[i]);
            publishProgress((int) ((i / (float) count) * 100));
            // Escape early if cancel() is called
            if (isCancelled()) break;
        }
        return totalSize;
    }

    protected void onProgressUpdate(Integer... progress) {
        setProgressPercent(progress[0]);
    }

    protected void onPostExecute(Long result) {
        showDialog("Downloaded " + result + " bytes");
    }
}
```

https://developer.android.com/reference/android/os/AsyncTask.html

ANDROID

```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
    protected Long doInBackground(URL... urls) {
        int count = urls.length;
        long totalSize = 0;
        for (int i = 0; i < count; i++) {
            totalSize += Downloader.downloadFile(urls[i]);
            publishProgress((int) ((i / (float) count) * 100));
            // Escape early if cancel() is called
            if (isCancelled()) break;
        }
        return totalSize;
    }

    protected void onProgressUpdate(Integer... progress) {
        setProgressPercent(progress[0]);
    }

    protected void onPostExecute(Long result) {
        showDialog("Downloaded " + result + " bytes");
    }
}
```

Progress value

https://developer.android.com/reference/android/os/AsyncTask.html

ANDROID

```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
    protected Long doInBackground(URL... urls) {
        int count = urls.length;
        long totalSize = 0;
        for (int i = 0; i < count; i++) {
            totalSize += Downloader.downloadFile(urls[i]);
            publishProgress((int) ((i / (float) count) * 100));
            // Escape early if cancel() is called
            if (isCancelled()) break;
        }
        return totalSize;
    }

    protected void onProgressUpdate(Integer... progress) {
        setProgressPercent(progress[0]);
    }

    protected void onPostExecute(Long result) {
        showDialog("Downloaded " + result + " bytes");
    }
}
```

Result value

https://developer.android.com/reference/android/os/AsyncTask.html

How does multi-threading work in Kotlin?

Kotlin combines the multi-threading options available in Android, with **coroutines** that allows for suspending functions

The main idea with suspend functions is to post-pone the execution, but, on the main thread when it is free

KOTLIN

Coroutines

Design Pattern -> Concurrency

You use them when you want to simplify code that executes asynchronously:

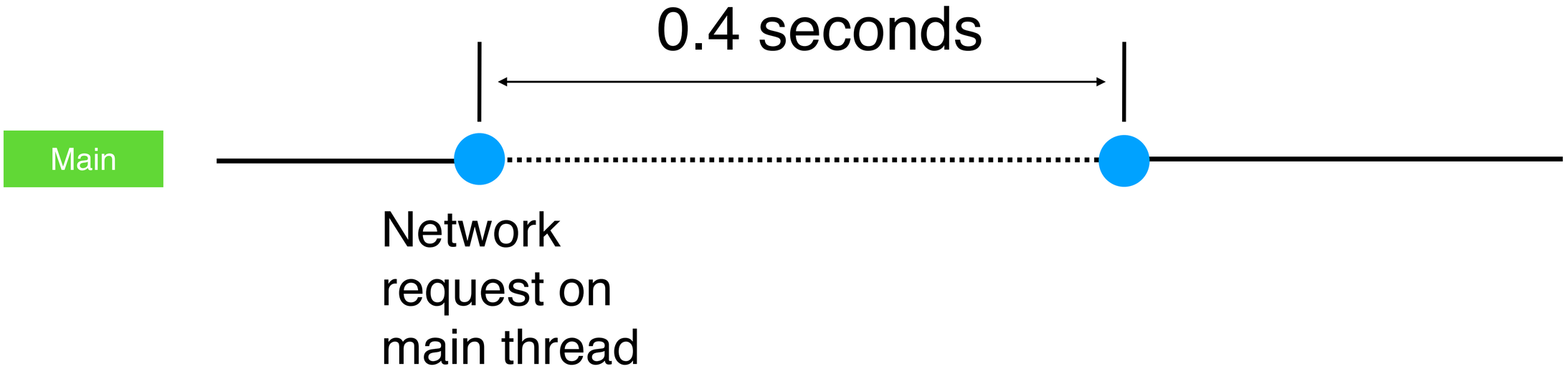
-Manage Long-running tasks, so they don’t block the main thread

-Provide Main-safety. That allows to call any suspend function from the main thread

https://developer.android.com/kotlin/coroutines

KOTLIN

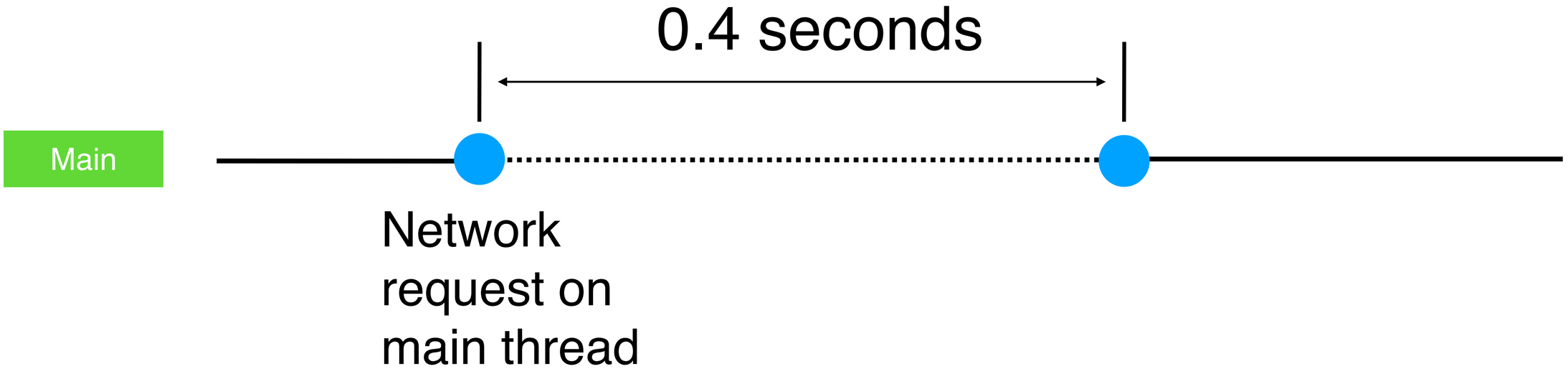
Coroutines - Long Running task



https://medium.com/androiddevelopers/coroutines-on-android-part-i-getting-the-background-3e0e54d20bb

KOTLIN

Coroutines - Long Running task



The main thread is blocked while the request is executed

https://medium.com/androiddevelopers/coroutines-on-android-part-i-getting-the-background-3e0e54d20bb

KOTLIN

Coroutines - Long Running task

Network request on main thread

How many CPU cycles were wasted? 1 billion cycles

(Pixel 2 CPU cycle -> 0.0000000004 seconds)

https://medium.com/androiddevelopers/coroutines-on-android-part-i-getting-the-background-3e0e54d20bb

If multi-threading is not an option, then it is better to avoid wasting CPU cycles on the main thread

KOTLIN

Coroutines - Long Running task

HTTP Request (off the main thread)

```
class ViewModel: ViewModel() {
   fun fetchDocs() {
       get("developer.android.com") { result ->
           show(result)
       }
    }
}
```

Retrofit

Volley

https://medium.com/androiddevelopers/coroutines-on-android-part-i-getting-the-background-3e0e54d20bb

KOTLIN

Coroutines - Long Running task

HTTP Request (off the main thread)

```
class ViewModel: ViewModel() {
   fun fetchDocs() {
       get("developer.android.com") { result ->
           show(result)
       }
    }
}
```

Retrofit

Volley

HTTP Request using coroutines

```
suspend fun fetchDocs() {
   val result = get("developer.android.com")
   show(result)
}

suspend fun get(url: String) = withContext(Dispatchers.IO){/*...*/}
```

https://medium.com/androiddevelopers/coroutines-on-android-part-i-getting-the-background-3e0e54d20bb

KOTLIN

Coroutines - Long Running task

HTTP Request using coroutines

```
suspend fun fetchDocs() {
   val result = get("developer.android.com")
   show(result)
}

suspend fun get(url: String) = withContext(Dispatchers.IO){/*...*/}
```

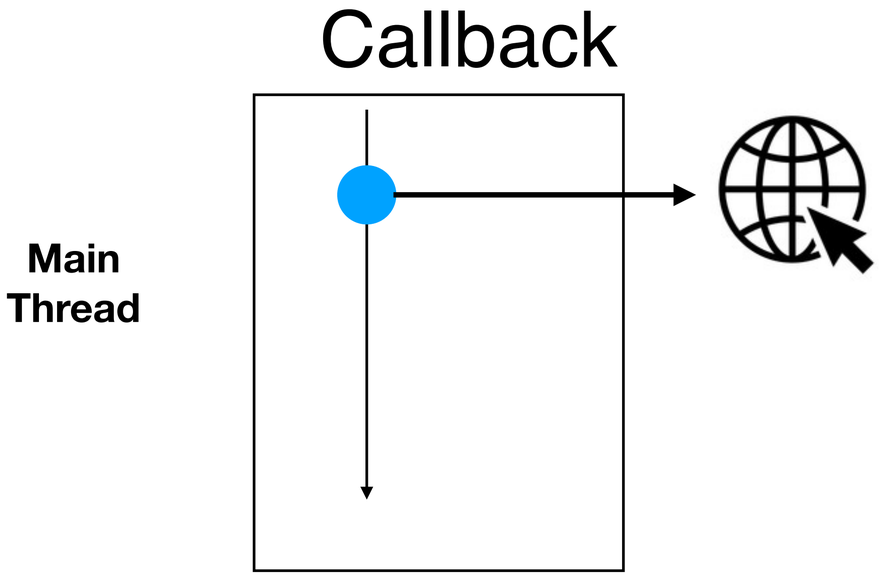
- **suspend** — pause the execution of the current coroutine, saving all local variables
- **resume** — continue a suspended coroutine from the place it was paused

You can only suspend functions called by other suspend functions, or using a coroutin

https://medium.com/androiddevelopers/coroutines-on-android-part-i-getting-the-background-3e0e54d20bb

KOTLIN

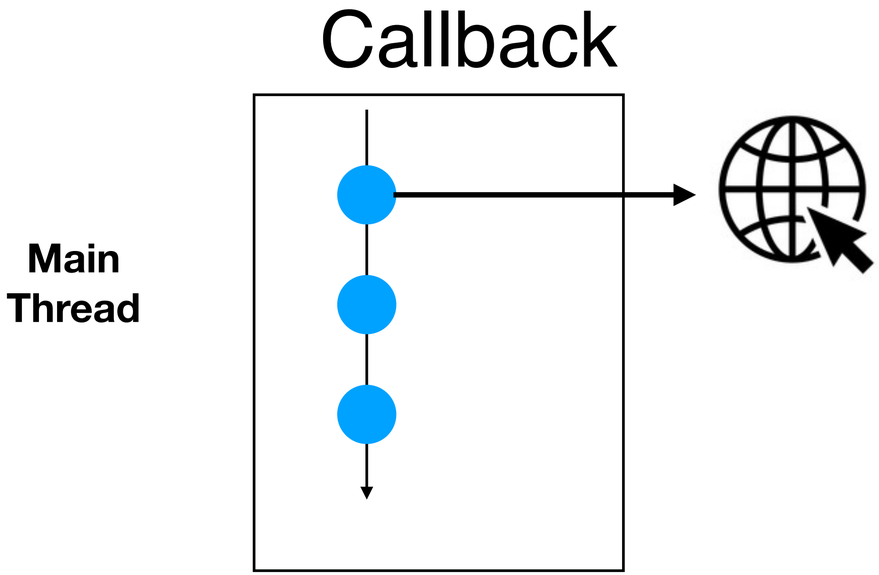
Coroutines - Long Running task



https://medium.com/androiddevelopers/coroutines-on-android-part-i-getting-the-background-3e0e54d20bb

KOTLIN

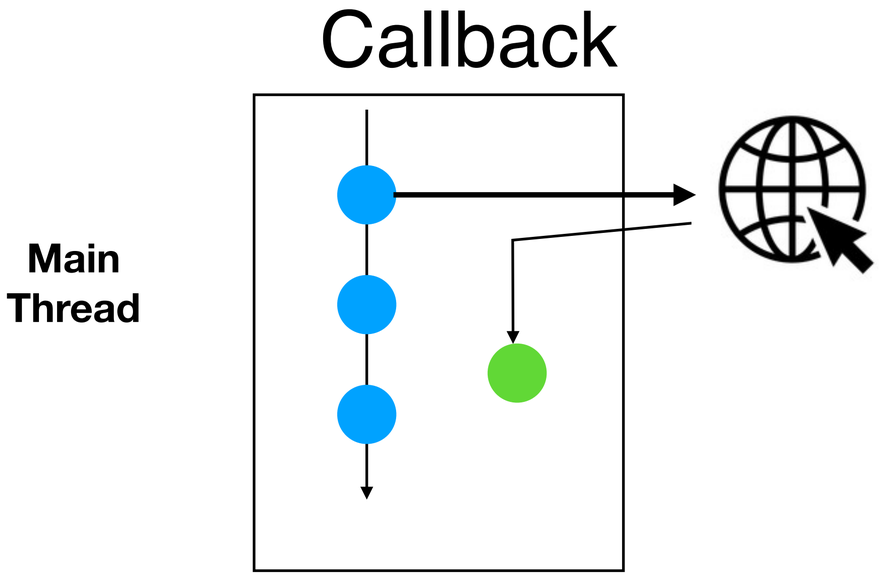
Coroutines - Long Running task



https://medium.com/androiddevelopers/coroutines-on-android-part-i-getting-the-background-3e0e54d20bb

KOTLIN

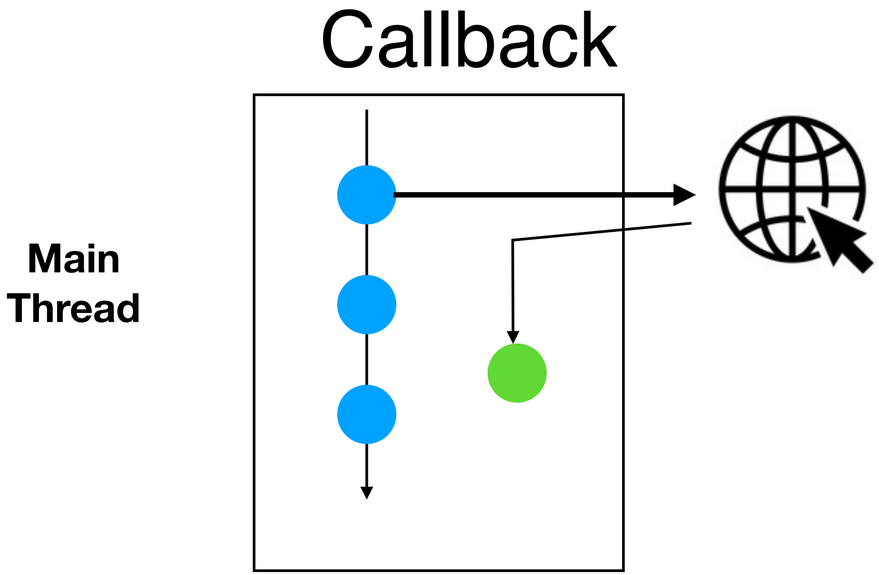
Coroutines - Long Running task



https://medium.com/androiddevelopers/coroutines-on-android-part-i-getting-the-background-3e0e54d20bb

KOTLIN

Coroutines - Long Running task

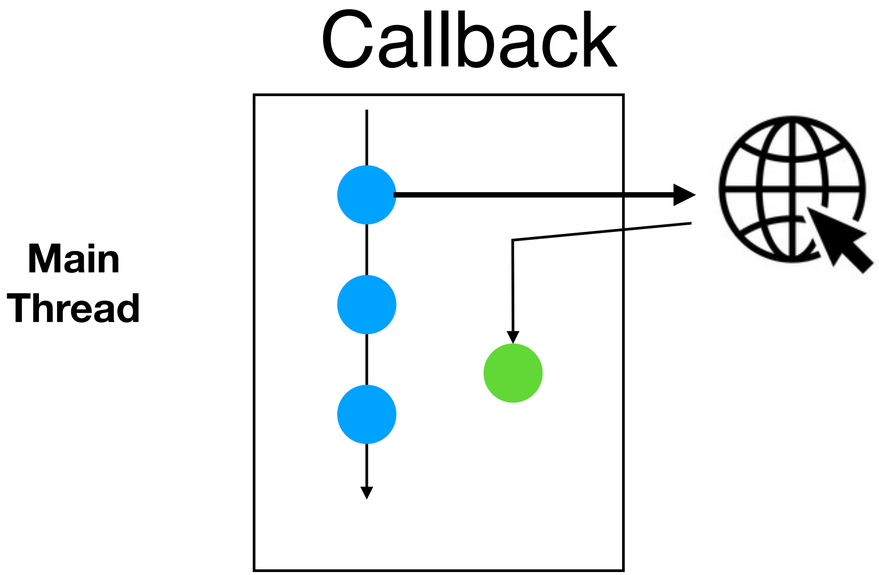



```
class ViewModel: ViewModel() {
   fun fetchDocs() {
       get("developer.android.com")
       {
         result -> show(result)
       }
    }
}
```

https://medium.com/androiddevelopers/coroutines-on-android-part-i-getting-the-background-3e0e54d20bb

KOTLIN

Coroutines - Long Running task



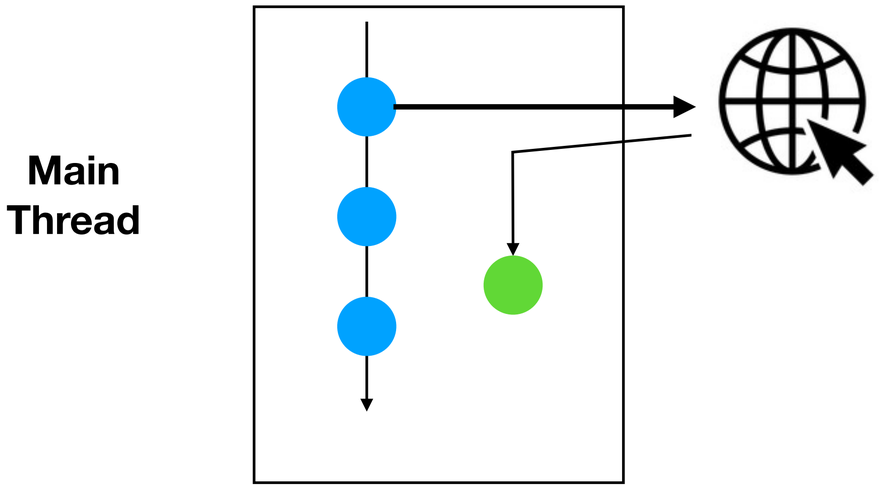

```
class ViewModel: ViewModel() {
    fun fetchDocs() {
        get("developer.android.com")
        {
          result -> show(result)
        }
    }
}
```

https://medium.com/androiddevelopers/coroutines-on-android-part-i-getting-the-background-3e0e54d20bb

KOTLIN

Coroutines - Long Running task

Callback




```
class ViewModel: ViewModel() {
    fun fetchDocs() {
        get("developer.android.com")
        {
          result -> show(result)
        }
    }
}
```

Coroutine

https://medium.com/androiddevelopers/coroutines-on-android-part-i-getting-the-background-3e0e54d20bb

KOTLIN

Coroutines - Main Safety

Coroutines will run on the main thread by default,
and suspend **does not mean background**.

Dispatchers are in charge of running the coroutines code.
They define if the coroutine runs in the main thread or in other thread

Dispatchers.Main
Main thread on Android, interact with the UI and perform light work
- Calling suspend functions - Call UI functions - Updating LiveData

Dispatchers.IO
Optimized for disk and network IO off the main thread
- Database* - Reading/writing files - Networking**

Dispatchers.Default
Optimized for CPU intensive work off the main thread
- Sorting a list - Parsing JSON - DiffUtils

https://medium.com/androiddevelopers/coroutines-on-android-part-i-getting-the-background-3e0e54d20bb

KOTLIN

Coroutines - Main Safety

Dispatchers.Main
Main thread on Android, interact with the UI and perform light work
- Calling suspend functions - Call UI functions - Updating LiveData

Dispatchers.IO
Optimized for disk and network IO off the main thread
- Database* - Reading/writing files - Networking**

Dispatchers.Default
Optimized for CPU intensive work off the main thread
- Sorting a list - Parsing JSON - DiffUtils

```
suspend fun fetchDocs() {
    val result = get("developer.android.com")
    show(result)
}
```

https://medium.com/androiddevelopers/coroutines-on-android-part-i-getting-the-background-3e0e54d20bb

KOTLIN

Coroutines - Main Safety

Dispatchers.Main
Main thread on Android, interact with the UI and perform light work
- Calling suspend functions - Call UI functions - Updating LiveData

Dispatchers.IO
Optimized for disk and network IO off the main thread
- Database* - Reading/writing files - Networking**

Dispatchers.Default
Optimized for CPU intensive work off the main thread
- Sorting a list - Parsing JSON - DiffUtils

```
suspend fun fetchDocs() {
   val result = get("developer.android.com")
   show(result)
}
```

```
suspend fun get(url: String) =
   withContext(Dispatchers.IO) {
       /* perform blocking network IO here */
    }
```

https://medium.com/androiddevelopers/coroutines-on-android-part-i-getting-the-background-3e0e54d20bb

KOTLIN

Coroutines - Main Safety

Dispatchers.Main
Main thread on Android, interact with the UI and perform light work
- Calling suspend functions - Call UI functions - Updating LiveData

Dispatchers.IO
Optimized for disk and network IO off the main thread
- Database* - Reading/writing files - Networking**

Dispatchers.Default
Optimized for CPU intensive work off the main thread
- Sorting a list - Parsing JSON - DiffUtils

```
suspend fun fetchDocs() {
    val result = get("developer.android.com")
    show(result)
}
```

```
suspend fun get(url: String) =
    withContext(Dispatchers.IO) {
        /* perform blocking network IO here */
    }
```

https://medium.com/androiddevelopers/coroutines-on-android-part-i-getting-the-background-3e0e54d20bb

How to run a Coroutine?

KOTLIN

Coroutines

You can start coroutines in one of two ways:

launch

Starts a new coroutine and doesn't return the result to the caller. Any work that is considered "fire and forget" can be started using launch.

```
fun onDocsNeeded() {
    viewModelScope.launch {
        fetchDocs()
    }
}
```

async

Starts a new coroutine and allows you to return a result with a suspend function called await.

```
suspend fun fetchTwoDocs() =
    coroutineScope {
        val deferredOne = async { fetchDoc(1) }
        val deferredTwo = async { fetchDoc(2) }
        deferredOne.await()
        deferredTwo.await()
    }
```

https://developer.android.com/kotlin/coroutines

launch is a bridge from regular functions into coroutines

```
fun onDocsNeeded() {
    viewModelScope.launch {
        fetchDocs()
    }
}
```

KOTLIN

Coroutines

You can start coroutines in one of two ways:

launch

Starts a new coroutine and doesn't return the result to the caller. Any work that is considered "fire and forget" can be started using launch.

```
fun onDocsNeeded() {
    viewModelScope.launch {
        fetchDocs()
    }
}
```

async

Starts a new coroutine and allows you to return a result with a suspend function called await.

```
suspend fun fetchTwoDocs() =
    coroutineScope {
        val deferreds = listOf(
            async { fetchDoc(1) },
            async { fetchDoc(2) }
        )
        deferreds.awaitAll()
    }
```

https://developer.android.com/kotlin/coroutines

KOTLIN

Coroutines

You can start coroutines in one of two ways:

launch

Starts a new coroutine and doesn't return the result to the caller. Any work that is considered "fire and forget" can be started using `launch`.

```
fun onDocsNeeded() {
    viewModelScope.launch {
        fetchDocs()
    }
}
```

?

async

Starts a new coroutine and allows you to return a `result` with a suspend function called `await`.

```
suspend fun fetchTwoDocs() =
    coroutineScope {
        val deferreds = listOf(
            async { fetchDoc(1) },
            async { fetchDoc(2) }
        )
        deferreds.awaitAll()
    }
```

https://developer.android.com/kotlin/coroutines

KOTLIN

Coroutines scopes

viewModelScope: While the view model is active

lifeCycleScope: Activity/fragment life cycle

KOTLIN

Coroutines

You can modify UI after doing a background work without doing to many steps

```
class MyViewModel : ViewModel() {

    fun launchDataLoad() {
        viewModelScope.launch {
            sortList()
            // Modify UI
        }
    }

    /**
    * Heavy operation that cannot be done in the Main Thread
    */
    suspend fun sortList() = withContext(Dispatchers.Default) {
        // Heavy work
    }
}
```

https://developer.android.com/kotlin/coroutines

KOTLIN

Coroutines

Possible Problems:

-Leaked Work

-Unable to keep track

-Problems handling errors

https://developer.android.com/kotlin/coroutines

KOTLIN

Coroutines

Possible Problems:

-Leaked Work.

-Unable to keep track.

-Problems handling errors

```
suspend fun loadLots() {
  coroutineScope {
    repeat(1_000) {
      launch { fetchDocs() }
    }
  }
}
```

https://medium.com/androiddevelopers/coroutines-on-android-part-ii-getting-started-3bff117176dd

KOTLIN

Coroutines

Check these links out:

Theory:

https://developer.android.com/kotlin/coroutines

https://medium.com/androiddevelopers/coroutines-on-android-part-i-getting-the-background-3e0e54d20bb

https://medium.com/androiddevelopers/coroutines-on-android-part-ii-getting-started-3bff117176dd

Practical Examples:

https://medium.com/androiddevelopers/coroutines-on-android-part-iii-real-work-2ba8a2ec2f45

And what about flutter?

Flutter is single-threaded, therefore, multi-threading is not available. Therefore, to avoid blocking the main thread, **async functions** should be used.

Multi-threading can be enabled by using multiple **isolates**

FLUTTER

Futures

```
Future<void> fetchUserOrder() {
  // Imagine that this function is fetching user info from
     another service or database
  return Future.delayed(Duration(seconds: 3),
         () =>   print('Large Latte'));
}

void main() {
  fetchUserOrder();
  print('Fetching user order...');
}
```

What is the output? Type the snippet in http://dartpad.dev

FLUTTER

Futures

```
Future<void> fetchUserOrder() {
  // Imagine that this function is fetching user info from
     another service or database
  return Future.delayed(Duration(seconds: 3),
         () =>   print('Large Latte'));
}

void main() {
  fetchUserOrder();
  print('Fetching user order...');
}
```

```
Console

  Fetching user order...
  Large Latte
```

Why ?

Because a future is code block expected to run in the future (when the main thread is free) and without blocking the main thread

FLUTTER

Futures

```
Future<void> fetchUserOrder() {
  // Imagine that this function is fetching user info from
     another service or database
  return Future.delayed(Duration(seconds: 3),
         () =>   print('Large Latte'));
}

void main() {
  fetchUserOrder();
  print('Fetching user order...');
}
```

The print is executed first because the future in the fetchUserOrder is delayed

FLUTTER

Futures

```
Future<void> fetchUserOrder() {
  // Imagine that this function is fetching user info from
     another service or database
  return Future( () =>   print('Large Latte'));
}

void main() {
  fetchUserOrder();
  print('Fetching user order...');

}
```

Try now with a future that is not delayed

FLUTTER

Futures

```dart
Future<void> fetchUserOrder() {
  // Imagine that this function is fetching user info from
     another service or database
  return Future( () =>   print('Large Latte'));
}

void main() {
  fetchUserOrder();
  print('Fetching user order...');

}
```

```
Console

  Fetching user order...
  Large Latte
```

FLUTTER

Futures

```
Future<void> fetchUserOrder() {
  // Imagine that this function is fetching user info from
     another service or database
  return Future( () =>   print('Large Latte'));
}

void main() {
  fetchUserOrder();
  print('Fetching user order...');
  print('......');
  print('.........');

}
```

What is the output?

FLUTTER

Futures

```dart
Future<void> fetchUserOrder() {
  // Imagine that this function is fetching user info from
     another service or database
  return Future( () =>   print('Large Latte'));
}

void main() {
  fetchUserOrder();
  print('Fetching user order...');
  print('......');
  print('.........');

}
```

Console

```
Fetching user order...
......
.........
Large Latte
```

FLUTTER

Futures

```dart
Future<void> fetchUserOrder() {
  // Imagine that this function is fetching user info from
     another service or database
  return Future( () =>   print('Large Latte'));
}

void main() {
  fetchUserOrder();
  print('Fetching user order...');
  print('......');
  print('.........');
}
```

At this point the main thread is free.. So the future is executed

FLUTTER

Futures

Let’s return the value now for being printed at the main function

```
Future<String> fetchUserOrder() {
  // Imagine that this function is fetching user info from
     another service or database
  return Future( () =>   'Large Latte');
}

void main() {
  print( fetchUserOrder() );
  print('Fetching user order...');

}
```

FLUTTER

Futures

```
Future<String> fetchUserOrder() {
  // Imagine that this function is fetching user info from
     another service or database
  return Future( () =>   'Large Latte');
}

void main() {
  print( fetchUserOrder() );
  print('Fetching user order...');

}
```

What is the output?

FLUTTER

Futures

```
Future<String> fetchUserOrder() {
  // Imagine that this function is fetching user info from
     another service or database
  return Future( () =>   'Large Latte');
}

void main() {
  print( fetchUserOrder() );
  print('Fetching user order...');

}
```

```
Console

  Instance of '_Future<String>'
  Fetching user order...
```

Why ?

Because the future is in state **uncompleted**

Therefore, you need to:

- Implemente a “then” block for the future (non blocking option)

- Wait for a value using the **async** and **await** keywords (this blocks the main thread)

FLUTTER

Futures

```
Future<String> fetchUserOrder() {
  // Imagine that this function is fetching user info from
     another service or database
  return Future( () =>   'Large Latte');
}

void main() {
  var myFuture = fetchUserOrder();
  myFuture.then( (message) {
     print( message );
  });

  print('Fetching user order...');

}
```

FLUTTER

Futures

```
Future<String> fetchUserOrder() {
  // Imagine that this function is fetching user info from
     another service or database
  return Future( () =>   'Large Latte');
}

void main() {
  var myFuture = fetchUserOrder();
  myFuture.then( (message) {
     print( message );
  });

  print('Fetching user order...');

}
```

FLUTTER

Futures

```
Future<String> fetchUserOrder() {
  // Imagine that this function is fetching user info from
     another service or database
  return Future( () =>   'Large Latte');
}

void main() {
  var myFuture = fetchUserOrder();
  myFuture.then( (message) {
     print( message );
  });

  print('Fetching user order...');

}
```

Console

```
Fetching user order...
Your order is Large Latte
```

FLUTTER

Futures

```
Future<String> fetchUserOrder() {
  // Imagine that this function is fetching user info from
     another service or database
  return Future( () =>   'Large Latte');
}

void main() {
  var myFuture = fetchUserOrder();
  myFuture.then( (message) {
     print( message );
  });

  print('Fetching user order...');

}
```

The code to be executed when the value is returned

What about the async/await way for futures?

FLUTTER

Futures

Let’s go back to a previous example

```
Future<String> fetchUserOrder() {
  // Imagine that this function is fetching user info from
     another service or database
  return Future( () =>   'Large Latte');
}

void main() {
  print( fetchUserOrder() );
  print('Fetching user order...');

}
```

```
Console

  Instance of '_Future<String>'
  Fetching user order...
```

FLUTTER

Futures

To make it work (blocking the main thread) we need to use async/ await

```
Future<String> fetchUserOrder() {
  // Imagine that this function is fetching user info from
  //another service or database
  return Future( () =>   'Large Latte');
}

void main() async {
  var order = await fetchUserOrder();
  print( "Your order is $order" );
  print('Fetching user order...');

}
```

```
Console

  Your order is Large Latte
  Fetching user order...
```

FLUTTER

Futures

```
Future<String> fetchUserOrder() {
  // Imagine that this function is fetching user info from
  //another service or database
  return Future( () =>   'Large Latte');
}

void main() async {
  var order = await fetchUserOrder();
  print( "Your order is $order" );
  print('Fetching user order...');

}
```

- To define an async function, add async before the function body
- The await keyword works only in async functions