

In one line: GCD submits closures to *queues* and *blocks threads* to wait; **Swift Concurrency** *suspends tasks* at `await` (the thread is freed) and lets the **compiler** prove safety with **actors** (serialised state), `@MainActor` (UI) and `Sendable` (what may cross). Rule zero in both: **UIKit only on the main thread**.

How it works — GCD

- DispatchQueue.main = **serial**, bound to the main thread (UI). DispatchQueue.global(qos:) = shared **concurrent** pool. DispatchQueue(label:) = serial unless attributes: .concurrent.
- async returns at once; sync **blocks the caller** until the block ran. main.sync from main (or sync onto the serial queue you are on) = **deadlock**.
- QoS**, high→low: .userInteractive > .userInitiated > .default > .utility > .background.
- DispatchGroup (enter/leave/notify) = wait for N jobs; DispatchSemaphore(value: N) = limit to N; .barrier on a *private concurrent* queue = reader-writer.
- URLSession completion handlers run on a background queue** (the session's delegate OperationQueue) — *never* main. Hop before UI: DispatchQueue.main.async {...}. Forgot .resume()? Tasks start suspended.

How it works — async/await

- await **suspends, never blocks**: the thread returns to a cooperative pool (one per core, no thread explosion); the task may resume on another thread.
- Task {} inherits actor, priority, task-locals. Task.detached {} inherits **nothing** (loses @MainActor). Both unstructured.
- Structured**: async let x = f() (fixed count), withThrowingTaskGroup (dynamic count). Parent waits for children; cancel flows down, errors flow up. Two plain awaits in a row run **one after the other**; async let a = f(); async let b = g() starts both at once.
- Cancellation is cooperative**: task.cancel() only sets a flag. Check Task.isCancelled or try Task.checkCancellation() (throws CancellationError).
- withCheckedThrowingContinuation bridges a callback API — resume **exactly once** (0 = hang, 2 = trap).
- actor**: reference type; one task at a time touches its state; outside calls need await. nonisolated opts a member out (lets only). **Reentrant**: at every await inside it, other calls may run ⇒ re-check state after await.
- @MainActor = global actor on the main thread; on a type, method or property. Off-main: await MainActor.run {}.
- Sendable = safe to cross isolation: value types of Sendable parts, final class with only let Sendable props, actors (implicitly). @unchecked Sendable = “I lock it myself”. Swift 6 makes violations *errors*.

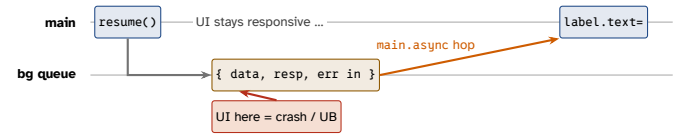
Example — the main-thread hop

```
// 1. Completion handler: runs on a BACKGROUND queue
URLSession.shared.dataTask(with: url) { data, _, _ in
    let img = data.flatMap(UIImage.init(data:))
    DispatchQueue.main.async { self.imageView.image = img } // hop!
}.resume() // tasks start suspended

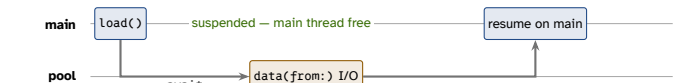
// 2. async/await inside @MainActor: resumes on main, no hop
@MainActor func load() async throws {
    let (data, _) = try await URLSession.shared.data(from: url)
    imageView.image = UIImage(data: data) // already on main
}
```

Picture — which thread runs what

1 · completion handler



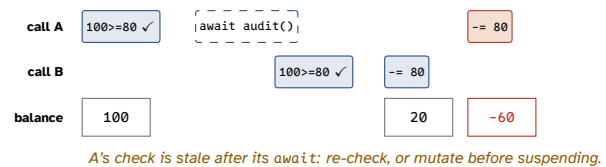
2 · await in @MainActor



3 · main.sync from main



Picture — actor reentrancy



Interview traps

- ! “Which thread is the completion on?” (you missed it): URLSession’s **background** queue → hop to main for UI. `async/await` resumes on *your* actor — `@MainActor` needs no hop; a nonisolated func does.
- ! `await` ≠ blocking. `sync/semaphore/sleep()` block a thread; `Task.sleep` suspends.
- ! `Task.detached` in a `@MainActor` VM runs off-main.
- ! Cancelling does not stop a loop that never checks.
- ! `DispatchGroup`: an early return without `leave()` → `notify` never fires; use `defer`.
- ! Actor as a “make it fast” tool: it *serialises* — use a `TaskGroup` for parallel work.

Remember

GCD blocks threads, await suspends tasks. “Background brings the data, main paints it.” QoS: Interactive, Initiated, Default, Utility, Background — “I I D U B”.

Likely questions

- Task vs .detached? — inherits actor/priority vs nothing.
- async let vs TaskGroup? — fixed vs dynamic number of children.
- Why main.sync from main hangs? — it waits for a block that needs main.
- What is Sendable? — safe to share across isolation domains.
- @MainActor replaces? — `main.async` for UI work.