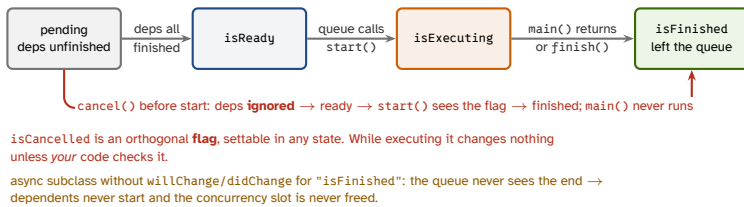
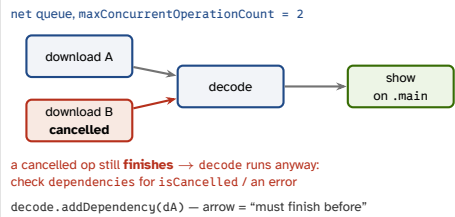


In one line: An Operation is a **single-shot**, KVO-observable unit of work; an OperationQueue starts it when isReady (every dependency isFinished), caps parallelism with maxConcurrentOperationCount, and drops it when it posts isFinished. **Cancellation is a flag**, dependencies are **ordering only** (no data), and an async subclass must drive its own state **with KVO** — or it never finishes.

State machine (KVO keys the queue watches)



Dependencies — a graph across queues



How it works

- Sync op:** override `main()`; finished when it returns (async work started inside is *not* awaited). **Async op:** override `start()` (never `super.start()`), `isAsynchronous = true`, own `isExecuting/isFinished` behind a lock, KVO for both, finish exactly once — also when cancelled before starting.
- A queue **always** calls `start()` on a separate thread and ignores `isAsynchronous`; it matters only when you call `start()` yourself.
- `b.addDependency(a)`: works across queues; “finished” includes cancelled; a cycle never becomes ready (silent hang). Results: `b` reads `a.output` (or an adapter op copies it).
- `maxConcurrentOperationCount` defaults to -1 (system decides); 1 = serial, but order = readiness → `queuePriority` → insertion.
- `qualityOfService` (`.userInteractive ... .background`) sets the thread's priority and energy class; `queuePriority` (`.veryLow ... .veryHigh`) only reorders *ready* ops in *one* queue.
- `BlockOperation`: its blocks run concurrently, finished when all return. `addBarrierBlock` (iOS 13) waits for everything queued before it. `isSuspended` stops *starting* ops. `OperationQueue.main` = serial, main thread; `underlyingQueue` targets a `DispatchQueue`.
- `completionBlock` runs after finish on an **arbitrary thread**, and is set to `nil` once it starts (iOS 8+).

Example — KVO-correct async Operation

```
class AsyncOperation: Operation {
    private let lock = NSLock()
    private var _exec = false, _done = false
    private func locked<T>(_ f: () -> T) -> T {
        lock.lock(); defer { lock.unlock() }; return f()
    }
    override var isAsynchronous: Bool { true }
    override var isExecuting: Bool { locked { _exec } }
    override var isFinished: Bool { locked { _done } }
    override func start() {
        // never super.start()
        if isCancelled { finish(); return } // still must finish
        willChangeValue(forKey: "isExecuting")
        locked { _exec = true }
        didChangeValue(forKey: "isExecuting")
        run { self.finish() } // subclass: async work
    }
    func run(_ done: @escaping () -> Void) { done() }
    final func finish() {
        // exactly once
        let keys = ["isExecuting", "isFinished"]
        keys.forEach(willChangeValue(forKey:))
        locked { _exec = false; _done = true }
        keys.forEach(didChangeValue(forKey:))
    }
}
```

Related: concurrency (GCD, actors) · concurrency-patterns (bounded parallelism) · async-sequences-streams · KVO (observation-and-combine) · background-execution · crashes-symbolication (0x8badf00d)

Operation vs GCD vs async/await

Need	Operation	GCD	Swift concurrency
dependency graph	<code>addDependency</code> , cross-queue	group notify, by hand	<code>async let</code> , ordered <code>await</code>
cancel a graph	<code>cancelAll()</code> sets flags	work item <code>cancel()</code> : only if not started	<code>task.cancel()</code> reaches child tasks
cap parallelism	<code>maxConcurrent...</code>	semaphore / serial queue	task group: seed <i>n</i> , add one per finish
observe state	KVO, progress (iOS 13)	—	—
delay	—	<code>asyncAfter</code>	<code>Task.sleep(for:)</code>

Operation still wins for: a **cancellable graph** across queues, a hard concurrency cap, KVO/Progress UI, subclassable reusable *types*, legacy code. New code on `async/await`: task groups + `Task.checkCancellation()`.

Interview traps

- ! Async op without the manual KVO → stuck “executing” forever.
- ! `cancel()` does not stop a running op or its `URLSessionTask` — check `isCancelled` and cancel the task yourself.
- ! Cancelled dependency = finished → the dependent still runs.
- ! `op.completionBlock = { use(op.result) }` captures op strongly — a cycle until it runs; use `[unowned op]`. Hop to main for UI.
- ! Adding the same instance twice / after it finished → `NSInvalidArgumentException`. Single-shot.
- ! `waitUntilAllOperationsAreFinished()` / `waitUntilFinished:` `true` on main → hang, watchdog 0x8badf00d.

Remember

Ready → Executing → Finished; cancel is only a flag; no KVO, no finish.

Likely questions

- When Operation over GCD? — dependencies, graph cancel, concurrency cap, KVO.
- Async Operation? — override `start`, own the flags, KVO both on finish.
- Priority vs QoS? — reorder ready ops in one queue vs system-wide CPU/energy class.
- 100 downloads, 4 at a time? — `maxConcurrentOperationCount = 4` or a task group seeded with 4.