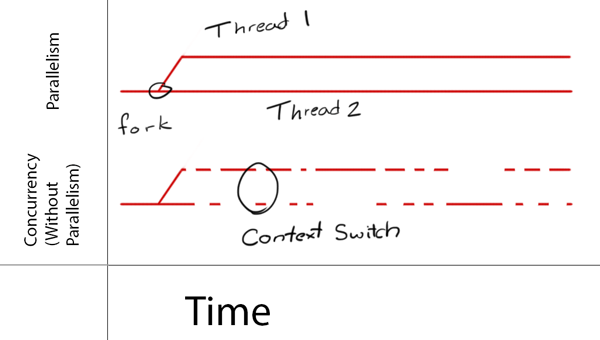
**Thread and Queue**

**Queue**

**Part 1**



* (concurrency) Với nhưng thiết bị có 1 core thì nó dùng cách chuyển đổi nội dung (context switch) để thực hiện concurrency. (xử lý nhiều việc luân phiên cùng lúc)
* (parallelism) Với các thiết bị nhiều core thì mỗi core đảm nhận 1 việc. (nó cũng có thể xử lý concurency)

GCD trên ios quản lý shared thread pool. Sau khi thêm block code vào GCD nó sẽ quyết định xem nên thực thi block code ở thread nào.

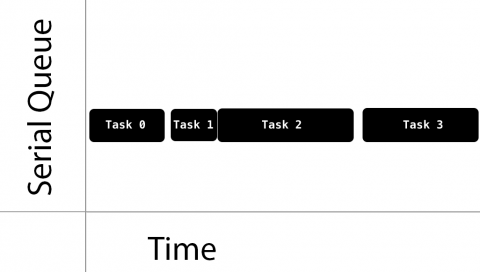
GCD sẽ dựa trên system resource để quyết định xem có cần xử lý song song không. It’s important to note that parallelism requires concurrency, but concurrency does not guarantee parallelism.

Basically, concurrency is about *structure* while parallelism is about *execution*.

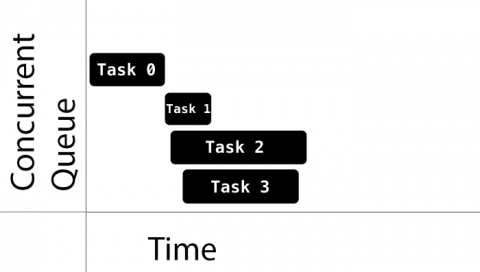
DispatchQueue xử lý công việc theo mô hình FIFO (First in first out)

Dispatch queue là Thread-safe.

* Serial queue: nghĩa là 1 lúc chỉ có 1 task được thực thi



* Concurrent queue: cho phép nhiều task thực hiện cùng 1. Task nào add trước thì thực hiện trước nhưng không có nghĩa là xong trước. Và không thể khẳng định khi nào và bao nhiêu task tiếp theo được bắt đầu.



GCD cung cấp 3 loại queues:

1. Main queue: chạy serial queue trên main thread
2. Global queue: concurrent queues shared by whole system. Có 5 loại quality service:

* Background: có độ ưu tiên thấp nhất. (dùng để thực hiện công việc khi app chạy ở background)
* Utility: có độ ưu tiên chỉ cao hơn background. (dùng để chạy các long-running task hay các tiến trình người dùng không theo dõi thường xuyên)
* Default: có độ ưu tiên cao hơn background và ultility. (dùng để thực hiện các công việc khởi tạo hoặc người dùng thường xuyên theo dõi).
* userInitiated: có độ ưu tiên chỉ thấp hơn userInteractive. ( dùng để thực hiện các công việc cần kết quả ngay tức thì. Vd: load nội dung của mail muốn hiển thị)
* userInteractive: có độ ưu tiên cao nhất. (dùng để thực hiện các công việc liên quan đến tương tác với người dùng. Vd: animation)
* unspecified: chưa xác định độ ưu tiên

1. Custom queues: queue do dev tạo ra, có thể là serial hoặc concurrent. Requests in these queues actually end up in one of the global queues.

Synchronous vs asynchronous

* Synchronous: trả quyền kiếm soát về the caller sau khi task đó hoàn thành.
* Asynchronous: trả quyền kiềm soát về cho caller ngay lập tức, the caller có thể thực hiện các công việc khác ngay lập tức.

Managing task

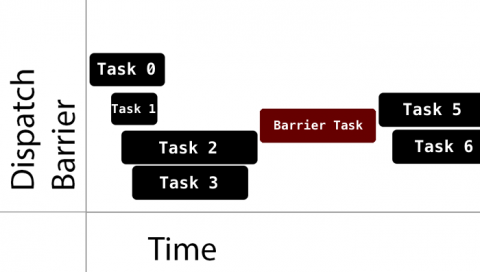
* Có thể coi task như các closure.

Mỗi task được submit tới DispatchQueue là 1 DispatchWorkItem.  You can configure the behavior of a DispatchWorkItem such as its QoS class or whether to spawn a new detached thread.

Managing singletions

Mục đích: để biến singletion thành thread-safe sử dụng flag barriers

Khi submit 1 DispatchWorkItem to dispatchqueue ta có thể set flags. Khi submit barriers flag, các task được submit trước nó phải được hoàn thành trước khi nó được xử lý và trước khi nó xử lý xong thì chỉ 1 mình nó được xử lý.



Chú ý: khi xử dụng barriers

* Khi sử dụng barriers trên global queue phải cần thận do global được sử dụng cho toàn bộ hệ thống.
* Sử dụng trong custom serial queue là dư thừa

Chú ý: khi xử dụng sync với thread khác

* Trên main queue: dễ tạo ra deadlock nó khi block main thread và nó lại block chính nó.
* Trên Global queue: hữ dụng để đợi task khác hoàn thành
* Trên custom serial queue: có thể block chính nó

**Part 2**

**DispathGroup**

* Enter: start task
* Leave: done task
* Wait: đợi task hoàn thành (có thể set timeout) (sẽ block thread hiện tại lại cho đến khi các task hoàn thành hoặc vượt timeout)
* Notify(queue:work:) cung cấp asynchronous completion closure. Thực hiện code trong closure khi các task đã hoàn thành. (không block thread)

**ConcurrentPerform**

DispatchQueue.concurrentPerform(iterations:execute:)

Concurrent Run loop: thực hiện các vòng lặp (iterations) đồng thời. (không quan tâm đến thứ tự, return khi các task hoàn thành)

let \_ = DispatchQueue.global(qos: .userInitiated) : báo cho GCD sử dụng qos: .usertinitiated cho concurrentPerform

**Canceling Dispatch Blocks**

Chỉ có thể cancel khi nó đang đợi được xử lý. Không thể cancel khi nó đã xử lý xong hoặc đang xử lý.

### Testing Asynchronous Code

Test chạy trên main thread, nên nó là serial

Nếu trong các func test mà chạy asycnchronous thì func tiếp theo sẽ được chạy đồng thời với task trước đó.

Xem chi tiết trong link

Custom queue so với global queue:

* Có thể thêm label để dễ debug
* Có thể suspend được
* Barriers

Lưu ý:

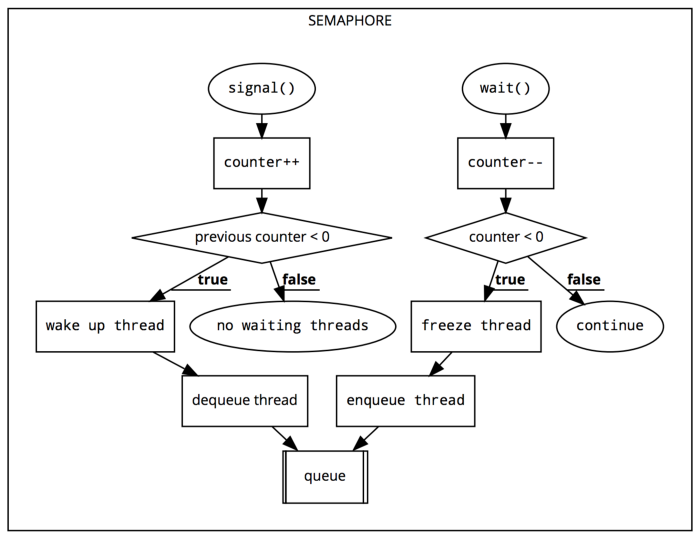
* Không nên sử dụng concurrentPerform với số vòng lặp nhỏ
* Nên dùng 1 cách cần thận vì dễ sinh lỗi và mất nhiều thời gian đê thực hiện.
* Khiến code phức tạp hơn

**DispatchSemaphore**

DispatchSemaphore dùng để quản lý số lượng task được thực hiện trong 1 thời điểm.

Trong DispatchSemaphore sử dụng 1 biến Counter value để quyết định xem có freeze thread hiện tại hay không. Giá trị của Counter value thay đổi khi ta gọi function signal() hoặc wait().

* Gọi wait() trước khi thực hiện task. Nó sẽ hỏi semaphore xem có cho phép thực hiện công việc không dựa vào số task tối đa được thực hiện. Nếu không được nó sẽ freeze thread đó lại. (Counter value giảm đi 1)
* Nếu counter value (sau khi trừ) < 0 thì thread bị freeze
* Nếu counter value (sau khi trừ) >= 0 thì thread không bị freeze
* Gọi signal() khi thực hiện xong task. Nó sẽ thông báo cho semaphore biết có 1 công việc vừa thực hiện xong. (Counter value tăng thêm 1)
* Nếu counter value (trước khi cộng) < 0 thì semaphore sẽ giải phóng 1 thread bị freeze
* Nếu counter value (trước khi cộng) >= 0 thì lúc này các công việc đã hoàn tất



Chú ý:

* Dễ gây ra thread explosion

**NSRecursiveLock**

Dùng để ngăn/block việc 1 shared, non thread-safe, resource bị tương tác bởi 1 hoặc nhiều thread cùng 1 lúc.

Nếu Thread A gọi lock() qua 1 biến NSLock và sau đó có 1 Thread B khác hoặc chính Thread A đó gọi lock() trước khi Thread A unlock() thì thread gọi sau sẽ bị block. Đối với việc gọi lock() qua biến NSRecursiveLock, nếu lock() được gọi từ cùng 1 Thread thì Thread đó sẽ không bị block

**Run loop**

**Định nghĩa**

Run loop là object quản lý events và messages mà nó cần và cung cấp entry function để thực thi logic của event.

After the thread executes this function, it will stay in the *“accept message-> wait-> processing -> Sleep till next message -> accept message”* loop inside the function until the end of the loop (such as a message passed to quit) and the function returns.

NSRunLoop và CFRunLoopRef phục vụ cùng 1 mục đích

CFRunLoopRef có trong CoreFoundation framework. Nó cung cấp APIs các function thuần C. Nó là thread safe

NSRunLoop nó là 1 wrapper của CFRunLoopRef cung cấp Object-oriented APIs, các APIs này không phải thread-safe.

Hệ thống mặc định có 5 modes registered:

1. kCFRunLoopDefaultMode: App’s default mode, main thread thường chạy trên mode này.
2. UITrackingRunLoopMode: mode theo dõi giao diện, dùng cho ScrollView để track touch và slide, để đảm bảo giao diện không bị ảnh hường bới các modes khắc khi sliding.
3. UIInitializationRunLoopMode: mode đầu tiên được gọi khi app khởi động, nó sẽ không được sử dụng cho đến khi khởi động xong
4. GSEventReceiveRunLoopMode: Internal Mode phục vụ event hệ thống, thường không được sử dụng
5. KCFRunLoopCommanModes: là placeholder mode và không có effect gì đặc biệt.

Mục đích của run loop:

1. Giữ cho app chạy và tiếp nhận user input
2. Quyết định xem lúc nào thì event sẽ được xử lý.
3. Call decoupling
4. Save CPU time

Các đầu vào của run loop:

1. Port-based: events bất đồng bộ từ port. Ví dụ: mach-ports, socket, message ports, file descriptors.
2. Timer: events từ timer object.
3. Perform selector: cho phép gọi selector từ thread khác. (khi thêm selector thì run loop phải active)
4. Custom: tự thiết lập các signal cho input

Lưu ý:

* Do mặc định run loop không chạy trên background thread nên Timer sẽ không hoạt động (phải run run loop trên background thread đó thì timer mới hoạt động, phải run run loop sau timer bởi khi đó thì timer mới có trong timers list)
* Muốn dừng run loop thì phải qua hàm CFRunloopStop()

**Thread**

**Part 1**

Thuật ngữ

* The term *thread* is used to refer to a separate path of execution for code.
* The term *process* is used to refer to a running executable, which can encompass multiple threads.
* The term *task* is used to refer to the abstract concept of work that needs to be performed.

Thread có 3 trạng thái chính: running, ready, blocked

Chú ý:

* **Thread explosion:** khi submit task đến concurrent queue mà queue đó đang bị block (vd: do semaphore, sync, …) có thể tại thời điểm đó có những task vẫn đang đợi được xử lý nhưng hệ thống sẽ tạo ra các thread mới ( việc tạo thread sẽ tốn nhiều tài nguyên hệ thống mà trong trường hợp này là không cần thiết)

Vd: ta submit 5000 task đến 1 concurrent queue và chỉ có 10 task được xử lý 1 lúc nhưng hệ thống lại tạo ra 64 thread thì lúc này có 54 thread đang bị block lại (nhưng thread này coi như vô dụng, không cần thiết) (việc cung cấp số lượng thread tối đa do hệ thống quyết định, ta yêu cầu 1000 thread không có nghĩa hệ thống sẽ cung cấp 1000 thread)

**Link**

Dispatch Queue:

* Part 1: <https://www.raywenderlich.com/5370-grand-central-dispatch-tutorial-for-swift-4-part-1-2>
* Part 2: <https://www.raywenderlich.com/5371-grand-central-dispatch-tutorial-for-swift-4-part-2-2>

RunLoop:

* <https://medium.com/@prafullkumar77/ios-run-loop-what-why-when-7febead400b7>
* <https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/Multithreading/RunLoopManagement/RunLoopManagement.html#//apple_ref/doc/uid/10000057i-CH16-SW5>
* <https://thanhvu.dev/2019/07/04/runloop/>

Thread:

* Part 1: <https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/Multithreading/AboutThreads/AboutThreads.html#//apple_ref/doc/uid/10000057i-CH6-SW2>

Chưa đọc:

* <https://www.freecodecamp.org/news/ios-concurrency/#:~:text=Thread%20explosion&text=This%20can%20happen%20when%20you,and%20threads%20aren't%20cheap>.