# Multithreading

* The main UI of the app runs on the main thread
* For better user experience, image downloading will happen in the background.
* There can be multiple background threads but only one frontend thread.
* System level alert are happening in background thread.
* Loading indicator means something going on in the background.

## Queue

* In iOS queue is made and the OS executes one queue and one thread and so on.
* So we dont deal thread, we just deal with queue.
* There are 3 types of queue
  + Main Queue
  + Concurrent Queue
  + Serial Queue
* Main Queue is UI One
* Serial queue is one after another.

## Serial Queues

When you choose to create a queue as serial queue, the queue can only execute one task at a time. All tasks in the same serial queue will respect each other and execute serially. However, they don’t care about tasks in separate queues which means that you can still execute tasks concurrently by using multiple serial queues. For example, you can create two serial queues, each queue executes only one task at a time but up to two tasks could still execute concurrently.

Serial queues are awesome for managing a shared resource. It provides guaranteed serialized access to the shared resource and prevents race conditions. Imagine that there is a single ticket booth but there are a bunch of people who want to buy cinema tickets, here the staff at the booth is a shared resource. It’ll be chaotic if the staff has to serve these people all at the same time. To handle this situation, people are required to queue up (serial queue), so that the staff can serve the customers one at a time.

Again, it doesn’t mean the cinema can only handle one customer at a time. If it sets up two more booths, it can serve three customers at one time. This is why I said you can still perform multiple tasks in parallel by using several serial queues.

The advantages of using serial queues are:

1. Guaranteed serialized access to a shared resource that avoids race condition.
2. Tasks are executed in a predictable order. When you submit tasks in a serial dispatch queue, they will be executed in the same order as they are inserted.
3. You can create any number of serial queues.

// dispatchQueue manages the execution of work items

dispatch\_queue\_t queVar = dispatch\_queue\_create("my first queue", NULL);

// async means your program will not wait for the code to finish.

// So basically the whole file will not wait for this block to complete

sync is the opposite

dispatch\_async(queVar, ^{

NSData \*data = [NSData dataWithContentsOfURL:[NSURL URLWithString:@"http://wwwfacebook.com.jpg"]];

dispatch\_async(dispatch\_get\_main\_queue(), ^{

self.imgView.image = [UIImage imageWithData:data];

});

});

1. dispatch\_queue\_t queVar = dispatch\_queue\_create("my first queue", DISPATCH\_QUEUE\_SERIAL);

## Concurrent Queues

As the name suggests, concurrent queues allows you to execute multiple tasks in parallel. The tasks (blocks of codes) starts in the order in which they are added in the queue. But their execution all occur concurrently and they don’t have to wait for each other to start. Concurrent queues guarantee that tasks start in same order but you will not know the order of execution, execution time or the number of tasks being executed at a given point.

For example, you submit three tasks (task #1, #2 and #3) to a concurrent queue. The tasks are executed concurrently and are started in the order in which they were added to the queue. However, the execution time and finish time vary. Even it may take some time for task #2 and task #3 to start, they both can complete before task #1. It’s up to the system to decide the execution of the tasks.

dispatch\_queue\_t aQueue = dispatch\_get\_global\_queue(DISPATCH\_QUEUE\_PRIORITY\_DEFAULT, 0);

// async means your program will not wait for the code to finish.

// So basically the whole file will not wait for this block to complete

sync is the opposite

The following is concurrent queue

dispatch\_async(dispatch\_get\_global\_queue(DISPATCH\_QUEUE\_PRIORITY\_HIGH,0), ^{

NSData \*data = [NSData dataWithContentsOfURL:[NSURL URLWithString:@"http://wwwfacebook.com.jpg"]];

dispatch\_async(dispatch\_get\_main\_queue(), ^{

self.imgView.image = [UIImage imageWithData:data];

});

});

dispatch\_queue\_t queVar = dispatch\_queue\_create("my first queue", DISPATCH\_QUEUE\_CONCURRENT);

## Using Queues

Now that we have explained both serial and concurrent queues, it’s time to see how we can use them. By default, the system provides each application with a single serial queue and four concurrent queues. The main dispatch queue is the globally available serial queue that executes tasks on the application’s main thread. It is used to update the app UI and perform all tasks related to the update of UIViews. There is only one task to be executed at a time and this is why the UI is blocked when you run a heavy task in the main queue.

Besides the main queue, the system provides four concurrent queues. We call them Global Dispatch queues. These queues are global to the application and are differentiated only by their priority level. To use one of the global concurrent queues, you have to get a reference of your preferred queue using the function dispatch\_get\_global\_queue which takes in the first parameter one of these values:

* DISPATCH\_QUEUE\_PRIORITY\_HIGH
* DISPATCH\_QUEUE\_PRIORITY\_DEFAULT
* DISPATCH\_QUEUE\_PRIORITY\_LOW
* DISPATCH\_QUEUE\_PRIORITY\_BACKGROUND

These queue types represent the priority of execution. The queue with HIGH has the highest priority and BACKGROUND has the lowest priority. So you can decide the queue you use based on the priority of the task. Please also note that these queues are being used by Apple’s APIs so your tasks are not the only tasks in these queues.

Concurrent queues are very good in performing loops concurrently

for (i = 0; i < count; i++) {

printf("%u\n",i);

}

dispatch\_queue\_t queue = dispatch\_get\_global\_queue(DISPATCH\_QUEUE\_PRIORITY\_DEFAULT, 0);

dispatch\_apply(count, queue, ^(size\_t i) ;

printf("%u\n",i);

});

## GCD

* GCD is a C API
* Grand central dispatch is C-based, lower level way to manage multi-threading
* NSOperation is built on GCD.
* dispatch\_async is the grand central dispatch. It will make a new thread if it has to. It will use a previous thread if it has to.

## GCD vs NSOperation

* GCD is a low-level C-based API.
* NSOperation and NSOperationQueue are Objective-C classes.
* NSOperationQueue is objective C wrapper over GCD. If you are using NSOperation, then you are implicitly using *Grand Central Dispatch.*
* **GCD advantage over NSOperation:**
  + *implementation*
    - For GCD implementation is very light-weight
    - NSOperationQueue is complex and heavy-weight
* **NSOperation advantages over GCD:**
  + *Control On Operation*
    - you can Pause, Cancel, Resume an NSOperation
  + *Dependencies*
    - you can set up a dependency between two NSOperations
    - operation will not started until all of its dependencies return true for finished.
  + *State of Operation*
    - can monitor the state of an operation or operation queue. ready ,executing or finished
  + *Max Number of Operation*
    - you can specify the maximum number of queued operations that can run simultaneously
* **When to Go for** GCD **or** NSOperation
  + when you want more control over queue (all above mentioned) use NSOperation and for simple cases where you want less overhead (you just want to do some work "into the background" with very little additional work) use GC.

## NSOperation

* // Allocated here for succinctness.
* NSOperationQueue \*q = [[NSOperationQueue alloc] init];
* /\* Data to process \*/
* NSData \*data = [@"Hello, I'm a Block!" dataUsingEncoding: NSUTF8StringEncoding];
* /\* Push an expensive computation to the operation queue, and then
* \* display the response to the user on the main thread. \*/
* [q addOperationWithBlock: ^{
* /\* Perform expensive processing with data on our background thread \*/
* NSString \*string = [[NSString alloc] initWithData: data encoding: NSUTF8StringEncoding];
* /\* Inform the user of the result on the main thread, where it's safe to play with the UI. \*/
* /\* We don't need to hold a string reference anymore \*/
* }];

# Semaphores

* Method of locking a resource that can be used by multiple resources. Ig you want 2 thread access on your resource but no more. Then you use semaphore.
* #import <dispatch/dispatch.h>
* dispatch\_semaphore\_t sema;
* -(void) doSomething:(NSString \*)s {
  + for (int i = 0; i < 50; i++) {
    - dispatch\_semaphore\_wait(sema, DISPATCH\_TIME\_FPREVER);
    - NSLog(@"%@\t%d", s, i);
    - dispatch\_semaphore\_signal(sema);
  + }
* -(void)viewDidLoad {
  + [super viewDidLoad];
  + sema = dispatch\_semaphore\_create(2);
  + dispatch\_queue\_t queue = dispatch\_get\_global\_queue(DISPATCH\_QUEUE\_PRIORITY\_DEFAULT, 0);
  + dispatch\_async(queue, ^(void){[self doSomething:@"Task 1"];});
  + dispatch\_async(queue, ^(void){[self doSomething:@"Task 2"];});
  + dispatch\_async(queue, ^(void){[self doSomething:@"Task 3"];});
* }

# Synchronized(Lock)

* If you’ve done any concurrent programming in Objective-C, then you’ve seen the @synchronized construct. The @synchronized construct does just what a lock does: it prevents different threads from executing the same code at the same time. But it’s more convenient and more readable in some cases compared to allocating, locking and unlocking an NSLock.
* @implementation ThreadSafeQueue
* {
* NSMutableArray \*\_elements;
* NSLock \*\_lock;
* }
* - (instancetype)init
* {
* self = [super init];
* if (self) {
* \_elements = [NSMutableArray array];
* \_lock = [[NSLock alloc] init];
* }
* return self;
* }
* - (void)push:(id)element
* {
* [\_lock lock];
* [\_elements addObject:element];
* [\_lock unlock];
* }
* @end
* @implementation ThreadSafeQueue
* {
* NSMutableArray \*\_elements;
* }
* - (instancetype)init
* {
* self = [super init];
* if (self) {
* \_elements = [NSMutableArray array];
* }
* return self;
* }
* - (void)increment
* {
* @synchronized (self) {
* [\_elements addObject:element];
* }
* }
* @end