Desarrollo de Apps para iOS

Concurrencia y Usabilidad

IWEB,LSWC 2013-2014
Santiago Pavón

ver: 2013.11.25

Objetivos

- La interface de usuario:
  - que no se quede bloqueada mientras estamos realizando un cálculo muy largo, descargando recursos de la red, ...
  - que siempre responda ágilmente a las acciones del usuario.

Main Run Loop

- Todas las aplicaciones tienen un thread (main thread) donde se ejecuta el main run loop.
  - Procesa los eventos, ejecuta las acciones de nuestros controles (target-action), actualiza el interface de usuario.

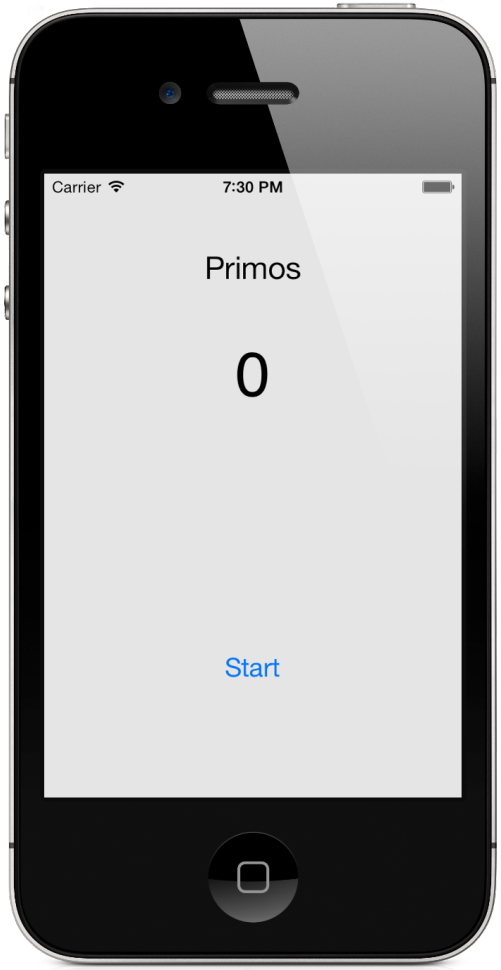
Ejemplo: Números Primos

- El ejemplo ejecuta una acción en el main thread que tarda mucho en terminar.
  - **Se congela la interface de usuario.**
- La label:
  - muestra el último primo encontrado.

```
@property (nonatomic,weak)
    IBOutlet UILabel* primeLabel;
```

- El botón:
  - empezar a calcular

```
-(IBAction) findPrimes;
```



```
- (BOOL) isPrime:(int)n {
    NSUInteger r = sqrt(n);
    for (NSUInteger i=2 ; i<n ; i++) {
        if (n%i == 0) return NO;
    }
    return YES;
}


- (IBAction) findPrimes {
    for (NSUInteger n=2 ;; n++) {
        if ([self isPrime:n]) {
            NSLog(@"Number %u is prime",n);
            self.primeLabel.text = [NSString stringWithFormat:@"%u",n];
        }
    }
}
```

Run Loop

Planificar en el Run Loop

- Podemos planificar la ejecución de un método en el run loop con:

  `[NSObject performSelector:withObject:afterDelay:]`

  - Existen muchos métodos perform… *(Consultar documentación)*

- En el ejemplo: haremos que el método findPrimes no busque todos los primos, sólo el siguiente.
  - La ejecución del método debe ser corta para no bloquear el main run loop.
    - Si el método requiere mucho tiempo de ejecución, conviene descomponerlo en varios métodos que requieran menos tiempo de ejecución cada uno de ellos.

- performSelector:withObject:afterDelay:
- performSelector:withObject:afterDelay:inModes:
- performSelectorOnMainThread:withObject:waitUntilDone:
- performSelectorOnMainThread:withObject:waitUntilDone:modes:
- performSelector:onThread:withObject:waitUntilDone:
- performSelector:onThread:withObject:waitUntilDone:modes:
- performSelectorInBackground:withObject:
+ cancelPreviousPerformRequestsWithTarget:
+ cancelPreviousPerformRequestsWithTarget:selector:object:

```
@property (nonatomic,assign) NSUInteger counter;


- (IBAction) findPrimes {

    while ( ! [self isPrime:++self.counter] );

    self.primeLabel.text =
                [NSString stringWithFormat:@"%u",self.counter];

    [self performSelector:@selector(findPrimes)
               withObject:self
               afterDelay:0];
}
```

Temporizadores

Temporizadores

- Son objetos que planifican la ejecución de un método en ciertos instantes de tiempo.
  - Típicamente para ejecuciones repetitivas.
- El método se ejecuta en el mismo thread del run loop donde se planificó.
  - Si es el main thread, la ejecución del método debe durar poco para no parar el main run loop.
    - Si el método requiere mucho tiempo de ejecución, conviene descomponerlo en varios métodos que requieran menos tiempo de ejecución cada uno de ellos.

NSTimer

- Crear un temporizador y planificarlo en el run loop:

```
NSTimer *timer = [NSTimer scheduledTimerWithTimeInterval:0.1
                                                  target:self
                                                  selector:@selector(findPrimes:)
                                                  userInfo:nil
                                                  repeats:YES];
```

  - Ejecuta el selector del objeto target cada 0.1 segundos.
  - El selector toma como parámetro el objeto **NSTimer** que lo llamó.

- Parar el temporizador:

```
[timer invalidate];
```

- Estado del temporizador:

```
[timer isValid];
```

```
@property (nonatomic,assign) NSUInteger counter;


-(IBAction) findPrimes {

    [NSTimer scheduledTimerWithTimeInterval:0.1
                                     target:self
                                   selector:@selector(findNextPrime:)
                                   userInfo:nil
                                    repeats:YES];
}

- (void) findNextPrime:(NSTimer*)timer {

   while ( ! [self isPrime:++self.counter]);

   self.primeLabel.text = [NSString stringWithFormat:@"%u",self.counter];
}
```

Concurrencia

Concurrencia

- El uso de temporizadores o la planificación de métodos en el main run loop:
  - Es sencillo de utilizar si tenemos pocas tareas concurrentes.
    - Aunque partir un método que requiere mucho tiempo de ejecución en métodos que hagan tareas más cortas puede ser complicado.
  - Pero si necesitamos ejecutar bastantes tareas simultáneamente, la complejidad del código puede dispararse.
  - No es concurrencia de verdad
    - Si la ejecución de los métodos dura mucho, el main run loop estará muy ocupado y la interface de usuario no responderá con agilidad.

UIKit no es Thread-safe

- Cuidado con la concurrencia:
  - La mayor parte del UIKit no es thread-safe.
    - por motivos de eficiencia.
  - Todo el trabajo de la interface de usuario debe hacerse en main thread.

Threads

Threads

- El sistema operativo proporciona pthreads (POSIX Threads API).

  `/usr/include/pthread.h`

- El framework Foundation proporciona el recubrimiento **`NSThread`**.
  - son funciones de C.
  - más fáciles de manejar que directamente los pthreads.

NSThread

- Para cada instancia de **`NSThread`**:
  - se crea automáticamente un run loop.
  - hay que crear manualmente un **`NSAutoreleasePool`**.
- Varias formas de crearlos:
  - Invocando:

```
[NSThread detachNewThreadSelector:
          toTarget:
          withObject:]
```

    - Su ejecución empieza inmediatamente.
  - Invocando:

```
[[NSThread alloc] initWithTarget:selector:object:]
```

    - Su ejecución empieza al llamar al método **`start`**.
  - También se puede crear una clase derivada de NSThread y sobreescribir el método **`main`**.

Creo el thread

```
- (IBAction) findPrimes {
    [NSThread detachNewThreadSelector:@selector(doFindPrimes)
                             toTarget:self
                           withObject:nil];
}

- (void) doFindPrimes {
   for (NSUInteger n=2 ;; n++) {
      @autoreleasepool {
         if ([self isPrime:n]) {
            [self performSelectorOnMainThread:@selector(updateLastPrime:)
                                   withObject:[NSNumber numberWithUnsignedInteger:n]
                                waitUntilDone:NO];
            [NSThread sleepForTimeInterval:0.001];
         }
      }
   }
}

- (void) updateLastPrime:(NSNumber*)n {
    self.primeLabel.text = [NSString stringWithFormat:@"%u",n.unsignedIntegerValue];
}
```

Hay que crear un AutoReleasePool

UIKit no es Thread-safe.
Lo envío al main thread.

Peligros

Peligros

- La programación concurrente es bastante complicada:
  - Condiciones de carrera, acceso simultáneo a secciones críticas y datos compartidos, bloqueos, inversión de prioridades, etc.
  - Obtendremos resultados y comportamientos erróneos.
  - Muy difícil de depurar.
  - El uso de muchos threads puede ralentizar un programa.

Cerrojos

- Control de acceso a secciones críticas y a datos compartidos:

  `NSLock`

  `@synchonized(objetolock)`

- Señalización entre threads:

  `NSCondition`

```
NSLock * myLock = [[NSLock alloc] init];

[myLock lock];
// zona protegida
[myLock unlock];



@synchronized(self) {
    // zona protegida
}
```

```
NSCondition * cond = [[NSCondition alloc] init];

// En el Thread Productor:
  [cond lock];
  // Meter al saco
  [cond signal];
  [cond unlock];


// En el Thread Consumidor:
  [cond lock];
  while (saco esta vacio)
    [cond wait];
  // Sacar del saco
  [cond unlock];
```

Bloqueos

- Ocurre cuando un thread está bloqueado esperando por una condición que nunca va a ocurrir.
- Ejemplo:
  - Un thread tiene el cerrojo A y espera por el cerrojo B.
  - Otro thread tiene el cerrojo B y espera por el cerrojo A.
- Recomendación:
  - no llamar a un bloque de código protegido con un cerrojo desde un bloque de código protegido con otro cerrojo.

Dormir un rato

- Si el programa tiene muchos threads consumiendo mucha cpu, el programa puede congestionarse.
  - La interface de usuario dejará de responder ágilmente.
  - Esto se agrava más en un terminal con pocos recursos.
- Solución: Podemos hacer que los threads se duerman un rato para favorecer a los demás threads.

```
[NSThread sleepForTimeInterval:0.1];
```

```
NSDate * d = [NSDate dateWithTimeIntervalSinceNow:0.1];
[NSThread sleepUntilDate:d];
```

- Muy importante: no dormir nunca el main thread.

Operations

Operation Queue y Operations

- Abstracción de alto nivel para no tener que usar threads.
- Una operación (**`NSOperation`**) es un objeto que representa una unidad de trabajo a realizar.
  - con dependencias y prioridades
- Las colas de operaciones (**`NSOperationQueue`**):
  - Ejecutan simultáneamente todas las operaciones que se añaden a la cola, pero respetando las restricciones que indiquemos (prioridades y dependencias)
  - Ejecutan las operaciones usando eficientemente los recursos disponibles.
    - Cuando queda un thread libre se saca una operación de la cola.

NSOperation

- Clase base de las operaciones:
  - Las operaciones que añadiremos a las operation queues son **subclases** de **`NSOperation`**.
- En las subclases tenemos que:
  - Sobreescribir el método **main** de la subclase.
    - Contiene las sentencias que ejecutará la operación.
  - El método **main** no puede lanzar excepciones
    - protegerlo con: `@try {} @catch(NSException*e) {}`
  - Crear un **AutoReleasePool** propio:
    - Threads diferentes no pueden compartir el mismo autorelease pool.

```
- (void) main {
    @try {
        @autoreleasepool {

            // las sentencias a ejecutar

        }
    }
    @catch (NSException * e) {
        NSLog(@"Excepcion: %@",e);
    }
}
```

```objc
#import <Foundation/Foundation.h>

@class PrimeOperation;


@protocol PrimeOperationDelegate

- (void) operationFoundNextPrime:(PrimeOperation*)po;

@end


@interface PrimeOperation : NSOperation

@property (nonatomic,weak) id<PrimeOperationDelegate> delegate;
@property (nonatomic,readonly,assign) NSUInteger prime;

- (id)initWithStartPrime:(NSUInteger)startPrime;

@end
```

PrimeOperation.h

```objc
#import "PrimeOperation.h"

@interface PrimeOperation ()
@property (nonatomic,assign) NSUInteger prime;
@end

@implementation PrimeOperation

- (id)initWithStartPrime:(NSUInteger)startPrime
{
    if (self = [super init]) {
        _prime = startPrime;
    }
    return self;
}

- (void) main {
   @try {
      @autoreleasepool {
         if ( ! self.isCancelled) {
            while ( ! [self isPrime:++self.prime]);

            if (self.delegate && [(NSObject *)self.delegate respondsToSelector:@selector(operationFoundNextPrime:)])

               [(NSObject *)self.delegate performSelectorOnMainThread:@selector(operationFoundNextPrime:)
                                                           withObject:self
                                                        waitUntilDone:NO];
            [NSThread sleepForTimeInterval:0.1];
         }
      }
   }
   @catch (NSException * e) {
      NSLog(@"Execepcion: %@",e);
   }
}

- (BOOL) isPrime:(NSUInteger)n { … }

@end
```

```objc
@property (nonatomic,strong) NSOperationQueue * queue;


- (void)viewDidLoad
{
    [super viewDidLoad];
    self.queue = [[NSOperationQueue alloc] init];
}

-(IBAction) findPrimes
{
    PrimeOperation * ope = [[PrimeOperation alloc] initWithStartPrime:1];
    ope.delegate = self;

    [self.queue addOperation:ope];
}

- (void) operationFoundNextPrime:(PrimeOperation*)po
{
    // Actualizar GUI
    self.primeLabel.text = [NSString stringWithFormat:@"%u",po.prime];

    // Lanzar otra operacion
    PrimeOperation * ope = [[PrimeOperation alloc] initWithStartPrime:po.prime];
    ope.delegate = self;
    [self.queue addOperation:ope];
}
```

NSInvocationOperation

- Es una **subclase** de **`NSOperation`** ya hecha.
- Permite especificar el objeto y el selector a usar al ejecutar la operación.

```
[[NSInvocationOperation alloc]
              initWithTarget:self
                    selector:@selector(metodo:)
                      object:arg];
```

```
@property (nonatomic,strong) NSOperationQueue * queue;

@property (nonatomic) NSUInteger lastPrime;

- (void)viewDidLoad
{
    [super viewDidLoad];
    self.queue = [[NSOperationQueue alloc] init];
}

-(IBAction) findPrimes
{
    NSInvocationOperation * ope = [[NSInvocationOperation alloc]
                                   initWithTarget:self
                                   selector:@selector(findNextPrime)
                                   object:nil];
    [self.queue addOperation:ope];
}

- (void) findNextPrime
{
    while ( ! [self isPrime:++self.lastPrime]);

    NSInvocationOperation * ope = [[NSInvocationOperation alloc]
                                   initWithTarget:self
                                   selector:@selector(updatePrime)
                                   object:nil];
    [[NSOperationQueue mainQueue] addOperation:ope];

}

- (void) updatePrime
{
    self.primeLabel.text = [NSString stringWithFormat:@"%d",self.lastPrime];

    NSInvocationOperation * ope = [[NSInvocationOperation alloc]
                                   initWithTarget:self
                                   selector:@selector(findNextPrime)
                                   object:nil];
    [self.queue addOperation:ope];
}
```

NSBlockOperation

- Es una subclase de **NSOperation** ya hecha.
- La operación ejecutará el block o los blocks dados.

```
+(id)blockOperationWithBlock:(void(^)(void))block
-(void)addExecutionBlock:(void(^)(void))block
```

```objc
@property (nonatomic,strong) NSOperationQueue * queue;

@property (nonatomic) NSUInteger lastPrime;

- (void)viewDidLoad
{
    [super viewDidLoad];
    self.queue = [[NSOperationQueue alloc] init];
}

-(IBAction) findPrimes
{
   [self createOperation];
}

- (void) createOperation
{
    // NSOperation para calcular el siguiente primo.
    NSBlockOperation * ope = [NSBlockOperation blockOperationWithBlock:^{

        while ( ! [self isPrime:++self.lastPrime]);

        // NSOperation para actualizar el GUI y lanzar una NSOperation nueva.
        // Se encola en la cola del Main Thread
        [[NSOperationQueue mainQueue] addOperation:[NSBlockOperation blockOperationWithBlock:^{

            self.primeLabel.text = [NSString stringWithFormat:@"%d",self.lastPrime];

            // Lanzar otra NSOperation
            [self createOperation];
        }]];
    }];

    [self.queue addOperation:ope];
}
```

Completion Block

- A un objeto NSOperation puede asignársele un completion block.
  - Se invoca cuando la operación ha terminado de ejecutarse.
    - Tanto si termina con éxito o es cancelada.
      - Consultar el valor de la propiedad **isCancelled**.

```
-(void)setCompletionBlock:(void(^)(void))block;

-(void(^)(void))completionBlock;
```

- El contexto de ejecución del bloque será algún thread secundario.
  - No es el Main Thread.

Dependencias entre Operaciones

- Podemos indicar que la operación B no puede ejecutarse hasta que no haya terminado la ejecución de la operación A.

```
[opB addDependency:opA];
```

- Consultar dependencias:

```
NSArray * dependencias = [opB dependencies];
```

- Eliminar dependencias:

```
[opB removeDependency:opA];
```

Prioridad de las Operaciones

- Las operaciones tienen una prioridad.
  - La usan las colas para decidir que operación ejecutar y cuanto tiempo se ejecutará.

```
[opA setQueuePriority:NSOperationQueuePriorityNormal];
```

- Valores:

```
NSOperationQueuePriorityVeryLow
NSOperationQueuePriorityLow
NSOperationQueuePriorityNormal
NSOperationQueuePriorityHigh
NSOperationQueuePriorityVeryHigh
```

- Consultar la prioridad:

```
NSOperationQueuePriority * pri = [opA queuePriority];
```

Cancelar una Operación

- La ejecución de una operación puede cancelarse:

```
[opA cancel];
```

- Cancelar una operación sólo hace que la propiedad **isCancelled** valga **YES**.
- El método **main** debe consultar esta propiedad con frecuencia, y terminar si vale **YES**.
- Cancelar una operación pendiente en la cola no la saca de esta.
  - Hay que esperar hasta que empiece a ejecutarse. El método **main** debe comprobar inicialmente si la operación fue cancelada, y si es así, terminar. Las operaciones sólo se eliminan de la cola cuando termina su ejecución.

Propiedad de las Operaciones

- **`isReady`**
  - YES si la operación está lista para ejecutarse, es decir, no hay dependencias por las que esperar.
- **`isCancelled`**
  - YES si la operación fue cancelada.
- **`isExecuting`**
  - YES si la operación es está ejecutando (ha empezado y no ha acabado su ejecución).
- **`isFinished`**
  - YES si terminó la ejecución de la operación.
- **`isConcurrent`**
  - Sobreescribir para devolver YES si la operación creará su propio thread para ejecutarse.
  - Hay que sobreescribir también otros métodos: **start**(), **isExecuting**, **isFinished**.

NSOperation: Más Métodos

```
-(void)setCompletionBlock:(void(^)(void))block

-(void)waitUntilFinished

. . .
```

NSOperationQueue

- Son los objetos que gestionan la ejecución de las operaciones.
- Para crear una cola:

```
NSOperationQueue * queue = [[NSOperationQueue alloc] init];
```

- Para añadir una operación a la cola:

```
[queue addOperation:opA];
```

  - La operación se ejecutará cuanto exista un thread disponible y la operación esté lista (isReady indicará que no hay dependencias pendientes).

- Por defecto, las colas deciden cuantos threads usarán dependiendo del hardware disponible.
  - Podemos modificar el número de threads a usar con el método:

```
setMaxConcurrentOperationCount:
```

    - Tendremos una cola serie si sólo usamos un thread.

- Una cola puede suspenderse para que no ejecute ninguna operación más.

```
[queue setSuspended:YES];
```

NSOperationQueue: Más Métodos

```
-(void)addOperations:(NSArray *)ops
    waitUntilFinished:(BOOL)wait

-(void)addOperationWithBlock:(void(^)(void))block

-(void)waitUntilAllOperationsAreFinished

-(void)cancelAllOperations

+(id)currentQueue
+(id)mainQueue

. . .
```

Grand Central Dispatch

GCD

- GCD es un **API C** para manejar el multithreading.
  - Nos oculta los detalles de la multiprogramación.
  - No tenemos que preocuparnos de los recursos disponibles. GCD se encarga de gestionarlos.
  - Se programa muy fácilmente.
  - Desde iOS 7 los objetos del GCD son objetos Objective-C.
- GCD nos proporciona colas a las que enviaremos tareas.
  - Una tarea es un objeto block.
    - También pueden usarse punteros a funciones.
  - En algún momento GCD asignará un thread libre a esa tarea para que se ejecute.
  - El encolado es **thread-safe**.
- Hay tres tipos de colas:
  - **main**: ejecuta tareas en serie en el main thread.
  - **concurrente**: el comienzo de la ejecución de las tareas es FIFO, y pueden ejecutarse concurrentemente.
  - **serie**: las tareas se ejecutan de una en una en modo FIFO.

Cola: **main**

- Ejecuta tareas en **serie** en el **Main Thread**.
- Esta cola la crea el sistema automáticamente.
- Para obtener esta cola:

```
dispatch_queue_t dispatch_get_main_queue(void);
```

Cola: **Concurrente**

- Para ejecutar muchas tareas **concurrentemente**.
  - El comienzo de la ejecución de las tareas es FIFO.
- GCD crea automáticamente tres colas con distintas propiedades.
  - No hay que crearlas, ni retenerlas, ni liberarlas.
- Para obtener estas colas:

```
dispatch_queue_t dispatch_get_global_queue(long priority,
                                           unsigned long flags);
```

  - Valores para **prioridad**:

    `DISPATCH_QUEUE_PRIORITY_HIGH`
    `DISPATCH_QUEUE_PRIORITY_DEFAULT`
    `DISPATCH_QUEUE_PRIORITY_LOW`
  - **flags**: es 0 (reservado para su uso en el futuro).

Cola: **serie** (o **concurrente**)

- Pueden crearse varias colas serie (o concurrente).
  - Las aplicaciones deben crearlas explícitamente y gestionarlas.

```
dispatch_queue_t dispatch_queue_create(
        const char *label, dispatch_queue_attr_t attr);
```

- Parámetros:
  - **label**: es un string de C. Identifica la cola para ayuda en la depuración.
  - Valores de **attr**:

```
DISPATCH_QUEUE_SERIAL (o NULL) // para crear una cola serie.
DISPATCH_QUEUE_CONCURRENT      // para crear una cola concurrente.
```

- Si la cola creada es serie, ejecuta sus tareas de una en una y en orden FIFO. Pueden terminar en cualquier orden.
  - Si una operación se bloquea, sólo se bloquea su cola. Las demás colas continúan ejecutando sus tareas.
- Si la cola creada es concurrente, los bloques se desencolan en orden FIFO, y se ejecutan concurrentemente (si hay recursos disponibles para ello).
- Usos: sacar del main thread tareas que pueden bloquearse, realizar tareas muy largas en otro thread, proteger zonas críticas, …

Encolar Tareas

- Enviar una tarea a una cola:

```
void dispatch_async(dispatch_queue_t queue,
                    dispatch_block_t block);

void dispatch_sync(dispatch_queue_t queue,
                   dispatch_block_t block);
```

  - donde:

```
typedef void (^dispatch_block_t)(void);
```

  - **dispatch_async** no es bloqueante.
  - **dispatch_sync** es bloqueante. Se espera hasta que el bloque ha terminado

- Importante: GCD es un API C y no entiende de excepciones. Los block deben capturar todas las excepciones producidas antes de terminar su ejecución.

API

- `dispatch_after`
- `dispatch_apply`
- `dispatch_once`
- `dispatch_resume`
- `dispatch_suspend`
- `dispatch_time`
- `dispatch_set_context`
- `dispatch_get_context`
- `dispatch_group_create`
- `dispatch_group_enter`
- `dispatch_group_leave`
- `dispatch_group_async`
- `dispatch_group_wait`
- `dispatch_semaphore_create`
- `dispatch_semaphore_signal`
- `dispatch_semaphore_wait`
- `dispatch_source_*`
- etc. . .

Ejemplo: Cola Global

```
-(IBAction) findPrimes
{
    [self globalPrimes];
}

- (void) globalPrimes {

    dispatch_queue_t global_queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_LOW,0);

    dispatch_async(global_queue, ^{
        for (NSUInteger n=2 ;; n++) {
            if ([self isPrime:n]) {

                // Actualiza GUI
                dispatch_async(dispatch_get_main_queue(), ^{
                    self.primeLabel.text = [NSString stringWithFormat:@"%u",n];
                });

                // Dormir para no comerse toda la CPU.
                [NSThread sleepForTimeInterval:0.1];
            }
        }
    });
}
```

Bucle infinito calculando primos

Actualizo el GUI en el main thread

Ejemplo: Cola Serie

```
@property (strong,nonatomic) dispatch_queue_t queue;

- (void)viewDidLoad
{
    [super viewDidLoad];
    self.queue = dispatch_queue_create("Cola Serie", DISPATCH_QUEUE_SERIAL);
}

-(IBAction) findPrimes
{
    [self nextPrimeAfterValue:1];
}

- (void) nextPrimeAfterValue:(NSUInteger)startValue
{
    dispatch_async(self.queue, ^{

        NSUInteger counter = startValue;

        while ( ! [self isPrime:++counter]);

        // Actualiza GUI y crea otra tarea.
        dispatch_async(dispatch_get_main_queue(), ^{
            self.primeLabel.text = [NSString stringWithFormat:@"%u",counter];

            [self nextPrimeAfterValue:counter];
        });
    });
}
```

Crear cola serie

Actualizo el GUI en el main thread

Crear otra tarea

Ejemplo: Bajarse una Foto

```
NSString *surl = @"http://www.dit.upm.es/figures/logos/dit08.gif";

surl = [surl stringByAddingPercentEscapesUsingEncoding:NSUTF8StringEncoding];

NSURL * url = [NSURL URLWithString:surl];

dispatch_queue_t queue = dispatch_queue_create("Download queue", NULL);

dispatch_async(queue, ^{

   NSData *data = [NSData dataWithContentsOfURL:url];

   dispatch_async(dispatch_get_main_queue(), ^{
      UIImage *image = [UIImage imageWithData:data];
      self.fotoImageView.image = image;
   });
});
```

Ejemplo: Zona Crítica

```
@property (strong,nonatomic) dispatch_queue_t queue;

self.queue = dispatch_queue_create("Cola Acceso", DISPATCH_QUEUE_SERIAL);

- (void) meter:(NSInteger)n {
    dispatch_async(self.queue, ^{

        self.total += n;

        dispatch_async(dispatch_get_main_queue(), ^{
            self.totalLabel.text = [NSString stringWithFormat:@"%d",self.total];
        });
    });
}

- (void) sacar:(NSInteger)n {
    dispatch_async(self.queue, ^{

        self.total -= n;

        dispatch_async(dispatch_get_main_queue(), ^{
            self.totalLabel.text = [NSString stringWithFormat:@"%d",self.total];
        });
    });
}
```

Cola para serializar tareas

Así es Thread Safe

Crear cola serie

Zona crítica

Actualizo el GUI en el main thread

Así es Thread Safe

Zona crítica

Actualizo el GUI en el main thread

Recomendaciones

Recomendaciones

- En general el uso de threads no se recomienda por su dificultad.
  - Difícil determinar cuantos threads usar dependiendo de los recursos disponibles, y gestionarlos.
    - O ajustar su número dinámicamente según la carga actual del sistema.
  - Difícil conseguir que se ejecuten eficientemente.
  - Difícil sincronización entre threads.
- Pero también hay casos en los que el uso de threads es mejor.
- Las tecnologías recomendadas para implementar concurrencia en las aplicaciones son:

  `Grand Central Dispatch`

  `Operation Queues`

  - No nos preocupamos de la creación y gestión de los threads.
  - La idea es definir tareas concretas a realizar, y dejar que el sistema las realice.