iOS Multiple Thread & GCD

Threading Programming Guide
[https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/Multithreading/CreatingThreads/CreatingThreads.html]

Concurrency Programming Guide
[https://developer.apple.com/library/content/documentation/General/Conceptual/ConcurrencyProgrammingGuide/Introduction/Introduction.html#//apple_ref/doc/uid/TP40008091]

- Organization of this Document
    - Concurrency & application design
    - Operation Queues
    - Dispatch Queues
    - Dispatch Sources
    - Migrating away from threads

---

1. Concurrency & application design
    1. Asynchronous design approach
    2. GCD (Grand Central Dispatch) —
    3. Operation queues

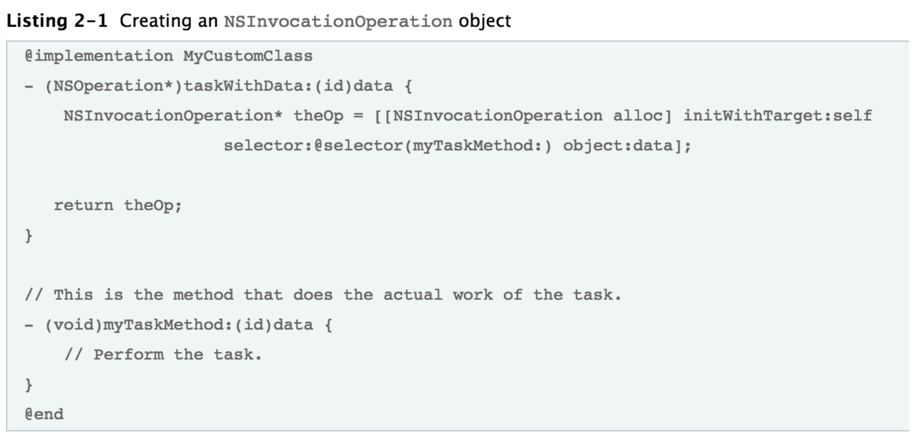
    Move away from threads

        1. Dispatch Queues: C-based, execute tasks serially/concurrently but always first-in, first-out order.
        2. Dispatch Sources: could monitor timer/ signal handlers/ descriptor-related events/ process-related events/ mach port events/ custom events.
        3. Operation queues: implement by *NSOperationQueue* sdks

    Asynchronous Design Tech

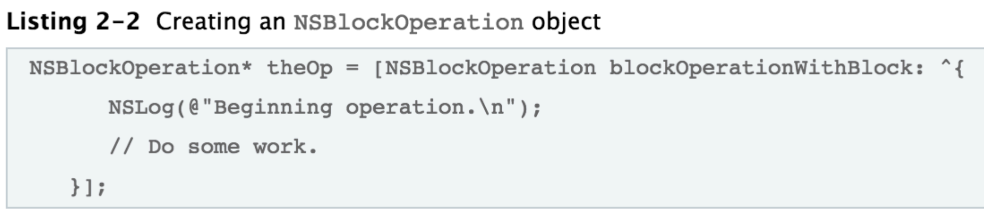
---

2. Operation Queues
    1. NSOperation class is abstract base class must be subclassed.
        1. NSInvocationOperation
            1. Create an NSInvocationOperation object


Listing 2-1 Creating an `NSInvocationOperation` object

```
@implementation MyCustomClass
- (NSOperation*)taskWithData:(id)data {
    NSInvocationOperation* theOp = [[NSInvocationOperation alloc] initWithTarget:self
                    selector:@selector(myTaskMethod:) object:data];

   return theOp;
}

// This is the method that does the actual work of the task.
- (void)myTaskMethod:(id)data {
    // Perform the task.
}
@end
```

2. NSBlockOperation — execute 1 or more block objects concurrently. (iOS 10.6 or later)


Listing 2-2 Creating an `NSBlockOperation` object

```
NSBlockOperation* theOp = [NSBlockOperation blockOperationWithBlock: ^{
      NSLog(@"Beginning operation.\n");
      // Do some work.
   }];
```

init(block: @escaping() -> Void)
Add more blocks using *addExecutionBlock*: function method. // no parameters and no return value
var executionBlocks
https://eezytutorials.com/ios/nsblockoperation-by-example.php#.WZ-oKJOGOxI

```
NSBlockOperation    *eezyBlockOperation = [NSBlockOpera
tion    blockOperationWithBlock:^{
    NSInteger    result =0 ;
    for     (NSInteger    i=0; i<10000; i++) {
        result = i;
    }
    NSLog(@"Your operation block %d",result);
 }];
NSLog(@"Before add block ");
NSOperationQueue    *operationQueue = [[NSOperationQueu
e    alloc]init];
[operationQueue addOperation:eezyBlockOperation];
NSLog(@"After add block");
  2014-04-04 08:24:27.155 iOS-Tutorial[503:a0b] Before
add block
  2014-04-04 08:24:27.156 iOS-Tutorial[503:a0b] After a
dd block
  2014-04-04 08:24:27.156 iOS-Tutorial[503:1403] Your o
```

```
peration block 9999
```

Example

```
NSBlockOperation *eezyBlockOperation = [NSBlockOperation bl
    NSInteger result =0 ;
    for (NSInteger i=0; i<10000; i++) {
        result = i;
    }
    NSLog(@"Your operation block %d",result);
 }];
[eezyBlockOperation addExecutionBlock:^{
    NSLog(@"Added execution block ");
}];
NSLog(@"Before add block ");
NSOperationQueue *operationQueue = [[NSOperationQueue alloc
[operationQueue addOperation:eezyBlockOperation];
NSLog(@"After add block");
```

Output

```
8:32:31.619 iOS-Tutorial[540:a0b] Before addd block
8:32:31.620 iOS-Tutorial[540:a0b] After add block
8:32:31.620 iOS-Tutorial[540:1403] Your operation block 9999
8:32:31.620 iOS-Tutorial[540:3703] Added execution block
```

Example

```
NSBlockOperation *eezyBlockOperation = [NSBlockOperatio
    NSInteger result =0 ;
    for (NSInteger i=0; i<10000; i++) {
        result = i;
    }
    NSLog(@"Your operation block %d",result);
}];
[eezyBlockOperation addExecutionBlock:^{
    NSLog(@"Added execution block ");
}];
NSLog(@"%@",[eezyBlockOperation executionBlocks]);

```

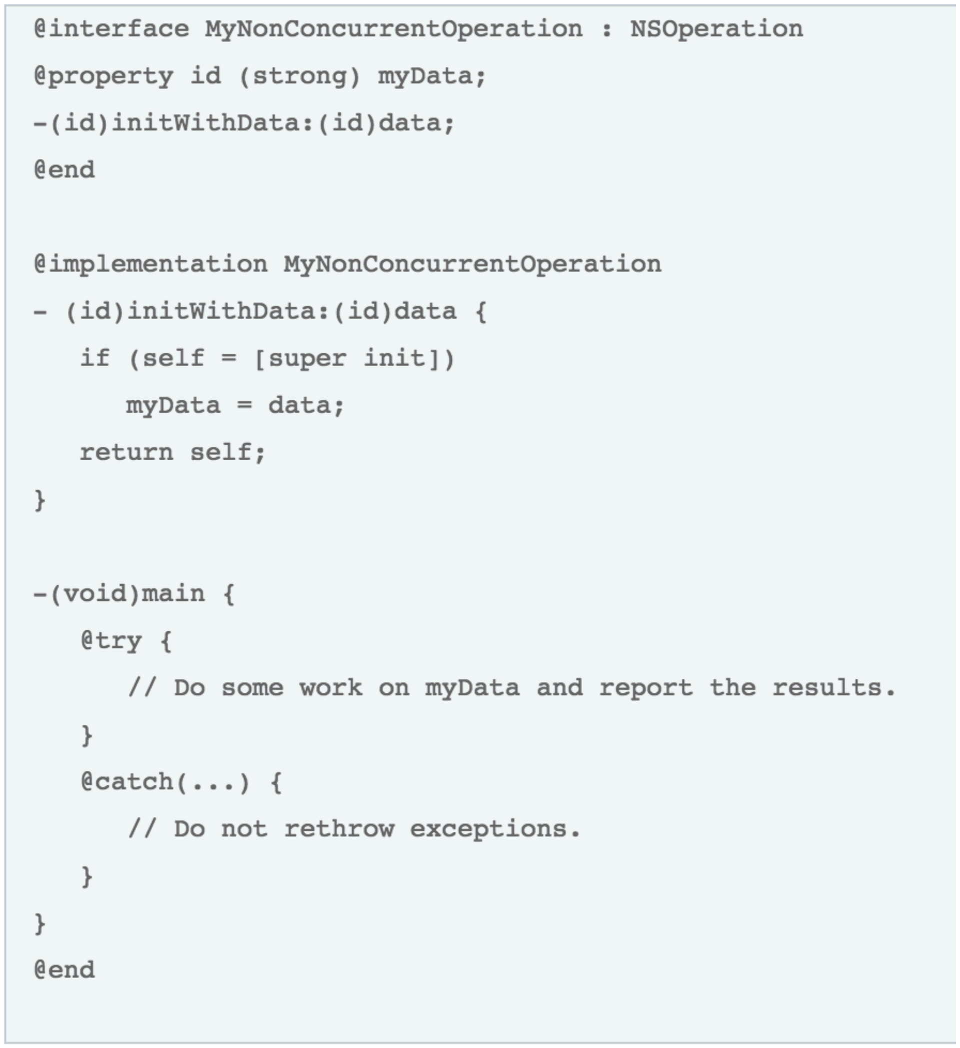
Output

```
2014-04-04 08:35:45.239 iOS-Tutorial[570:a0b] (
    "<__NSGlobalBlock__: 0x4080>",
    "<__NSGlobalBlock__: 0x40a8>"
)

```

3. NSOperation (Customized Operation)

- isCancelled
- isConcurrent
- isExecuting
- isFinished
- isReady
- dependencies
- queuePriority
- completionBlock

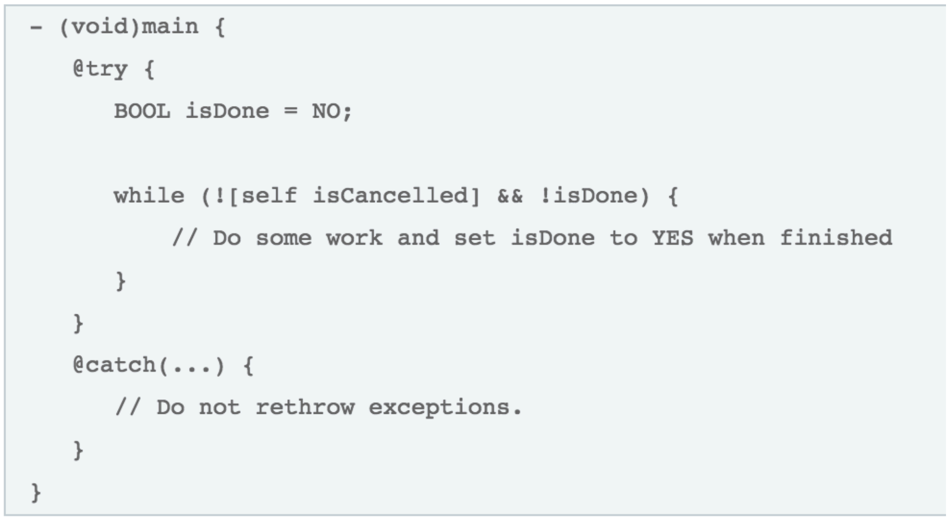
Performing the main task


Listing 2-3 Defining a simple operation object

```
@interface MyNonConcurrentOperation : NSOperation
@property id (strong) myData;
-(id)initWithData:(id)data;
@end

@implementation MyNonConcurrentOperation
- (id)initWithData:(id)data {
   if (self = [super init])
      myData = data;
   return self;
}

-(void)main {
   @try {
      // Do some work on myData and report the results.
   }
   @catch(...) {
      // Do not rethrow exceptions.
   }
}
@end
```

responding to cancellation events

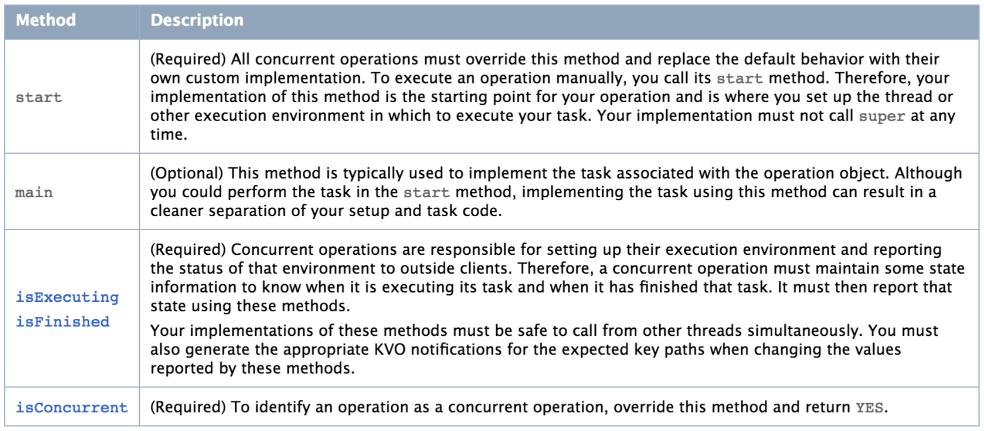
Listing 2-4 Responding to a cancellation request



```
- (void)main {
   @try {
      BOOL isDone = NO;

      while (![self isCancelled] && !isDone) {
          // Do some work and set isDone to YES when finished
      }
   }
   @catch(...) {
      // Do not rethrow exceptions.
   }
}
```

Configuring operations for concurrent execution
1. Operation objects execute in a synchronous manner by default.
2. run asynchronous override function:

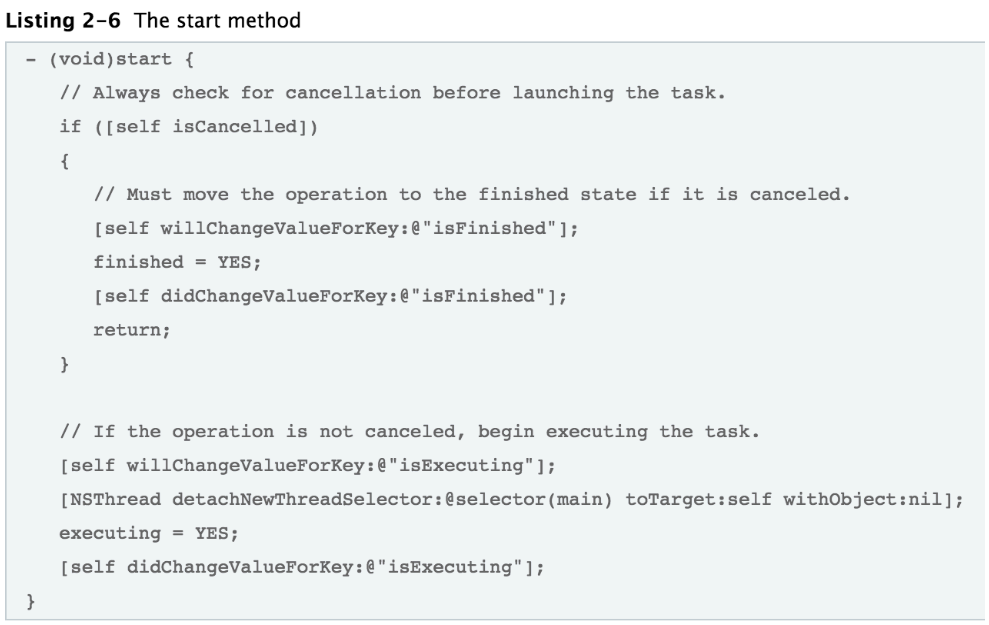
Table 2-2 Methods to override for concurrent operations



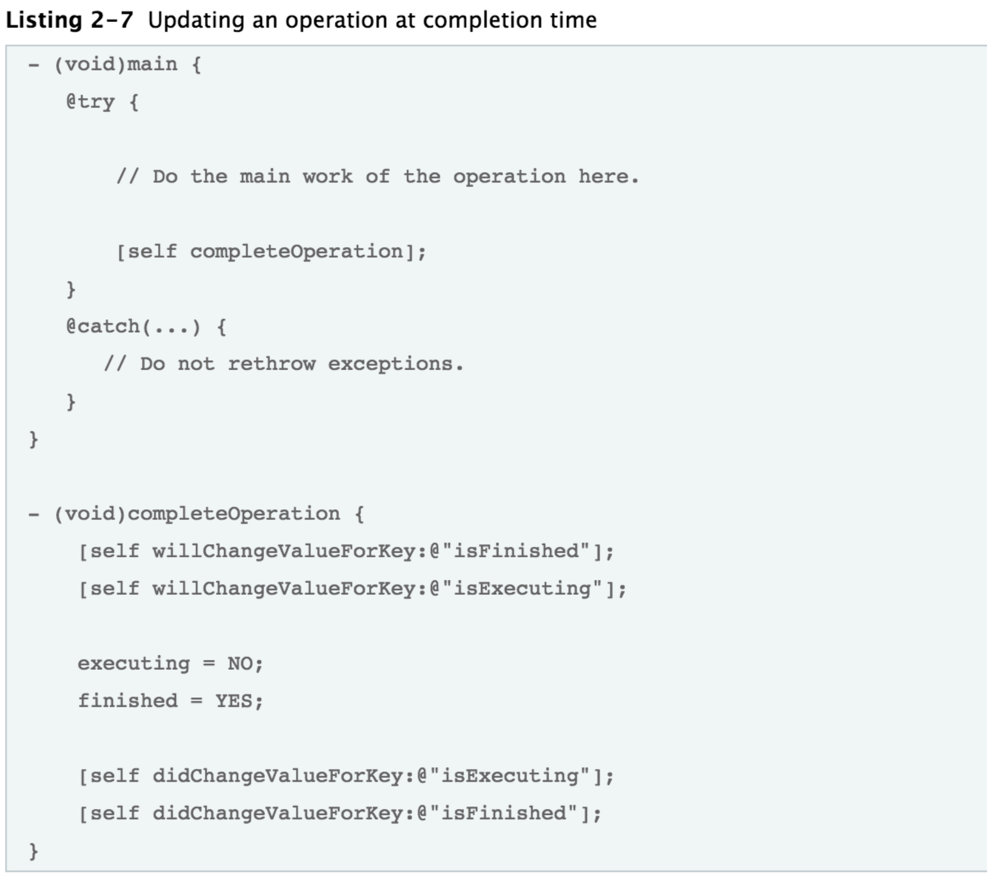
Method	Description
`start`	(Required) All concurrent operations must override this method and replace the default behavior with their own custom implementation. To execute an operation manually, you call its `start` method. Therefore, your implementation of this method is the starting point for your operation and is where you set up the thread or other execution environment in which to execute your task. Your implementation must not call `super` at any time.
`main`	(Optional) This method is typically used to implement the task associated with the operation object. Although you could perform the task in the `start` method, implementing the task using this method can result in a cleaner separation of your setup and task code.
`isExecuting` `isFinished`	(Required) Concurrent operations are responsible for setting up their execution environment and reporting the status of that environment to outside clients. Therefore, a concurrent operation must maintain some state information to know when it is executing its task and when it has finished that task. It must then report that state using these methods. Your implementations of these methods must be safe to call from other threads simultaneously. You must also generate the appropriate KVO notifications for the expected key paths when changing the values reported by these methods.
`isConcurrent`	(Required) To identify an operation as a concurrent operation, override this method and return `YES`.

Listing 2-5 Defining a concurrent operation

```
@interface MyOperation : NSOperation {
    BOOL        executing;
    BOOL        finished;
}
- (void)completeOperation;
@end

@implementation MyOperation
- (id)init {
    self = [super init];
    if (self) {
        executing = NO;
        finished = NO;
    }
    return self;
}

- (BOOL)isConcurrent {
    return YES;
}

- (BOOL)isExecuting {
    return executing;
}

- (BOOL)isFinished {
    return finished;
}
@end
```


Listing 2-6 The start method

```
- (void)start {
   // Always check for cancellation before launching the task.
   if ([self isCancelled])
   {
      // Must move the operation to the finished state if it is canceled.
      [self willChangeValueForKey:@"isFinished"];
      finished = YES;
      [self didChangeValueForKey:@"isFinished"];
      return;
   }

   // If the operation is not canceled, begin executing the task.
   [self willChangeValueForKey:@"isExecuting"];
   [NSThread detachNewThreadSelector:@selector(main) toTarget:self withObject:nil];
   executing = YES;
   [self didChangeValueForKey:@"isExecuting"];
}
```


Listing 2-7 Updating an operation at completion time

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

       // Do the main work of the operation here.

       [self completeOperation];
   }
   @catch(...) {
      // Do not rethrow exceptions.
   }
}

- (void)completeOperation {
    [self willChangeValueForKey:@"isFinished"];
    [self willChangeValueForKey:@"isExecuting"];

    executing = NO;
    finished = YES;

    [self didChangeValueForKey:@"isExecuting"];
    [self didChangeValueForKey:@"isFinished"];
}
```

Above is basic customize operation, if implement support dependencies on something besides other operation objects, override `isReady` method and force it return NO until custom dependencies were satisfied. (call super if still support default dependency management system)

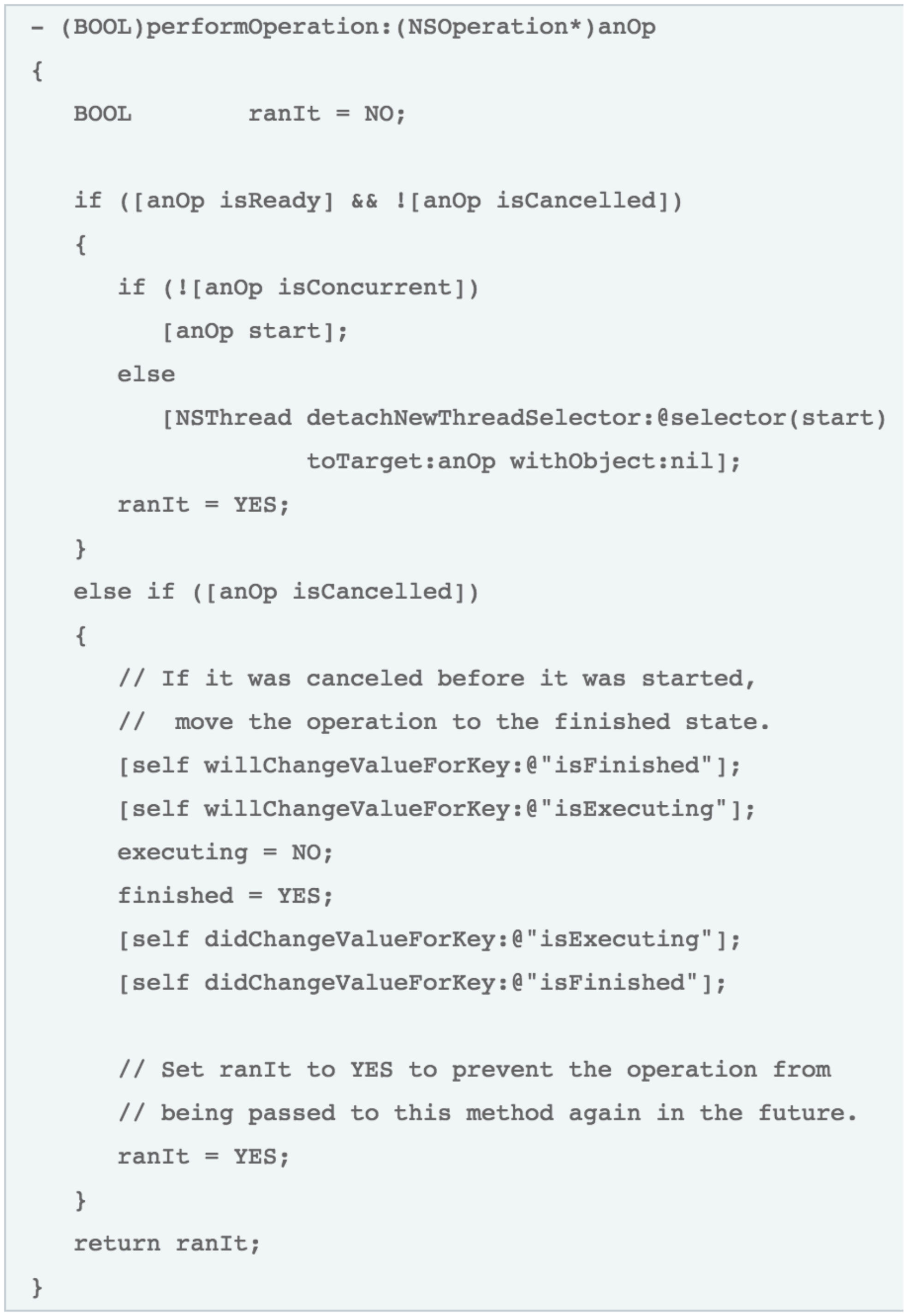
- dependencies if you override `*addDependency*:` or `*removeDependency*:`

4. Customizing execution behavior of an operation object
   - configuring interoperation dependencies
     (*addDependency*) cannot begin executing until target object finishes
   Should always configure dependencies before running your operations/ adding them to an operation queue.
   - changing operation's execution priority. Only works in same operation queue.
     default - normal, but could call setQueuePriority:
    NSOperationQueuePriorityNormal/VeryLow/VeryHigh
   - changing underlying thread priority (OS X 10.6 and later)
     thread priority from 0.0 to 1.0. default 0.5. *setThreadPriority*:
     if create a concurrent operation, override start method, and also must configure thread priority by self.
   - setting up a completion block (OS X 10.6 and later)

5. Executing Operations
   1. use OperationQueue
   ```
   NSOperationQueue* aQueue = [[NSOperationQueue alloc] init];
    [aQueue addOperation: anOp];
    [aQueue addOperations: anArrayOfOps waitUntilFinished: NO];
   [aQueue addOperationWithBlock:^{ … }];
   ```
   before add to queue, set configuration and modification to operation object.
   Could *setMaxConcurrentOperationCount*:
   difference with GCD: serial Operation Queue not same as serial dispatch Queue in GCD, since its order execution based on readiness of operation and priority.

   2. without a queue, and using its start method.

Listing 2-8 Executing an operation object manually



```
- (BOOL)performOperation:(NSOperation*)anOp
{
   BOOL        ranIt = NO;

   if ([anOp isReady] && ![anOp isCancelled])
   {
      if (![anOp isConcurrent])
         [anOp start];
      else
         [NSThread detachNewThreadSelector:@selector(start)
                   toTarget:anOp withObject:nil];
      ranIt = YES;
   }
   else if ([anOp isCancelled])
   {
      // If it was canceled before it was started,
      //  move the operation to the finished state.
      [self willChangeValueForKey:@"isFinished"];
      [self willChangeValueForKey:@"isExecuting"];
      executing = NO;
      finished = YES;
      [self didChangeValueForKey:@"isExecuting"];
      [self didChangeValueForKey:@"isFinished"];

      // Set ranIt to YES to prevent the operation from
      // being passed to this method again in the future.
      ranIt = YES;
   }
   return ranIt;
}
```

6. Canceling Operations
   If in an operation queue, only way to dequeue is to cancel it. `cancel` or `cancelAllOperations`

        7. `WaitiUnitilFinished` method from NSOperation.
           never wait for an operation from app main thread. and introduce more serialization into your app.
           `waitUntilAllOperationsAreFinished` method of NSOperationQueue
        8. Suspend and resuming Queues.
           `setSuspended:` method for a queue does not cause already executing operations to pause in mid of their tasks.
        9.

3. Dispatch Queue
    1. simpler to use and more efficient at executing tasks than corresponding threaded code.
    2. all dispatch queues are FIFO data structure.
    3. Types of dispatch queue:

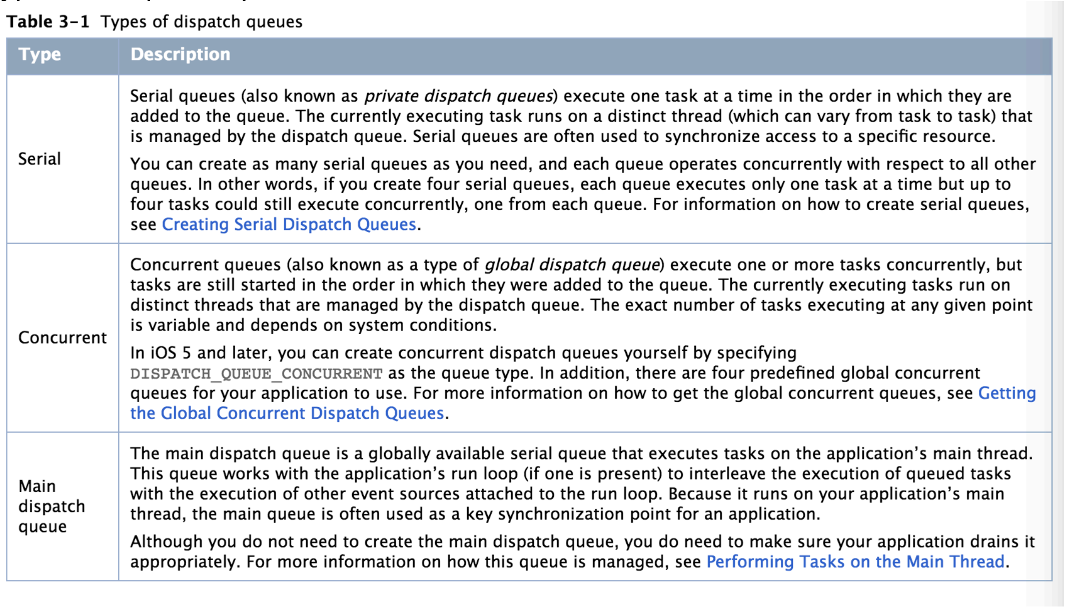


Table 3-1 Types of dispatch queues

Type	Description
Serial	Serial queues (also known as *private dispatch queues*) execute one task at a time in the order in which they are added to the queue. The currently executing task runs on a distinct thread (which can vary from task to task) that is managed by the dispatch queue. Serial queues are often used to synchronize access to a specific resource. You can create as many serial queues as you need, and each queue operates concurrently with respect to all other queues. In other words, if you create four serial queues, each queue executes only one task at a time but up to four tasks could still execute concurrently, one from each queue. For information on how to create serial queues, see Creating Serial Dispatch Queues.
Concurrent	Concurrent queues (also known as a type of *global dispatch queue*) execute one or more tasks concurrently, but tasks are still started in the order in which they were added to the queue. The currently executing tasks run on distinct threads that are managed by the dispatch queue. The exact number of tasks executing at any given point is variable and depends on system conditions. In iOS 5 and later, you can create concurrent dispatch queues yourself by specifying `DISPATCH_QUEUE_CONCURRENT` as the queue type. In addition, there are four predefined global concurrent queues for your application to use. For more information on how to get the global concurrent queues, see Getting the Global Concurrent Dispatch Queues.
Main dispatch queue	The main dispatch queue is a globally available serial queue that executes tasks on the application's main thread. This queue works with the application's run loop (if one is present) to interleave the execution of queued tasks with the execution of other event sources attached to the run loop. Because it runs on your application's main thread, the main queue is often used as a key synchronization point for an application. Although you do not need to create the main dispatch queue, you do need to make sure your application drains it appropriately. For more information on how this queue is managed, see Performing Tasks on the Main Thread.

    4. dispatch queue advantages over threads:
       1. simplicity of the work-queue programming model.
          Since thread, you need code both of creation and management of the threads.
       2. predictability in dispatch queues.
          e.g.: 2 tasks access same shared resource but run on different threads,
                both could modify resource, then need lock ensure both tasks not modify same time.
                In this case, use serial dispatch queue ensure only one task modify resource at any given time.
          this type of queue-based synchronization more efficient than lock since locks always require an expensive kernel trap in both contested and uncontested cases, whereas dispatch queue works primarily in app's process space and only calls down to kernel when absolutely necessary.

pay less memory than threads
3. system decide total number of task executing at any one time.
4. system decide queue priority levels into account when choosing new tasks to start. [Providing a Clean Up Function For a Queue](#).
5. tasks in queue must be ready to execute at the time they added to the queue. diff from operation
6. private dispatch queues are Reference-counted objects.[Memory Management for Dispatch Queues](#).

5. queue-related technologies:

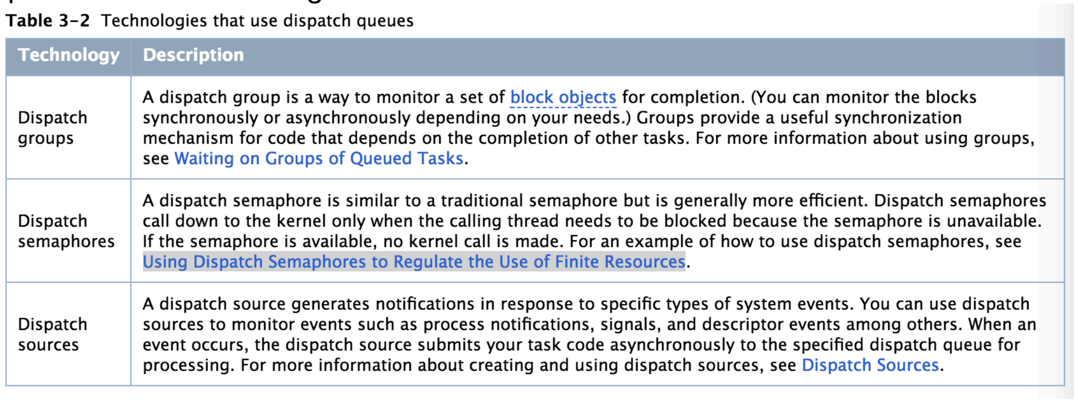
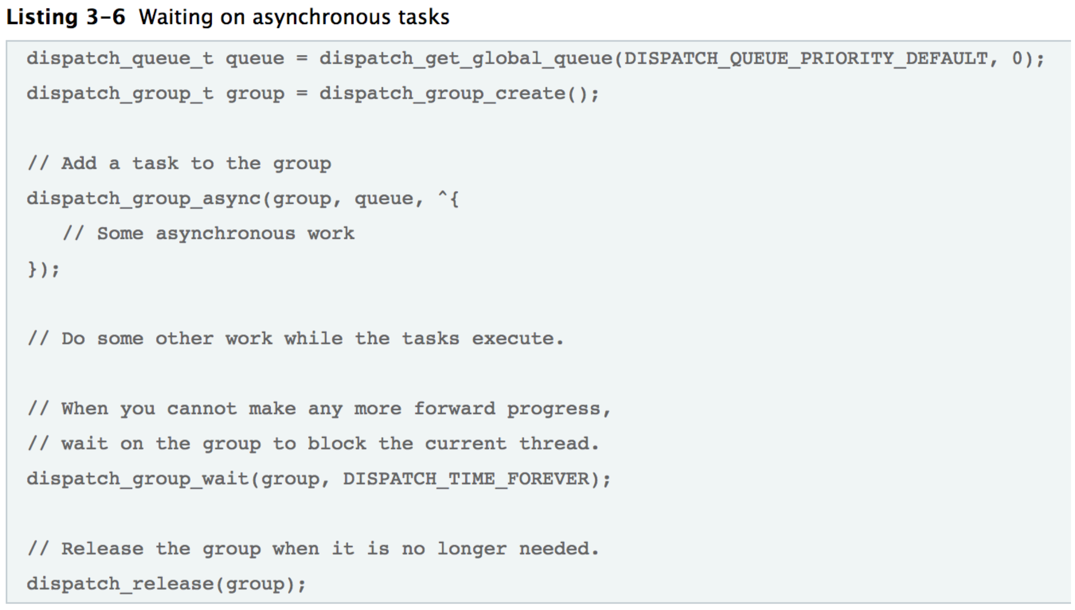


Table 3-2 Technologies that use dispatch queues

Technology	Description
Dispatch groups	A dispatch group is a way to monitor a set of block objects for completion. (You can monitor the blocks synchronously or asynchronously depending on your needs.) Groups provide a useful synchronization mechanism for code that depends on the completion of other tasks. For more information about using groups, see Waiting on Groups of Queued Tasks.
Dispatch semaphores	A dispatch semaphore is similar to a traditional semaphore but is generally more efficient. Dispatch semaphores call down to the kernel only when the calling thread needs to be blocked because the semaphore is unavailable. If the semaphore is available, no kernel call is made. For an example of how to use dispatch semaphores, see Using Dispatch Semaphores to Regulate the Use of Finite Resources.
Dispatch sources	A dispatch source generates notifications in response to specific types of system events. You can use dispatch sources to monitor events such as process notifications, signals, and descriptor events among others. When an event occurs, the dispatch source submits your task code asynchronously to the specified dispatch queue for processing. For more information about creating and using dispatch sources, see Dispatch Sources.

- dispatch_group
  block a thread until several tasks finish executing

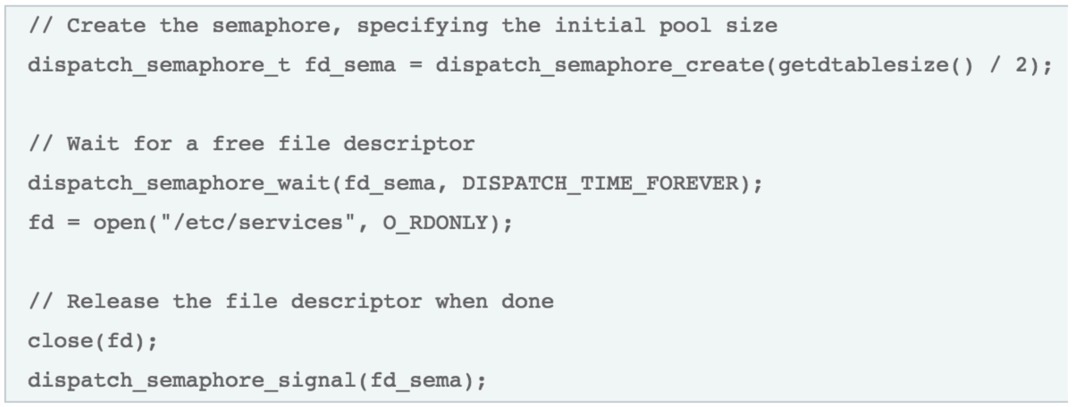


Listing 3-6 Waiting on asynchronous tasks

```
dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
dispatch_group_t group = dispatch_group_create();

// Add a task to the group
dispatch_group_async(group, queue, ^{
   // Some asynchronous work
});

// Do some other work while the tasks execute.

// When you cannot make any more forward progress,
// wait on the group to block the current thread.
dispatch_group_wait(group, DISPATCH_TIME_FOREVER);

// Release the group when it is no longer needed.
dispatch_release(group);
```

Dispatch queues & thread safe:
- thread safe, submit tasks to dispatch queue from any thread on system without first taking a lock/synchronize
- not call dispatch_sync from a task that is executing on same queue that you pass to your function call.

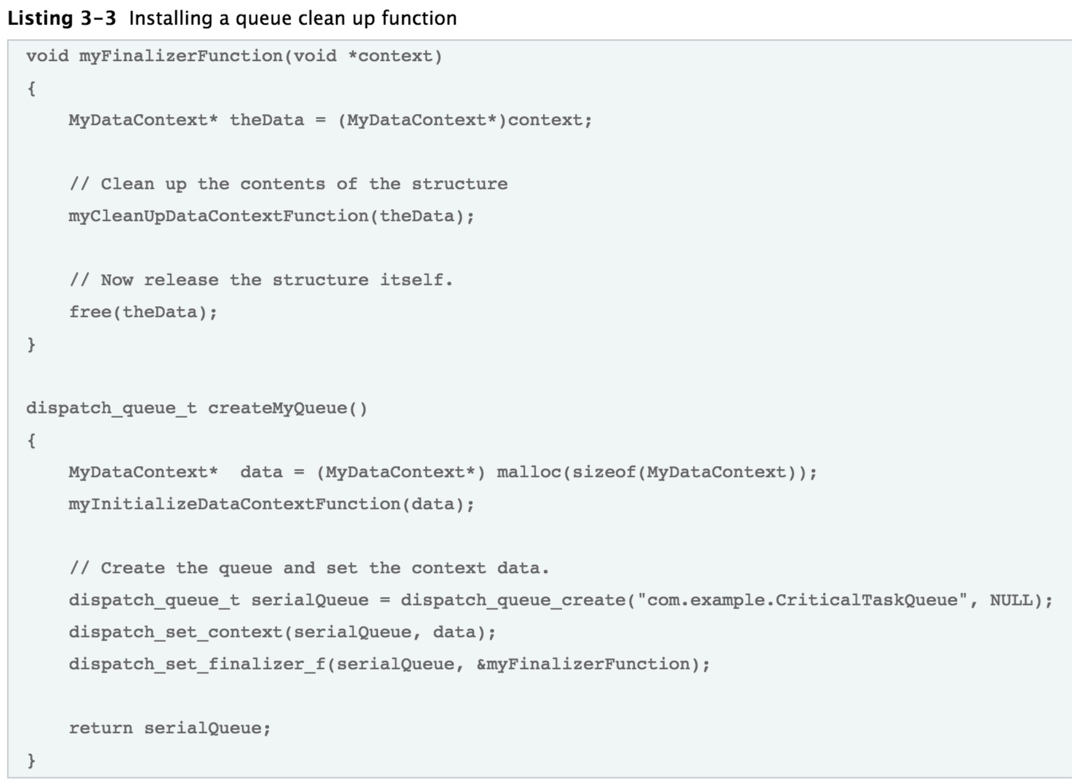
- dispatch_semaphores: GCD do not need to call down into kernel if resources are available
    dispatch_semaphore_create, *dispatch_semaphore_wait*, *dispatch_semaphore_signal*



```
// Create the semaphore, specifying the initial pool size
dispatch_semaphore_t fd_sema = dispatch_semaphore_create(getdtablesize() / 2);

// Wait for a free file descriptor
dispatch_semaphore_wait(fd_sema, DISPATCH_TIME_FOREVER);
fd = open("/etc/services", O_RDONLY);

// Release the file descriptor when done
close(fd);
dispatch_semaphore_signal(fd_sema);
```

- dispatch_sources

6. Implementing Tasks Using Blocks
   block using compiler time encapsulates it living in heap and pass around whole app.
   - advantages: use variables from outside their lexical scope with *__block*.
   - about overhead to creating blocks and executing them on a queue.
   - share data with diff tasks in the same queue is to use context pointer of dispatch queue.see [Storing Custom Context Information with a Queue](#).
       - dispatch_set_context, dispatch_get_context functions.
   dispatch_set_finalizer_f to clean up data associate with queue and the function is called only if context pointer is not NULL.



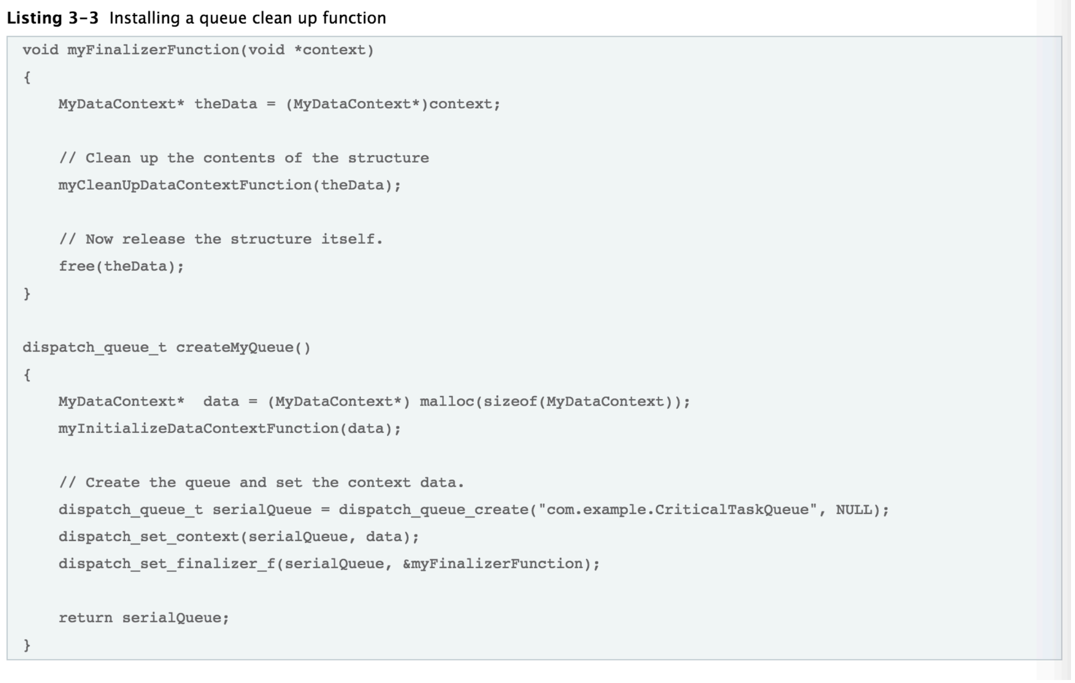
Listing 3-3 Installing a queue clean up function

```
void myFinalizerFunction(void *context)
{
    MyDataContext* theData = (MyDataContext*)context;

    // Clean up the contents of the structure
    myCleanUpDataContextFunction(theData);

    // Now release the structure itself.
    free(theData);
}

dispatch_queue_t createMyQueue()
{
    MyDataContext*  data = (MyDataContext*) malloc(sizeof(MyDataContext));
    myInitializeDataContextFunction(data);

    // Create the queue and set the context data.
    dispatch_queue_t serialQueue = dispatch_queue_create("com.example.CriticalTaskQueue", NULL);
    dispatch_set_context(serialQueue, data);
    dispatch_set_finalizer_f(serialQueue, &myFinalizerFunction);

    return serialQueue;
}
```

- create own autorelease if your block creates more than a few objc objects to free up memory.

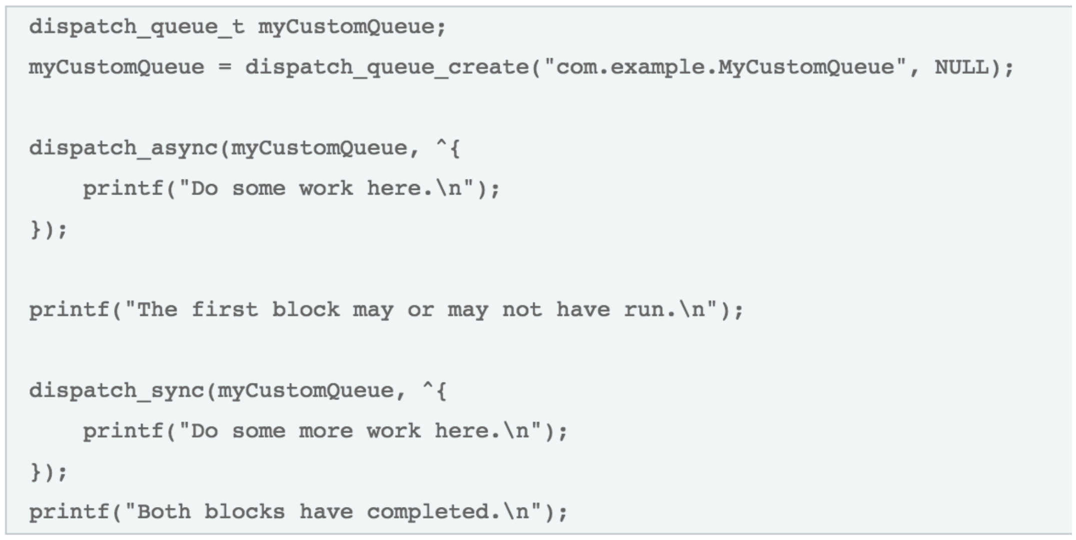
7. Creating and managing Dispatch Queue
   1. getting global concurrent dispatch queue
   2. creating serial dispatch queues
   3. get common queues at runtime
   4. memory management for dispatch queues

   dispatch_queue_t aQueue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
   dispatch_queue_t queue = dispatch_queue_create("com.example.MyQueue", NULL);
   dispatch_get_current_queue
   dispatch_get_main_queue
   dispatch_get_global_queue
   dispatch_retain, dispatch_release
   dispatch_set_context, dispatch_get_context
   dispatch_set_finalizer_f

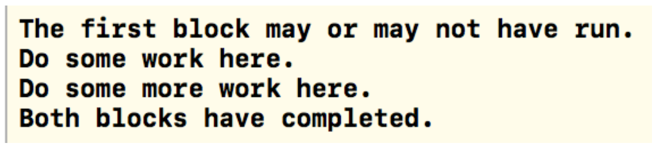

Listing 3-3 Installing a queue clean up function

```
void myFinalizerFunction(void *context)
{
    MyDataContext* theData = (MyDataContext*)context;

    // Clean up the contents of the structure
    myCleanUpDataContextFunction(theData);

    // Now release the structure itself.
    free(theData);
}

dispatch_queue_t createMyQueue()
{
    MyDataContext*  data = (MyDataContext*) malloc(sizeof(MyDataContext));
    myInitializeDataContextFunction(data);

    // Create the queue and set the context data.
    dispatch_queue_t serialQueue = dispatch_queue_create("com.example.CriticalTaskQueue", NULL);
    dispatch_set_context(serialQueue, data);
    dispatch_set_finalizer_f(serialQueue, &myFinalizerFunction);

    return serialQueue;
}
```

8. Adding tasks to a Queue
   1. adding single task to a queue
      dispatch_(a)sync, dispatch_(a)sync_f



```
dispatch_queue_t myCustomQueue;
myCustomQueue = dispatch_queue_create("com.example.MyCustomQueue", NULL);

dispatch_async(myCustomQueue, ^{
    printf("Do some work here.\n");
});

printf("The first block may or may not have run.\n");

dispatch_sync(myCustomQueue, ^{
    printf("Do some more work here.\n");
});
printf("Both blocks have completed.\n");
```



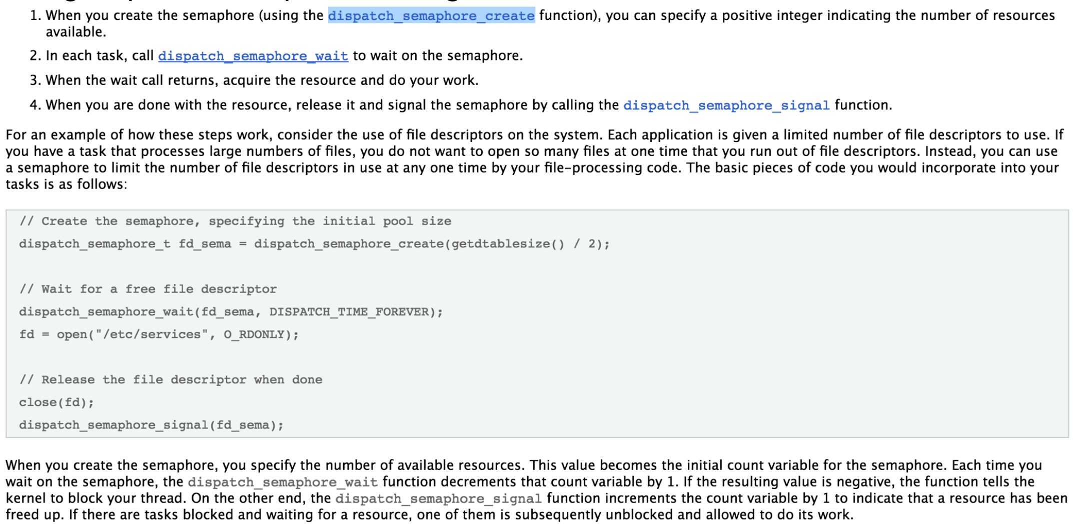
```
The first block may or may not have run.
Do some work here.
Do some more work here.
Both blocks have completed.
```

   2. performing a completion block when a task is done
      dispatch_apply, dispatch_apply_f // Passing in a concurrent queue allows

you to perform multiple loop iterations simultaneously and is the most common way to use these functions

9. Suspending and Resuming queues
   dispatch_suspend, dispatch_resume

10. Using dispatch semaphores to regulate the use of finite resources



1. When you create the semaphore (using the `dispatch_semaphore_create` function), you can specify a positive integer indicating the number of resources available.
2. In each task, call `dispatch_semaphore_wait` to wait on the semaphore.
3. When the wait call returns, acquire the resource and do your work.
4. When you are done with the resource, release it and signal the semaphore by calling the `dispatch_semaphore_signal` function.

For an example of how these steps work, consider the use of file descriptors on the system. Each application is given a limited number of file descriptors to use. If you have a task that processes large numbers of files, you do not want to open so many files at one time that you run out of file descriptors. Instead, you can use a semaphore to limit the number of file descriptors in use at any one time by your file-processing code. The basic pieces of code you would incorporate into your tasks is as follows:

```
// Create the semaphore, specifying the initial pool size
dispatch_semaphore_t fd_sema = dispatch_semaphore_create(getdtablesize() / 2);

// Wait for a free file descriptor
dispatch_semaphore_wait(fd_sema, DISPATCH_TIME_FOREVER);
fd = open("/etc/services", O_RDONLY);

// Release the file descriptor when done
close(fd);
dispatch_semaphore_signal(fd_sema);
```

When you create the semaphore, you specify the number of available resources. This value becomes the initial count variable for the semaphore. Each time you wait on the semaphore, the `dispatch_semaphore_wait` function decrements that count variable by 1. If the resulting value is negative, the function tells the kernel to block your thread. On the other end, the `dispatch_semaphore_signal` function increments the count variable by 1 to indicate that a resource has been freed up. If there are tasks blocked and waiting for a resource, one of them is subsequently unblocked and allowed to do its work.

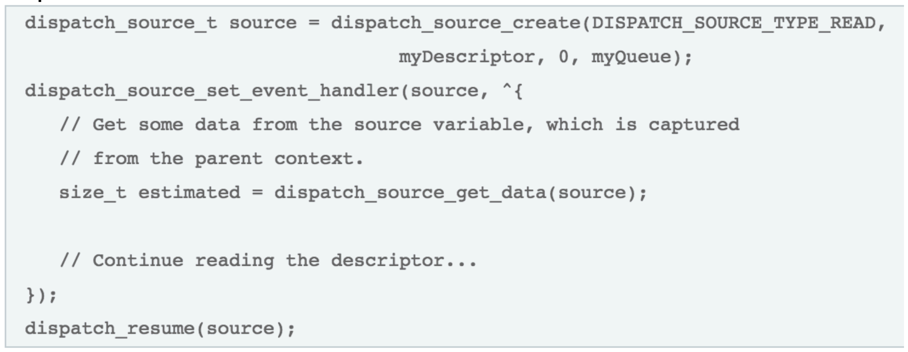
11. Waiting on Groups of Queued Tasks
    dispatch_group_t group = dispatch_group_create()
    dispatch_group_async(group, queue, ^{ … })
    dispatch_group_wait(group, DISPATCH_TIME_FOREVER)
    dispatch_release(group)

4. Dispatch Sources —
   Fundamental data type coordinates processing of specific low-level system events.
   GCD support following types of dispatch sources:
   - timer dispatch sources
   - signal dispatch sources
   - descriptor sources notify you of various file — and socket-based operations (data reading/writing, file deleting, moving or rename in file system, file meta info changed)
   - process dispatch sources notify you process-related events (a process exits, fork/exec type call, signal delivered to process)
   - mach port dispatch sources
   - custom dispatch sources

   Dispatch sources replace async callback functions that are typically used to process system-related events.
   Unlike tasks that you submit to a queue manually, dispatch sources provide a continuous source of events for your application. A dispatch source remains attached to its dispatch queue until you cancel it explicitly. While attached, it submits its

associated task code to the dispatch queue whenever the corresponding event occurs. Some events, such as timer events, occur at regular intervals but most occur only sporadically as specific conditions arise. For this reason, dispatch sources retain their associated dispatch queue to prevent it from being released prematurely while events may still be pending.

1. Creating Dispatch Sources
    1. dispatch_source_create
    2. configure dispatch source
        1. dispatch_source_set_event_handler, dispatch_source_set_event_handler_f

           ```
           dispatch_source_t source = dispatch_source_create(DISPATCH_SOURCE_TYPE_READ,
                                          myDescriptor, 0, myQueue);
           dispatch_source_set_event_handler(source, ^{
              // Get some data from the source variable, which is captured
              // from the parent context.
              size_t estimated = dispatch_source_get_data(source);

              // Continue reading the descriptor...
           });
           dispatch_resume(source);
           ```
        2. void (^dispatch_block_t)(void) // block-based event handler
        3. void (*dispatch_function_t)(void *) // function-based event handler, single context pointer
           Function-based event handlers take a single context pointer, containing the dispatch source object, and return no value. Block-based event handlers take no parameters and have no return value.

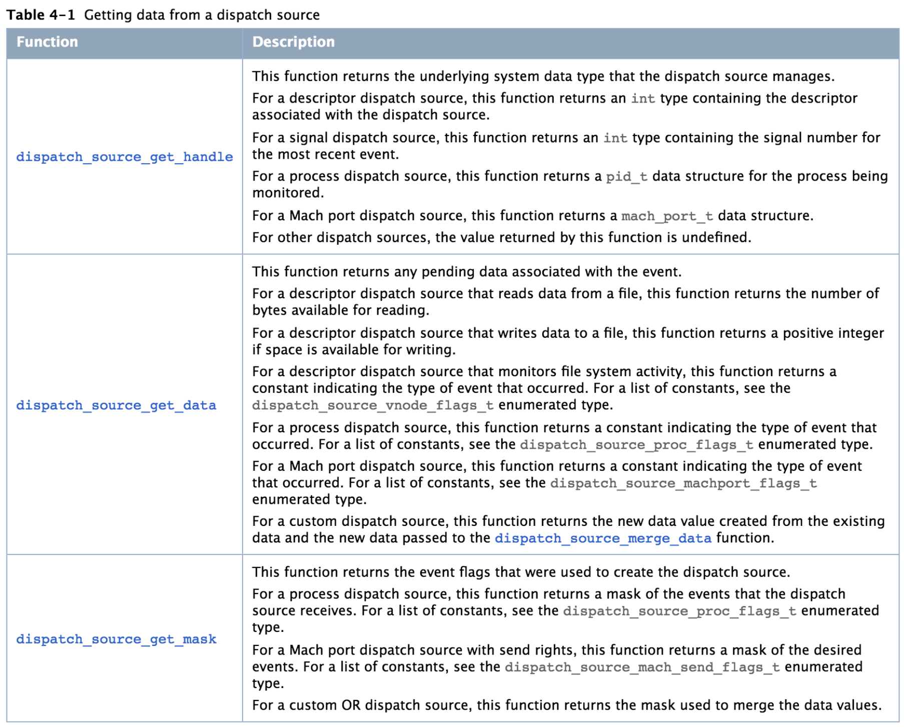
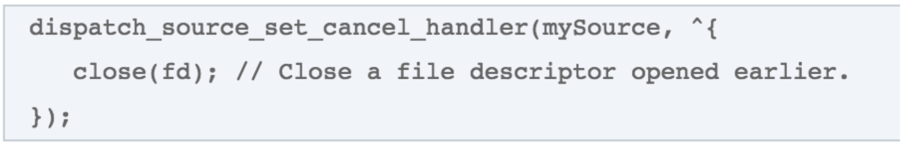


Table 4-1 Getting data from a dispatch source

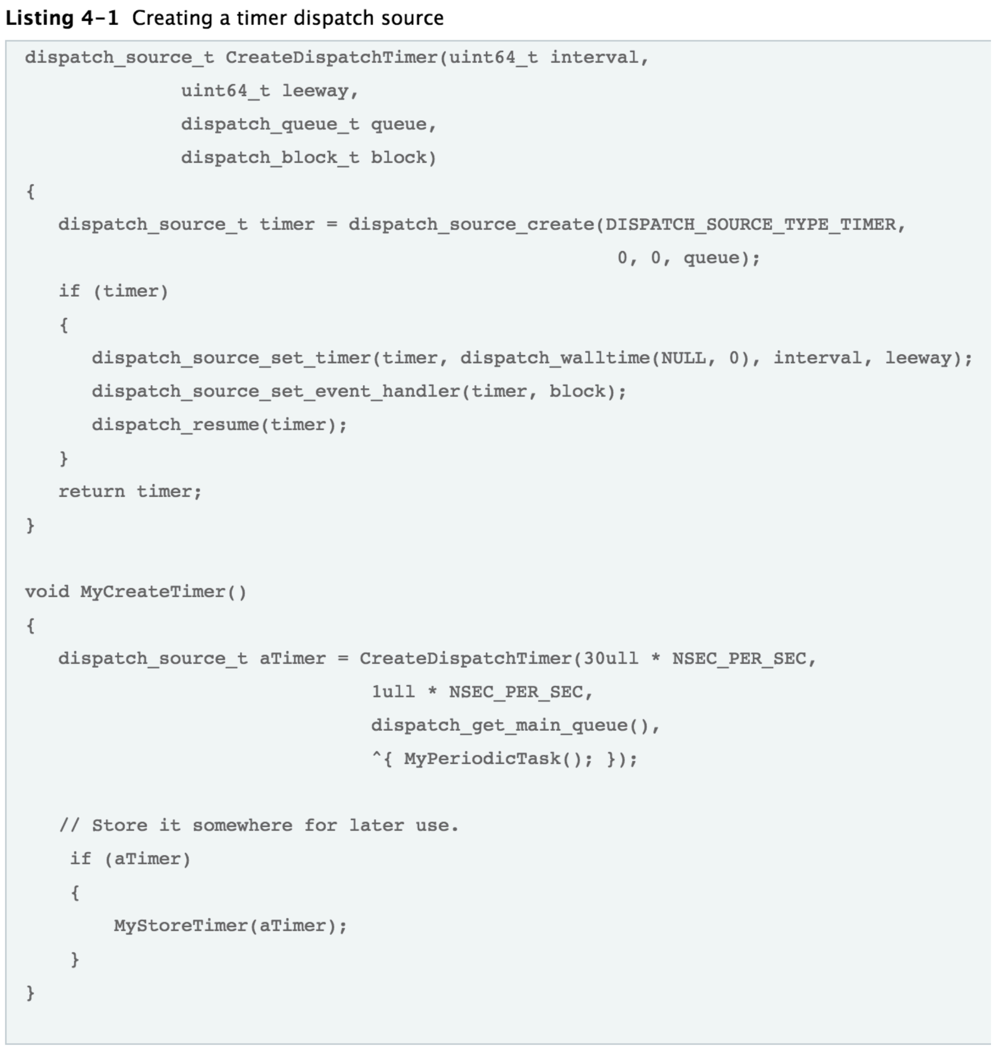
Function	Description
dispatch_source_get_handle	This function returns the underlying system data type that the dispatch source manages. For a descriptor dispatch source, this function returns an `int` type containing the descriptor associated with the dispatch source. For a signal dispatch source, this function returns an `int` type containing the signal number for the most recent event. For a process dispatch source, this function returns a `pid_t` data structure for the process being monitored. For a Mach port dispatch source, this function returns a `mach_port_t` data structure. For other dispatch sources, the value returned by this function is undefined.
dispatch_source_get_data	This function returns any pending data associated with the event. For a descriptor dispatch source that reads data from a file, this function returns the number of bytes available for reading. For a descriptor dispatch source that writes data to a file, this function returns a positive integer if space is available for writing. For a descriptor dispatch source that monitors file system activity, this function returns a constant indicating the type of event that occurred. For a list of constants, see the `dispatch_source_vnode_flags_t` enumerated type. For a process dispatch source, this function returns a constant indicating the type of event that occurred. For a list of constants, see the `dispatch_source_proc_flags_t` enumerated type. For a Mach port dispatch source, this function returns a constant indicating the type of event that occurred. For a list of constants, see the `dispatch_source_machport_flags_t` enumerated type. For a custom dispatch source, this function returns the new data value created from the existing data and the new data passed to the `dispatch_source_merge_data` function.
dispatch_source_get_mask	This function returns the event flags that were used to create the dispatch source. For a process dispatch source, this function returns a mask of the events that the dispatch source receives. For a list of constants, see the `dispatch_source_proc_flags_t` enumerated type. For a Mach port dispatch source with send rights, this function returns a mask of the desired events. For a list of constants, see the `dispatch_source_mach_send_flags_t` enumerated type. For a custom OR dispatch source, this function returns the mask used to merge the data values.

        4.
    3. optionally assign event handler to dispatch source
        1. installing cancellation handler // optimal, if you have custom behaviors tie to dispatch source.
           For dispatch sources use descriptor/ Mach port must provide cancellation handler to close/ release



```
dispatch_source_set_cancel_handler(mySource, ^{
   close(fd); // Close a file descriptor opened earlier.
});
```

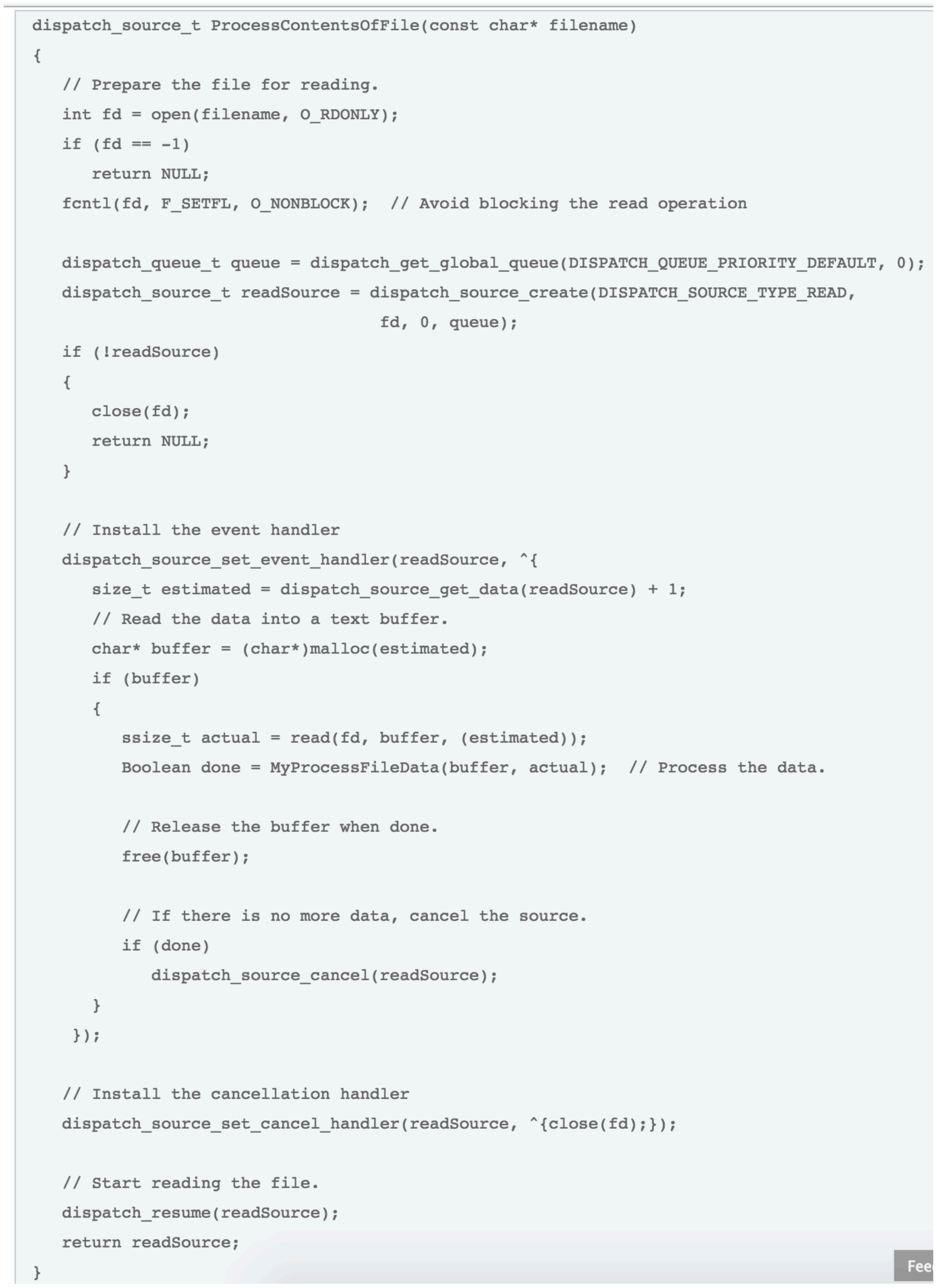
        2. changing the target queue — dispatch_set_target_queue (might change priority at which dispatch source's events are processed)
    4. call dispatch_resume function to start processing events
2. Dispatch Source Examples
    1. Timer dispatch: every 30 sec and leeway 1 sec



Listing 4-1 Creating a timer dispatch source

```
dispatch_source_t CreateDispatchTimer(uint64_t interval,
              uint64_t leeway,
              dispatch_queue_t queue,
              dispatch_block_t block)
{
   dispatch_source_t timer = dispatch_source_create(DISPATCH_SOURCE_TYPE_TIMER,
                                                     0, 0, queue);
   if (timer)
   {
      dispatch_source_set_timer(timer, dispatch_walltime(NULL, 0), interval, leeway);
      dispatch_source_set_event_handler(timer, block);
      dispatch_resume(timer);
   }
   return timer;
}

void MyCreateTimer()
{
   dispatch_source_t aTimer = CreateDispatchTimer(30ull * NSEC_PER_SEC,
                               1ull * NSEC_PER_SEC,
                               dispatch_get_main_queue(),
                               ^{ MyPeriodicTask(); });

   // Store it somewhere for later use.
    if (aTimer)
    {
        MyStoreTimer(aTimer);
    }
}
```

2. Reading data from descriptor (file/socket)



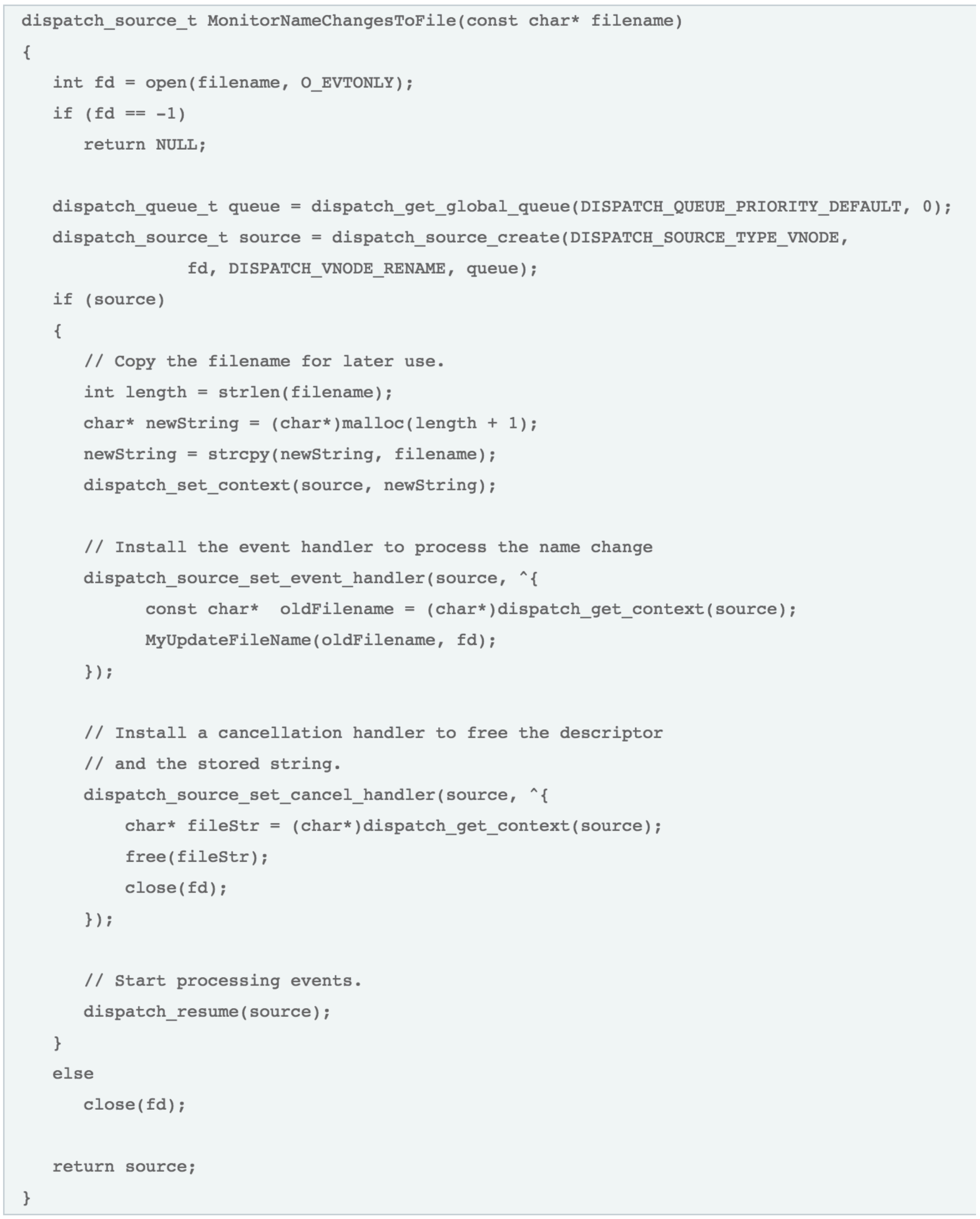
```
dispatch_source_t ProcessContentsOfFile(const char* filename)
{
   // Prepare the file for reading.
   int fd = open(filename, O_RDONLY);
   if (fd == -1)
      return NULL;
   fcntl(fd, F_SETFL, O_NONBLOCK);  // Avoid blocking the read operation

   dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
   dispatch_source_t readSource = dispatch_source_create(DISPATCH_SOURCE_TYPE_READ,
                                  fd, 0, queue);
   if (!readSource)
   {
      close(fd);
      return NULL;
   }

   // Install the event handler
   dispatch_source_set_event_handler(readSource, ^{
      size_t estimated = dispatch_source_get_data(readSource) + 1;
      // Read the data into a text buffer.
      char* buffer = (char*)malloc(estimated);
      if (buffer)
      {
         ssize_t actual = read(fd, buffer, (estimated));
         Boolean done = MyProcessFileData(buffer, actual);  // Process the data.

         // Release the buffer when done.
         free(buffer);

         // If there is no more data, cancel the source.
         if (done)
            dispatch_source_cancel(readSource);
      }
    });

   // Install the cancellation handler
   dispatch_source_set_cancel_handler(readSource, ^{close(fd);});

   // Start reading the file.
   dispatch_resume(readSource);
   return readSource;
}
```

3. Writing data to descriptor (file/socket)

Listing 4-3 Writing data to a file

```
dispatch_source_t WriteDataToFile(const char* filename)
{
    int fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC,
                      (S_IRUSR | S_IWUSR | S_ISUID | S_ISGID));
    if (fd == -1)
        return NULL;
    fcntl(fd, F_SETFL); // Block during the write.

    dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
    dispatch_source_t writeSource = dispatch_source_create(DISPATCH_SOURCE_TYPE_WRITE,
                            fd, 0, queue);
    if (!writeSource)
    {
        close(fd);
        return NULL;
    }

    dispatch_source_set_event_handler(writeSource, ^{
        size_t bufferSize = MyGetDataSize();
        void* buffer = malloc(bufferSize);

        size_t actual = MyGetData(buffer, bufferSize);
        write(fd, buffer, actual);

        free(buffer);

        // Cancel and release the dispatch source when done.
        dispatch_source_cancel(writeSource);
    });

    dispatch_source_set_cancel_handler(writeSource, ^{close(fd);});
    dispatch_resume(writeSource);
    return (writeSource);
}
```

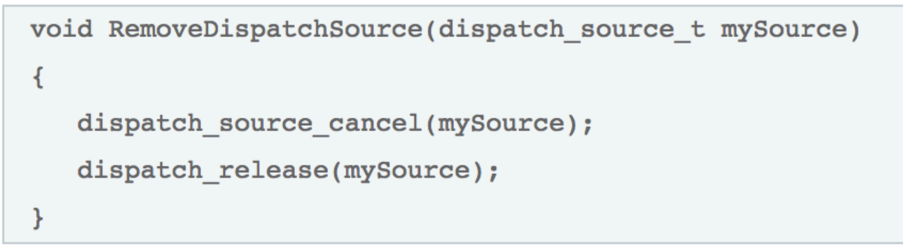
4. monitor file system object for changes



Listing 4-4 Watching for filename changes

```
dispatch_source_t MonitorNameChangesToFile(const char* filename)
{
   int fd = open(filename, O_EVTONLY);
   if (fd == -1)
      return NULL;

   dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
   dispatch_source_t source = dispatch_source_create(DISPATCH_SOURCE_TYPE_VNODE,
                fd, DISPATCH_VNODE_RENAME, queue);
   if (source)
   {
      // Copy the filename for later use.
      int length = strlen(filename);
      char* newString = (char*)malloc(length + 1);
      newString = strcpy(newString, filename);
      dispatch_set_context(source, newString);

      // Install the event handler to process the name change
      dispatch_source_set_event_handler(source, ^{
            const char*  oldFilename = (char*)dispatch_get_context(source);
            MyUpdateFileName(oldFilename, fd);
      });

      // Install a cancellation handler to free the descriptor
      // and the stored string.
      dispatch_source_set_cancel_handler(source, ^{
          char* fileStr = (char*)dispatch_get_context(source);
          free(fileStr);
          close(fd);
      });

      // Start processing events.
      dispatch_resume(source);
   }
   else
      close(fd);

   return source;
}
```

3. Canceling a Dispatch Source



```
void RemoveDispatchSource(dispatch_source_t mySource)
{
   dispatch_source_cancel(mySource);
   dispatch_release(mySource);
}
```

4. Suspending and Resuming Dispatch Source

            1. dispatch_resume, dispatch_suspend

5. Migrating Away from Threads
   (skip: https://developer.apple.com/library/content/documentation/General/Conceptual/ConcurrencyProgrammingGuide/ThreadMigration/ThreadMigration.html#//apple_ref/doc/uid/TP40008091-CH105-SW1)
    1. advantages:
        1. Reduce memory penalty your app pays for storing thread stacks in app's memory space
        2. Eliminates code to create configure your threads
        3. Eliminates code manage and schedule work on threads