# Dispatch Queues

Grand Central Dispatch (GCD) dispatch queues are a powerful tool for performing tasks. Dispatch queues let you execute arbitrary blocks of code either asynchronously or synchronously with respect to the caller. You can use dispatch queues to perform nearly all of the tasks that you used to perform on separate threads. The advantage of dispatch queues is that they are simpler to use and much more efficient at executing those tasks than the corresponding threaded code.

This chapter provides an introduction to dispatch queues, along with information about how to use them to execute general tasks in your application. If you want to replace existing threaded code with dispatch queues, you can find some additional tips for how to do that in Migrating Away from Threads.

## About Dispatch Queues

Dispatch queues are an easy way to perform tasks asynchronously and concurrently in your application. A *task* is simply some work that your application needs to perform. For example, you could define a task to perform some calculations, create or modify a data structure, process some data read from a file, or any number of things. You define tasks by placing the corresponding code inside either a function or a block object and adding it to a dispatch queue.

A dispatch queue is an object-like structure that manages the tasks you submit to it. All dispatch queues are first-in, first-out data structures. Thus, the tasks you add to a queue are always started in the same order that they were added. GCD provides some dispatch queues for you automatically, but others you can create for specific purposes. Table 3-1 lists the types of dispatch queues available to your application and how you use them.

**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.<br>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.<br>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.<br>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. |

When it comes to adding concurrency to an application, dispatch queues provide several advantages over threads. The most direct advantage is the simplicity of the work-queue programming model. With threads, you have to write code both for the work you want to perform and for the creation and management of the threads themselves. Dispatch queues let you focus on the work you actually want to perform without having to worry about the thread creation and management. Instead, the system handles all of the thread creation and management for you. The advantage is that the system is able to manage threads much more efficiently than any single application ever could. The system can scale the number of threads dynamically based on the available resources and current system conditions. In addition, the system is usually able to start running your task more quickly than you could if you created the thread yourself.

Although you might think rewriting your code for dispatch queues would be difficult, it is often easier to write code for dispatch queues than it is to write code for threads. The key to writing your code is to design tasks that are self-contained and able to run asynchronously. (This is actually true for both threads and dispatch queues.) However, where dispatch queues have an advantage is in predictability. If you have two tasks that access the same shared resource but run on different threads, either thread could modify the resource first and you would need to use a lock to ensure that both tasks did not modify that resource at the same time. With dispatch queues, you could add both tasks to a serial dispatch queue to ensure that only one task modified the resource at any given time. This type of queue-based synchronization is more efficient than locks because locks always require an expensive kernel trap in both the contested and uncontested cases, whereas a dispatch queue works primarily in your application's process space and only calls down to the kernel when absolutely necessary.

Although you would be right to point out that two tasks running in a serial queue do not run concurrently, you have to remember that if two threads take a lock at the same time, any concurrency offered by the threads is lost or significantly reduced. More importantly, the threaded model requires the creation of two threads, which take up both kernel and user-space memory. Dispatch queues do not pay the same memory penalty for their threads, and the threads they do use are kept busy and not blocked.

Some other key points to remember about dispatch queues include the following:

- Dispatch queues execute their tasks concurrently with respect to other dispatch queues. The serialization of tasks is limited to the tasks in a single dispatch queue.
- The system determines the total number of tasks executing at any one time. Thus, an application with 100 tasks in 100 different queues may not execute all of those tasks concurrently (unless it has 100 or more effective cores).
- The system takes queue priority levels into account when choosing which new tasks to start. For information about how to set the priority of a serial queue, see Providing a Clean Up Function For a Queue.
- Tasks in a queue must be ready to execute at the time they are added to the queue. (If you have used Cocoa operation objects before, notice that this behavior differs from the model operations use.)
- Private dispatch queues are reference-counted objects. In addition to retaining the queue in your own code, be aware that dispatch sources can also be attached to a queue and also increment its retain count. Thus, you must make sure that all dispatch sources are canceled and all retain calls are balanced with an appropriate release call. For more information about retaining and releasing queues, see Memory Management for Dispatch Queues. For more information about dispatch sources, see About Dispatch Sources.

For more information about interfaces you use to manipulate dispatch queues, see *Grand Central Dispatch (GCD) Reference*.

# Queue-Related Technologies

In addition to dispatch queues, Grand Central Dispatch provides several technologies that use queues to help manage your code. Table 3-2 lists these technologies and provides links to where you can find out more information about them.

**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. |

# Implementing Tasks Using Blocks

Block objects are a C-based language feature that you can use in your C, Objective-C, and C++ code. Blocks make it easy to define a self-contained unit of work. Although they might seem akin to function pointers, a block is actually represented by an underlying data structure that resembles an object and is created and managed for you by the compiler. The compiler packages up the code you provide (along with any related data) and encapsulates it in a form that can live in the heap and be passed around your application.

One of the key advantages of blocks is their ability to use variables from outside their own lexical scope. When you define a block inside a function or method, the block acts as a traditional code block would in some ways. For example, a block can read the values of variables defined in the parent scope. Variables accessed by the block are copied to the block data structure on the heap so that the block can access them later. When blocks are added to a dispatch queue, these values must typically be left in a read-only format. However, blocks that are executed synchronously can also use variables that have the `__block` keyword prepended to return data back to the parent's calling scope.

You declare blocks inline with your code using a syntax that is similar to the syntax used for function pointers. The main difference between a block and a function pointer is that the block name is preceded with a caret (`^`) instead of an asterisk (`*`). Like a function pointer, you can pass arguments to a block and receive a return value from it. Listing 3-1 shows you how to declare and execute blocks synchronously in your code. The variable `aBlock` is declared to be a block that takes a single integer parameter and returns no value. An actual block matching that prototype is then assigned to `aBlock` and declared inline. The last line executes the block immediately, printing the specified integers to standard out.

**Listing 3-1** A simple block example

```
int x = 123;
int y = 456;

// Block declaration and assignment
void (^aBlock)(int) = ^(int z) {
    printf("%d %d %d\n", x, y, z);
};

// Execute the block
aBlock(789);   // prints: 123 456 789
```

The following is a summary of some of the key guidelines you should consider when designing your blocks:

- For blocks that you plan to perform asynchronously using a dispatch queue, it is safe to capture scalar variables from the parent function or method and use them in the block. However, you should not try to capture large structures or other pointer-based variables that are allocated and deleted by the calling context. By the time your block is executed, the memory referenced by that pointer may be gone. Of course, it is safe to allocate memory (or an object) yourself and explicitly hand off ownership of that memory to the block.
- Dispatch queues copy blocks that are added to them, and they release blocks when they finish executing. In other words, you do not need to explicitly copy blocks before adding them to a queue.
- Although queues are more efficient than raw threads at executing small tasks, there is still overhead to creating blocks and executing them on a queue. If a block does too little work, it may be cheaper to execute it inline than dispatch it to a queue. The way to tell if a block is doing too little work is to gather metrics for each path using the performance tools and compare them.
- Do not cache data relative to the underlying thread and expect that data to be accessible from a different block. If tasks in the same queue need to share data, use the context pointer of the dispatch queue to store the data instead. For more information on how to access the context data of a dispatch queue, see Storing Custom Context Information with a Queue.
- If your block creates more than a few Objective-C objects, you might want to enclose parts of your block's code in an @autorelease block to handle the memory management for those objects. Although GCD dispatch queues have their own autorelease pools, they make no guarantees as to when those pools are drained. If your application is memory constrained, creating your own autorelease pool allows you to free up the memory for autoreleased objects at more regular intervals.

For more information about blocks, including how to declare and use them, see *Blocks Programming Topics*. For information about how you add blocks to a dispatch queue, see Adding Tasks to a Queue.

# Creating and Managing Dispatch Queues

Before you add your tasks to a queue, you have to decide what type of queue to use and how you intend to use it. Dispatch queues can execute tasks either serially or concurrently. In addition, if you have a specific use for the queue in mind, you can configure the queue attributes accordingly. The following sections show you how to create dispatch queues and configure them for use.

## Getting the Global Concurrent Dispatch Queues

A concurrent dispatch queue is useful when you have multiple tasks that can run in parallel. A concurrent queue is still a queue in that it dequeues tasks in a first-in, first-out order; however, a concurrent queue may dequeue additional tasks before any previous tasks finish. The actual number

of tasks executed by a concurrent queue at any given moment is variable and can change dynamically as conditions in your application change. Many factors affect the number of tasks executed by the concurrent queues, including the number of available cores, the amount of work being done by other processes, and the number and priority of tasks in other serial dispatch queues.

The system provides each application with four concurrent dispatch queues. These queues are global to the application and are differentiated only by their priority level. Because they are global, you do not create them explicitly. Instead, you ask for one of the queues using the `dispatch_get_global_queue` function, as shown in the following example:

```
dispatch_queue_t aQueue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,
0);
```

In addition to getting the default concurrent queue, you can also get queues with high- and low-priority levels by passing in the `DISPATCH_QUEUE_PRIORITY_HIGH` and `DISPATCH_QUEUE_PRIORITY_LOW` constants to the function instead, or get a background queue by passing the `DISPATCH_QUEUE_PRIORITY_BACKGROUND` constant. As you might expect, tasks in the high-priority concurrent queue execute before those in the default and low-priority queues. Similarly, tasks in the default queue execute before those in the low-priority queue.

**Note:** The second argument to the `dispatch_get_global_queue` function is reserved for future expansion. For now, you should always pass `0` for this argument.

Although dispatch queues are reference-counted objects, you do not need to retain and release the global concurrent queues. Because they are global to your application, retain and release calls for these queues are ignored. Therefore, you do not need to store references to these queues. You can just call the `dispatch_get_global_queue` function whenever you need a reference to one of them.

## Creating Serial Dispatch Queues

Serial queues are useful when you want your tasks to execute in a specific order. A serial queue executes only one task at a time and always pulls tasks from the head of the queue. You might use a serial queue instead of a lock to protect a shared resource or mutable data structure. Unlike a lock, a serial queue ensures that tasks are executed in a predictable order. And as long as you submit your tasks to a serial queue asynchronously, the queue can never deadlock.

Unlike concurrent queues, which are created for you, you must explicitly create and manage any serial queues you want to use. You can create any number of serial queues for your application but should avoid creating large numbers of serial queues solely as a means to execute as many tasks simultaneously as you can. If you want to execute large numbers of tasks concurrently, submit them to one of the global concurrent queues. When creating serial queues, try to identify a purpose for each queue, such as protecting a resource or synchronizing some key behavior of your application.

Listing 3-2 shows the steps required to create a custom serial queue. The `dispatch_queue_create` function takes two parameters: the queue name and a set of queue attributes. The debugger and performance tools display the queue name to help you track how your tasks are being executed. The queue attributes are reserved for future use and should be `NULL`.

**Listing 3-2** Creating a new serial queue

```
dispatch_queue_t queue;

queue = dispatch_queue_create("com.example.MyQueue", NULL);
```

In addition to any custom queues you create, the system automatically creates a serial queue and binds it to your application's main thread. For more information about getting the queue for the main thread, see Getting Common Queues at Runtime.

## Getting Common Queues at Runtime

Grand Central Dispatch provides functions to let you access several common dispatch queues from your application:

- Use the `dispatch_get_current_queue` function for debugging purposes or to test the identity of the current queue. Calling this function from inside a block object returns the queue to which the block was submitted (and on which it is now presumably running). Calling this function from outside of a block returns the default concurrent queue for your application.
- Use the `dispatch_get_main_queue` function to get the serial dispatch queue associated with your application's main thread. This queue is created automatically for Cocoa applications and for applications that either call the `dispatch_main` function or configure a run loop (using either the `CFRunLoopRef` type or an `NSRunLoop` object) on the main thread.
- Use the `dispatch_get_global_queue` function to get any of the shared concurrent queues. For more information, see Getting the Global Concurrent Dispatch Queues.

## Memory Management for Dispatch Queues

Dispatch queues and other dispatch objects are reference-counted data types. When you create a serial dispatch queue, it has an initial reference count of 1. You can use the `dispatch_retain` and `dispatch_release` functions to increment and decrement that reference count as needed. When the reference count of a queue reaches zero, the system asynchronously deallocates the queue.

It is important to retain and release dispatch objects, such as queues, to ensure that they remain in memory while they are being used. As with memory-managed Cocoa objects, the general rule is that if you plan to use a queue that was passed to your code, you should retain the queue before you use it and release it when you no longer need it. This basic pattern ensures that the queue remains in memory for as long as you are using it.

> **Note:** You do not need to retain or release any of the global dispatch queues, including the concurrent dispatch queues or the main dispatch queue. Any attempts to retain or release the queues are ignored.

Even if you implement a garbage-collected application, you must still retain and release your dispatch queues and other dispatch objects. Grand Central Dispatch does not support the garbage collection model for reclaiming memory.

## Storing Custom Context Information with a Queue

All dispatch objects (including dispatch queues) allow you to associate custom context data with the object. To set and get this data on a given object, you use the `dispatch_set_context` and `dispatch_get_context` functions. The system does not use your custom data in any way, and it is up to you to both allocate and deallocate the data at the appropriate times.

For queues, you can use context data to store a pointer to an Objective-C object or other data structure that helps identify the queue or its intended usage to your code. You can use the queue's finalizer function to deallocate (or disassociate) your context data from the queue before it is deallocated. An example of how to write a finalizer function that clears a queue's context data is shown in Listing 3-3.

## Providing a Clean Up Function For a Queue

After you create a serial dispatch queue, you can attach a finalizer function to perform any custom clean up when the queue is deallocated. Dispatch queues are reference counted objects and you can use the `dispatch_set_finalizer_f` function to specify a function to be executed when the reference count of your queue reaches zero. You use this function to clean up the context data associated with a queue and the function is called only if the context pointer is not `NULL`.

Listing 3-3 shows a custom finalizer function and a function that creates a queue and installs that finalizer. The queue uses the finalizer function to release the data stored in the queue's context pointer. (The `myInitializeDataContextFunction` and `myCleanUpDataContextFunction` functions referenced from the code are custom functions that you would provide to initialize and

clean up the contents of the data structure itself.) The context pointer passed to the finalizer function contains the data object associated with the queue.

**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;
}
```

# Adding Tasks to a Queue

To execute a task, you must dispatch it to an appropriate dispatch queue. You can dispatch tasks synchronously or asynchronously, and you can dispatch them singly or in groups. Once in a queue, the queue becomes responsible for executing your tasks as soon as possible, given its constraints and the existing tasks already in the queue. This section shows you some of the techniques for dispatching tasks to a queue and describes the advantages of each.

## Adding a Single Task to a Queue

There are two ways to add a task to a queue: asynchronously or synchronously. When possible, asynchronous execution using the `dispatch_async` and `dispatch_async_f` functions is preferred over the synchronous alternative. When you add a block object or function to a queue, there is no way to know when that code will execute. As a result, adding blocks or functions asynchronously lets you schedule the execution of the code and continue to do other work from the calling thread. This is especially important if you are scheduling the task from your application's main thread—perhaps in response to some user event.

Although you should add tasks asynchronously whenever possible, there may still be times when you need to add a task synchronously to prevent race conditions or other synchronization errors. In these instances, you can use the `dispatch_sync` and `dispatch_sync_f` functions to add the task to the

queue. These functions block the current thread of execution until the specified task finishes executing.

> **Important:** You should never call the `dispatch_sync` or `dispatch_sync_f` function from a task that is executing in the same queue that you are planning to pass to the function. This is particularly important for serial queues, which are guaranteed to deadlock, but should also be avoided for concurrent queues.

The following example shows how to use the block-based variants for dispatching tasks asynchronously and synchronously:

```
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");
```

## Performing a Completion Block When a Task Is Done

By their nature, tasks dispatched to a queue run independently of the code that created them. However, when the task is done, your application might still want to be notified of that fact so that it can incorporate the results. With traditional asynchronous programming, you might do this using a callback mechanism, but with dispatch queues you can use a completion block.

A completion block is just another piece of code that you dispatch to a queue at the end of your original task. The calling code typically provides the completion block as a parameter when it starts the task. All the task code has to do is submit the specified block or function to the specified queue when it finishes its work.

Listing 3-4 shows an averaging function implemented using blocks. The last two parameters to the averaging function allow the caller to specify a queue and block to use when reporting the results. After the averaging function computes its value, it passes the results to the specified block and dispatches it to the queue. To prevent the queue from being released prematurely, it is critical to retain that queue initially and release it once the completion block has been dispatched.

**Listing 3-4** Executing a completion callback after a task

```
void average_async(int *data, size_t len,
   dispatch_queue_t queue, void (^block)(int))
{
   // Retain the queue provided by the user to make
   // sure it does not disappear before the completion
   // block can be called.
   dispatch_retain(queue);
```

```
    // Do the work on the default concurrent queue and then
    // call the user-provided block with the results.
    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
        int avg = average(data, len);
        dispatch_async(queue, ^{ block(avg);});

        // Release the user-provided queue when done
        dispatch_release(queue);
    });
}
```

## Performing Loop Iterations Concurrently

One place where concurrent dispatch queues might improve performance is in places where you have a loop that performs a fixed number of iterations. For example, suppose you have a `for` loop that does some work through each loop iteration:

```
for (i = 0; i < count; i++) {
    printf("%u\n",i);
}
```

If the work performed during each iteration is distinct from the work performed during all other iterations, and the order in which each successive loop finishes is unimportant, you can replace the loop with a call to the `dispatch_apply` or `dispatch_apply_f` function. These functions submit the specified block or function to a queue once for each loop iteration. When dispatched to a concurrent queue, it is therefore possible to perform multiple loop iterations at the same time.

You can specify either a serial queue or a concurrent queue when calling `dispatch_apply` or `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. Although using a serial queue is permissible and does the right thing for your code, using such a queue has no real performance advantages over leaving the loop in place.

**Important:** Like a regular `for` loop, the `dispatch_apply` and `dispatch_apply_f` functions do not return until all loop iterations are complete. You should therefore be careful when calling them from code that is already executing from the context of a queue. If the queue you pass as a parameter to the function is a serial queue and is the same one executing the current code, calling these functions will deadlock the queue.

Because they effectively block the current thread, you should also be careful when calling these functions from your main thread, where they could prevent your event handling loop from responding to events in a timely manner. If your loop code requires a noticeable amount of processing time, you might want to call these functions from a different thread.

Listing 3-5 shows how to replace the preceding `for` loop with the `dispatch_apply` syntax. The block you pass in to the `dispatch_apply` function must contain a single parameter that identifies the current loop iteration. When the block is executed, the value of this parameter is `0` for the first iteration, `1` for the second, and so on. The value of the parameter for the last iteration is `count - 1`, where `count` is the total number of iterations.

**Listing 3-5** Performing the iterations of a `for` loop concurrently

```
dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,
0);

dispatch_apply(count, queue, ^(size_t i) {
```

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

You should make sure that your task code does a reasonable amount of work through each iteration. As with any block or function you dispatch to a queue, there is overhead to scheduling that code for execution. If each iteration of your loop performs only a small amount of work, the overhead of scheduling the code may outweigh the performance benefits you might achieve from dispatching it to a queue. If you find this is true during your testing, you can use striding to increase the amount of work performed during each loop iteration. With striding, you group together multiple iterations of your original loop into a single block and reduce the iteration count proportionately. For example, if you perform 100 iterations initially but decide to use a stride of 4, you now perform 4 loop iterations from each block and your iteration count is 25. For an example of how to implement striding, see Improving on Loop Code.

### Performing Tasks on the Main Thread

Grand Central Dispatch provides a special dispatch queue that you can use to execute tasks on your application's main thread. This queue is provided automatically for all applications and is drained automatically by any application that sets up a run loop (managed by either a `CFRunLoopRef` type or `NSRunLoop` object) on its main thread. If you are not creating a Cocoa application and do not want to set up a run loop explicitly, you must call the `dispatch_main` function to drain the main dispatch queue explicitly. You can still add tasks to the queue, but if you do not call this function those tasks are never executed.

You can get the dispatch queue for your application's main thread by calling the `dispatch_get_main_queue` function. Tasks added to this queue are performed serially on the main thread itself. Therefore, you can use this queue as a synchronization point for work being done in other parts of your application.

### Using Objective-C Objects in Your Tasks

GCD provides built-in support for Cocoa memory management techniques so you may freely use Objective-C objects in the blocks you submit to dispatch queues. Each dispatch queue maintains its own autorelease pool to ensure that autoreleased objects are released at some point; queues make no guarantee about when they actually release those objects.

If your application is memory constrained and your block creates more than a few autoreleased objects, creating your own autorelease pool is the only way to ensure that your objects are released in a timely manner. If your block creates hundreds of objects, you might want to create more than one autorelease pool or drain your pool at regular intervals.

For more information about autorelease pools and Objective-C memory management, see *Advanced Memory Management Programming Guide*.

## Suspending and Resuming Queues

You can prevent a queue from executing block objects temporarily by suspending it. You suspend a dispatch queue using the `dispatch_suspend` function and resume it using the `dispatch_resume` function. Calling `dispatch_suspend` increments the queue's suspension reference count, and calling `dispatch_resume` decrements the reference count. While the reference count is greater than zero, the queue remains suspended. Therefore, you must balance all suspend calls with a matching resume call in order to resume processing blocks.

**Important:** Suspend and resume calls are asynchronous and take effect only between the execution of blocks. Suspending a queue does not cause an already executing block to stop.

# Using Dispatch Semaphores to Regulate the Use of Finite Resources

If the tasks you are submitting to dispatch queues access some finite resource, you may want to use a dispatch semaphore to regulate the number of tasks simultaneously accessing that resource. A dispatch semaphore works like a regular semaphore with one exception. When resources are available, it takes less time to acquire a dispatch semaphore than it does to acquire a traditional system semaphore. This is because Grand Central Dispatch does not call down into the kernel for this particular case. The only time it calls down into the kernel is when the resource is not available and the system needs to park your thread until the semaphore is signaled.

The semantics for using a dispatch semaphore are as follows:

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.

# Waiting on Groups of Queued Tasks

Dispatch groups are a way to block a thread until one or more tasks finish executing. You can use this behavior in places where you cannot make progress until all of the specified tasks are complete. For example, after dispatching several tasks to compute some data, you might use a group to wait on those tasks and then process the results when they are done. Another way to use dispatch groups is as an alternative to thread joins. Instead of starting several child threads and then joining with each of them, you could add the corresponding tasks to a dispatch group and wait on the entire group.

Listing 3-6 shows the basic process for setting up a group, dispatching tasks to it, and waiting on the results. Instead of dispatching tasks to a queue using the `dispatch_async` function, you use the `dispatch_group_async` function instead. This function associates the task with the group and queues it for execution. To wait on a group of tasks to finish, you then use the `dispatch_group_wait` function, passing in the appropriate group.

**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 and Thread Safety

It might seem odd to talk about thread safety in the context of dispatch queues, but thread safety is still a relevant topic. Any time you are implementing concurrency in your application, there are a few things you should know:

- Dispatch queues themselves are thread safe. In other words, you can submit tasks to a dispatch queue from any thread on the system without first taking a lock or synchronizing access to the queue.
- Do not call the `dispatch_sync` function from a task that is executing on the same queue that you pass to your function call. Doing so will deadlock the queue. If you need to dispatch to the current queue, do so asynchronously using the `dispatch_async` function.
- Avoid taking locks from the tasks you submit to a dispatch queue. Although it is safe to use locks from your tasks, when you acquire the lock, you risk blocking a serial queue entirely if that lock is unavailable. Similarly, for concurrent queues, waiting on a lock might prevent other tasks from executing instead. If you need to synchronize parts of your code, use a serial dispatch queue instead of a lock.
- Although you can obtain information about the underlying thread running a task, it is better to avoid doing so. For more information about the compatibility of dispatch queues with threads, see Compatibility with POSIX Threads.

For additional tips on how to change your existing threaded code to use dispatch queues, see Migrating Away from Threads.