Managing Concurrency with NSOperation

Recent years have seen an end to the continual rise of processor clock speeds, and a move toward multi-core as the way forward. This trend has implications for software developers, because in order to make use of the extra processing power available in multi-core systems, you have to rewrite parts of your application to work concurrently. Even if you do not need that extra number-crunching performance in your app, there are many operations—such as downloading files—that can be more efficient when performed simultaneously.

Multi-threading has a reputation for being difficult and unpredictable, and most developers prefer to steer clear of it. With Leopard, three classes have been added for Cocoa developers to take advantage of multi-core, and concurrency in general, while avoiding the intricacy of low-level multi-threading: `NSOperation`, `NSInvocationOperation`, and `NSOperationQueue`.

The `NSOperation` and `NSOperationQueue` classes alleviate much of the pain of multi-threading, allowing you to simply define your tasks, set any dependencies that exist, and fire them off. Each task, or *operation*, is represented by an instance of an `NSOperation` class; the `NSOperationQueue` class takes care of starting the operations, ensuring that they are run in the appropriate order, and accounting for any priorities that have been set.

In this tutorial, you will see how `NSOperation` and `NSOperationQueue` can be used to build two simple applications. The first, simply called *Image Download*, performs batch downloads of images to the user Downloads folder. The second, more advanced application, known as *Post Op*, not only downloads images, but also combines them to form a 3D poster.

Both applications take a search term entered by the user, and retrieve a corresponding list of image URLs from the Yahoo! Image Search web service. `NSOperation` and `NSOperationQueue` are then used to download the images, and—in the case of the more advanced application—draw them into a poster to give the appearance of photos floating in three dimensional space.

NOTE: Both projects require Xcode 3.0 or later.

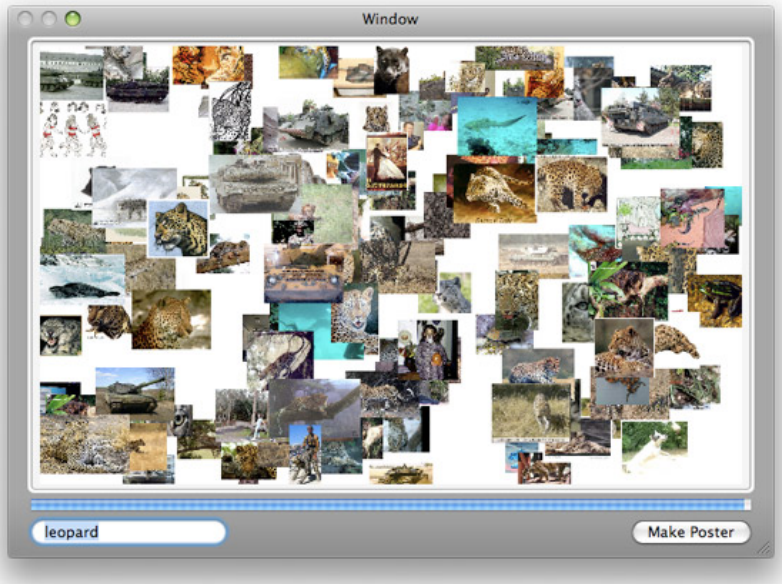

Figure 1: The Post Op application builds 3D posters from downloaded images.

Image Download demonstrates the most basic usage of the operation classes: performing a homogeneous set of completely independent tasks. Post Op shows more advanced usage, with a mixture of operations, and reasonably complex dependencies.

NSOperation and NSOperationQueue

There are a number of different ways that you can use `NSOperation`, but the most common is to write a custom subclass and override one method: `main`. The `main` method gets called to perform the operation when the `NSOperationQueue` schedules it to run. `NSOperation` classes written in this way are known as

Related Articles

Threading Programming Guide
NSOperation Class Reference
NSOperationQueue Class Reference

Resources

ImageDownload (.dmg, 536KB) Xcode project
Post Op (.dmg, 540KB) Xcode project

non-concurrent operations, because the developer is not responsible for spawning threads—multi-threading is all handled by the super class. (Don't be confused by the terminology: just because an operation is *non-concurrent*, does not mean it cannot be executed concurrently, it simply means that you don't have to handle the concurrency yourself.)

If you need more control over threading and the run-time environment of your operations, you can make use of *concurrent operations*. To do this, you subclass `NSOperation` and override the `start` method. In the `start` method, you can spawn threads and setup the environment before calling the `main` method. You are also required to maintain the state of the `NSOperation` by setting properties like `isExecuting` and `isFinished`. In short, concurrent operations give you a lot more control, but also demand more effort—for most tasks non-concurrent operations suffice.

The `NSInvocationOperation` class provides a third, light-weight means of utilizing the operation classes. `NSInvocationOperation` is a concrete subclass of `NSOperation` that makes it trivial to attach an operation to an existing method. `NSInvocationOperation` objects can be added to an `NSOperationQueue`, just as any other `NSOperation`, so that you get the same benefits of scheduling and multi-threading without the overhead of subclassing.

In addition to actually performing a task, `NSOperation` also provides various means of influencing the order of execution. In particular, you can set the priority of an `NSOperation`, and dependencies between `NSOperation` instances. An `NSOperation` will not be run by an `NSOperationQueue` until all of its dependencies are finished running, whether they have been added to the same `NSOperationQueue` or not. Setting priorities is useful for helping the `NSOperationQueue` choose between operations that are ready to be executed: the `NSOperation` with the highest priority will run first.

Subclassing NSOperation

In the coming sections, you will see how the operation classes work in practice, beginning in the ImageDownload (.dmg, 536KB) source code. Image Download uses non-concurrent operations, with `DownloadOperation`—a subclass of `NSOperation`—responsible for downloading the images used in the poster.

```
@interface DownloadOperation : NSOperation {
    NSURL *url;
    NSString *downloadPath;
}

@property (readonly, copy) NSURL *url;
@property (readonly, copy) NSString *downloadPath;

-(id)initWithURL:(NSURL *)url downloadPath:(NSString *)downloadPath;

@end
```

Each `DownloadOperation` downloads one image from a URL, and saves it to a path in the file system. The URL and path are set via the `initWithURL:downloadPath:` initializer.

The `DownloadOperation` class performs the image download in the `main` method, which is inherited from `NSOperation`.

```
@implementation DownloadOperation

...

-(void)main {
    if ( self.isCancelled ) return;
    if ( nil == self.url ) return;
    NSData *imageData = [NSData dataWithContentsOfURL:self.url];
    if ( self.isCancelled ) return;
    [imageData writeToFile:self.downloadPath atomically:NO];
}

@end
```

The `main` method is fairly straightforward: It downloads the image data from the URL using the `NSData` initializer `dataWithContentsOfURL:`, and then writes it to file with the `writeToFile:` method. Despite this simplicity, it does contain a subtle, but important, detail: The `NSOperation` property `isCancelled` is checked regularly, and the method returns if it is set to `YES`. A cancelled operation will not be forcibly terminated, but is expected to exit its `main` method as soon as possible. You should add checks of the `isCancelled` property to all of your `main` methods.

Running Operations

Image Download works as follows: the user enters a search term, and clicks the Download button. The

button's action is the `downloadImages:` method of the `MainController` class; when invoked, this method calls the `startOperations` method, which downloads the URLs from Yahoo!, and then creates `DownloadOperation` objects to download the corresponding images. The operations are submitted to an `NSOperationQueue`, which begins running them.

The `startOperations` method begins by creating a new `NSOperationQueue`.

```
-(void)startOperations {
    // Setup the operation queue.
    // Cancel any previous operations that might be running
    [operationQueue cancelAllOperations];
    self.operationQueue = [NSOperationQueue new];
    [operationQueue setMaxConcurrentOperationCount:8];
```

The `cancelAllOperations` is used to cancel any operations that might still be running in an old operation queue from a previous search, and the `setMaxConcurrentOperationCount:` has been invoked to limit the number of threads that the `NSOperationQueue` will spawn. This is not a requirement; if you don't do it, the `NSOperationQueue` will automatically determine how many threads it should spawn.

The `downloadImageURLs` method is then invoked to download the image URLs and store them in the `imageURLs` instance variable.

```
    // Download URLs from Yahoo!
    [self downloadImageURLs];
```

The code in `downloadImageURLs` is interesting in itself, making use of Cocoa's XML classes to parse the data supplied by the Web Service, but is not particularly relevant to `NSOperation` and will not be covered here.

The `startOperations` continues by creating a folder for the images called 'Image Download' in the user's Downloads folder, and then the `DownloadOperation` objects themselves.

```
    // Create download folder
    NSArray *searchPaths =
NSSearchPathForDirectoriesInDomains(NSDownloadsDirectory,
        NSUserDomainMask, YES);
    NSString *downloadFolder =
        [[searchPaths lastObject] stringByAppendingPathComponent:@"Image
Download"];
    if ( ![[NSFileManager defaultManager] fileExistsAtPath:downloadFolder] ) {
        [[NSFileManager defaultManager] createDirectoryAtPath:downloadFolder
attributes:nil];
    }

    // Create the image download operations, and add them to the queue
    NSUInteger count = 0;
    for ( NSURL *url in imageURLs ) {
        NSString *extension = [[url path] pathExtension];
        NSString *filename = [NSString stringWithFormat:@"Image %d.%@", ++count,
extension];
        NSString *downloadPath = [downloadFolder
stringByAppendingPathComponent:filename];
        DownloadOperation *operation = [[DownloadOperation alloc] initWithURL:url
            downloadPath:downloadPath];
        [operationQueue addOperation:operation];
    }
}
```

Each image is given a name beginning with 'Image', and followed by a number and an extension. The extension is retrieved from the image URL.

After each `DownloadOperation` is created, it is added to the `NSOperationQueue` using the `addOperation:` instance method. From this point forth the operation can be run by the operation queue—the `NSOperationQueue` does not have to wait for all operations to be added first. If you need to, you can prevent operations from running by temporarily suspending the operation queue using the `setSuspended:` method.

In many cases, this is all there is to utilizing the Cocoa operation classes. No messy threading or locking, just a set of independent operations—`NSOperation` and `NSOperationQueue` take care of the details.

Advanced Usage

The Image Download application demonstrates by far the most common usage of `NSOperation`, but there will be some cases that demand more, and the operation classes can handle those too.

The Post Op (.dmg, 540KB) source code includes examples of more advanced usage of `NSOperation`, including setting dependencies between operations, utilizing `NSInvocationOperation` to turn preexisting methods into operations and dealing with threading issues when accessing shared resources and non-threadsafe frameworks (for information about thread safety in AppKit, refer to the Threading Programming Guide). These more advanced usages will be covered in the following sections.

Methods as Operations

The `NSInvocationOperation` subclass allows you to turn existing methods into `NSOperation`s. In Post Op, an `NSInvocationOperation` is used in the `startOperations` method of the `MainController` class to download the URLs of the images in the poster, before the download operations begin running.

```
// Add the index downloading operation
NSInvocationOperation *indexOperation = [[NSInvocationOperation alloc]
initWithTarget:self
    selector:@selector(downloadImageURLs) object:nil];
[operationQueue addOperation:indexOperation];
```

The `NSInvocationOperation` initializer is passed the selector of the `downloadImageURLs` method, which retrieves the image URLs; the target of the invocation, the `MainController` object `self`; and an optional argument, which is passed back in the invocation if it is non-nil. When the `NSInvocationOperation` is run by the operation queue, it invokes the method corresponding to the selector passed into the initializer. If you already have methods that perform well defined operations, this can be a very light-weight way to use `NSOperation`.

Setting Dependencies

Although many applications of the operation classes will involve independent tasks, others will have dependencies. In Post Op, for example, in order to avoid storing downloaded images in memory or on disk, they are added to the poster immediately, and then deleted. The poster is built up from back to front, with images that are to appear further away drawn first. Downloads of images in the foreground thus depend upon those in the background having already been completed.

To simplify the process of setting the dependencies, and to refresh the poster image in the user interface after each layer is drawn, an extra `NSInvocationOperation` instance is added for each layer. The invocation operations invoke the `refreshPosterImage` method, which updates the poster image in the user interface, as shown in Figure 2.

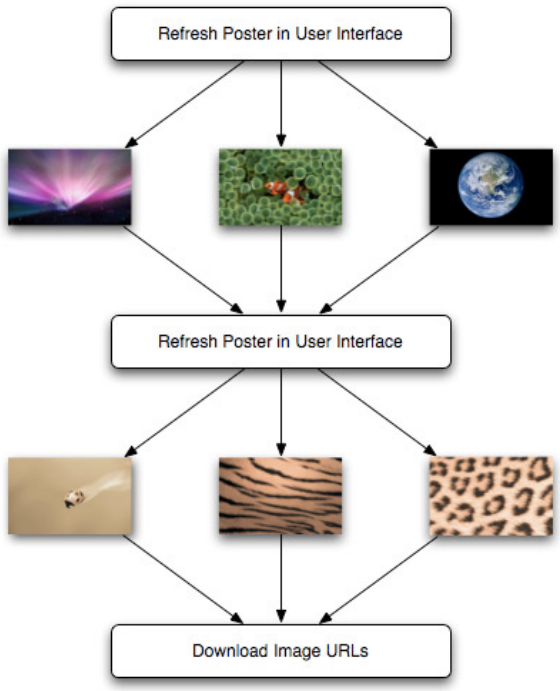

Figure 2: Dependencies between operations in Post Op. Extra 'layer' operations reduce the number of dependencies, and refresh the user interface.

Creating the download operations involves a double loop: the outer loop runs over layers and the inner loop over images in each layer.

```
    // Loop over layers (z-index) from back to front, creating operations to
download images.
    // Images for each layer are dependent on those in the previous layer.
    NSInteger zIndex;
    NSInvocationOperation *lastLayerOperation = nil;
    for ( zIndex = 0; zIndex < NUMBER_OF_LAYERS; zIndex++ ) {

        // Create one operation for each layer, to refresh interface, and to make
setting up
        // dependencies a bit easier.
        NSInvocationOperation *layerOperation = [[NSInvocationOperation alloc]
initWithTarget:self
            selector:@selector(refreshPosterImage) object:nil];
        [operationQueue addOperation:layerOperation];

        // Setup download operations for images in this layer
        NSInteger imageIndex;
        for ( imageIndex = 0; imageIndex < NUMBER_OF_IMAGES / NUMBER_OF_LAYERS;
++imageIndex ) {
            DownloadOperation *operation = [DownloadOperation new];
            operation.zIndex = zIndex;
            operation.delegate = self;
            if ( nil != lastLayerOperation ) {
                [operation addDependency:lastLayerOperation];  // This operation
depends on last layer
            }
            else {
                [operation addDependency:indexOperation]; // First layer images
depend on URLs download
            }
            [layerOperation addDependency:operation];     // Next layer depends on
this operation
            [operationQueue addOperation:operation];
        }

        // Update previous layer variable
        lastLayerOperation = layerOperation;
    }
```

Dependencies are set using the `NSOperation` method `addDependency:`. Without the layer operations, dependencies would have to be added between each image and all of the images in the previous layer. By introducing the layer operations, each image only depends on a single layer operation. Images in the first layer are made dependent on the URL download operation (*i.e.*, `indexOperation`).

Threading Issues

Although the operation classes make dealing with concurrency considerably simpler than traditional low-level multi-threading, and generally lead to cleaner code design, you still have to be aware that operations will generally run on multiple threads. There are two cases where extra code may be needed to avoid threading bugs: when multiple operations modify a shared resource; and when a non-threadsafe framework is used.

In Post Op, image URLs are stored in a mutable set in the `MainController` class. This set represents a shared resource because each `DownloadOperation` instance needs to extract a URL from it before beginning to download. (The reason the URLs are not passed directly to the `DownloadOperation` initializer, as in Image Download, is that they have yet to be downloaded when the `DownloadOperation` instances are created.) The `urlForDownloadOperation:` delegate method is invoked from the `main` method of `DownloadOperation`; this method can potentially be called by multiple threads at once and has to be written to handle this eventuality.

```
-(NSURL *)urlForDownloadOperation:(DownloadOperation *)operation; {
    NSURL *url;
    @synchronized (imageURLs) {
        url = [imageURLs anyObject];
        [imageURLs removeObject:url];
    }
    return url;
}
```

A `@synchronized` block is used to make sure that the `imageURLs` set, which contains the URLs for the images to be downloaded, remains in a valid state. A `@synchronized` block can only be executed by one thread at a time. Without the `@synchronized` block, it is possible for two or more threads to retrieve the same URL, and each try to remove it with the call to `removeObject:`. This is known as a *race condition*, and will usually result in a crash, or unpredictable results at the very least. Note that this is only a problem

when the shared resource is being modified: when it is accessed without modification there is no problem, provided the object in question is threadsafe.

The second threading gotcha arises when you need to invoke methods from a non-threadsafe framework (for information on thread safety, see the Threading Programming Guide). The `NSObject` method `performSelectorOnMainThread:withObject:waitUntilDone:` method can be used to ensure that such invocations take place on the main thread. In Post Op, the `DownloadOperation` class draws its image into the poster using AppKit; to do this, it invokes a delegate method: `processImageForDownloadOperation:`. But rather than invoke the delegate method directly, it uses `performSelectorOnMainThread:withObject:waitUntilDone` to ensure that the drawing takes place on the main thread.

```
-(void)main {
    if ( self.isCancelled ) return;
    if ( nil != delegate ) {
        NSURL *url = [delegate urlForDownloadOperation:self];
        if ( nil == url ) return;
        self.downloadedImage = [[NSImage alloc] initWithContentsOfURL:url];
        if ( self.isCancelled ) return;
        [delegate
performSelectorOnMainThread:@selector(processImageForDownloadOperation:)
            withObject:self waitUntilDone:YES];
    }
}
```

Running Post Op

Running Post Op is very straightforward. Just enter a search term, or series of terms, and press the Make Poster button. Post Op will download 200 images, which can take a few minutes, and produce a desktop sized poster from them. You can save a TIFF image of the poster by choosing File > Save Poster….

You can use the Activity Monitor utility to observe the behavior of Post Op as it downloads images. In particular, the '# Threads' column shows how the number of threads gets varied in time by the `NSOperationQueue`. Note how the number of threads changes as Post Op downloads and draws each layer, growing as download operations are created, and shrinking again as the program waits for the last images in a layer to finish downloading.

Next Operation

The `NSOperation` classes are a Leopard hidden gem, making task-based concurrency much more approachable for Cocoa developers. There are many different ways of using `NSOperation`, but you have now seen the most common usages. For more details, and other possibilities, a good place to start is the Threading Programming Guide. The `NSOperation` and `NSOperationQueue` class references are also very useful, and the Coding Headstart 'Simpler Threading with NSOperation' (found in the Mac Dev Center: Coding Head Starts, login required) provides a hands-on introduction.

Updated: 2008-09-30