

Weaving the Leopard's Pelt

Simulating Fibers on OS X

Andrew Pontious
apontious@umbar.com

Abstract

A fiber is a thread-like mechanism on Windows NT and later that must be manually switched to, unlike normal threads, which are designed to be started and run concurrently by the system. On OS X, there are no specific APIs for creating and running fibers, and why would you want to?

This paper describes two scenarios where it might be useful, a ported codebase that assumes it will always control program flow, and an application framework that provides synchronous functions for asynchronous system APIs. Both are implemented in simplified form, the former in Cocoa, the latter in Carbon.

Scenarios

The ported codebase and application framework scenarios described above are real-world cases. So first, let's take a closer look at them.

TADS Port

Michael Roberts debuted TADS, the Text Adventure Development System, in 1988. As with most modern text adventure gaming systems, the TADS compiler creates not a full application, but a compiled game file that still requires an interpreter to be played on a particular platform, similar to the way Java works. Roberts originally wrote the compiler and interpreter for both the Mac and PC platforms.

Because of its genesis as a multi-platform system, the TADS interpreter's cross-platform "engine" code is written from the ground up to be aware of multi-platform issues. For example, it allows for use of platform-specific file references, such as FSSpecs and FSRefs on the Macintosh. But it made one crucial assumption; flow of execution is controlled by the TADS engine without interruption. Events are handled via platform-specific implementation calls without giving up program control, without allowing callbacks of any sort. This is a valid assumption on Windows,

and is a valid assumption for what we now call Wait Next Event-based Classic and Carbon applications. But it is not a valid assumption for newer, RAEL-based Carbon applications or any Cocoa applications.

Isthmus Application Framework

As part of my implementation of a new generation of Macintosh TADS compilers, I'm writing a more general-purpose application framework called Isthmus. (There are a number of reasons for this, which I won't get in to now.)

Even in Mac OS 9, there are Toolbox APIs that require callbacks, notably the Notification Manager. In Mac OS X, this list expands to include sheet-based Navigation Services APIs. I tried for longer than I'd like to admit to write my framework so that it could handle these sorts of callback-based APIs in a general way. This meant that any activity that would require these APIs needed to be chopped up into two pieces: the piece before the call, and the piece that would resume the desired logic after the callback. This turned out to be very difficult to set up, and would be a great burden for users of the framework to support.

It would be much easier if there were some way to subsume the call/callback

structure needed by these APIs into a single, synchronous framework call.

The Solution

“Inverse Threads”

The common thread (so to speak) in these two cases is the need to defer to the OS X event loop without actually giving up flow of execution. That sounds like a task for threads, one thread to hold the engine or framework flow, and one for the OS.

The first realization here is that although we need two threads, using a default multithreaded architecture is overcomplicating things. We don't actually need two simultaneous, preemptive threads; the tasks here do not need to be done in parallel. Instead, they need to be done sequentially.

This realization was brought to my attention by my frequent collaborator Mac Murrett, who called the resulting idea “inverse threads”: set up your two threads so only one is ever running at a particular point in time. The other is always blocked. So while you have some of the benefits of two threads, in particular two unbroken flows of execution, you don't have all of the drawbacks, such as the need to carefully synchronize data access or the need for the code itself to be reentrant.

What You Don't Get

One benefit you *don't* get is one of the key features of Windows fibers, namely lower system overhead. On Windows, threads are managed in the kernel, so the cost of repeated switches from user mode to kernel mode is incurred in an application that employs threads. Real fibers are managed entirely in user mode, so that cost is absent. Our “fibers,” on the other hand, will require all the overhead of normal threads.

Second Thread Requirement

The second realization is that the ported (from case 1) or client (from case 2) code

that invokes the synchronous API always has to be in the second, non-OS thread. The OS thread, which is what stops and resumes OS event handling, has to always be available to do that, and it can't be if it is already involved in the code listed above. This is a fairly crippling restriction, reducing the scenarios where someone can use this technique down pretty much to the two listed above. You can't just drop in this technique into an application that wasn't designed from the ground up for it.

Let's see how this will work:

TADS engine: The TADS engine needs to be embedded in an application specifically written for its Mac OS X port. Therefore, the two-thread technique can be written in to it from the start.

Isthmus Framework: A framework also controls program flow: client code is invoked at particular places, but otherwise the framework is in control, so it can also go ahead and start a second thread at the beginning where those client code access points will be invoked.

Unfortunately, both TADS and Isthmus are too large to include here as examples. Instead, I will present drastically simplified versions of them in Cocoa and Carbon to demonstrate how the techniques work.

Example 1: Atoll

In my first example, I will make a tiny framework – called “Atoll” as the junior cousin to Isthmus, and written, like Isthmus, in Carbon – that starts up the required second thread for client code entry points. Atoll will have one such entry point, called `deactivated()`, which is called when the application is moved into the background. It will also have a synchronous API, called `bounceDockIconTillForemost()`, that only returns once the application is

foremost again. To implement that API, I will need to defer to the main thread for event handling, which would normally require a callback, but which I will make part of the synchronous function.

The client code will then call `bounceDockIconTillForemost()` from within its overridden version of `deactivated()`, and after it returns, show an alert complaining about being sent to the background.

Which Thread API?

First design decision: what sort of threads should I use under Carbon? The latest and greatest Carbon API is Multiprocessing Services, a.k.a. “MPTasks”. This is great for creating preemptive threads. But among Apple’s APIs, I find something older that is in fact much more suitable: the Thread Manager, which creates cooperative threads, as is needed in many Mac OS 9 contexts. In fact, the entire “fiber” approach can be considered a cooperative threading approach, though calling it “the fiber approach” makes it sound cooler. So I’ll use cooperative threads for this exercise.

The Carbon App

I fire up the latest version of Xcode, 1.2, on the latest version of Mac OS X, 10.3.4. I make a new Carbon project, called “HeyYou”. I change the only code file in the project, `main.c`, to `main.cpp` so it can handle C++ code.

Then I make the C++ code and header files for my sole `Atoll` framework class, `Atoll::App`. I call those files `App.cpp` and `App.h` and add them to the project.

I make two public member functions for `Atoll::App`. The first is `run()`; I will call this from `main()` to get things started. The second is the synchronous API `bounceDockIconTillForemost()`.

Then I make the client code entry point, `deactivated()`. It is both virtual and

protected. Lastly, I add an empty virtual destructor, so `Atoll::App` can be safely subclassed.

Now that I’ve written the interface, I get down to the implementation. I move most of the code currently in `main()` to `Atoll::App::run()`, making a few tweaks for C++-style error handling: now the `OSStatus` error code is thrown instead of returned.¹ In its place in `main.cpp`, I put this:

```
#include "App.h"

int main(int argc, char* argv[])
{
    OSStatus status(0);

    try
    {
        Atoll::App app;
        app.run();
    }
    catch(OSStatus exception)
    { status = exception; }

    return status;
}
```

We’ll tweak this code again when we subclass `Atoll::App`.

The Second Thread

Next, I add the code to start up the second thread, for client code access points.

To do this, I need several new declarations in my `Atoll::App` class. I need to be able to refer to both the client code thread and the main thread, I need a function to run in the client code thread, and I need to know what kind of event I’m responding to. So I add a private area to the `Atoll::App` class with the following:

```
ThreadID    mMainThread;
ThreadID    mClientCodeThreadID;

UInt32      mCurrentEventKind;
```

¹ See the sample project for more details.

```
static void *
    ClientCodeThread(void *
                    threadParam);
```

Next, I insert the code to use most of the above items into my `Atoll::App::run()` function, after the existing call to `DisposeNibReference()`:

```
err = GetCurrentThread(
    &mMainThread);
if(err != noErr) throw err;

err =
    NewThread(
        kCooperativeThread,
        NewThreadEntryUPP(
            ClientCodeThread),
        this, // Thread parameter
        0, // Use default stack size
        kCreateIfNeeded,
        NULL, // Disregard result
        &mClientCodeThreadID);
if(err != noErr) throw err;

// Actually start thread.
err = YieldToThread(
    mClientCodeThreadID);
if(err != noErr) throw err;
```

Note error handling here and throughout the code featured in this paper is very basic, and not suitable for anything other than an example. Also, technically, I'm leaking a UPP.

Also note I'm passing in an `Atoll::App` instance pointer as context to the Thread Manager. It will return it to me when it invokes my function. Since my function is static, this is a way to let it call instance methods.

The above code creates a new cooperative thread and saves off the IDs for both the second thread and the main thread. It also starts up the second thread's function. That function's only job is to wait to be yielded to again in order to invoke client code entry points. At this point, the only entry point occurs when the application is moved to the background, so the thread function calls

the client code entry point under that condition:

```
/*static*/ void*
Atoll::App::ClientCodeThread(
    void * threadParam)
{
    Atoll::App *app =
        reinterpret_cast<Atoll::App
*>(threadParam);
    while(true)
    {
        // Yield when first called
        // and after each entry
        // point.
        OSErr err =
            YieldToThread(
                app->mMainThread);
        assert(err == noErr);

        switch(
            app->mCurrentEventKind)
        {
            // Client code entry
            // point.
            case kEventAppDeactivated:
                app->deactivated();
                break;
            default:
                break;
        }
    }
    return NULL;
}
```

Handling Events

Now, I write the code that causes our framework to be told about both deactivate and activate events, so we can yield to the second thread to, respectively, start the `deactivate()` function and finish the `bounceDockIconTillForemost()` function.

I add another private static member function to `Atoll::App` and give it this simple implementation:

```
/*static*/ OSStatus
Atoll::App::HandleEvents(
    EventHandlerCallRef
        eventHandlerCallRef,
    EventRef eventRef,
    void * userData)
{
```

```

#pragma unused(
    eventHandlerCallRef,
    eventRef)

Atoll::App *app =
    reinterpret_cast
        <Atoll::App *>(userData);

app->mCurrentEventKind =
    GetEventKind(eventRef);

return YieldToThread(
    app->mClientCodeThreadID);
}

```

Then I need to register this event handler for just de/activate events, so I add the following code to

`Atoll::App::run()` just above the other code we added for the second thread:

```

EventTypeSpec eventTypeSpec[] =
{
    { kEventClassApplication,
      kEventAppDeactivated },
    { kEventClassApplication,
      kEventAppActivated }
};

err =
    InstallApplicationEventHandler(
        NewEventHandlerUPP(
            (EventHandlerProcPtr)
                HandleEvents),
        GetEventTypeCount(
            eventTypeSpec),
        eventTypeSpec,
        this, // User data
        NULL);
if(err != noErr) throw err;

```

Note that again I'm leaking a UPP, something that should be fixed in real-world code, and that again I'm passing in the application instance as context.

Bouncing the Icon

For the final step in the Atoll implementation, I need to flesh out the `bounceDockIconTillForemost()` API:

```

// Assumes application is in
// background.
void

```

```

Atoll::App::bounceDockIconTillForemost()
{
    NMRec nmRec = {0};
    nmRec.qType = nmType;
    // This makes process icon
    // bounce in Dock on OS X.
    nmRec.nmMark = 1;

    OSStatus err =
        NMInstall(&nmRec);
    if(err != noErr) throw err;

    // Main thread will yield to
    // this thread again when
    // application is brought to
    // front.
    err =
        YieldToThread(mMainThread);
    if(err != noErr) throw err;
}

```

To be thorough, I should check that this is only called from the client thread and when the application is in the background. These checks are left as an exercise for the reader.

Review

Since the Atoll implementation is finished, let's take a moment to review what it does. `run()` starts the second thread, whose function, `ClientCodeThread()`, yields over and over to the main thread each time it is resumed. The main thread yields back under two circumstances:

1. When the application is deactivated. This causes `ClientCodeThread()` to call the `deactivated()` client code entry point.
2. When the application is activated. `ClientCodeThread()` does nothing here, since this case is designed for when the

`bounceDockIconTillForemost()` has yielded to the main thread. That function merely continues, which returns control to the client code that called it. That code eventually returns control to `ClientCodeThread()`, which resumes the main thread again.

"Hey You!": Overriding Atoll

Now, I need to make some client code to actually use this system. I make a new class called `HeyYouApp` that subclasses `Atoll::App`. It overrides the `deactivated()` entry point.

```
#include "App.h"

class HeyYouApp :
    public Atoll::App
{
protected:

    virtual void deactivated();
};
```

I implement `deactivated()` as follows:

```
#include "HeyYouApp.h"

void HeyYouApp::deactivated()
{
    bounceDockIconTillForemost();

    DialogRef alert;
    OSStatus err =
        CreateStandardAlert(
            kAlertStopAlert,
            CFSTR("Hey, you! Don't put
me in the background!"),
            NULL, NULL, &alert);
    if(err != noErr) throw err;

    DialogItemIndex itemIndex;
    err = RunStandardAlert(
        alert, NULL,
        &itemIndex);
    if(err != noErr) throw err;
}
```

The alert I show afterward is merely to demonstrate that program flow only continues once the application is foremost.

I need to modify `main.cpp` to make an instance of `HeyYouApp` instead of `Atoll::App`. After that, I build and run the application.

To test it, after the application is running in the foreground, I click on the Desktop. The `HeyYou` application icon starts bouncing in the Dock. I bring the application to the front again, and see an alert that tells me "Hey you! Don't

put me in the background!". I click its OK button to dismiss it.

Notice that the framework interface available to client code is much simpler than the framework's actual implementation. The client can only override `deactivate()`, and can only call `bounceDockIconTillForemost()`. No threads or callbacks to worry about, no muss, no fuss.

More generally, for some additional complexity in a framework, you can get a much simpler client interface that can be used in many OS X applications.

Example 2: Smidgens

Since a tad is "A small amount or degree", I will call my miniature, sample implementation of TADS an even smaller measurement, "Smidgens".

A Smidgens game is just a dictionary of key-value strings. If you type in a command that matches a key, the matching value is printed as the response. The Smidgens engine is the logic that figures out what value to print and controls the game flow.

Since that's the simplest part of this exercise, first I'll write the Smidgens engine, including the interfaces for the platform-specific APIs that Smidgens will call to do things like ask for the user's input and display the engine's response.

Then I'll "port" Smidgens to Mac OS X by embedding it in a Cocoa application that will handle those platform-dependent tasks.

The Cocoa App

I fire up Xcode again, but this time, I make a document-based Cocoa application. I'll use the default `MyDocument` classes that that project provides for my purposes.

The Smidgen Engine

I make `Smidgens.h` and `Smidgens.m` and add them to the project.

I add the following to `Smidgens.h`:

```
void RunSmidgensEngine(
    NSDictionary *gameEntries,
    void *context);

// Must be implemented by
// platforms:
NSString *GetUserInput(
    void *context);
void ShowEngineResponse(
    NSString *response,
    NSString *input,
    void *context);
```

The first function is the interface to start the Smidgens engine. It has a context pointer that we'll use to pass along our document instance. The second two functions are implemented in my "port", so the `Smidgens.m` code file only contains the first function:

```
void RunSmidgensEngine(
    NSDictionary *gameEntries,
    void *context)
{
    NSString *input = nil;

    while(
        ![input isEqualToString:
            @"quit"])
    {
        input =
            GetUserInput(context);

        if([input length] > 0)
        {
            NSString *output =
                [gameEntries
                 objectForKey:input];
            if(output == nil)
                output =
@"I'm sorry, I didn't understand
that.";

            ShowEngineResponse(
                output, input, context);
        }
    }
}
```

The engine executes, asking for user input, until the user types in "quit". The

engine acquires new user input by calling the platform-dependent function `GetUserInput()`. It then executes the logic to figure out what its response should be. Finally, it calls the function `ShowEngineResponse()` to hand off responsibility for exactly how to show the response to the user to the platform-dependent code again.

The Second Thread, Redux

Starting the second thread and switching between threads in Cocoa requires using different APIs than Carbon. The threads are preemptive. I can't stop them, but I can make them wait by using an `NSConditionalLock`. So I add one of those to my `MyDocument` declaration. I also add some convenience methods to `MyDocument` to make use of the lock. And I add a method that will be run during the thread, the same job that `ClientCodeThread()` performed in the Carbon application. The full declaration, so far, looks like this:

```
#import <Cocoa/Cocoa.h>

@interface MyDocument :
    NSDocument
{
    NSConditionLock *threadLock;
}

- (void)stopForSmidgensThread;
- (void)stopForUI;

- (void)runSmidgensThread:
    (id)dummy;

@end
```

But instead of implementing any of those methods right away, first I look at where and how to start the second thread. It should be after everything in `MyDocument` has been initialized, including its window. The Cocoa project template already includes a method definition for such an entry point, called `windowControllerDidLoadNib`. I change it to look like the following:

```

enum
{
    kUIMayRun = 0,
    kSmidgensThreadMayRun
};

- (void)
windowControllerDidLoadNib:
    (NSWindowController *)
        aController
{
    [super
     windowControllerDidLoadNib:
         aController];

    // Must exist and be locked
    // before thread starts.
    threadLock =
        [[NSConditionLock alloc]
         initWithCondition:kUIMayRun];

    [NSThread
     detachNewThreadSelector:
         @selector(
             runSmidgensThread:)
     toTarget:self
     withObject:nil];

    [self stopForSmidgensThread];
}

```

Notice the enumeration, whose entries describe the two possible conditions: either the UI is running, or the Smidgens engine thread is running.

I make the lock before I make the thread. Then I make the thread, pointing it at the `MyDocument` instance's own `runSmidgensThread:` method. Finally, I defer to the engine thread via one of the thread control utility methods mentioned above. I do this because I know the Smidgens engine will turn around and immediately call a function that resumes the main thread.

Since I've made a lock object that I don't delete anywhere else, I add a new `dealloc` method to handle deleting it:

```

- (void)dealloc
{
    [threadLock release];
    [super dealloc];
}

```

Next, I implement those thread control methods, which are very simple and complementary:

```

- (void)stopForSmidgensThread
{
    [threadLock
     unlockWithCondition:
         kSmidgensThreadMayRun];

    // Wait until Smidgens engine
    // thread changes condition.
    [threadLock
     lockWhenCondition:
         kUIMayRun];
}

- (void)stopForUI
{
    [threadLock
     unlockWithCondition:
         kUIMayRun];

    // Wait until UI/main thread
    // changes condition.
    [threadLock
     lockWhenCondition:
         kSmidgensThreadMayRun];
}

```

The first line of each method releases the pending other thread. The second line forces the current thread to wait. Note a real-world implementation would put in some error-checking code to verify that these messages are only sent from the correct thread.

Finally, I implement the thread method:

```

- (void)runSmidgensThread:
    (id)dummy
{
    // Must manually create pool
    // for a thread.
    NSAutoreleasePool *pool =
        [[NSAutoreleasePool alloc]
         init];

    NSDictionary *gameEntries =
        [[NSDictionary alloc]
         initWithObjectsAndKeys:
             @"It's dark. You might
             be eaten by a grue!", @"north",
             @"A gaping pit blocks
             your way.", @"south",

```

```

        @"There's a towering
cliff.", @"east",
        @"You wander around a
forest, get bored, and come
back.", @"west",
        nil];

RunSmidgensEngine(
    gameEntries, self);

[gameEntries release];

[pool release];
}

```

Since threads don't start with their own autorelease pool, I have to make and manage one manually. After I do that, I make the text entries for the game and fire up the Smidgens engine.

I now have everything written except the platform-dependent implementations of the Smidgens functions `GetUserInput()` and `ShowEngineResponse()`. Doing that will require that I set up the user interface.

The User Interface

For my quick-and-dirty interface, I bring up Interface Builder and open the `MyDocument.nib` file. It already has a window. I make it a bit smaller and add several controls to it:

- An `NSTextView` that fills most of the window, which will hold the Smidgens responses to the user.
- An `NSTextField` for where the user will type his or her input. I make this the window's `initialFirstResponder`.
- A divider (`NSBox`) to separate the two.

`MyDocument` will need to access the first two, so I need to add outlets for that, and `MyDocument` also needs to be able to respond when the user presses the Return button in the input field, so I need an action for that. Here's the new code for the `MyDocument` declaration, with ellipses for code that was already there:

```

@interface MyDocument :
    NSDocument
{
    ...

    IBOutlet NSTextField *
        inputTextField;
    IBOutlet NSTextView *
        outputTextView;
}

...

- (IBAction)acceptUserInput:
    (NSTextField *)sender;

@end

```

I connect all of this together in Interface Builder.

All this new `MyDocument` stuff won't be of any use unless I can tie it back to the Smidgens engine. I add the `GetUserInput()` and `ShowEngineResponse()` definitions to `MyDocument.m`, but how can I tie those global functions to `MyDocument`? The easiest way is to make new `MyDocument` methods to actually perform these tasks, and then send those messages from the global functions. So here are two more declarations for `MyDocument.h`:

```

- (NSString *)getUserInput;
- (void)showEngineResponse:
    (NSString *)response
    input:(NSString *)input;

```

And here are the global functions, sending those messages:

```

NSString *GetUserInput(
    void *context)
{
    return
        [(MyDocument *)context
         getUserInput];
}

void ShowEngineResponse(
    NSString *response,
    NSString *input,
    void *context)
{
    [(MyDocument *)context

```

```

        showEngineResponse:response
        input:input];
}

```

Almost done! The last step is to implement all the remaining methods in MyDocument:

```

- (NSString *)getUserInput
{
    [self stopForUI];

    return
        [inputTextField stringValue];
}

- (IBAction)acceptUserInput:
    (NSTextField *)sender
{
    [self stopForSmidgensThread];
}

- (void)showEngineResponse:
    (NSString *)response
    input:(NSString *)input
{
    [outputTextView insertText:
        [NSString stringWithFormat:
            @">%@\n%@\n\n",
            input, response]];

    [inputTextField
    setStringValue:@""];
}

```

getUserInput, called from the engine thread, yields to the main thread so it can handle user input: keystrokes, blinking the cursor, possibly moving the window around and selecting menus, etc. Control is returned via the second method, acceptUserInput:, which is invoked when the user hits the Return key. getUserInput then returns the string the user typed.

The last method, showEngineResponse:input:, which is invoked from the Smidgens engine once it knows what the response string is, fills in the output view with that string, after formatting it a bit, and clears the input field.

When I build and run this project, it all works just fine. I can type in commands

like “north” and “west”, and I get the desired responses.

The End of the Thread

If I type “quit”, however, the application hangs. Why? Because the Smidgens thread ends without releasing control back to the main thread. I can solve this by adding the following line at the end of the runSmidgensThread: method:

```

[threadLock unlockWithCondition:
    kUIMayRun];

```

However, the window is still there. Ideally, “quit” should close the entire document. There is a standard NSDocument method to do this, close, but that message should be sent from the main thread, not the engine thread. I need to inform the main thread that this step needs to be taken, which means a new flag.

I add the following line to MyDocument.h:

```

bool quitFlag;

```

I then add the following right above the unlockWithCondition: line, which is the last line in the runSmidgensThread: method:

```

quitFlag = YES;

```

Lastly, I change acceptUserInput: to look like this:

```

- (IBAction)acceptUserInput:
    (NSTextView *)sender
{
    [self stopForSmidgensThread];

    if(quitFlag)
        [self close];
}

```

When I rerun the project and type in “quit”, now the document window disappears.

Voila! I now have my own little text adventure game system.

<http://umbar.com> in the future for information about their progress.

Downside #1: Unresponsiveness

Now that I've shown that these techniques are feasible, I'd like to talk about a downside in addition to the ones mentioned at the start of the paper, a downside that may seem obvious in retrospect. While the ported and client code is simplified by not using the full power of multithreading, if that code starts a task that takes too long, the resulting application will seem unresponsive to the user, since the UI is waiting for that code to finish. Is it better to have the application be a little less responsive in exchange for a simpler and therefore possibly less buggy codebase? Developers will have to weigh these considerations based on their specific circumstances.

Downside #2: Exception Handling

The second downside involves exceptions. There is no standard way to transfer an exception from one thread to another. This is especially a problem for the framework scenario: what if `HeyYouApp::deactivated()` wanted to throw an exception to tell the framework code that something had gone wrong? It can't propagate via the normal C++ mechanism.

I have not investigated a solution at the time of this writing, but it will probably involve either limiting which kinds of exceptions can be thrown, or possibly requiring all exceptions to implement a certain interface. Exceptions of that type would then be caught and copied to the other thread.

Conclusion

Simulating fibers on OS X isn't hard. I just did it twice in the course of a few pages! These examples should get you started if you want to implement them for yourself.

Both the Isthmus application framework and the Mac OS X TADS interpreter port are works in progress. Check out