

Running a Child Process with Standard Input and Output

This thread has been locked by a moderator.

Running a child process using `Process` (or `NSTask` in Objective-C) is easy, but piping data to and from the child's `stdin` and `stdout` is surprisingly tricky. I regularly see folks confused by this. Moreover, it's easy to come up with a solution that works most of the time, but suffers from weird problems that only show up in the field [1].

I recently had a couple of DTS incidents from folks struggling with this, so I sat down and worked through the details. Pasted below is the results of that effort, namely, a single function that will start a child process, pass it some data on `stdin`, read the data from the child's `stdout`, and call a completion handler when everything is done.

There are some things to note here, some obvious, some not so much:

- I've included Swift and Objective-C versions of the code. Both versions work the same way. The Swift version has all the comments. If you decide to base your code on the Objective-C version, copy the comments from there.
- I didn't bother collecting `stderr`. That's not necessary in many cases and, if you need it, it's not hard to extend the code to handle that case.
- I use `Dispatch I/O` rather than `FileHandle` to manage the I/O channels. `Dispatch I/O` is well suited to this task. In contrast, `FileHandle` has numerous problems working with pipes. For the details, see [Whither FileHandle?](#).
- This single function is way longer than I'd normally tolerate. This is partly due to the extensive comments and partly due to my desire to maintain focus. When wrapping `Process` it's very easy to run afoul of architecture astronaut-ism. Indeed, I have a much more full-featured `Process` wrapper sitting on my hard disk, but that's going to stay there in favour of this approach (-:
- Handling a child process correctly involves some gnarly race conditions. The code has extensive comments explaining how I deal with those.

If you have any questions or comments about this, put them in a new thread. Make sure to tag that thread with *Foundation* and *Inter-process communication* so that I see it.

Share and Enjoy

—
Quinn "The Eskimo!" @ Developer Technical Support @ Apple
`let myEmail = "eskimo" + "1" + "@" + "apple.com"`

[1] Indeed, [this post](#) shows that I've made this sort of mistake myself)-:

Foundation Inter-process communication

Reply Posted 1 year ago by eskimo

Add a Comment

Replies

```
/// Runs the specified tool as a child process, supplying `stdin` and capturing `stdout`.
///
/// - important: Must be run on the main queue.
///
/// - Parameters:
///   - tool: The tool to run.
///   - arguments: The command-line arguments to pass to that tool; defaults to the empty array.
///   - input: Data to pass to the tool's `stdin`; defaults to empty.
///   - completionHandler: Called on the main queue when the tool has terminated.

func launch(tool: URL, arguments: [String] = [], input: Data = Data(), completionHandler: @escaping CompletionHandler) {
    // This precondition is important; read the comment near the `run()` call to
    // understand why.
    dispatchPrecondition(condition: .onQueue(.main))

    let group = DispatchGroup()
    let inputPipe = Pipe()
    let outputPipe = Pipe()

    var errorQ: Error? = nil
    var output = Data()

    let proc = Process()
    proc.executableURL = tool
    proc.arguments = arguments
    proc.standardInput = inputPipe
    proc.standardOutput = outputPipe
    group.enter()
    proc.terminationHandler = { _ in
        // This bounce to the main queue is important; read the comment near the
        // `run()` call to understand why.
        DispatchQueue.main.async {
            group.leave()
        }
    }

    // This runs the supplied block when all three events have completed (task
    // termination and the end of both I/O channels).
    //
    // - important: If the process was never launched, requesting its
    // termination status raises an Objective-C exception (ouch!). So, we only
    // read `terminationStatus` if `errorQ` is `nil`.

    group.notify(queue: .main) {
        if let error = errorQ {
            completionHandler(.failure(error), output)
        } else {
            completionHandler(.success(proc.terminationStatus), output)
        }
    }

    do {
        func posixErr(_ error: Int32) -> Error { NSError(domain: NSPOSIXErrorDomain, code: Int(error), userInfo: nil) }

        // If you write to a pipe whose remote end has closed, the OS raises a
        // `SIGPIPE` signal whose default disposition is to terminate your
        // process. Helpful! `F_SETNOSIGPIPE` disables that feature, causing
        // the write to fail with `EPIPE` instead.

        let fcntlResult = fcntl(inputPipe.fileHandleForWriting.fileDescriptor, F_SETNOSIGPIPE, 1)
        guard fcntlResult == 0 else { throw posixErr(errno) }

        // Actually run the process.

        try proc.run()

        // At this point the termination handler could run and leave the group
        // before we have a chance to enter the group for each of the I/O
        // handlers. I avoid this problem by having the termination handler
        // dispatch to the main thread. We are running on the main thread, so
        // the termination handler can't run until we return, at which point we
        // have already entered the group for each of the I/O handlers.
        //
        // An alternative design would be to enter the group at the top of this
        // block and then leave it in the error handler. I decided on this
        // design because it has the added benefit of all my code running on the
        // main queue and thus I can access shared mutable state, like `errorQ`,
        // without worrying about thread safety.

        // Enter the group and then set up a Dispatch I/O channel to write our
        // data to the child's `stdin`. When that's done, record any error and
        // leave the group.
        //
        // Note that we ignore the residual value passed to the
        // `write(offset:data:queue:ioHandler:)` completion handler. Earlier
        // versions of this code passed it along to our completion handler but
        // the reality is that it's not very useful. The pipe buffer is big
        // enough that it usually soaks up all our data, so the residual is a
        // very poor indication of how much data was actually read by the
        // client.

        group.enter()
        let writeIO = DispatchIO(type: .stream, fileDescriptor: inputPipe.fileHandleForWriting.fileDescriptor, queue: .main) { _ in
            // `FileHandle` will automatically close the underlying file
            // descriptor when you release the last reference to it. By holding
            // on to `inputPipe` until here, we ensure that doesn't happen. And
            // as we have to hold a reference anyway, we might as well close it
            // explicitly.
            //
            // We apply the same logic to `readIO` below.
            try! inputPipe.fileHandleForWriting.close()
        }
        let inputDD = input.withUnsafeBytes { DispatchData(bytes: $0) }
        writeIO.write(offset: 0, data: inputDD, queue: .main) { isDone, _, error in
            if isDone || error != 0 {
                writeIO.close()
                if errorQ == nil && error != 0 { errorQ = posixErr(error) }
                group.leave()
            }
        }

        // Enter the group and then set up a Dispatch I/O channel to read data
        // from the child's `stdin`. When that's done, record any error and
        // leave the group.

        group.enter()
        let readIO = DispatchIO(type: .stream, fileDescriptor: outputPipe.fileHandleForReading.fileDescriptor, queue: .main) { _ in
            try! outputPipe.fileHandleForReading.close()
        }
        readIO.read(offset: 0, length: .max, queue: .main) { isDone, chunkQ, error in
            output.append(contentsOf: chunkQ ?? .empty)
            if isDone || error != 0 {
                readIO.close()
                if errorQ == nil && error != 0 { errorQ = posixErr(error) }
                group.leave()
            }
        }
    } catch {
        // If either the `fcntl` or the `run()` call threw, we set the error
        // and manually call the termination handler. Note that we've only
        // entered the group once at this point, so the single leave done by the
        // termination handler is enough to run the notify block and call the
        // client's completion handler.
        errorQ = error
        proc.terminationHandler!(proc)
    }

    /// Called when the tool has terminated.
    ///
    /// This must be run on the main queue.
    ///
    /// - Parameters:
    ///   - result: Either the tool's termination status or, if something went
    ///   wrong, an error indicating what that was.
    ///   - output: Data captured from the tool's `stdout`.

    typealias CompletionHandler = (_ result: Result<Int32, Error>, _ output: Data) -> Void
}
```

Posted 1 year ago by eskimo

Add a Comment

```
// IMPORTANT: This is a line-for-line port of the Swift code, so see that code
// for comments explaining what's going on here.

extern void launchToolWithArgumentsAndInput(
    NSURL * toolURL,
    NSArray<NSString *> * arguments,
    NSData * input,
    LaunchToolCompletionHandler completionHandler
) {
    dispatch_queue_t queue = dispatch_get_main_queue();
    dispatch_assert_queue(queue);
    dispatch_group_t group = dispatch_group_create();
    NSPipe * inputPipe = [NSPipe pipe];
    NSPipe * outputPipe = [NSPipe pipe];

    __block NSError * errorQ = nil;
    NSMutableData * output = [NSMutableData data];

    NSTask * task = [[NSTask alloc] init];
    task.executableURL = toolURL;
    task.arguments = arguments;
    task.standardInput = inputPipe;
    task.standardOutput = outputPipe;
    dispatch_group_enter(group);
    task.terminationHandler = ^(NSTask * _) {
        #pragma unused(_)
        dispatch_async(queue, ^{
            dispatch_group_leave(group);
        });
    };

    dispatch_group_notify(group, queue, ^{
        int status = errorQ == nil ? task.terminationStatus : 0;
        completionHandler(errorQ, status, output);
    });

    BOOL success = fcntl(inputPipe.fileHandleForWriting.fileDescriptor, F_SETNOSIGPIPE, 1) == 0;
    if ( ! success ) {
        errorQ = [NSError errorWithDomain:NSPOSIXErrorDomain code:errno userInfo:nil];
    }
    if (success) {
        success = [task launchAndReturnError:&errorQ];
    }
    if ( ! success ) {
        task.terminationHandler(task);
        return;
    }

    dispatch_group_enter(group);
    dispatch_io_t writeIO = dispatch_io_create(DISPATCH_IO_STREAM, inputPipe.fileHandleForWriting.fileDescriptor, queue, ^(int _) {
        #pragma unused(_)
        BOOL s = [inputPipe.fileHandleForWriting closeAndReturnError:NULL];
        assert(s);
    });
    dispatch_data_t inputDD = dispatch_data_create(input.bytes, input.length, NULL, DISPATCH_DATA_DESTRUCTOR_DEFAULT);
    dispatch_io_write(writeIO, 0, inputDD, queue, ^(bool isDone, dispatch_data_t __, int error) {
        #pragma unused(_)
        if (isDone || (error != 0)) {
            dispatch_io_close(writeIO, 0);
            if ((errorQ == nil) && (error != 0)) { errorQ = [NSError errorWithDomain:NSPOSIXErrorDomain code:error userInfo:nil];
        }
        dispatch_group_leave(group);
    });

    dispatch_group_enter(group);
    dispatch_io_t readIO = dispatch_io_create(DISPATCH_IO_STREAM, outputPipe.fileHandleForReading.fileDescriptor, queue, ^(int _) {
        #pragma unused(_)
        BOOL s = [outputPipe.fileHandleForReading closeAndReturnError:NULL];
        assert(s);
    });
    dispatch_io_read(readIO, 0, SIZE_MAX, queue, ^(bool isDone, dispatch_data_t __nullable chunkQ, int error) {
        if (chunkQ != nil) {
            dispatch_data_apply(chunkQ, ^{
                #pragma unused(1, 2)
                [output appendBytes:buffer length:size];
                return true;
            });
        }
        if (isDone || (error != 0)) {
            dispatch_io_close(readIO, 0);
            if ((errorQ == nil) && (error != 0)) { errorQ = [NSError errorWithDomain:NSPOSIXErrorDomain code:error userInfo:nil];
        }
        dispatch_group_leave(group);
    });
}
```

Posted 1 year ago by eskimo

Add a Comment

This site contains user submitted content, comments and opinions and is for informational purposes only. Apple disclaims any and all liability for the acts, omissions and conduct of any third parties in connection with or related to your use of the site. All postings and use of the content on this site are subject to the Apple Developer Forums Participation Agreement.

> Developer > Forums			
Platforms	Topics & Technologies	Resources	Programs
iOS	Accessibility	Documentation	Apple Developer Program
iPadOS	Accessories	Curriculum	Apple Developer Enterprise Program
macOS	App Extensions	Downloads	App Store Small Business Program
tvOS	App Store	Forums	MFI Program
watchOS	Audio & Video	Videos	News Partner Program
	Augmented Reality		Video Partner Program
Tools	Business	Support	Security Bounty Program
Swift	Design	Support Articles	Security Research Device Program
SwiftUI	Distribution	Contact Us	
SF Symbols	Education	Bug Reporting	
Swift Playgrounds	Fonts	System Status	Events
TestFlight	Games		App Accelerators
Xcode	Health & Fitness	Account	App Store Awards
Xcode Cloud	In-App Purchase	Apple Developer	Apple Design Awards
	Localization	App Store Connect	Apple Developer Academies
	Maps & Location	Certificates, IDs, & Profiles	Entrepreneur Camp
	Machine Learning	Feedback Assistant	Tech Talks
	Security		WWDC
	Safari & Web		