MATT GALLAGHER

Gathering system information in Swift with sysctl

March 8, 2016 by Matt Gallagher
Tags: Swift, debug analysis

In the previous article, I looked at gathering stack traces to record what your own process is doing. In debug analysis, though, information about *what* a process has done is only half the picture: we often need to know about the environment in which the process ran to understand *why* the process has behaved a certain way.

In this article, I'll look at gathering a narrow set of basic information about the host system for the purpose of debug analysis. System information can be obtained through a number of different APIs, each with their own advantages, disadvantages and idiosyncracies but I'll be focussing on a core function available across OS X and iOS: `sysctl`. The function itself is cumbersome and full of classic C quirks so I'll also share a Swift wrapper for `sysctl` to make it slightly less irksome.

Contents

Introduction

As with the previous article, this article concerns debug analysis. Specifically, analyzing information about what has happened after a problem occurs to try and determine what

want to look at capturing information about the host system.

To illustrate what I mean, let's look at what's in a typical Mac OS X diagnostic report. We can look at any of the ".diag" files in the "/Library/Logs/DiagnosticReports" folder. A typical dianostic report on my computer contains:

1. The date
2. Name and version information for program that was running when the report was created
3. Specific details about what error or condition triggered the report
4. A stack trace for the program that was running
5. The following information about the host computer…

```
OS Version:     Mac OS X 10.11.3 (Build 15D21)
Architecture:   x86_64
Hardware model:  MacPro4,1
Active cpus:    8
```

This is the host information I want to gather.

NSProcessInfo and UIDevice

Let's look and see if we can gather this information from any common Cocoa location.

The Foundation singleton `NSProcessInfo.processInfo()` has properties `operatingSystemVersionString` and `activeProcessorCount` which could give:

```
OS Version:     Version 10.11.3 (Build 15D21)
Active cpus:    8
```

the iOS-only singleton `UIDevice.currentDevice()` also has `systemName` and `model` which would let you amend that to:

```
OS Version:     iPhone OS Version 9.2.1 (Build 13D20)
Hardware model:  iPhone
Active cpus:    2
```

Unfortunately though, "iPhone" is not a very helpful model description (this ran on my iPhone 6s which has a true model name of "iPhone8,1").

Objective-C/Swift APIs. For this information, we need to look elsewhere.

uname

Cocoa classes don't really help us get the hardware model. Instead, let's turn to a C function named `uname`. Calling `uname` function fills in a struct named `utsname` with the following values:

```
sysname = "Darwin"
nodename = "Matt-Gallaghers-iPhone"
release = "15.0.0"
version = "Darwin Kernel Version 15.0.0: Wed Dec  9 22:19:38 PST 2015; root:xnu-3248.1
machine = "iPhone8,1"
```

We have the full model name. We can combine this with information from `NSProcessInfo` and we have all the basic information we need, right?

Let's try the same thing on a "MacPro4,1" running Mac OS X…

```
sysname = "Darwin"
nodename = "MacPro.local"
release = "15.3.0"
version = "Darwin Kernel Version 15.3.0: Thu Dec 10 18:40:58 PST 2015; root:xnu-3248.1
machine = "x86_64"
```

The model name is gone, replaced instead by the CPU family. So we can't get the "Hardware model" on the Mac using `uname`.

Looking for the source

Why is `uname` inconsistent between platforms? What's happening?

Let's look at where `uname` gets its information and see what's going on. We can view the source code for `uname` on opensource.apple.com. The `machine` field is filled in by the following code:

```
mib[0] = CTL_HW;
mib[1] = HW_MACHINE;
len = sizeof(name->machine);
if (sysctl(mib, 2, &name->machine, &len, NULL, 0) == -1)
   rval = -1;
```

function named `sysctl`.

Of course, `sysctl` isn't the source either. Following that rabbit hole all the way down gives:

1. sysctl gets its information from different OID handlers
2. sysctl_hw_generic handles the information for most of the `CTL_HW` OIDs, including `HW_MACHINE`.
3. PEGetMachineName handles the `HW_MACHINE` OID.
4. Depending on CPU, one of the IOPlatformExpert::getMachineName implementations (essentially a driver for the CPU) will return the machine name.

The value is hardcoded into the `getMachineName` function so this is the true source, although it's largely irrelevant to us since the `sysctl` API remains the final layer that we can easily access.

Does this answer why `uname` is inconsistent between OS X and iOS? Let's have a look at the output from `sysctl` on these platforms:

	OS X	iOS
`HW_MACHINE`	x86_64	iPhone8,1
`HW_MODEL`	MacPro4,1	N71mAP

"sysctl" results for selected "CTL_HW" subkeys on iOS and OS X

That "N71mAP" value is the iPhone's CPU model – not completely the same as "x86_64" for an Intel W3520 but similar. So it looks like the inconsistency is due to `HW_MACHINE` and `HW_MODEL` results from `sysctl` getting swapped around – without access to the source code, I don't if this is a mistake or a deliberate decision (it looks like an accidental mixup) but in any case, the iOS behavior has remained steady since the iOS platform was released.

With this knowledge, we can finally get a "Hardware model" value by using `sysctl` to get the `HW_MODEL` for OS X and `HW_MACHINE` for iOS systems.

What else can `sysctl` do?

For my own purposes, I rarely go much deeper; basic machine and model information is enough to satisfy the diagnostic information needs for which I employ `sysctl`.

only a small fraction of a huge range of values you can get from `sysctl`.

You can see *almost* all of these values on OS X by running `sysctl -A` on the command line. More than 1000 keys and values will be shown.

I say "almost" though because a few keys are not shown. Curiously, there's a handful of values that are hidden from this list by default, including `HW_MODEL` and `HW_MACHINE`. To get the full list of values, you can download the source to the `sysctl` command line tool and on line 992, change the final `0` argument passed to `show_var` to `1`. Running the result gives a few dozen extra values you can query.

On iOS, there are a couple hundred fewer `sysctl` values available (806 on my iPhone versus 1098 on my Mac) with many of the missing values omitted from the hardware – `CTL_HW` – section. While this is annoying, fortunately most of the relevant traits of an iOS system (CPU type, capabilities and clock rate) are locked to the model so it's not a major catastrophe. In any case, be wary of the fact that `sysctl` on iOS may return errors (specifically, a POSIX error 2) for many values that are valid on OS X.

Improving sysctl's interface with a nested set of wrappers

The `sysctl` function itself is not incredibly complicated but it is a little ugly in Swift:

```
public func sysctl(name: UnsafeMutablePointer<Int32>, namelen: u_int,
   oldp: UnsafeMutablePointer<Void>, oldlenp: UnsafeMutablePointer<Int>,
   newp: UnsafeMutablePointer<Void>, newlenp: Int) -> Int32
```

You pass a C array of `Int32` which uniquely identifies the value you're after and you pass a buffer via `oldp` that's `oldlenp` long and the value will be written there (I'm going to completely ignore using `sysctl` to *set* values since it's very rare to do that in an app).

The reason why `sysctl` feels so cumbersome in Swift is:

- Creating an array of `Int32` and passing that by pointer for the first parameter is a nuisance in Swift
- You basically need to call `sysctl` twice: once with `oldp` equal to `nil` to get the size required for the result buffer and then a second time with a properly allocated buffer.

interpret correctly.
- There are a few different ways in which failure can occur and we want to reduce these different ways to idiomatic Swift errors or preconditions.

For these reasons, I use a wrapper around `sysctl` which has the following interface:

```
public func sysctl(levels: [Int32]) throws -> [Int8]
```

This lets you write `let modelAsArrayOfChar = sysctl([CTL_HW, HW_MODEL])` to get the hardware model as an array of `Int8`.

Of course, a `[Int8]` isn't particularly useful so I call this function from inside subsequent functions that further refine the process:

```
public func sysctlString(levels: Int32...) throws -> String
```

This function lets you pass in the `levels` as a comma separated list and converts the result to a regular Swift `String` so we can get the model as a string in a single line: `let modelString = try sysctlAsString(CTL_HW, HW_MODEL)`.

An alternative overload lets you use the `sysctl` names instead of `Int32` identifiers:

```
public func sysctlString(name: String) throws -> String
```

for which we'd write `let modelString = try sysctlAsString("hw.model")`.

There's still more that we can do: we can eliminate the `try`, entirely. Risking fatal errors should always be kept to a minimum but for core `sysctl` values, the error code path is effectively unreachable (see 'Effectively unreachable code paths' in Partial Functions in Swift, Part1) so forcing "no error" with `try!` is a valid approach for these code values.

This then leads to the final wrapper around these functions, a static struct exposing the core values, without error handling:

```
public struct Sysctl {
   public static var model: String
}
```

`HW_MACHINE` and `HW_MODEL` results on iOS so that its behavior is more in line with OS X. The result is that you can use `Sysctl.model` to get the "Hardware model" on either OS X or iOS.

Usage

> The project containing this `sysctl` wrapper is available on github: mattgallagher/CwlUtils.

The CwlSysctl.swift file is fully self-contained so you can just copy the file, if that's all you need. Otherwise, the ReadMe.md file for the project contains detailed information on cloning the whole repository and adding the framework it produces to your own projects.

Conclusion

The `sysctl` function is a fundamental kernel/machine information tool on OS X and iOS. There's more information about network interfaces in SystemConfiguration and there's more about attached hardware devices in the IOKit registry, but these locations don't hold all of the same information that `sysctl` offers (and IOKit isn't available on iOS anyway).

Despite the fundamental nature of this function, its clumsy interface often leaves it as an API of last resort behind Foundation classes and simpler C functions. I hope I've shown that if you're interested in a description of the host's "Hardware model", it should be first choice.

Previous article:
Tracking tasks with stack traces in Swift

Next article:
Errors: unexpected, composite, non-pure, external.