Implementing Your Own Crash Reporter

This thread has been locked by a moderator.

72k

I often get questions about third-party crash reporting. These usually show up in one of two contexts:

- Folks are trying to implement their own crash reporter.
- Folks have implemented their own crash reporter and are trying to debug a problem based on the report it generated.

This is a complex issue and this post is my attempt to untangle some of that complexity.

If you have a follow-up question about anything I've raised here, please put it in a new thread with the *Debugging* tag.

IMPORTANT All of the following is my own direct experience. None of it should be considered official DTS policy. If you have questions that need an official answer, perhaps you're trying to convince your boss that implementing your own crash reporter is a very bad idea :-), open a DTS tech support incident and we can discuss things there.

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

Scope

First, I can only speak to the technical side of this issue. There are other aspects that are beyond my remit:

- I don't work for App Review, and only they can give definitive answers about what will or won't be allowed on the store.
- Implementing your own crash reporter has significant privacy implications.

IMPORTANT If you implement your own crash reporter, discuss the privacy impact with a lawyer.

This post assumes that you are implementing your own crash reporter. A lot of folks use a crash reporter from another third party. From my perspective these are the same thing. If you use a custom crash reporter, you are responsible for its behaviour, both good and bad, regardless of where the actual code came from.

Note If you use a crash reporter from another third party, run the tests outlined in *Preserve the Apple Crash Report* to verify that it's working well.

General Advice

I strongly advise against implementing your own crash reporter. It's very easy to create a basic crash reporter that works well enough to debug *simple* problems. It's impossible to implement a *good* crash reporter, one that's reliable, binary compatible, and sufficient to debug complex problems. The bulk of this post is a low-level explanation of that impossibility.

Rather than attempting the impossible, I recommend that you lean in to Apple's crash reporter. In recent years it's acquired some really cool new features:

- If you're creating an App Store app, the Xcode organiser gives you easy, interactive access to Apple crash reports.
- If you're an enterprise developer, consider switching to Custom App Distribution. This yields all the benefits of App Store distribution without your app being generally available on the store.
- iOS 14 and macOS 12 report crashes in MetricKit.

If you previously dismissed Apple crash reports as insufficient, I encourage you to reconsider that decision.

Why Is This Impossible?

Earlier I said "*It's impossible* to implement a good crash reporter", and I want to explain why I'm confident enough in my conclusions to use that specific word. There are two fundamental problems here:

- On iOS (and the other iOS-based platforms, watchOS and tvOS) your crash reporter must run inside the crashed process. That means it can never be 100% reliable. If the process is crashing then, by definition, it's in an undefined state. Attempting to do real work in that state is just asking for problems [1].
- To get good results your crash reporter must be *intimately* tied to system implementation details. These can change from release to release, which invalidates the assumptions made by your crash reporter. This isn't a problem for the Apple crash reporter because it ships with the system. However, a crash reporter that's built in to your product is always going to be brittle.

I'm speaking from hard-won experience here. I worked for DTS during the PowerPC-to-Intel transition, and saw a lot of folks with custom crash reporters struggle through that process.

Still, this post exists because lots of folks ignore this reality, so the subsequent sections contain advice about specific technical issues.

WARNING Do not interpret any of the following as encouragement to implement your own crash reporter. I strongly advise against that. However, if you ignore my advice then you should at least try to minimise the risk, which is what the rest of this document is about.

[1] On macOS it's possible for your crash reporter to run out of process, just like the Apple crash reporter. However, *possible* is not the same as easy. In fact, running out of process can make things worse. It prevents you from getting critical state for the crashed process without being tightly bound to OS implementation details. It would be nice if Apple provided APIs for this sort of thing, but that's currently not the case.

Preserve the Apple Crash Report

You must ensure that your crash reporter doesn't disrupt the Apple crash reporter. This is important for three reasons:

- Some fraction of your crashes will not be caused by your code but by problems in framework code, and accurate Apple crash reports are critical in diagnosing such issues.
- When dealing with really hard-to-debug problems, you need the more obscure info that's shown in the Apple crash report.
- If you're working with someone from Apple (via a bug report, DTS incident, or whatever), they're going to want an accurate Apple crash report. If your crash reporter is disrupting the Apple crash reporter — either preventing it from generating crash reports entirely [1], or distorting those crash reports — that limits how much they can help you.

To avoid these issues I recommend that you test your crash reporter's impact on the Apple crash reporter. The basic idea is:

1. Create a program that generates a set of specific crashes.
2. Run through each crash.
3. Verify that your crash reporter produces sensible results.
4. Verify that the Apple crash reporter produces the same results as it does without your crash reporter

With regards step 1, your test suite should include:

- An un-handled language exception thrown by your code
- An un-handled language exception thrown by the OS (accessing an `NSArray` out of bounds is an easy way to get this)
- Various machine exceptions (at a minimum, memory access, illegal instruction, and breakpoint exceptions)
- Stack overflow

Make sure to test all of these cases on both the main thread and a secondary thread.

With regards step 4, check that the resulting Apple crash report includes correct values for:

- The exception info
- The crashed thread
- That thread's state
- Any application-specific info, and especially the last exception backtrace

[1] A particularly pathological behaviour here is to end your crash reporter by calling `exit`. This completely suppresses the Apple crash report. Some third-party language runtimes 'helpfully' include such a crash reporter, which makes it very hard to debug problems that occur within your process but outside of that language.

Signals

Many third-party crash reporters use UNIX signals to catch the crash. This is a shame because using Mach exception handling, the mechanism used by the Apple crash reporter, is generally a better option. However, there are two reasons to favour UNIX signals over Mach exception handling:

- On iOS-based platforms your crash reporter must run in-process, and doing in-process Mach exception handling is not feasible.
- Folks are a lot more familiar with UNIX signals. Mach exception handling, and Mach messaging in general, is pretty darned obscure.

If you use UNIX signals for your crash reporter, be aware that this API has some gaping pitfalls. First and foremost, your signal handler can only use *async signal safe* functions [1]. You can find a list of these functions in `sigaction` man page [2] [3].

WARNING This list does not include `malloc`. This means that a crash reporter's signal handler cannot use Objective-C or Swift, as there's no way to constrain how those language runtimes allocate memory. That means you're stuck with C or C++, but even there you have to be careful to comply with this constraint.

The Operative: It's worse than you know.

Captain Malcolm Reynolds: It usually is.

Many crash reports use functions like `backtrace` (see its man page) to get a backtrace from their signal handler. There's two problems with this:

- `backtrace` is not an async signal safe function.
- `backtrace` uses a naive algorithm that doesn't deal well with cross signal handler stack frames [4].

The latter point is particularly worrying, because it hides the identity of the stack frame that triggered the signal.

If you're going to backtrace out of a signal, you must use the crashed thread's state (accessible via the handlers `ucp` parameter) to start your backtrace.

Apropos that, if your crash reporter wants to log the state of the crashed thread, that's the place to get it.

Your signal handler must be prepared to be called by multiple threads. A typical crashing signal (like `SIGSEGV`) is delivered to the thread that triggered the machine exception. While your signal handler is running on that thread, other threads in your process continue to run. One of these threads could crash, causing it to call your signal handler.

It's a good idea to suspend all threads in your process early in your signal handler. However, there's no way to completely eliminate this window.

Note The need to suspend all the other threads in your process is further evidence that sticking to async signal safe functions is required. An unsafe function might depend on a thread you've suspended.

A typical crashing signal is delivered on the thread that triggered the machine exception. If the machine exception was caused by a stack overflow, the system won't have enough stack space to call your signal handler. You can tell the system to switch to an alternative stack (see the discussion of `SA_ONSTACK` in the `sigaction` man page) but that isn't a complete solution (because of the thread issue discussed immediately above).

Finally, there's the question of how to exit from your signal handler. **You must not call** `exit`. There's two problems with doing that:

- `exit` is not async signal safe. In fact, `exit` can run arbitrary code via handlers registered with `atexit`. If you want to exit the process, call `_exit`.
- Exiting the process is a bad idea anyway, because it will prevent the Apple crash reporter from running. This is very poor form. For an explanation as to why, see *Preserve the Apple Crash Report* (above).

A better solution is to *unregister* your signal handler (set it to `SIG_DFL`) and then return. This will cause the crashed process to continue execution, crash again, and generate a crash report via the Apple crash reporter.

[1] While the common signals caught by a crash reporter are not technically async signals (except `SIGABRT`), you still have to treat them as async signals because they can occur on any thread at any time.

[2] It's reasonable to extend this list to other routines that are implemented as thin shims on a system call. For example, I have no qualms about calling `vm_read` (see below) from a signal handler.

[3] Be aware, however, that even this list has caveats. See my Async Signal Safe Functions vs Dyld Lazy Binding post for details.

[4] Cross signal handler stack frames are pushed on to the stack by the kernel when it runs a signal handler on a thread. As there's no API to learn about the structure of these frames, there's no way to backtrace across one of these frames in isolation. I'm happy to go into details but it's really not relevant to this discussion [5]. If you're interested, start a new thread with the *Debugging* tag and we can chat there.

[5] (Ang, my footnotes have footnotes!) The exception to this is where you're trying to generate a crash report for code running in a signal handler. That's not easy, and frankly you're better off avoiding signal handlers in general. Where possible, handle signals via a Dispatch event source.

Reading Memory

A signal handler must be very careful about the memory it touches, because the contents of that memory might have been corrupted by the crash that triggered the signal. My general rule here is that the signal handler can safely access:

- Its code
- Its stack (subject to the constraints discussed earlier)
- Its arguments
- Immutable global state

In the last point, I'm using *immutable* to mean *immutable after startup*. It's reasonable to set up some global state when the process starts, before installing your signal handler, and then rely on it in your signal handler.

Changing any global state after the signal handler is installed is dangerous, and if you need to do that you must be careful to ensure that your signal handler sees consistent state, even though a crash might occur halfway through your change.

You can't protect this global state with a mutex because mutexes are not async signal safe (and even if they were you'd deadlock if the mutex was held by the thread that crashed). You should be able to use atomic operations for this, but atomic operations are notoriously hard to use correctly (if I had a dollar for every time I've pointed out to a developer they're using atomic operations incorrectly, I'd be very badly paid (<: but that's still a lot of developers!).

If your signal handler reads other memory, it must take care to avoid crashing while doing that read. There's no BSD-level API for this [1], so I recommend that you use `vm_read`.

[1] The traditional UNIX approach for doing this is to install a signal handler to catch any memory access exceptions triggered by the read, but now we're talking signal handling within a signal handler and that's just silly.

Writing Files

If your want to write a crash report from your signal handler, you must use low-level UNIX APIs (`open`, `write`, `close`) because only those low-level APIs are documented to be async signal safe. You must also set up the path in advance because the standard APIs for determining where to write the file (`NSFileManager`, for example) are not async signal safe.

Offline Symbolication

Do not attempt to do symbolication from your signal handler. Rather, write enough information to your crash report to support offline symbolication. Specifically:

- The addresses to symbolicate
- For each Mach-O image in the process:
  - The image path
  - The image UUID
  - The image load address

You can get most of the Mach-O image information using the APIs in `<mach-o/dyld.h>` [1]. Be aware, however, that these APIs are not async signal safe. You'll need to get this information in advance and cache it for your signal handler to access.

This is complicated by the fact that the list of Mach-O images can change as you process loads and unloads code. This requires you to share mutable state with your signal handler, which is exactly what I recommend against in *Reading Memory*.

Note You can learn about images loading and unloading using `_dyld_register_func_for_add_image` and `_dyld_register_func_for_remove_image` respectively.

[1] I believe you'll need to parse the Mach-O load commands to get the image UUID.

What to Include

When deciding what to include in a crash report, there's a three-way balance to be struck:

1. The more information you include, the easier it is to diagnose problems.
2. Some information is hard to obtain, either because there's no public API to get that information, or because the API is not available to your crash reporter.
3. Some information is so privacy-sensitive that it has no place in a crash report.

Apple's crash reporter strikes its own balance here, and I recommend that you try to include everything that it includes, subject to the limitations described in point 2.

Here's what I'd considered to be a minimal list:

- Information about the machine exception that triggered the crash
- For memory access exceptions, the address of the access that triggered the crash
- Backtraces of all the threads (sometimes the backtrace of a non-crashing thread can yield critical information about the crash)
- The crashed thread
- Its thread state
- A list of Mach-O images, as discussed in the *Offline Symbolication* section

IMPORTANT Make sure you report the thread backtraces in a consistent order. Without that it's hard to correlate information across crash reports.

Change History

- 2019-02-12 — First posted.
- 2019-02-13 — Made minor editorial changes. Added a new footnote to the *Signals* section.
- 2019-02-14 — Clarified the complexities of an out-of-process crash reporter. Added the *What to Include* section. Enhanced the *Signals* section to cover reentrancy and stack overflow. Made minor editorial changes.
- 2019-02-15 — Expanded the introduction to the *Preserve the Apple Crash Report* section.
- 2019-05-13 — Added a reference to my Async Signal Safe Functions vs Dyld Lazy Binding post.
- 2021-02-27 — Fixed the formatting. Made minor editorial changes.
- 2021-09-10 — Expanded the *General Advice* section to include pointers to Apple crash report resources, including MetricKit. Split the second half of that section out in to a new *Why Is This Impossible?* section. Made minor editorial changes.
- 2022-05-16 — Fixed a broken link.

Debugging

Reply

Posted 3 years ago by eskimo

Add a Comment