Timeouts and cancellation for humans

Thu 11 January 2018

[Psst: want to come work on NumPy with me? We're hiring!]

Your code might be perfect and never fail, but unfortunately the outside world is less reliable. Sometimes, other people's programs crash or freeze. Networks go down; printers catch on fire. Your code needs to be prepared for this: every time you read from the network, attempt to acquire an inter-process lock, or send an HTTP request, there are at least three possibilities you need to think about:

- It might succeed.
- It might fail
- It might hang forever, never succeeding or failing: days pass, leaves fall, winter comes, yet still our request waits, yearning for a response that will never come.

The first two are straightforward enough. To handle that last case, though, you need timeouts. Pretty much every place your program interacts with another program or person or system, it needs a timeout, and if you don't have one, that's a latent bug.

Let's be honest: if you're like most developers, your code probably has *tons* of bugs caused by missing timeouts. Mine certainly does. And it's weird – since this need is so ubiqituous, and so fundamental to doing I/O correctly, you'd think that every programming environment would provide easy and robust ways to apply timeouts to arbitrary operations. But... they don't. In fact, most timeout APIs are so tedious and error-prone that it's just not practical for developers to reliably get this right. So don't feel bad – it's not your fault your code has all those timeout bugs, it's the fault of those I/O libraries!

But now I'm, uh, writing an I/O library. And not just any I/O library, but one whose whole selling point is that it's obsessed with being easy to use. So I wanted to make sure that in my library – Trio – you can easily and reliably apply timeouts to arbitrary I/O operations. But designing a user-friendly timeout API is a surprisingly tricky task, so in this blog post I'm going to do a deep dive into the landscape of possible designs – and in particular the many precursors that inspired me – and then explain what I came up with, and why I think it's a real improvement on the old state-of-the-art. And finally, I'll discuss how Trio's ideas could be applied more broadly, and in particular, I'll demonstrate a prototype implementation for good old synchronous Python.

So – what's so hard about timeout handling?

Contents:

Simple timeouts don't support abstraction

The simplest and most obvious way to handle timeouts is to go through each potentially-blocking function in your API, and give it a `timeout` argument. In the Python standard library you'll see this in APIs like `threading.Lock.acquire`:

```
lock = threading.Lock()

# Wait at most 10 seconds for the lock to become available
lock.acquire(timeout=10)
```

If you use the `socket` module for networking, it works the same way, except that the timeout is set on the socket object instead of passed to every call:

```
sock = socket.socket()

# Set the timeout once
sock.settimeout(10)
# Wait at most 10 seconds to establish a connection to the remote host
sock.connect(...)
# Wait at most 10 seconds for data to arrive from the remote host
sock.recv(...)
```

This is a little more convenient than having to remember to pass in explicit timeouts every time (and we'll discuss the convenience issue more below) but it's important to understand that this is a purely cosmetic change. The semantics are the same as we saw with `threading.Lock`: each method call gets its own separate 10 second timeout.

So what's wrong with this? It seems straightforward enough. And if we always wrote code directly against these low level APIs, then it would probably be sufficient. But – programming is about abstraction. Say we want to fetch a file from S3. We might do that with boto3, using S3.Client.get_object. What does `S3.Client.get_object` do? It makes a series of HTTP requests to the S3 servers, by calling into the requests library for each one. And then each call to `requests` internally makes a series of calls to the `socket` module to do the actual network communication [1].

From the user's point of view, these are three different APIs that fetch data from a remote service:

```
s3client.get_object(...)
requests.get("https://...")
sock.recv(...)
```

Sure, they're at different levels of abstraction, but the whole idea of abstracting away such details is that the user doesn't have to care. So if our plan is to use `timeout=` arguments everywhere, then we should expect these each to take a `timeout=` argument:

```
s3client.get_object(..., timeout=10)
requests.get("https://...", timeout=10)
sock.recv(..., timeout=10)
```

Now here's the problem: if this is how we're doing things, then actually implementing these functions is a pain in the butt. Why? Well, let's take a simplified example. When processing HTTP response, there comes a point when we've seen the `Content-Length` header, and now we need to read that many bytes to fetch the actual response body. So somewhere inside `requests` there's a loop like:

```
def read_body(sock, content_length):
    body = bytearray()
    while len(body) < content_length:
        max_to_receive = content_length - len(body)
        body += sock.recv(max_to_receive)
    assert len(body) == content_length
    return body
```

Now we'll modify this loop to add timeout support. We want to be able to say "I'm willing to wait at most 10 seconds to read the response body". But we can't just pass the timeout argument through to `recv`, because imagine the first call to `recv` takes 6 seconds – now for our overall operation to complete in 10 seconds, our second `recv` call has to be given a timeout of 4 seconds. With the `timeout=` approach, every time we pass between levels of abstraction we need to write some annoying gunk to recalculate timeouts:

```
def read_body(sock, content_length, timeout):
    read_body_deadline = timeout + time.monotonic()
    body = bytearray()
    while len(body) < content_length:
        max_to_receive = content_length - len(body)
        recv_timeout = read_body_deadline - time.monotonic()
        body += sock.recv(max_to_receive, timeout=recv_timeout)
    assert len(body) == content_length
    return body
```

(And even this is actually simplified because we're pretending that `sock.recv` takes a `timeout` argument – if you wanted to this for real you'd have to call `settimeout` before every socket method, and then probably use some `try`/`finally` thing to set it back or else risk confusing some other part of your program.)

In practice, nobody does this – all the higher-level Python libraries I know of that take `timeout=` arguments, just pass them through unchanged to the lower layers. And this breaks abstraction. For example, here are two popular Python APIs you might use today, and they look like they take similar `timeout=` arguments:

```
import threading
lock = threading.Lock()
lock.acquire(timeout=10)

import requests
requests.get("https://...", timeout=10)
```

But in fact these two `timeout=` arguments mean totally different things. The first one means "try to acquire the lock, but give up after 10 seconds". The second one means "try to fetch the given URL, but give up if at any point any individual low-level socket operation takes more than 10 seconds". Probably the whole reason you're using `requests` is that you don't want to think about low-level sockets, but sorry, you have to anyway. In fact it is currently **not possible** to guarantee that `requests.get` will return in **any** finite time: if a malicious or misbehaving server sends at least 1 byte every 10 seconds, then our `requests` call above will keep resetting its timeout over and over and never return.

I don't mean to pick on `requests` here – this problem is everywhere in Python APIs. I'm using `requests` as the example because Kenneth Reitz is famous for his obsession with making its API as obvious and intuitive as possible, and this is one of the rare places where he's failed. I think this is the only part of the requests API that gets a big box in the documentation warning you that it's counterintuitive. So like... if even Kenneth Reitz can't get this right, I think we can conclude that "just slap a `timeout=` argument on it" does not lead to APIs fit for human consumption.

Absolute deadlines are composable (but kinda annoying to use)

If `timeout=` arguments don't work, what can we do instead? Well, here's one option that some people advocate. Notice how in our `read_body` example above, we converted the incoming relative timeout ("10 seconds from the moment I called this function") into an absolute deadline ("when the clock reads 12:01:34.851"), and then converted back before each socket call. This code would get simpler if we wrote the whole API in terms of `deadline=` arguments, instead of `timeout=` arguments. This makes things simple for library implementors, because you can just pass the deadline down your abstraction stack:

```
def read_body(sock, content_length, deadline):
    body = bytearray()
    while len(body) < content_length:
        max_to_receive = content_length - len(body)
        body += sock.recv(max_to_receive, deadline=deadline)
    assert len(body) == content_length
    return body

 # Wait 10 seconds total for the response body to be downloaded
 deadline = time.monotonic() + 10
 read_body(sock, content_length, deadline)
```

(A well-known API that works like this is Go's socket layer.)

But this approach also has a downside: it succeeds in moving the annoying bit out of the library internals, and and instead puts it on the person using the API. At the outermost level where timeout policy is being set, your library's users probably want to say something like "give up after 10 seconds", and if all you take is a `deadline=` argument then they have to do the conversion by hand every time. Or you could have every function take both `timeout=` and `deadline=` arguments, but then you need some boilerplate in every function to normalize them, raise an error if both are specified, and so forth. Deadlines are an improvement over raw timeouts, but it feels like there's still some missing abstraction here.

Cancel tokens

Cancel tokens encapsulate cancellation state

Here's the missing abstraction: instead of supporting two different arguments:

```
# What users do:
requests.get(..., timeout=...)
# What libraries do:
requests.get(..., deadline=...)
# How we implement it:
def get(..., deadline=None, timeout=None):
    deadline = normalize_deadline(deadline, timeout)
    ...
```

we can encapsulate the timeout expiration information into an object with a convenience constructor:

```
class Deadline:
    def __init__(self, deadline):
        self.deadline = deadline

def after(timeout):
    return Deadline(time.monotonic() + timeout)

# Wait 10 seconds total for the URL to be fetched
requests.get("https://...", deadline=after(10))
```

That looks nice and natural for users, but since it uses an absolute deadline internally, it's easy for library implementors too.

And once we've gone this far, we might as well make things a bit more abstract. After all, a timeout isn't the only reason you might want to give up on some blocking operation; "give up after 10 seconds have passed" is a special case of "give up after <some arbitrary condition becomes true>". If you were using `requests` to implement a web browser, you'd want to be able to say "start fetching this URL, but give up when the 'stop' button gets pressed". And libraries mostly treat this `Deadline` object as totally opaque in any case – they just pass it through to lower-level calls, and trust that eventually some low-level primitives will interpret it appropriately. So instead of thinking of this object as encapsulating a deadline, we can start thinking of it as encapsulating an arbitrary "should we give up now" check. And in honor of its more abstract nature, instead of calling it a `Deadline` let's call this new thing a `CancelToken`:

```
# This library is only hypothetical, sorry
from cancel_tokens import cancel_after, cancel_on_callback

# Returns an opaque CancelToken object that enters the "cancelled"
# state after 10 seconds.
cancel_token = cancel_after(10)
# So this request gives up after 10 seconds
requests.get("https://...", cancel_token=cancel_token)

# Returns an opaque CancelToken object that enters the "cancelled"
# state when the given callback is called.
cancel_callback, cancel_token = cancel_on_callback()
# Arrange for the callback to be called if someone clicks "stop"
stop_button.on_press = cancel_callback
# So this request gives up if someone clicks 'stop'
requests.get("https://...", cancel_token=cancel_token)
```

So promoting the cancellation condition to a first-class object makes our timeout API easier to use, and *at the same time* makes it dramatically more powerful: now we can handle not just timeouts, but also arbitrary cancellations, which is a very common requirement when writing concurrent code. (For example, it lets us express things like: "run these two redundant requests in parallel, and as soon as one of them finishes then cancel the other one".) This is a *great* idea. As far as I know, it originally comes from Joe Duffy's cancellation tokens work in C#, and Go context objects are essentially the same idea. Those folks are pretty smart! In fact, cancel tokens also solve some other problems that show up in traditional cancellation systems.

Cancel tokens are level-triggered and can be scoped to match your program's needs

In our little tour of timeout and cancellation APIs, we started with timeouts. If you start with cancellation instead, then there's another common pattern you'll see in lots of systems: a method that lets you cancel a single thread (or task, or whatever your framework uses as a thread-equivalent), by waking it up and throwing in some kind of exception. Examples include asyncio's

Task.cancel, Curio's Task.cancel, pthread cancellation, Java's Thread.interrupt, C#'s Thread.Interrupt, and so forth. In their honor, I'll call this the "thread interrupt" approach to cancellation.

In the thread-interrupt approach, cancellation is a point-in-time *event* that's directed at a *fixed-size entity*: one call → one exception in one thread/task. There are two issues here.

The problem with scale is fairly obvious: if you have a single function you'd like to call normally *but* you might need to cancel it, then you have to spawn a new thread/task/whatever just for that:

```
http_thread = spawn_new_thread(requests.get, "https://...")
# Arrange that http_thread.interrupt() will be called if someone
# clicks the stop button
stop_button.on_click = http_thread.interrupt
try:
    http_response = http_thread.wait_for_result()
except Interrupted:
    ...
```

Here the thread isn't being used for concurrency; it's just an awkward way of letting you delimit the scope of the cancellation.

Or, what if you have a big complicated piece of work that you want to cancel – for example, something that internally spawns multiple worker threads? In our example above, if `requests.get` spawned some additional backgrounds threads, they might be left hanging when we cancel the first thread. Handling this correctly would require some complex and delicate bookkeeping.

Cancel tokens solve this problem: the work they cancel is "whatever the token was passed into", which could be a single function, or a complex multi-tiered set of thread pools, or anything in between.

The other problem with the thread-interrupt approach is more subtle: it treats cancellation as an *event*. Cancel tokens, on the other hand, model cancellation as a *state*: they start out in the uncancelled state, and eventually transition into the cancelled state.

This is subtle, but it makes cancel tokens less error-prone. One way to think of this is the edge-triggered/level-triggered distinction: thread-interrupt APIs provide edge-triggered notification of cancellations, as compared to level-triggered for cancel tokens. Edge-triggered APIs are notoriously tricky to use. You can see an example of this in Python's threading.Event: even though it's called "event", it actually has an internal boolean state; cancelling a cancel token is like setting an Event.

That's all pretty abstract. Let's make it more concrete. Consider the common pattern of using a `try`/`finally` to make sure that a connection is shut down properly. Here's a rather artificial example of a function that makes a Websocket connection, sends a message, and then makes sure to close it, regardless of whether `send_message` raises an exception: [2]

```
def send_websocket_messages(url, messages):
    open_websocket_connection(url)
    try:
        for message in messages:
            ws.send_message(message)
    finally:
        ws.close()
```

Now suppose we start this function running, but at some point the other side drops off the network and our `send_message` call hangs forever. Eventually, we get tired of waiting, and cancel it.

With a thread-interrupt style edge-triggered API, this causes the `send_message` call to immediately raise an exception, and then our connection cleanup code automatically runs. So far so good. But here's an interesting fact about the websocket protocol: it has a "close" message you're supposed to send before closing the connection. In general this is a good thing; it allows for cleaner shutdowns. So when we call `ws.close()`, it'll try to send this message. But... in this case, the reason we're trying to close the connection is because we've given up on the other side accepting any new messages. So now `ws.close()` also hangs forever.

If we used a cancel token, this doesn't happen:

```
def send_websocket_messages(url, messages, cancel_token):
    open_websocket_connection(url, cancel_token=cancel_token)
    try:
        for message in messages:
            ws.send_message(message, cancel_token=cancel_token)
    finally:
        ws.close(cancel_token=cancel_token)
```

Once the cancel token is triggered, then *all* future operations on that token are cancelled, so the call to `ws.close` doesn't get stuck. It's a less error-prone paradigm.

It's kind of interesting how so many older APIs could get this wrong. If you follow the path we did in this blog post, and start by thinking about applying a timeout to a complex operation composed out of multiple blocking calls, then it's obvious that if the first call uses up the whole timeout budget, then any future calls should fail immediately. Timeouts are naturally level-triggered. And then when we generalize from timeouts to arbitrary cancellations, the insight carries over. But if you only think about timeouts for primitive operations then this never arises; or if you start with a generic cancellation API and then use it to implement timeouts (like e.g. Twisted and asyncio do), then the advantages of level-triggered cancellation are easy to miss.

Cancel tokens are unreliable in practice because humans are lazy

So cancel tokens have really great semantics, and are certainly better than raw timeouts or deadlines, but they still have a usability problem: to write a function that supports cancellation, you have to accept this boilerplate argument and then make sure to pass it on to every subroutine you call. And remember, a correct and robust program has to support cancellation in *every function that ever does I/O, anywhere in your stack*. If you ever get lazy and leave it out, or just forget to pass it through to any particular subroutine call, then you have a latent bug.

Humans suck at this kind of boilerplate. I mean, not you, I'm sure you're a very diligent programmer who makes sure to implement correct cancellation support in every function and also flosses every day. But... perhaps some of your co-workers are not so diligent? Or maybe you depend on some library that someone else wrote – how much do you trust your third-party vendors to get this right? As the size of your stack grows then the chance that everyone everywhere always gets this right approaches zero.

Can I back that up with any real examples? Well, consider this: in both C# and Go, the most prominent languages that use this approach and have been advocating it for a number of years, the underlying networking primitives *still do not have cancel token support* [3]. These are like... THE fundamental operations that might hang for reasons outside your control and that you need to be prepared to time out or cancel, but... I guess they just haven't gotten around to implementing it yet? Instead their socket layers support an older mechanism for setting timeouts or deadlines on their socket objects, and if you want to use cancel tokens you have to figure out how to bridge between the two different systems yourself.

The Go standard library does provide one example of how to do this: their function for establishing a network connection (basically the equivalent of Python's `socket.connect`) does accept a cancel token. Implementing this requires 40 lines of source code, a background task, and the first try had a race condition that took a year to be discovered in production. So... in Go if you want to use cancel tokens (or `Context`s, in Go parlance), then I guess that's what you need to implement every time you use any socket operation? Good luck?

I don't mean to make fun. This stuff is hard. But C# and Go are huge projects maintained by teams of highly-skilled full-time developers and backed by Fortune 50 companies. If they can't get it right, who can? Not me. I'm one human trying to reinvent I/O in Python. I can't afford to make things that complicated.

Cancel scopes: Trio's human-friendly solution for timeouts and cancellation

Remember way back at the beginning of this post, we noted that Python socket methods don't take individual timeout arguments, but instead let you set the timeout once on the socket so it's implicitly passed to every method you call? And in the section just above, we noticed that C# and Go do pretty much the same thing? I think they're on to something. Maybe we should accept that when you have some data that has to be passed through to every function you call, that's something the computer should handle, rather than making flaky humans do the work – but in a general way that supports complex abstractions, not just sockets.

How cancel scopes work

Here's how you impose a 10 second timeout on an HTTP request in Trio:

```
# The primitive API:
with trio.open_cancel_scope() as cancel_scope:
    cancel_scope.deadline = trio.current_time() + 10
    await request.get("https://...")
```

Of course normally you'd use a convenience wrapper, like:

```
# An equivalent but more idiomatic formulation:
with trio.move_on_after(10):
    await requests.get("https://...")
```

But since this post is about the underlying design, we'll focus on the primitive version. (Credit: the idea of using `with` blocks for timeouts is something I first saw in Dave Beazley's Curio, though I changed a bunch. I'll hide the details in a footnote: [4].)

You should think of `with open_cancel_scope()` as creating a cancel token, but it doesn't actually expose any `CancelToken` object publically. Instead, the cancel token is pushed onto an invisible internal stack, and automatically applied to any blocking operations called inside the `with` block. So `requests` doesn't have to do anything to pass this through – when it eventually sends and receives data over the network, those primitive calls will automatically have the deadline applied.

The `cancel_scope` object lets us control cancellation status: you can change the deadline, issue an explicit cancellation by calling `cancel_scope.cancel()`, and so forth. If you know C#, it's analogous to a CancellationTokenSource. One useful trick it allows is implementing the kind raise-an-error-if-the-timeout-fires API that people are used to, on top of the more primitive cancel scope unwinding semantics.

When an operation is cancelled, it raises a `Cancelled` exception, which is used to unwind the stack back out to the appropriate `with open_cancel_scope` block. Cancel scopes can be nested; `Cancelled` exceptions know which scope triggered them, and will keep propagating until they reach the corresponding `with` block. (As a consequence, you should always let the Trio runtime take care of raising and catching `Cancelled` exceptions, so that it can properly keep track of these relationships.)

Supporting nesting is important because some operations may want to use timeouts internally as an implementation detail. For example, when you ask Trio to make a TCP connection to a hostname that has multiple IP addresses associated with it, it uses a "happy eyeballs" algorithm to run multiple connections attempts in parallel with a staggered start. This requires an internal timeout to decide when it's time to initiate the next connection attempt. But users shouldn't have to care about that! If you want to say "try to connect to `example.com:443`, but give up after 10 seconds", then that's just:

```
with trio.move_on_after(10):
    tcp_stream = await trio.open_tcp_stream("example.com", 443)
```

And everything works; thanks to the cancel scope nesting rules, it turns out `open_tcp_stream` handles this correctly with no additional code.

Where do we check for cancellation?

Writing code that's correct in the face of cancellation can be tricky. If a `Cancelled` exception were to suddenly materialize in a place the user wasn't prepared for it – perhaps when their code was half-way through manipulating some delicate data structure – it could corrupt internal state and cause hard-to-track-down bugs. On the other hand, a timeout and cancellation system doesn't do much good if you don't notice cancellations relatively promptly. So an important challenge for any system is to first pick a "goldilocks rule" that checks often enough, but not too often, and then somehow communicate this rule to users so that they can make sure their code is prepared.

In Trio's case, this is pretty straightforward. We already, for other reasons, use Python's async/await syntax to annotate blocking functions. The main thing does is let you look at the text of any function and immediately see which points might block waiting for something to happen. Example:

```
async def user_defined_function():
    print("Hello!")
    await trio.sleep(1)
    print("Goodbyte!")
```

Here we can see that the call to `trio.sleep` blocks, because it has the special `await` keyword. You can't call `trio.sleep` – or any other of Trio's built-in blocking primitives – without using this keyword, because they're marked as async functions. And then Python enforces that if you want to use the `await` keyword, then you have to mark the calling function as async as well, which means that all *callers* of `user_defined_function` will also use the `await` keyword. This makes sense, since if `user_defined_function` calls a blocking function, that makes it a blocking function too. In many other systems, whether a function might block is something you can only determine by examining all of its potential callees, and all their callees, etc.; async/await takes this global runtime property and makes it visible at a glance in the source code.

Trio's cancel scopes then piggy-back on this system: we declare that whenever you see an `await`, that's a place where you might have to handle a `Cancelled` exception – either because it's a call to one of Trio's primitives which directly check for cancellation, or because it's a call to a function that indirectly calls one of those primitives, and thus might see a `Cancelled` exception come bubbling out. This has several nice properties. It's extremely easy to explain to users. It covers all the functions where you absolutely need timeout/cancellation support to avoid infinite hangs – only functions that block can get stuck blocking forever. It means that any function that does I/O on a regular basis also automatically checks for cancellation on a regular basis, so most of the time you don't need to worry about this (though for the occasional long-running pure computation, you may want to add some explicit cancellation checks by calling `await` trio.sleep(0) – which you have to do anyway to let the scheduler work!). Blocking functions tend to have a large variety of failure modes, so in many cases any cleanup required to handle Cancelled exceptions will be shared with that needed to handle, for example, a misbehaving network peer. And Trio's cooperative multi-tasking system also uses the await points to mark places where the scheduler might switch to another task, so you already have to be careful about leaving data structures in inconsistent states across an await. Cancellation and async/await go together like peanut butter and chocolate.

An escape hatch

While checking for cancellation at all blocking primitive calls makes a great default, there are some very rare cases where you want to disable this and take explicit control over cancellation. They're so rare that I don't have a simple example to use here (though there are a few arcane examples in the Trio source that you can grep for if you're really curious). To provide this escape hatch, you can set a cancel scope to "shield" its contents from outside cancellations. It looks like this:

```
with trio.move_on_after(10):
    with trio.open_cancel_scope() as inner_scope:
        inner_scope.shield = True
        # Sleeps for 20 seconds, ignoring the overall 10 second
        # timeout
        await trio.sleep(20)
```

To support composition, shielding is sensitive to the cancel scope stack: it only blocks outer cancel scopes from applying, and has no effect on inner scopes. In our example above, our shield doesn't have any affect on any cancel scopes that might be used *inside* trio.sleep – those still behave normally. Which is good, because whatever trio.sleep does internally is its own private implementation detail. And in fact, trio.sleep *does* use a cancel scope internally! [5]

One reason that shield is an attribute on cancel scopes instead of having a special "shield scope" is that it makes it convenient to implement this kind of nesting, because we can re-use cancel scope's existing stack structure. The other reason is that anywhere you're disabling external timeouts, you need to think about what you're going to do instead to make sure things can't hang forever, and having a cancel scope right there makes it easy to apply a new timeout that's under the local code's control:

```
# Demonstrating that the shielding scope can be used to avoid hangs
# after disabling outside timeouts:
with trio.move_on_after(10) as outer_scope:
    with trio.move_on_after(15) as inner_scope:
        inner_scope.shield = True
        # Returns after 15 seconds, when the shielding scope expires:
        await trio.sleep(1000000)
```

Now if you're a Trio user please forget you read this section; if you think you need to use shielding then you almost certainly should rethink what you're trying to do. But if you're an I/O runtime implementer looking to add cancel scope support, then this is an important feature.

Cancel scopes and concurrency

Finally, there's one more feature of Trio that should be mentioned here. So far in this essay, I haven't discussed concurrency much at all; timeouts and cancellation are largely independent, and everything above applies even to straightforward single-threaded synchronous code. But we did make some assumptions that might seem trivial: that if you call a function inside a with block, then (a) the execution will actually happen inside the with block, and (b) any exceptions it throws will propagate back to the with block so it can catch them. Unfortunately, many threading and concurrency libraries violate this, specifically in the case where some work is spawned or scheduled:

```
# This looks innocent enough:
with move_on_after(10):
    do_the_thing()

# But it isn't:
def do_the_thing():
    # Using some made-up API similar to what most systems use:
    start_task_in_background(some_worker_that_will_actually_do_the_thing)
```

If we were only looking at the `with` block alone, this would seem perfectly innocent. But when we look at how `do_the_thing` is implemented, we realize that it's likely that we'll exit the `with` block before the background task finishes, so there's some ambiguity: should the timeout apply to the background task or not? And then if it does apply, then how should we handle the `Cancelled` exception? For most system, unhandled exceptions in background threads/tasks are simply discarded.

However, these problems don't arise in Trio, because of its unique approach to concurrency. Trio's nursery system means that child tasks are always integrated into the call stack, which effectively becomes a call tree. Concretely, the way this is enforced is that Trio has no global `start_task_in_background` primitive; instead, if you want to spawn a child task, you have to first open a "nursery" block (for the child to live in, get it?), and then the lifetime of that child is tied to the `with` block that created the nursery:

```
with move_on_after(10):
    await do_the_thing()

async def do_the_thing():
    async with trio.open_nursery() as nursery:
        nursery.start_soon(some_worker_that_will_actually_do_the_thing)
        # Now the 'async with' block won't complete until the
        # child task has finished, and if the child has an unhandled
        # exception then it will be re-raised here in the parent.
        # Which makes this example pretty silly -- the "background
        # task" acts just like a function call. Which is the point :-)
```

This system has many advantages, but the relevant one here is that it preserves the key assumptions that cancel scopes rely on. Any given nursery is either inside or outside the cancel scope – we can tell by checking whether the `with open_cancel_scope` block encloses the `async with open_nursery` block. And then it's straightforward to say that if a nursery is inside a cancel scope, then that scope should apply to all children in that nursery. This means that if we apply a timeout to a function, it can't "escape" by spawning a child task – the timeout applies to the child task too. (The exception is if you pass an outside nursery into the function, then it can spawn tasks into that nursery, which can escape the timeout. But then this is obvious to the caller, because they have to provide the nursery – the point is to make it clear what's going on, not to make it impossible to spawn background tasks.)

Summary

Returning to our initial example: I've been doing some initial work on porting `requests` to run on Trio (you can help!), and so far it looks like the Trio version will not only handle timeouts better than the traditional synchronous version, but that it will be able to do this using *zero lines of code* – all the places where you'd want to check for cancellation are the ones where Trio does so automatically, and all the places where you need special care to handle the resulting exceptions are places where `requests` is prepared to handle arbitrary exceptions for other reasons.

There are no free lunches; cancellation handling can still be a source of bugs, and requires care when writing code. But Trio's cancel scopes are dramatically easier to use – and therefore more reliable – than any other system I've found. Hopefully we can make timeout bugs the exception rather than the rule.

Who else can benefit from cancel scopes?

So... that's great if you're using Trio. Is this something that only works in Trio's context, or is it more general? What kind of adaptations would need to be made to use this in other environments?

If you want to implement cancel scopes, then you'll need:

- Some kind of implicit context-local storage to track the cancel scope stack. If you're using threads, then thread-local storage works; if you're using something more exotic, then you'll need to figure out the equivalent in your system. (So for example, in Go you'd need goroutine-local storage, which famously doesn't exist.) This can be a bit tricky; for example in Python, we need something like PEP 568 to iron out some bad interactions between cancel scopes and generators.
- A way to delimit the boundaries of a cancel scope. Python's `with` blocks work great; other options would include dedicated syntax, or restricting cancel scopes to individual function calls like `with_timeout(10, some_fn, arg1, arg2)` (though this could force awkward factorings, and you'd need to figure out some way to expose the cancel scope object).
- A strategy for unwinding the stack back to the appropriate cancel scope after a timeout/cancellation occurs. Exceptions work great, so long as you have a way to catch them at cancel scope boundaries – this is another reason that Python's `with` blocks work so well for this. But if your language uses, say, error code returns instead of exceptions, then I'm sure you could build some stack unwinding convention out of those.
- A story for how cancel scopes integrate with your concurrency API (if any). Of course the ideal is something like Trio's nursery system (which also has many other advantages, but that's a whole 'nother blog post). But even without that, you could for example deem that any new tasks spawned inside a cancel scope inherit that cancel scope, regardless of when they finish. (Unless they opt out using something like the shielding feature.)
- Some rule to determine which operations are cancellable and communicate that to the user. As noted above, async/await works perfectly for this, but if you aren't using async/await then other conventions are certainly possible. Languages with rich static type systems might be able to exploit them somehow. Worst case you could just be careful to document it on each function.
- Cancel scope integration for all of the blocking I/O primitives you care about. This is reasonably straightforward if you're building a system from scratch. Async systems have an advantage here because integrating everything into an event loop already forces you to reimplement all your I/O primitives in some uniform way, which gives you an excellent opportunity to add uniform cancellation handling at the same time.

Synchronous, single-threaded Python

Our original motivating examples involved `requests`, an ordinary synchronous library. And pretty much everything above applies equally to synchronous or concurrent code. So I think it's interesting to explore the idea of using these in classic synchronous Python. Maybe we can fix `requests` so it doesn't have to apologize for its `timeout` argument!

There are a few limitations we'll have to accept:

- It won't be ubiquitous – libraries will have to make sure that they only use "scope-enabled" blocking operations. Perhaps in the long run we could imagine this becoming part of the standard library and integrated into all the standard primitives, but even then there will still be third-party extension libraries that do their own I/O without going through the standard library. On the other hand, a library like `requests` can be careful to only use scope-enabled libraries, and then document that it itself is scope-enabled. (This is perhaps the biggest advantage an async library like Trio has when it comes to timeouts and cancellation: being async doesn't make a difference per se, but an async library is forced to reimplement all the basic I/O primitives to integrate them into its I/O loop; and if you're reimplementing everything *anyway*, it's easy to make cancellation support consistent.)
- There's no marker like `await` to show which operations are cancellable. This means that users will have to take somewhat more care and check the documentation for individual functions – but that's still less work then what it currently takes to make timeouts work right.
- Python's underlying synchronous primitives generally only support cancellation due to timeouts, not arbitrary events, so we probably can't provide a `cancel_scope.cancel()` operation. But this limitation doesn't seem too onerous, because if you have a single-threaded synchronous program and the single thread is stuck in some blocking operation, then who's going to call `cancel()` anyway?

Summing up: it can't be quite as nice as what Trio provides, but it'd still be pretty darn useful, and certainly nicer than what we have now.

If this sounds interesting to you, check out the proof-of-concept that I implemented.

asyncio

One of the original motivations for this blog post was talking to Yury about whether we could retrofit any of Trio's improvements back into asyncio. Looking at asyncio through the lens of the above analysis, a few things jump out at us:

- There's some impedence mismatch between the cancel scope model of implicit stateful arbitrarily-scale cancel tokens, and asyncio's current task-oriented, edge-triggered cancellation (and then the `Future`s layer has a slightly different cancellation model again), so we'd need some story for how to meld those together. Or maybe it would be possible to migrate `Task`s to a stateful cancellation model?
- Without nurseries, there's no reliable way to propagate cancellation across tasks, and there are a lot of different operations that are sort of like spawning a task but at a different level of abstraction (e.g. `loop.call_soon`). You could have a rule that any new tasks always inherit their spawner's cancel scopes, but I'm not sure whether this would be a good idea or not – it needs some thought.
- Without a generic mechanism for propagating exceptions back up the stack, there's no way to reliably route `Cancelled` exceptions back to the original scope; generally asyncio simply prints and discards unhandled exceptions from `Task`s. Maybe that's fine?

Unfortunately asyncio's in a bit of a tricky position, because it's built on an architecture derived from the previous decade of experience with async I/O in Python... and then after that architecture was locked in, it added new syntax to Python that invalidated all that experience. But hopefully it's still possible to adapt some of these lessons – at least with some compromises.

Other languages

If you're working in another language, I'd love to hear how the cancel scope idea adapts – if at all. For example, it'll definitely need some adjustment for languages that don't use exceptions, or that are missing the kind of user-extensible syntax that Python's `with` blocks provide.

Now go forth and fix your timeout bugs!

Or if you want to read more about Trio, we have a friendly tutorial that people seem to like.

[1] In fact I'm glossing over several layers of abstraction here: it's really more like boto3 → botocore → requests → urllib3 → http.client → socket.

[2] In real life we'd probably use a `with` statement here, like:

```
def send_websocket_messages(url, messages):
    with open_websocket_connection(url) as ws:
        for message in messages:
            ws.send_message(message)
```

This makes the problem even *harder* to see, because now the nasty `ws.close` call that makes our program hang is entirely invisible.

[3] Incredibly, C#'s high-level async networking functions actually accept cancel token arguments and then ignore them.

[4] Curio's timeouts are derived from a thread-interrupt style cancellation model (similar to Java/C#'s `Thread.interrupt`), so timeout expiration is edge-triggered, didn't handle nesting at all until I complained about it to Dave, and only applies to the current task, not any child tasks that it might have spawned. Trio's cancel scopes are basically Curio's timeout blocks + C#'s cancel tokens + a more straightforward nesting model + shielding + nursery-based concurrency to make child tasks respect stack discipline. Keep reading to learn what all of these things are :-).

[5] Possibly interesting context: in other systems, it's common to have some kind of "call later" primitive, that schedules some code to run at a particular time. (This is a special case of the general "callback pattern" of registering some arbitrary code to run when a certain event occurs.) In those systems, you might expect cancel scope deadlines to be implemented with something like `call_later(deadline, scope.cancel())`. But one of Trio's core design principles is to reject the whole callback paradigm, on the grounds that it's a disguised way of spawning background tasks, and we think concurrency is hard enough without disguised background tasks. So in Trio, the way you implement `call_later`-like functionality is to spawn a task, and then have it sleep until the given time. Event notification is always done by waking up a task, not spawning a new one. And what *this* means is that cancel scope deadlines are actually Trio's core primitive for timekeeping! All other time-based operations like `sleep` are implemented on top of cancel scope deadlines.

Posted by Nathaniel J. Smith Thu 11 January 2018

Recent Posts

- Timeouts and cancellation for humans
- Control-C handling in Python and Trio
- Announcing Trio
- Why does calloc exist?
- Some thoughts on asynchronous API design in a post-async/await world

Social

- Atom
- Twitter
- Mastodon / GNU Social

Copyright © 2016–2018 Nathaniel J. Smith — Powered by Pelican