Appropriately using Ruby's Thread.handle_interrupt

August 31, 2023

Working on GoodJob, I spend a lot of time thinking about multithreaded behavior in Ruby. One piece of Ruby functionality that I don't see written about very often is `Thread.handle_interrupt`. *Big thanks to John Bachir and Matthew Draper for talking through its usage with me.*

Some background about interrupts: In Ruby, exceptions can be raised anywhere and at anytime in a thread by other threads (including the main thread, that's how `timeout` works). Even `rescue` and `ensure` blocks can be interrupted. Everywhere. Most of the time this isn't something you need to think about (unless you're using Timeout or `rack-timeout` or doing explicit multithreaded code). But if you are, it's important to think and code defensively.

Starting with an example:

```
thread = Thread.new do
  open_work
  do_work
ensure
  close_work
end

# wait some time
thread.kill # or thread.raise
```

In this example, it's possible that the exception raised by `thread.kill` will interrupt the middle of the `ensure` block. That's bad! And can leave that work in an inconsistent state.

Ruby's `Thread.handle_interrupt` is the defensive tool to use. It allows for modifying when those interrupting exceptions are raised:

```
# :immediate is the default and will interrupt immediately
Thread.handle_interrupt(Exception: :immediate) { close_work }

# :on_blocking interrupts only when the GVL is released
# e.g. IO outside Ruby
Thread.handle_interrupt(Exception: :on_blocking) { close_work }

# :never will never interrupt during that block
Thread.handle_interrupt(Exception: :never) { close_work }
```

`Thread.handle_interrupt` will modify behavior for the duration of the block, and it will then raise the interrupt after the block exits. It can be nested too:

```
Thread.handle_interrupt(Exception: :on_blocking) do
  ruby_stuff

  Thread.handle_interrupt(Exception: :never) do
   really_important_work
  end

  file_io # <= this can be interrupted
end
```

FYI BE AWARE: Remember, the interrupt behavior is only affected *within* the `handle_interrupt` block. The following code **has a problem**:

```
thread = Thread.new do
  open_work
  do_work
ensure
  Thread.handle_interrupt(Exception: :never) do
    close_work
  end
end
```

Can you spot it? It's right here:

```
ensure
  # <- Interrupts can happen right here
  Thread.handle_interrupt(Exception: :never) do
```

There's a "seam" right there between the `ensure` and the `Thread.handle_interrupt` where interrupts *can happen!* Sure, it's probably rare that an interrupt would hit right then and there, but if you went to the trouble to guard against it, it's likely very bad if it did happen. And it happens! "Why puma workers constantly hung, and how we fixed by discovering the bug of Ruby v2.5.8 and v2.6.6"

HOW TO USE IT APPROPRIATELY: This is the pattern you likely want:

```
thread = Thread.new do
  Thread.handle_interrupt(Exception: :never) do
    Thread.handle_interrupt(Exception: :immediately) do
      open_work
```

```
      do_work
    end
  ensure
    close_work
  end
end
```

That's right: have the `ensure` block nested within the outer `Thread.handle_interrupt(Exception: :never)` so that interrupts cannot happen in the `ensure`, and then use a second `Thread.handle_interrupt(Exception: :immediately)` to allow the interrupts to take place in the code *before* the `ensure` block.

There's another pattern you might also be able to use with `:on_blocking`:

```
thread = Thread.new do
  Thread.handle_interrupt(Exception: :on_blocking) do
    open_work
    do_work
  ensure
    Thread.handle_interrupt(Exception: :never) do
      close_work
    end
  end
end
```

Doesn't that have the problematic seam? Nope, because when under `:on_blocking` there isn't an operation taking place right there would release the GVL (e.g. no IO).

But it does get tricky if, for example, the `do_work` is some Ruby calculation that is unbounded (I dunno, maybe a problematic regex or someone accidentally decided to calculate prime numbers or something). Then the Ruby code will not be interrupted at all and your thread will hang. That's bad too. So you'd then need to do something like this:

```ruby
thread = Thread.new do
  Thread.handle_interrupt(Exception: :on_blocking) do
    open_work
    Thread.handle_interrupt(Exception: :immediately) do
      do_work
    end
  ensure
    Thread.handle_interrupt(Exception: :never) do
      close_work
    end
  end
end
```

See, it's ok to nest `Thread.handle_interrupt` and likely *necessary* to achieve the safety and defensiveness you're expecting.