Ruby trunk - Bug #15645

It is possible to escape `Mutex#synchronize` without releasing the mutex

03/06/2019 06:28 PM - jneen (Jeanine Adkisson)

Status:	Open		
Priority:	Normal		
Assignee:			
Target version:			
ruby -v:	ruby 2.6.1p33 (2019-01-30 revision 66950) [x86_64-linux]	**Backport:**	2.4: UNKNOWN, 2.5: UNKNOWN, 2.6: UNKNOWN

Description

Hello, I hope this finds you well.

I have a persistent deadlocking issue in a project that relies both on Mutex#synchronize and Thread#raise, and I believe I have reduced the problem to the following example, in which it is possible to exit a synchronize block without releasing the mutex.

```
mutex = Mutex.new
class E < StandardError; end

t1 = Thread.new do
  10000.times do
    begin
      mutex.synchronize do
        puts 'acquired'
        # sleep 0.01
        raise E if rand < 0.5
        puts 'releasing'
      end
    rescue E
      puts "interrupted"
    end

    puts "UNRELEASED MUTEX" if mutex.owned?
  end
end

t2 = Thread.new do
  1000.times do
    mutex.synchronize { sleep 0.01 }
    sleep 0.01
    t1.raise(E)
  end
end

t3 = Thread.new do
  1000.times do
    mutex.synchronize { sleep 0.01 }
    sleep 0.01
    t1.raise(E)
  end
end

t2.join
t3.join
```

I would expect mutex.owned? to always return false outside of the synchronize { ... } block, but when I run the above script, I see the following output:

```
; ruby tmp/testy.rb
acquired
interrupted
interrupted
```

```
UNRELEASED MUTEX
#<Thread:0x00005577aaa07860@tmp/testy.rb:4 run> terminated with exception (report_on_
exception is true):
Traceback (most recent call last):
        3: from tmp/testy.rb:5:in `block in <main>'
        2: from tmp/testy.rb:5:in `times'
        1: from tmp/testy.rb:7:in `block (2 levels) in <main>'
tmp/testy.rb:7:in `synchronize': deadlock; recursive locking (ThreadError)
```

I do not fully understand why this is possible, and it is possible there is a simpler example that would reproduce the issue. But it seems at least that it is necessary for two different threads to be running Thread#raise simultaneously.

Occasionally, especially if the timing of the sleep calls are tuned, the thread t1 will display an stack trace for an error E - which I believe is the expected behavior in the case that the error is raised during its rescue block.

Thank you for your time!

History

#1 - 03/09/2019 06:50 PM - lexi.lambda (Alexis King)

I know very little about Ruby internals, and I haven't worked with Ruby in years, but I saw a link to this bug and got pretty fascinated trying to understand what was going on. I'm pretty sure I've spotted the problem, though I haven't actually tested if I'm right.

Here's what I think is happening: Mutex#synchronize is implemented very simply, by calling rb_mutex_lock followed by a call to rb_ensure that sets up rb_mutex_unlock as an ensure handler before invoking the provided block:

```
VALUE
rb_mutex_synchronize(VALUE mutex, VALUE (*func)(VALUE arg), VALUE arg)
{
    rb_mutex_lock(mutex);
    return rb_ensure(func, arg, rb_mutex_unlock, mutex);
}
```

Naturally, this can go wrong if the stack is unwound *after* the mutex has been locked, but *before* control has left rb_mutex_lock, since the ensure handler has not yet been installed. If we peek inside the implementation of do_mutex_lock, we find that this is, indeed, possible. If a thread fails to acquire the mutex, it goes to sleep, and when it wakes up and acquires the GVL, it locks the mutex for itself (assuming the mutex hasn't been locked by some other thread in the meantime). However, before control leaves rb_mutex_lock, it catastrophically checks for interrupts! At the time of this writing, this happens on lines 287–293 of thread_sync.c:

```
if (interruptible_p) {
    RUBY_VM_CHECK_INTS_BLOCKING(th->ec); /* may release mutex */
    if (!mutex->th) {
        mutex->th = th;
        mutex_locked(th, self);
    }
}
```

The comment in question is right in that interrupt handlers may release the mutex, so it defensively reacquires it if control returns and the mutex is unlocked, but this problem is caused by an inverse situation: control *doesn't* return (the stack has been unwound), but the thread still holds the lock.

Therefore, the solution seems simple: **instead of using rb_mutex_lock inside of rb_mutex_synchronize, use mutex_lock_uninterruptible, which sets interruptible_p to FALSE and avoids the issue entirely.**

As a final note, it's possibly worth saying that this issue was introduced in commit 3586c9e087, which was part of Ruby 2.5.0. Earlier versions of Ruby did not have this bug. However, the program in the original bug report will still crash on versions of Ruby without the bug, and it would be correct to do so, since an exception can be delivered outside the extent of the rescue. Here's a modified version of the program that I believe avoids all nondeterministic behavior by using Thread.handle_interrupt and waiting until exceptions are masked before starting the raising thread. It's also slightly simpler, in that only one thread is competing for the mutex and only one is raising exceptions. On correct Ruby implementations, the following program should never terminate:

```
@mutex = Mutex.new
class E < StandardError; end

Thread.new do
  loop do
    @mutex.synchronize {}
  end
end

def start_raising_thread
  Thread.new do
    loop do
```

```
      @t.raise E
      Thread.pass
    end
  end
end

@t = Thread.new do
  Thread.handle_interrupt(E => :never) do
    start_raising_thread
    loop do
      begin
        Thread.handle_interrupt(E => :immediate) do
          @mutex.synchronize {}
        end
      rescue E
      end

      raise "UNRELEASED MUTEX" if @mutex.owned?
    end
  end
end

@t.join
```