Thread Safe Deferred Callback

Asynchronous programming involves being able to execute functions at a future occurrence of some event. Designing a thread-safe deferred callback class becomes a challenging interview question and we will learn about it in this
lesson .

Thread Safe Deferred Callback

Design and implement a thread-safe class that allows registeration of callback methods that are executed after a user specified time interval in seconds has elapsed.

Solution

Let us try to understand the problem without thinking about concurrency. Let's say our class exposes an API called `registerAction()` that'll take a parameter `action`, which will get executed after user specified seconds. Anyone calling this API should be able to specify after how many seconds should our class invoke the passed-in action.

One naive way to solve this problem is to have a busy thread that continuously loops over the list of actions and executes them as they become due. However, the challenge here is to design a solution that doesn't involve busy-waiting.

One possible solution is to have an execution thread that maintains a priority queue (min-heap) of actions ordered by the time remaining to execute each of the actions. The execution thread can sleep for the duration equal to the time duration before the earliest action in the min-heap becomes due for execution.

Consumer threads can come and add their desired actions in the min-heap within the critical section. The caveat here is that the execution thread will need to be woken up to recalculate the minimum duration it would sleep for before an action is due for execution. An action with an earlier due timestamp might have been added while the executor thread was sleeping on a duration calculated for an action due later than the one just added.

Consider this example: initially, the execution thread is sleeping for 30 mins before any action in the min-heap is due. A consumer thread comes along and adds an action to be executed after 5 minutes. The execution thread would need to wake up and reset itself to sleep for only 5 minutes instead of 30 minutes. Once we find an elegant way of achieving this, our problem is pretty much solved.

Let's see what the skeleton of our class would look like:

```ruby
class DeferredCallbackExecutor

  def initialize
    # Priority queue sorted by the earliest execution time
    @actions = PQueue.new {|a,b| b.executeAt > a.executeAt}
    @lock = Mutex.new
    @newCallbackArrived = ConditionVariable.new
  end

  # the method that is signaled and executes the due callbacks
  def start
    while true
    end
  end

  def registerAction(action)
  end
```

Note that we'll not focus on writing our own version of priority queue; instead, we borrow an existing implementation. When defining our priority queue `actions`, we also pass in the comparator that is used to sort actions based on when they will get executed. Each action has a instance field `executeAt` which helps us compare two actions:

instance field executeAt which helps us compare two actions.

```
@actions = PQueue.new { |a, b| b.executeAt > a.executeAt }
```

For any two actions a and b, the action with an earlier (numerically smaller) value for the `executeAt` field is ranked higher in the min-heap.

We also define a simple `DeferredAction` class, an object of which will be passed into the `registerAction()` method.

For guarding access to critical sections, we'll use an object of class `Mutex`. For coordination among threads, we'll use an object of class `ConditionVariable` which consumer threads can `signal()` to let the executor thread know that a new action has been added. Without the use of a condition variable, the execution thread would busy wait. The execution thread will `wait()` on the condition variable while the consumer threads will `signal()` it. Let's write out what we just discussed as code.

```
class DeferredCallbackExecutor

  def initialize
    # our array that acts as a priority queue
    @actions = PQueue.new {|a,b| b.executeAt) > a.executeAt) }
    @lock = Mutex.new
    @newCallbackArrived = ConditionVariable.new
  end

  # the method that is signaled and executes the due callbacks
  def start
    while true
    end
  end

  # register callback by pushing in the priority queue
  def registerAction(action)
    @lock.synchronize do
      @actions.push(action)
      @newCallbackArrived.signal()
    end
  end

end
```

```
class DeferredAction

  def initialize(executeAfter, message)
    @executeAt = DateTime.now.strftime("%s").to_i + executeAfter.to_i
    @message = message
  end

  attr_reader :executeAt
  attr_reader :message
end
```

Note in `registerAction()` method we acquire the mutex lock to protect the critical section which adds an action to the queue. Remember each condition variable has an associated lock and in order to `signal()` the condition variable `newCallbackArrived` we must acquire the `lock` mutex object, which we correctly do. Note the execution thread won't wake up from the `wait()` call until the consumer thread gives up the mutex associated with the condition variable, even though the condition variable has been signaled/notified.

Now let's come to the meat of our solution which is to design the execution thread's workflow. The thread will run the `start()` method and enter into a perpetual loop. The flow should be as follows:

- Initially the queue will be empty and the execution thread should just wait indefinitely on the condition variable to be notified.

- When the first callback arrives, we note how many seconds after its arrival does it need to be invoked and `wait()` on the condition variable for that many seconds. This time we use a variant of the wait method that takes in the numbers of seconds to wait.

- Now two things are possible at this point. No new actions arrive, in which case the executor thread completes waiting and polls the queue for tasks that should be executed and starts executing them.

  Or that another action arrives, in which case the consumer thread would signal the condition variable to wake up the execution thread and have it re-evaluate the duration it can sleep for before the earliest callback becomes due.

earliest callback becomes due.

This flow is captured in the code below:

```
  def start
    # sleepFor is re-calculated everytime a new action is registered
    @sleepFor = 0
    while true
      @lock.synchronize do

        # if the queue is empty, wait for items to be populated
        while (@actions.size == 0)
          @newCallbackArrived.wait(@lock)
        end

        # if the queue is not empty, execute the earliest due action
        while (@actions.size != 0)
          nextAction = @actions.top

          # calculate sleepFor by deducting the current time from th
e initialized
          # executeAt time of the action
          @sleepFor = (nextAction.executeAt).to_i - DateTime.now.strf
time("%s").to_i

          if (@sleepFor <= 0)
            break
          end

          # wait until either a new action registers or the sleepin
g duration completes
          @newCallbackArrived.wait(@lock, @sleepFor)
        end

        # remove the action executed
        actionToExecute = @actions.pop
        p "Action required to be execute at #{actionToExecute.execute
At} and executed at #{DateTime.now.strftime("%s").to_i} with messag
e: #{actionToExecute.message}"
      end
    end
  end
```

Pay attention to how we are using `while` loop to wrap the `wait()` method invocation on the condition variables! The idiomatic pattern to `wait()` on a condition variable requires us to check for the associated predicate in a while loop. Note that even in the case of a spurious wakeup the executor thread will either go back to waiting (first while loop) or recalculate a non-zero time to sleep before an action is due for execution and go back to waiting on the condition variable for that long.

The working of the above snippet is explained below:

- Initially, the queue (actions) is empty and the executor thread will simply `wait()` indefinitely on the condition variable to be notified. We correctly wrap the waiting in a while loop to conform to the idiomatic usage of condition variables.

- If the queue is not empty, say if the executor thread is created later than the consumer threads, then the executor thread will fall into the second while loop. If it's already time to execute an action, it'll immediately break-out of the loop or else sleep for time duration before the first action becomes due.

- For all other happy path cases, adding an action to the queue would always notify the awaiting executor thread to wake up and start processing the queue.

The complete code appears below in the code widget:

main.rb

pqueue.rb

```
# Priority queue with array based heap.
#
# A priority queue is like a standard queue, except that each inserted
# elements is given a certain priority, based on the result of the
# comparison block given at instantiation time. Also, retrieving an element
# from the queue will always return the one with the highest priority
# (see #pop and #top).
#
# The default is to compare the elements in repect to their #<=> method.
# For example, Numeric elements with higher values will have higher
# priorities.
#
```

```
# Note that as of version 2.0 the internal queue is kept in the reverse order
# from how it was kept in previous version. If you had used #to_a in the
# past then be sure to adjust for the priorities to be ordered back-to-front

# instead of the oterh way around.
#
class PQueue

  #
  VERSION = "2.1.0"  #:erb: VERSION = "<%= version %>"

  #
  # Returns a new priority queue.
  #
  # If elements are given, build the priority queue with these initial
  # values. The elements object must respond to #to_a.
  #
  # If a block is given, it will be used to determine the priority between
  # the elements. The block must must take two arguments and return `1`, `0`,
  # or `-1` or `true`, `nil` or `false. It should return `0` or `nil` if the
  # two arguments are considered equal, return `1` or `true` if the first
  # argument is considered greater than the later, and `-1` or `false` if
  # the later is considred to be greater than the first.
  #
  # By default, the priority queue retrieves maximum elements first
  # using the #<=> method.
  #
  def initialize(elements=nil, &block) # :yields: a, b
    @que = []
    @cmp = block || lambda{ |a,b| a <=> b }
    replace(elements) if elements
  end

  protected

  #
  # The underlying heap.
  #
  attr_reader :que #:nodoc:

  public

  #
  # Priority comparison procedure.
  #
  attr_reader :cmp

  #
  # Returns the size of the queue.
  #
  def size
    @que.size
  end

  #
  # Alias of size.
  #
  alias length size

  #
  # Add an element in the priority queue.
  #
  def push(v)
    @que << v
    reheap(@que.size-1)
    self
  end

  #
  # Traditional alias for #push.
  #
  alias enq push

  #
  # Alias of #push.
  #
  alias :<< :push

  #
  # Get the element with the highest priority and remove it from
  # the queue.
  #
  # The highest priority is determined by the block given at instantiation
  # time.
  #
  # The deletion time is O(log n), with n is the size of the queue.
  #
  # Return nil if the queue is empty.
  #
  def pop
    return nil if empty?
    @que.pop
  end

  #
  # Traditional alias for #pop.
  #
  alias deq pop

  # Get the element with the lowest priority and remove it from
  # the queue.
  #
  # The lowest priority is determined by the block given at instantiation
  # time.
  #
  # The deletion time is O(log n), with n is the size of the queue.
  #
  # Return nil if the queue is empty.
  #
  def shift
    return nil if empty?
    @que.shift
  end

  #
  # Returns the element with the highest priority, but
  # does not remove it from the queue.
  #
  def top
    return nil if empty?
    return @que.last
  end

  #
  # Traditional alias for #top.
```

```
  #
  alias peek top

  #
  # Returns the element with the lowest priority, but
  # does not remove it from the queue.
  #
  def bottom
    return nil if empty?
    return @que.first
  end

  #
  # Add more than one element at the same time. See #push.
  #
  # The elements object must respond to #to_a, or be a PQueue itself.
  #
  def concat(elements)
    if empty?
      if elements.kind_of?(PQueue)
        initialize_copy(elements)
      else
        replace(elements)
      end
    else
      if elements.kind_of?(PQueue)
        @que.concat(elements.que)
        sort!
      else
        @que.concat(elements.to_a)
        sort!
      end
    end
    return self
  end

  #
  # Alias for #concat.
  #
  alias :merge! :concat

  #
  # Return top n-element as a sorted array.
  #
  def take(n=@size)
    a = []
    n.times{a.push(pop)}
    a
  end

  #
  # Returns true if there is no more elements left in the queue.
  #
  def empty?
    @que.empty?
  end

  #
  # Remove all elements from the priority queue.
  #
  def clear
    @que.clear
    self
  end

  #
  # Replace the content of the heap by the new elements.
  #
  # The elements object must respond to #to_a, or to be
  # a PQueue itself.
  #
  def replace(elements)
    if elements.kind_of?(PQueue)
      initialize_copy(elements)
    else
      @que.replace(elements.to_a)
      sort!
    end
    self
  end

  #
  # Return a sorted array, with highest priority first.
  #
  def to_a
    @que.dup
  end

  #
  # Return true if the given object is present in the queue.
  #
  def include?(element)
    @que.include?(element)
  end

  #
  # Push element onto queue while popping off and returning the next element.
  # This is qquivalent to successively calling #pop and #push(v).
  #
  def swap(v)
    r = pop
    push(v)
    r
  end

  #
  # Iterate over the ordered elements, destructively.
  #
  def each_pop #:yields: popped
    until empty?
      yield pop
    end
    nil
  end

  #
  # Pretty inspection string.
  #
  def inspect
    "<#{self.class}: size=#{size}, top=#{top || "nil"}>"
  end

  #
  # Return true if the queues contain equal elements.
```

```
  #
  def ==(other)
    size == other.size && to_a == other.to_a
  end

  private

  #
  #
  #
  def initialize_copy(other)
    @cmp  = other.cmp
    @que  = other.que.dup
    sort!
  end

  #
  # The element at index k will be repositioned to its proper place.
  #
  # This, of course, assumes the queue is already sorted.
  #
  def reheap(k)
    return self if size <= 1

    que = @que.dup

    v = que.delete_at(k)
    i = binary_index(que, v)

    que.insert(i, v)

    @que = que

    return self
  end

  #
  # Sort the queue in accorance to the given comparison procedure.
  #
  def sort!
    @que.sort! do |a,b|
      case @cmp.call(a,b)
      when  0, nil   then  0
      when  1, true  then  1
      when -1, false then -1
      else
        warn "bad comparison procedure in #{self.inspect}"
        0
      end
    end
    self
  end

  #
  # Alias of #sort!
  #
  alias heapify sort!

  #
  def binary_index(que, target)
    upper = que.size - 1
    lower = 0
    while(upper >= lower) do
      idx  = lower + (upper - lower) / 2
      comp = @cmp.call(target, que[idx])

      case comp
      when 0, nil
        return idx
      when 1, true
        lower = idx + 1
      when -1, false
        upper = idx - 1
      else
      end
    end
    lower
  end

end # class PQueue
```

The above test adds ten actions to be executed one after another one second apart. Observe the desired execution times for the actions and the actual execution times. The two should match.

In the second test case, we first submit a callback that should get executed after eight seconds. Three seconds later another call back is submitted which should be executed after only one second. The callback being submitted later should execute first.

main.rb

pqueue.rb

```
# Priority queue with array based heap.
#
# A priority queue is like a standard queue, except that each inserted
# elements is given a certain priority, based on the result of the
# comparison block given at instantiation time. Also, retrieving an element
# from the queue will always return the one with the highest priority
# (see #pop and #top).
#
# The default is to compare the elements in repect to their #<=> method.
# For example, Numeric elements with higher values will have higher
# priorities.
```

```
#
# Note that as of version 2.0 the internal queue is kept in the reverse order
# from how it was kept in previous version. If you had used #to_a in the
# past then be sure to adjust for the priorities to be ordered back-to-front
# instead of the oterh way around.
#
class PQueue

  #
  VERSION = "2.1.0"  #:erb: VERSION = "<%= version %>"

  #
  # Returns a new priority queue.
  #
  # If elements are given, build the priority queue with these initial
  # values. The elements object must respond to #to_a.
  #
  # If a block is given, it will be used to determine the priority between
  # the elements. The block must must take two arguments and return `1`, `0`,
  # or `-1` or `true`, `nil` or `false. It should return `0` or `nil` if the
  # two arguments are considered equal, return `1` or `true` if the first
  # argument is considered greater than the later, and `-1` or `false` if
  # the later is considred to be greater than the first.
  #
  # By default, the priority queue retrieves maximum elements first
  # using the #<=> method.
  #
  def initialize(elements=nil, &block) # :yields: a, b
    @que = []
    @cmp = block || lambda{ |a,b| a <=> b }
    replace(elements) if elements
  end

  protected

  #
  # The underlying heap.
  #
  attr_reader :que #:nodoc:

  public

  #
  # Priority comparison procedure.
  #
  attr_reader :cmp

  #
  # Returns the size of the queue.
  #
  def size
    @que.size
  end

  #
  # Alias of size.
  #
  alias length size

  #
  # Add an element in the priority queue.
  #
```

```
def push(v)
  @que << v
  reheap(@que.size-1)
  self
end

#
# Traditional alias for #push.
#
alias enq push

#
# Alias of #push.
#
alias :<< :push

#
# Get the element with the highest priority and remove it from
# the queue.
#
# The highest priority is determined by the block given at instantiation
# time.
#
# The deletion time is O(log n), with n is the size of the queue.
#
# Return nil if the queue is empty.
#
def pop
  return nil if empty?
  @que.pop
end

#
# Traditional alias for #pop.
#
alias deq pop

# Get the element with the lowest priority and remove it from
# the queue.
#
# The lowest priority is determined by the block given at instantiation
# time.
#
# The deletion time is O(log n), with n is the size of the queue.
#
# Return nil if the queue is empty.
#
def shift
  return nil if empty?
  @que.shift
end

#
# Returns the element with the highest priority, but
# does not remove it from the queue.
#
def top
  return nil if empty?
  return @que.last
end

#
```

```
# Traditional alias for #top.
#
alias peek top

#
# Returns the element with the lowest priority, but
# does not remove it from the queue.
#
def bottom
  return nil if empty?
  return @que.first
end

#
# Add more than one element at the same time. See #push.
#
# The elements object must respond to #to_a, or be a PQueue itself.
#
def concat(elements)
  if empty?
    if elements.kind_of?(PQueue)
      initialize_copy(elements)
    else
      replace(elements)
    end
  else
    if elements.kind_of?(PQueue)
      @que.concat(elements.que)
      sort!
    else
      @que.concat(elements.to_a)
      sort!
    end
  end
  return self
end

#
# Alias for #concat.
#
alias :merge! :concat

#
# Return top n-element as a sorted array.
#
def take(n=@size)
  a = []
  n.times{a.push(pop)}
  a
end

#
# Returns true if there is no more elements left in the queue.
#
def empty?
  @que.empty?
end

#
# Remove all elements from the priority queue.
#
def clear
```

```
  @que.clear
  self
end


#
# Replace the content of the heap by the new elements.
#
# The elements object must respond to #to_a, or to be
# a PQueue itself.
#
def replace(elements)
  if elements.kind_of?(PQueue)
    initialize_copy(elements)
  else
    @que.replace(elements.to_a)
    sort!
  end
  self
end

#
# Return a sorted array, with highest priority first.
#
def to_a
  @que.dup
end

#
# Return true if the given object is present in the queue.
#
def include?(element)
  @que.include?(element)
end

#
# Push element onto queue while popping off and returning the next element.
# This is qquivalent to successively calling #pop and #push(v).
#
def swap(v)
  r = pop
  push(v)
  r
end

#
# Iterate over the ordered elements, destructively.
#
def each_pop #:yields: popped
  until empty?
    yield pop
  end
  nil
end

#
# Pretty inspection string.
#
def inspect
  "<#{self.class}: size=#{size}, top=#{top || "nil"}>"
end

#
```

```ruby
# Return true if the queues contain equal elements.
#
def ==(other)
  size == other.size && to_a == other.to_a
end

private

#
#
#
def initialize_copy(other)
  @cmp  = other.cmp
  @que  = other.que.dup
  sort!
end

#
# The element at index k will be repositioned to its proper place.
#
# This, of course, assumes the queue is already sorted.
#
def reheap(k)
  return self if size <= 1

  que = @que.dup

  v = que.delete_at(k)
  i = binary_index(que, v)

  que.insert(i, v)

  @que = que

  return self
end

#
# Sort the queue in accorance to the given comparison procedure.
#
def sort!
  @que.sort! do |a,b|
    case @cmp.call(a,b)
    when  0, nil   then  0
    when  1, true  then  1
    when -1, false then -1
    else
      warn "bad comparison procedure in #{self.inspect}"
      0
    end
  end
  self
end

#
# Alias of #sort!
#
alias heapify sort!

#
def binary_index(que, target)
  upper = que.size - 1
```

```
    lower = 0

    while(upper >= lower) do
      idx  = lower + (upper - lower) / 2
      comp = @cmp.call(target, que[idx])

      case comp
      when 0, nil
        return idx
      when 1, true
        lower = idx + 1
      when -1, false
        upper = idx - 1
      else
      end
    end
    lower
  end

end # class PQueue
```