# 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.

## Thread Safe Deferred Callback

> Design and implement a thread-safe class that allows registration 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 **`add_action()`** 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 which doesn't involve a busy thread.
>
> 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:

```
class DeferredCallbackExecutor():
    def __init__(self):
        self.actions = list()
        self.cond = Condition()
        self.sleep = 0

    def add_action(self, action):
        pass

    def start(self):
        while True:
            # execution logic comes here
            pass
```

We define a simple **`DeferredAction`** class, an object of which will be passed into the **`register_action`** method. This method will be adding the action to our min-heap which is represented by variable **`actions`**. Additionally, we'll be using the **`heapq`** module to heapify and perform other heap related operations on the **`actions`** list variable. We also define the **`__lt__()`** method in the **`DeferredAction`** class which helps compare two actions based on when they will get executed.

For guarding access to critical sections we'll use a **`Condition`** object. It serves dual purpose of locking critical sections and allowing consumer threads to signal the executor thread when a new action gets 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 **`notify()`** it. Let's write out what we just discussed as code.

```
class DeferredCallbackExecutor():
    def __init__(self):
        self.actions = list()
        self.cond = Condition()
        self.sleep = 0

    def add_action(self, action):
        # add exec_at time for the action
        action.execute_at = time.time() + action.exec_secs_aft
er

        self.cond.acquire()
        heapq.heappush(self.actions, action)
        self.cond.notify()
        self.cond.release()

    def start(self):
        while True:
            self.cond.acquire()
            # critical section where an action is removed from
 the queue and executed
            self.cond.release()


# class representing an action
class DeferredAction(object):
    def __init__(self, exec_secs_after, name, action):
        self.exec_secs_after = exec_secs_after
        self.action = action
        self.name = name

    def __lt__(self, other):
        return self.execute_at < other.execute_at
```

Note in **`register_action()`** method we acquire the underlying lock associated with the condition variable to protect the critical section which adds an action to the queue. Remember each condition variable has an associated lock that may be passed-in explicitly or is created under the hood if not passed-in. Note the execution thread won't wake up from the **`wait()`** call until the consumer thread gives up the lock 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.

This flow is captured in the code below:

```
    def start(self):

        while True:
            self.cond.acquire()

            while len(self.actions) is 0:
                self.cond.wait()

            while len(self.actions) is not 0:

                # calculate sleep duration
                next_action = self.actions[0]
                sleep_for = next_action.execute_at - math.floo
r(time.time())
                if sleep_for <= 0:
                    # time to execute action
                    break

            self.cond.wait(timeout=sleep_for)

            action_to_execute_now = heapq.heappop(self.actions
)
            action_to_execute_now.action(*(action_to_execute_n
ow,))

            self.cond.release()
```

(/learn)

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 to check for the associated condition in a while loop.

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:

```
        self.actions = list()
        self.cond = Condition()
        self.sleep = 0

    def add_action(self, action):
        # add exec_at time for the action
        action.execute_at = time.time() + action.exec_secs_after

        self.cond.acquire()
        heapq.heappush(self.actions, action)
        self.cond.notify()
        self.cond.release()

    def start(self):

        while True:
            self.cond.acquire()

            while len(self.actions) is 0:
                self.cond.wait()

            while len(self.actions) is not 0:

                # calculate sleep duration
                next_action = self.actions[0]
                sleep_for = next_action.execute_at - math.floor(time.time())
                if sleep_for <= 0:
                    # time to execute action
```

Four actions are submitted for deferred execution. A to be executed after 3 seconds, B to be executed after 2 seconds, C to be executed after 1 second and finally D to be executed after 7 seconds. The output shows the execution of the actions in the order C, B, A and D.

Back
... continued

Next
Implementing Semaphore

✓ Mark as Completed

Report an Issue

Ask a Question
(https://discuss.educative.io/tag/thread-safe-deferred-callback__interview-practise-problems__python-concurrency-for-senior-engineering-interviews)