Non-Blocking Queue

This lesson is a follow-up to the blocking queue question and explores the various ways in which we can make a blocking queue non-blocking.

Non-Blocking Queue

Problem

We have seen the blocking version of a queue in the previous question that blocks a producer or a consumer when the queue is full or empty respectively. In this problem, you are asked to implement a queue that is non-blocking. The requirement on non-blocking is intentionally left open to interpretation to see how you think through the problem.

This question is inspired by one of David Beazley's (https://www.dabeaz.com/) Python talks.

Solution

Let's first define the notion of **non-blocking**. If a consumer or a producer can successfully enqueue or dequeue an item, it is considered non-blocking. However, if the queue is full or empty then a producer or a consumer (respectively) need not wait until the queue can be added to or taken from.

First Cut

The trivial solution is to return a boolean value indicating the success of an operation. If the invoker of either **`enqueue()`** or **`dequeue()`** receives False, then it is the responsibility of the invoker to retry the operation at a later time. This trivial solution appears in the code widget below.

```
from threading import Thread
from threading import Lock
from threading import current_thread
from concurrent.futures import Future
import time
import random


class NonBlockingQueue:

    def __init__(self, max_size):
        self.max_size = max_size
        self.q = []
        self.lock = Lock()

    def dequeue(self):

        with self.lock:
            curr_size = len(self.q)

            if curr_size != 0:
                return self.q.pop(0)

            else:
                return False

    def enqueue(self, item):

        with self.lock:
            curr_size = len(self.q)

            if curr_size == self.max_size:
                return False

            else:
                self.q.append(item)
                return True


def consumer_thread(q):
    while 1:
        item = q.dequeue()

        if item == False:
            print("Consumer couldn't dequeue an item")
        else:
            print("\n{0} consumed item {1}".format(current_thread().getName(), item

        time.sleep(random.randint(1, 3))
```

```
def producer_thread(q):
    item = 1


    while 1:
        result = q.enqueue(item)
        if result is True:
            print("\n {0} produced item".format(current_thread().getName()), flush=
            item += 1


if __name__ == "__main__":
    no_block_q = NonBlockingQueue(5)

    consumerThread1 = Thread(target=consumer_thread, name="consumer", args=(no_bloc
    producerThread1 = Thread(target=producer_thread, name="producer", args=(no_bloc

    consumerThread1.start()
    producerThread1.start()

    time.sleep(15)
    print("Main thread exiting")
```

In the above example, note that we use a **`Lock`** object before we attempt to enqueue or dequeue an item in our internal queue. We must ensure serial access to shared data-structures.

Second Cut

If we want to get more sophisticated in our approach we can return an object of **`concurrent.futures.Future`** class to the invoker of the queue APIs incase the requested operation can't be completed at the time of invocation.

The invoker can, at a later time, retry the operation when the future gets resolved. The onus of appropriately resolving the future lies with our blocking queue class. If we intend to resolve the future at a later time, we also need to store a reference to it. We'll maintain two additional queues **`q_waiting_puts`** and **`q_waiting_gets`** to keep track of futures queues, **`q_waiting_puts`** and **`q_waiting_gets`**, to keep track of futures we return for put and get requests.

As an example, consider a producer thread wants to add an item to the queue when it is already full. The queue declines the operation and instead hands out a future to the producer. The producer can retry the enqueue operation at its own leisure when it finds the handed out future in a resolved state. Moreover, the queue resolves the futures in a FIFO order.

The **`dequeue()`** method changes as follows:

```
    def dequeue(self):

        result = None
        future = None

        with self.lock:
            curr_size = len(self.q)

            if curr_size != 0:
                result = self.q.pop(0)

                # remember to resolve a pending future for
                # a put request
                if len(self.q_waiting_puts) != 0:
                    self.q_waiting_puts.pop(0).set_result(True
)

            else:
                # queue is empty so create a future for a get
                # request
                future = Future()
                self.q_waiting_gets.append(future)

        return result, future
```

And the **`enqueue()`** method changes as follows:

```
    def enqueue(self, item):

        # print("size {0}".format(len(self.q_waiting_puts)))

        future = None
        with self.lock:
            curr_size = len(self.q)

            # queue is full so create a future for a put
            # request
            if curr_size == self.max_size:
                future = Future()
                self.q_waiting_puts.append(future)

            else:
                self.q.append(item)

                # remember to resolve a pending future for
                # a get request
                if len(self.q_waiting_gets) != 0:
                    future_get = self.q_waiting_gets.pop(0)
                    future_get.set_result(True)

        return future
```

The code widget below presents the complete code. Running the code widget will run a simulation with a fast producer and a slow consumer. The producer will constantly poll for received futures to get resolved and then try the enqueue operation again.

```
from threading import Thread
from threading import Lock
from threading import current_thread
from concurrent.futures import Future
import time
import random


class NonBlockingQueue:

    def __init__(self, max_size):
        self.max_size = max_size
        self.q = []
        self.q_waiting_puts = []
        self.q_waiting_gets = []
        self.lock = Lock()

    def dequeue(self):

        result = None
        future = None

        with self.lock:
            curr_size = len(self.q)

            if curr_size != 0:
                result = self.q.pop()

                if len(self.q_waiting_puts) > 0:
                    self.q_waiting_puts.pop().set_result(True)

            else:
                future = Future()
                self.q_waiting_gets.append(future)

        return result, future

    def enqueue(self, item):

        future = None
        with self.lock:
            curr_size = len(self.q)

            if curr_size == self.max_size:
                future = Future()
                self.q_waiting_puts.append(future)

            else:
                self.q.append(item)

                if len(self.q_waiting_gets) != 0:
```

```
                future_get = self.q_waiting_gets.pop()
                future_get.set_result(True)


        return future


def consumer_thread(q):
    while 1:
        item, future = q.dequeue()

        if item is None:
            print("Consumer received a future but we are ignoring it")
        else:
            print("\n{0} consumed item {1}".format(current_thread().getName(), item

        # slow down consumer thread
        time.sleep(random.randint(1, 3))


def producer_thread(q):
    item = 1
    while 1:
        future = q.enqueue(item)
        if future is not None:
            while future.done() == False:
                print("waiting for future to resolve")
                time.sleep(0.1)
        else:
            item += 1


if __name__ == "__main__":
    no_block_q = NonBlockingQueue(5)

    consumerThread1 = Thread(target=consumer_thread, name="consumer", args=(no_bloc
    producerThread1 = Thread(target=producer_thread, name="producer", args=(no_bloc

    consumerThread1.start()
    producerThread1.start()

    time.sleep(15)
    print("Main thread exiting")
```

Using Callbacks Instead of Busy Waiting

In the above solution we are busy waiting in a loop for the future to get resolved. The next evolution of our solution will use a callback that gets invoked when a future gets resolved instead of busy waiting. The **`Future`** object exposes a method **`add_done_callback()`** that takes in a callable which gets invoked when the future is resolved. We can refactor our **`producer_thread()`** as follows:

```
def producer_thread(q):
    item = 1
    while 1:
        future = q.enqueue(item)
        if future is not None:
            future.item = item
            future.q = q
            future.add_done_callback(retry_enqueue)

        item += 1
```

The **`retry_enqueue()`** method is defined as follows:

```
def retry_enqueue(future):

    item = future.item
    q = future.q
    new_future = q.enqueue(item)

    if new_future is not None:
        new_future.item = item
        new_future.q = q
        new_future.add_done_callback(retry_enqueue)
```

An important detail is to understand when the callback gets invoked once the corresponding future object is resolved, and by which thread. The official documentation states that a callback method is **always**

called in a thread belonging to the process that added them. We can conclude from this statement that the callback may be invoked by the same thread that resolves the future and immediately after the future object's `set_result()` method is called.

Does the above detail affect our design? Realize that the queue resolves a future while holding the `Lock` object. The callback gets invoked by the **same thread** that resolves the future. And within the callback method `retry_enqueue()` we again attempt an `enqueue()` operation. The astute reader should immediately realize that if the callback method is invoked whilst the thread is holding the lock, then in that sequence, the thread will attempt to acquire the same lock twice causing a deadlock. For the stated solution to work, we'll need to used a `RLock`. The complete solution appears in the code widget below.

```
from threading import Thread
from threading import Lock
from threading import RLock
from threading import current_thread
from concurrent.futures import Future
import time
import random


class NonBlockingQueue:

    def __init__(self, max_size):
        self.max_size = max_size
        self.q = []
        self.q_waiting_puts = []
        self.q_waiting_gets = []
        self.lock = RLock()
        #self.lock = Lock()

    def dequeue(self):

        result = None
        future = None

        with self.lock:
            curr_size = len(self.q)

            if curr_size != 0:
                result = self.q.pop(0)

                # remember to resolve a pending future for
                # a put request
                if len(self.q_waiting_puts) != 0:
                    self.q_waiting_puts.pop(0).set_result(True)

            else:
                # queue is empty so create a future for a get
                # request
                future = Future()
                self.q_waiting_gets.append(future)

        return result, future

    def enqueue(self, item):

        # print("size {0}".format(len(self.q_waiting_puts)))

        future = None
        with self.lock:
            curr_size = len(self.q)
            # queue is full so create a future for a put
            # request

            if curr_size == self.max_size:
                future = Future()
                self.q_waiting_puts.append(future)

            else:
                self.q.append(item)

                # remember to resolve a pending future for
                # a get request
                if len(self.q_waiting_gets) != 0:
                    future_get = self.q_waiting_gets.pop(0)
                    future_get.set_result(True)

        return future


def consumer_thread(q):
    while 1:
        item, future = q.dequeue()

        if item is None:
            print("\nConsumer received a future but we are ignoring it")
        else:
            print("\n{0} consumed item {1}".format(current_thread().getName(), item

        # slow down the consumer
        time.sleep(random.randint(1, 3))


def retry_enqueue(future):
    print("\nCallback invoked by thread {0}".format(current_thread().getName()))
    item = future.item
    q = future.q
    new_future = q.enqueue(item)

    if new_future is not None:
        new_future.item = item
        new_future.q = q
        new_future.add_done_callback(retry_enqueue)
    else:
        print("\n{0} successfully added on a retry".format(item))


def producer_thread(q):
    item = 1
    while 1:
        future = q.enqueue(item)
        if future is not None:
            future.item = item
            future.q = q
            future.add_done_callback(retry_enqueue)
        item += 1

        # slow down the producer
        time.sleep(0.1)


if __name__ == "__main__":
    no_block_q = NonBlockingQueue(5)

    consumerThread1 = Thread(target=consumer_thread, name="consumer", args=(no_bloc
    producerThread1 = Thread(target=producer_thread, name="producer", args=(no_bloc

    consumerThread1.start()
    producerThread1.start()

    time.sleep(15)
    print("\nMain thread exiting")
```

As an exercise, uncomment **line#18** and comment out **line#17** and re-run the widget. Observe that the program would enter a deadlock. The output from the code widget will show the consumer thread blocked. The widget executes successfully as the producer and consumer threads are marked as daemon threads.

So far, we have successfully changed the **`enqueue()`** method into a non-blocking one. Also, note that since we save the put futures in a queue, the order of the items to be put is maintained as long as the first retry for a failed put is successful. In case of the first retry fails, the new future is placed at the end of the **`q_waiting_puts`** queue.

Making dequeue ( ) Non-blocking

Making **`dequeue()`** non-blocking is trivial. Instead of setting true when resolving a get future to let the consumer know that an item is available in the queue, we can simply resolve the future to an item dequeued from the queue. This saves the consumer from making a **`dequeue()`** call. The

`enqueue()` method would change as follows:

```
    def enqueue(self, item):

        # print("size {0}".format(len(self.q_waiting_puts)))

        future = None
        with self.lock:
            curr_size = len(self.q)

            # queue is full so create a future for a put
            # request
            if curr_size == self.max_size:
                future = Future()
                self.q_waiting_puts.append(future)

            else:
                self.q.append(item)

                # remember to resolve a pending future for
                # a get request
                if len(self.q_waiting_gets) != 0:
                    future_get = self.q_waiting_gets.pop(0)
                    future_get.set_result(self.q.pop(0))
```

Additionally, we'll also need to change the **`consumer_thread()`** method and introduce a **`retry_dequeue()`** callback. Both the changes are shown below:

```
def retry_dequeue(future):
    item = future.result()
    print("\nretry_dequeue executed by thread {0} and {1} consumed on a retry".format(current_thread().getName(), item), flush=True)



def consumer_thread(q):
    while 1:
        item, future = q.dequeue()

        if item is None:
            future.add_done_callback(retry_dequeue)

        else:
            print("\n{0} consumed item {1}".format(current_thread().getName(), item), flush=True)

        # slow down the consumer
        time.sleep(1)
```

In the code widget below, we present the complete code which runs the consumer faster than the producer.

```
from threading import Thread
from threading import RLock
from threading import current_thread
from concurrent.futures import Future
import time
import random


class NonBlockingQueue:

    def __init__(self, max_size):
        self.max_size = max_size
        self.q = []
        self.q_waiting_puts = []
        self.q_waiting_gets = []
        self.lock = RLock()

    def dequeue(self):

        result = None
        future = None

        with self.lock:
            curr_size = len(self.q)

            if curr_size != 0:
                result = self.q.pop(0)

                # remember to resolve a pending future for
                # a put request
                if len(self.q_waiting_puts) != 0:
                    self.q_waiting_puts.pop(0).set_result(True)

            else:
                # queue is empty so create a future for a get
                # request
                future = Future()
                self.q_waiting_gets.append(future)

        return result, future

    def enqueue(self, item):

        future = None
        with self.lock:
            curr_size = len(self.q)

            # queue is full so create a future for a put
            # request
            if curr_size == self.max_size:
                future = Future()
                self.q_waiting_puts.append(future)

            else:
                self.q.append(item)

                # remember to resolve a pending future for
                # a get request
                if len(self.q_waiting_gets) != 0:
                    future_get = self.q_waiting_gets.pop(0)
                    future_get.set_result(self.q.pop(0))

        return future


def retry_dequeue(future):
    item = future.result()
    print("\nretry_dequeue executed by thread {0} and {1} consumed on a retry".form


def consumer_thread(q):
    while 1:
        item, future = q.dequeue()

        if item is None:
            future.add_done_callback(retry_dequeue)

        else:
            print("\n{0} consumed item {1}".format(current_thread().getName(), item

        # slow down the consumer
        time.sleep(1)


def retry_enqueue(future):
    print("\nCallback invoked by thread {0}".format(current_thread().getName()))
    item = future.item
    q = future.q
    new_future = q.enqueue(item)

    if new_future is not None:
        new_future.item = item
        new_future.q = q
        new_future.add_done_callback(retry_enqueue)
    else:
        print("\n{0} successfully added on a retry".format(item))


def producer_thread(q):
    item = 1
    while 1:
        future = q.enqueue(item)
        if future is not None:
            future.item = item
            future.q = q
            future.add_done_callback(retry_enqueue)

        item += 1

        # slow down the producer
        time.sleep(random.randint(1, 3))


if __name__ == "__main__":
    no_block_q = NonBlockingQueue(5)

    consumerThread1 = Thread(target=consumer_thread, name="consumer", args=(no_bloc
    producerThread1 = Thread(target=producer_thread, name="producer", args=(no_bloc

    consumerThread1.start()
    producerThread1.start()

    time.sleep(15)
    print("\nMain thread exiting")
```

The code widget might throw an exception at times because we mark the producer and consumer threads as daemon threads and don't implement a solution to gracefully shutdown the two threads to avoid adding complexity to the example.

Critique

Every solution comes with its pros and cons. Let's discuss the shortcomings of the solutions we presented.

- We may argue that in terms of space we end up retaining information about all the enqueues and dequeues that couldn't be completed immediately. In case of a fast producer and a slow consumer, the internal future queues may grow without bound, thus defeating the intent of a bounded queue.

- Queue is used to ensure FIFO order. Even though we resolve futures in a FIFO order but that doesn't guarantee that pending puts or gets happen in the same order. In the busy-wait solution, a consumer may check upon the state of a future late enough such that other consumers have already drained the queue. Similarly, in the callback version, unless we are guaranteed by the framework that a callback gets invoked as soon as the future gets resolved we can't promise a FIFO order.

← Back
Blocking Queue | Bounded Buffer | Co...

Next →
Rate Limiting Using Token Bucket Filter

✓ Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/non-blocking-queue__interview-practise-problems__python-concurrency-for-senior-engineering-interviews)