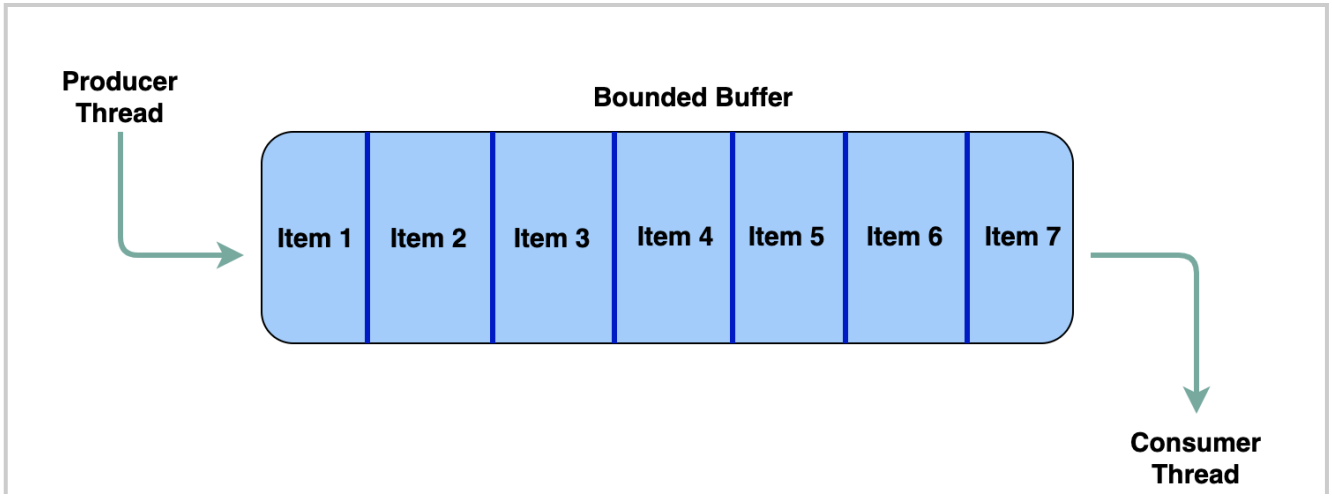
Blocking Queue | Bounded Buffer | Consumer Producer

Classical synchronization problem involving a limited size buffer which can have items added to it or removed from it by different producer and consumer threads. This problem is known by different names: consumer producer problem, bounded buffer problem or blocking queue problem.

Blocking Queue | Bounded Buffer | Consumer Producer

> A blocking queue is defined as a queue which blocks the caller of the enqueue method if there's no more capacity to add the new item being enqueued. Similarly, the queue blocks the dequeue caller if there are no items in the queue. Also, the queue notifies a blocked enqueuing thread when space becomes available and a blocked dequeuing thread when an item becomes available in the queue.



Solution

The **`queue.Queue`** module already provides a synchronized queue out of the box. We'll be creating one from scratch that can be used by multiple consumers and producers.

Our queue will have a finite size that is passed in via the constructor. Additionally, we'll use a list as the backing data structure for our queue. We can leverage the **`pop()`** and **`append()`** method on the list to emulate the operation of a queue. **`appends()`** adds an element to the front of the list and **`pop()`** retrieves the last element of the list, freeing us from keeping track of the front and back of the queue.

We'll need two variables: one to keep track of the maximum size of the queue and another to track the current size of the queue. Moreover, we'll also need a condition variable that either the producer or the consumer can wait on if the queue is full or empty respectively.

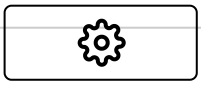
Initially, our class will look like as follows:

```
class BlockingQueue:

    def __init__(self, max_size):
        self.max_size = max_size
        self.curr_size = 0
        self.cond = Condition()
        self.q = []

    def enqueue(self):
        pass

    def dequeue(self):
        pass
```



Let's start with the **enqueue()** method, which takes in an item to be enqueued to the back of the queue. If the **`current size of the queue == capacity`** then we know we'll need to block the invoker of the method. We can do so by calling **`wait()`** on the condition variable wrapped with a while loop. The while loop is conditioned on the size of the queue being equal to the maximum capacity of the queue. The loop's predicate would become false as soon as another thread performs a dequeue.

Note that whenever we test for the value of the **`curr_size`** variable, we also need to make sure that no other thread is manipulating the **`curr_size`** variable. This can be achieved by acquiring the lock associated with the condition variable at the start of each method.

The enqueue method would look like as follows:

```
    def enqueue(self, item):

        self.cond.acquire()
        while self.curr_size == self.max_size:
            self.cond.wait()

        self.q.append(item)
        self.curr_size += 1

        self.cond.notifyAll()
        self.cond.release()
```

Note that in the end we are calling **`notifyAll()`** method. Since we just added an item to the queue, it is possible that a consumer thread might be blocked in the **`dequeue()`** method of the queue class waiting for an item to become available. So it's necessary we send a signal to wake up any waiting threads.

If no thread is waiting, then the signal will simply go unnoticed and be ignored, which wouldn't affect the correct working of our class. This would be an instance of a **missed signal**.

Now let's design the `dequeue()` method. Similar to the `enqueue()` method, we need to block the caller of the dequeue method if there's nothing to dequeue i.e. `self.curr_size == 0`. When this condition is true a consumer thread needs to wait on the condition variable.

After consuming an item, remember to call `notifyAll()` since if the queue were full then there might be producer threads blocked in the `enqueue()` method. This logic in code appears as below:

```
    def dequeue(self):

        self.cond.acquire()
        while self.curr_size == 0:
            self.cond.wait()

        item = self.q.pop(0)
        self.curr_size -= 1

        self.cond.notifyAll()
        self.cond.release()

        return item
```

We see the dequeue method is analogous to enqueue method. The complete code appears in the code widget below along with an example.


```
from threading import Thread
from threading import Condition
from threading import current_thread
import time
import random


class BlockingQueue:

    def __init__(self, max_size):
        self.max_size = max_size
        self.curr_size = 0
        self.cond = Condition()
        self.q = []

    def dequeue(self):

        self.cond.acquire()
        while self.curr_size == 0:
            self.cond.wait()

        item = self.q.pop(0)
        self.curr_size -= 1

        self.cond.notifyAll()
        self.cond.release()

        return item

    def enqueue(self, item):

        self.cond.acquire()
        while self.curr_size == self.max_size:
            self.cond.wait()

        self.q.append(item)
        self.curr_size += 1

        self.cond.notifyAll()
        print("\ncurrent size of queue {0}".format(self.curr_size), flush=True)
        self.cond.release()


def consumer_thread(q):
    while 1:
        item = q.dequeue()
        print("\n{0} consumed item {1}".format(current_thread().getName(), item), f
        time.sleep(random.randint(1, 3))


def producer_thread(q, val):
```

```
    item = val
    while 1:

        q.enqueue(item)
        item += 1
        time.sleep(0.1)


if __name__ == "__main__":
    blocking_q = BlockingQueue(5)

    consumerThread1 = Thread(target=consumer_thread, name="consumer-1", args=(block
    consumerThread2 = Thread(target=consumer_thread, name="consumer-2", args=(block
    producerThread1 = Thread(target=producer_thread, name="producer-1", args=(block
    producerThread2 = Thread(target=producer_thread, name="producer-2", args=(block

    consumerThread1.start()
    consumerThread2.start()
    producerThread1.start()
    producerThread2.start()

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

In the above example we have two producers and two consumers. We intentionally slow down the consumers using sleep. The simulation runs for 15 seconds. One producer adds values to the queue starting from 1 while the other adds values starting from 100. From the output, you can observe that the consumer threads see values from both the producer threads and at no point, the current size of the queue exceeds the maximum allowed size.

Follow Up Question

Does it matter if we use **`notify()`** or **`notifyAll()`** method in our implementation?

In both the **`enqueue()`** and **`dequeue()`** methods we use the **`notifyAll()`** method instead of the **`notify()`** method. The reason

behind the choice is very crucial to understand. Consider a situation with two producer threads and one consumer thread all working with a queue of size one. It's possible that when an item is added to the queue by one of the producer threads, the other two threads are blocked waiting on the condition variable. If the producer thread after adding an item invokes **`notify()`** it is possible that the other producer thread is chosen by the system to resume execution. The woken-up producer thread would find the queue full and go back to waiting on the condition variable, causing a deadlock. Invoking **`notifyAll()`** assures that the consumer thread also gets a chance to wake up and resume execution.

Back

Asynchronous with Executors

Next

Non-Blocking Queue

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/blocking-queue-bounded-buffer-consumer-producer__interview-practise-problems__python-concurrency-for-senior-engineering-interviews)