1188. Design Bounded Blocking Queue

Medium Concurrency Leetcode Link

Problem Description

The problem is about creating a data structure which is a bounded blocking queue with thread-safety in mind. A bounded blocking queue is a queue with a fixed maximum size, and it has the capability to block or wait when operations like enqueue (adding to the queue) and dequeue (removing from the queue) cannot be performed because the queue is full or empty, respectively. The queue supports three main operations:

- `enqueue(int element)`: This method adds an element to the end of the queue. If the queue has reached its capacity, the method should block the calling thread until there is space available to add the new element.
- `dequeue()`: This removes and returns the element at the front of the queue. If the queue is empty, this operation should block until there is an element available to dequeue.
- `size()`: This returns the current number of elements in the queue.

It is especially noted that the implementation will be tested in a multithreaded environment, where multiple threads could be calling these methods simultaneously. Therefore, it is crucial that the implementation ensures that all operations on the bounded blocking queue are thread-safe (i.e., function correctly when accessed from multiple threads).

Lastly, the use of any built-in bounded blocking queue implementations is prohibited as the goal is to understand how to create such a data structure from scratch, potentially in a job interview.

Intuition

The solution to the problem involves coordinating access to the queue to ensure that only one thread can perform an enqueue or dequeue operation at a time to maintain thread safety. This means protecting the internal state of the queue from race conditions that could lead to incorrect behavior or data corruption.

To achieve this, we employ two synchronization primitives called `Semaphores`. A semaphore maintains a set of permits, and a thread that wants to perform an operation must first acquire a permit. If no permit is available, the thread blocks until a permit is released by another thread. This behavior fits perfectly for our problem's requirements.

We use two semaphores in the solution:

1. `s1`, which starts with a number of permits equal to the capacity of the queue. This semaphore controls access to enqueue operation. A thread can enqueue if it can acquire a permit from `s1`, which signifies that there is room in the queue. After the enqueue operation, the thread releases a permit to `s2`, signaling that there is an item available to be dequeued.

2. `s2`, which starts with zero permits as the queue is initially empty. This semaphore controls access to the dequeue operation. For a thread to dequeue, it must acquire a permit from `s2`, which signifies that there is at least one item in the queue to be dequeued. After the dequeue operation, the thread releases a permit to `s1`, indicating that there is now additional space available in the queue.

The queue itself (`q`) is represented by a `deque` (double-ended queue) from Python's collections module, which allows for fast appending and popping from both ends. Note that accessing `len(q)` to get the size of the queue does not need to be serialized by semaphores, as it is not modifying the queue.

This approach enables us to limit the number of elements in the queue to the defined capacity, and to ensure that `enqueue` and `dequeue` operations wait for the queue to be not full or not empty, respectively, before proceeding, fulfilling the conditions for a bounded blocking queue in a thread-safe manner.

Solution Approach

In the provided solution, the implementation of the `BoundedBlockingQueue` class is done with two semaphores and a deque. Here's a walkthrough of the pattern and algorithms used:

- **Semaphores**: Two instances of the Semaphore class are used, `s1` and `s2`, each serving a distinct purpose. `s1` governs the ability to insert an item into the queue, and begins with permits equal to the capacity. `s2` reflects the number of items in the queue available to be dequeued and starts with no permits. This is a classic application of the "producer-consumer" problem's solution, where one semaphore is used to signal "empty slots" and another semaphore is used to signal "available items".

- **Deque**: A deque (double-ended queue) from the collections module is used to represent the queue's data structure. This provides efficient FIFO (first-in-first-out) operations needed for enqueueing (`append`) and dequeueing (`popleft`).

When `enqueue(element: int)` is called, the following steps are performed:

1. `self.s1.acquire()`: Attempt to acquire a permit from `s1`, which represents a free slot in the queue. If there are no free slots, this call will block until another thread calls `dequeue` and releases a permit on `s1`.
2. `self.q.append(element)`: Once a permit has been acquired (meaning there is space in the queue), the element is safely enqueued into the queue.
3. `self.s2.release()`: Releasing a permit on `s2` to signal that an element has been enqueued and is now available for dequeuing.

For `dequeue()`, the steps are the mirror image:

1. `self.s2.acquire()`: This acquires a permit from `s2`, which signifies that there is at least one element in the queue to be dequeued. If the queue is empty, this call will block until a thread calls `enqueue` and releases a permit on `s2`.
2. `ans = self.q.popleft()`: Removes the oldest (front) element from the queue safely because it's been ensured that the queue is not empty.
3. `self.s1.release()`: Releasing a permit on `s1` to signal that an element has been dequeued and there is now a free slot in the queue.

`size()` is a straightforward operation as it simply returns the current number of items in the queue, `len(self.q)`. It does not modify the queue, so it does not require interaction with semaphores.

The solution effectively serializes access to the mutable shared state (`self.q`), preventing race conditions by using semaphores to coordinate enqueue and dequeue actions. This guarantees that the queue never exceeds its capacity and avoids dequeue operations being called on an empty queue, thus satisfying thread safety and other requirements of the problem.

Example Walkthrough

Let's consider a small example to illustrate the solution approach for a `BoundedBlockingQueue` with a capacity of 2.

- Initially, we create the queue with a capacity of 2, initializing semaphore `s1` with 2 permits and semaphore `s2` with 0 permits.

- Imagine thread A calls `enqueue(1)`:

  1. It acquires a permit from `s1`, which now has 1 permit left.
  2. It then appends `1` to the `deque`, and the queue state becomes `[1]`.
  3. Finally, it releases a permit to `s2`, indicating that there is one item available for dequeuing.
- Now, thread B calls `enqueue(2)`:

  1. It acquires the remaining permit from `s1`, and `s1` now has 0 permits.
  2. It appends `2` to the `deque`, so the queue state becomes `[1,2]`.
  3. It releases another permit to `s2`, now `s2` has 2 permits indicating there are two items available to be dequeued.
- At this state, the queue is full. If another thread, say thread C, tries to `enqueue(3)`, it will be blocked as `s1` has no permits left, signifying the queue is at full capacity.

- Meanwhile, if thread D calls `dequeue()`:

  1. It acquires a permit from `s2` (which has 2 permits at this point), leaving 1 permit left in `s2`.
  2. It dequeues an element from the `deque` which is `1` (FIFO order), leaving the queue state as `[2]`.
  3. It releases a permit to `s1`, increasing the number of permits back to 1, signaling that there is now space for one more item in the queue.
- If thread C is still waiting to `enqueue(3)`, it can now proceed as a permit became available in `s1`.

  1. It acquires the permit from `s1`, and again `s1` has 0 permits.
  2. It appends `3` to the `deque`, so the queue state becomes `[2,3]`.
  3. It releases a permit to `s2`, which now has 2 permits, reflecting the two items in the queue.
- At any time, calling `size()` returns the number of items currently in the queue, which can be accessed by any thread without needing to acquire a permit.

This example demonstrates how the `BoundedBlockingQueue` enforces its bounds and provides thread-safe enqueueing and dequeuing operations using semaphores to manage its capacity and state.

Python Solution

```python
from threading import Semaphore
from collections import deque  # Ensure deque is imported

class BoundedBlockingQueue:
    def __init__(self, capacity: int):
        # Initialize the queue with given capacity.
        self.semaphore_empty_slots = Semaphore(capacity)  # Semaphore to track empty slots
        self.semaphore_filled_slots = Semaphore(0)        # Semaphore to track filled slots
        self.queue = deque()                               # Use deque for queue operations

    def enqueue(self, element: int) -> None:
        # Add an element to the end of the queue.
        self.semaphore_empty_slots.acquire()  # Decrease the counter of empty slots, wait if no empty slots
        self.queue.append(element)            # Add the element to the queue
        self.semaphore_filled_slots.release() # Increase the counter of filled slots, signaling dequeue if slots are filled

    def dequeue(self) -> int:
        # Remove and return an element from the front of the queue.
        self.semaphore_filled_slots.acquire()  # Decrease the counter of filled slots, wait if no filled slots
        element = self.queue.popleft()         # Remove the element from the queue
        self.semaphore_empty_slots.release()   # Increase the counter of empty slots, signaling enqueue if slots are available
        return element                         # Return the dequeued element

    def size(self) -> int:
        # Get the current number of elements in the queue.
        return len(self.queue)             # Return the size of the queue

```

Java Solution

```java
import java.util.ArrayDeque;
import java.util.Deque;
import java.util.concurrent.Semaphore;

// This class represents a thread-safe bounded blocking queue with a fixed capacity.
public class BoundedBlockingQueue {
    // Semaphore to control the number of elements that can be added (based on capacity).
    private final Semaphore enqueueSemaphore;
    // Semaphore to control the number of elements that can be removed (starts at 0).
    private final Semaphore dequeueSemaphore;
    // The queue to store elements.
    private final Deque<Integer> queue;

    // Constructor initializes the semaphores and queue with specified capacity.
    public BoundedBlockingQueue(int capacity) {
        enqueueSemaphore = new Semaphore(capacity);
        dequeueSemaphore = new Semaphore(0);
        queue = new ArrayDeque<>();
    }

    // Enqueues an element into the queue if there's available capacity.
    public void enqueue(int element) throws InterruptedException {
        // Acquire a permit from enqueueSemaphore discarding it, if available capacity is 0 waits.
        enqueueSemaphore.acquire();
        synchronized (this) {
            // Adds the element to the end of the queue.
            queue.offer(element);
        }
        // Release a permit to dequeueSemaphore, increasing the number of available elements to dequeue.
        dequeueSemaphore.release();
    }

    // Dequeues an element from the front of the queue.
    public int dequeue() throws InterruptedException {
        // Acquire a permit from dequeueSemaphore, waiting if necessary until an element is available.
        dequeueSemaphore.acquire();
        int element;
        synchronized (this) {
            // Remove and return the front element of the queue.
            element = queue.poll();
        }
        // Release a permit to enqueueSemaphore, increasing the available capacity.
        enqueueSemaphore.release();
        return element;
    }

    // Returns the current number of elements in the queue.
    public int size() {
        synchronized (this) {
            // The size of the queue is returned.
            return queue.size();
        }
    }
}

```

C++ Solution

```cpp
#include <queue>
#include <mutex>
#include <condition_variable>

class BoundedBlockingQueue {
public:
    // Constructor initializes the queue with a capacity limit.
    BoundedBlockingQueue(int capacity) : capacity_(capacity), count_(0) {
        // No need to initialize semaphores since we will use condition_variable and mutex
    }

    // Enqueue adds an element to the queue. If the queue is full, blocks until space is available.
    void enqueue(int element) {
        std::unique_lock<std::mutex> lock(mutex_);
        // Wait until there is space in the queue
        not_full_condition_.wait(lock, [this] { return count_ < capacity_; });
        queue_.push(element);
        ++count_;
        // Notify one waiting thread (if any) that an item was dequeued
        not_empty_condition_.notify_one();
    }

    // Dequeue removes and returns an element from the queue. If the queue is empty, blocks until an element is available.
    int dequeue() {
        std::unique_lock<std::mutex> lock(mutex_);
        // Wait until there is an item to dequeue
        not_empty_condition_.wait(lock, [this] { return count_ > 0; });
        int value = queue_.front();
        queue_.pop();
        --count_;
        // Notify one waiting thread (if any) that space is now available
        not_full_condition_.notify_one();
        return value;
    }

    // Get the current size of the queue.
    int size() {
        std::lock_guard<std::mutex> lock(mutex_);
        return count_;
    }

private:
    std::queue<int> queue_; // The queue that holds the elements
    std::mutex mutex_; // Mutex to protect access to the queue
    std::condition_variable not_full_condition_; // Condition variable to block enqueue when queue is full
    std::condition_variable not_empty_condition_; // Condition variable to block dequeue when queue is empty
    int capacity_; // Maximum number of items in the queue
    int count_; // Current number of items in the queue
};

```

Typescript Solution

```typescript
const queue: number[] = []; // The queue that holds the elements
let capacity: number; // Maximum number of items in the queue
let count: number = 0; // Current number of items in the queue

// Utilizing these for synchronization could be complex without encapsulation
const mutex = new Mutex(); // A pretend Mutex since JavaScript/TypeScript doesn't have one
const notFullCondition = new ConditionVariable(); // A pretend condition variable for when the queue is not full
const notEmptyCondition = new ConditionVariable(); // A pretend condition variable for when the queue is not empty

// Initialization of the queue with a capacity limit.
function initializeQueue(initialCapacity: number): void {
    capacity = initialCapacity;
}

// Adds an element to the queue. If the queue is full, it is supposed to block until space is available.
async function enqueue(element: number): Promise<void> {
    await mutex.lock();
    try {
        // Wait until there is space in the queue
        while (count >= capacity) {
            await notFullCondition.wait(mutex);
        }
        queue.push(element);
        count++;
        // Notify one waiting thread (if any) that an item was enqueued
        notEmptyCondition.notifyOne();
    } finally {
        mutex.unlock();
    }
}

// Removes and returns an element from the queue. If the queue is empty, it is supposed to block until an element is available.
async function dequeue(): Promise<number> {
    await mutex.lock();
    try {
        // Wait until there is an item to dequeue
        while (count === 0) {
            await notEmptyCondition.wait(mutex);
        }
        const value = queue.shift();
        count--;
        // Notify one waiting thread (if any) that space is now available
        notFullCondition.notifyOne();
        return value;
    } finally {
        mutex.unlock();
    }
}

// Returns the current size of the queue.
function size(): number {
    return count; // No need for synchronization in a single-threaded environment
}

// Note that Mutex and ConditionVariable are not native JS/TS classes.
// These would need to be implemented or a library would have to be used to simulate them.

```

Time and Space Complexity

For this `BoundedBlockingQueue` implementation using semaphores, we will analyze the time and space complexities of its operations.

Time Complexity

- `__init__`: Initializing the queue involves setting up two semaphores and the underlying deque. This operation is constant time, `O(1)`, as it involves only a fixed number of operations, regardless of the capacity of the queue.

- `enqueue`: The enqueue operation involves two semaphore operations (acquire and release) and an append operation on a deque. The semaphore operations are generally `O(1)` assuming they don't block; if they do block, the time complexity is dependent on external factors such as contention from other threads. Appending to the deque is an `O(1)` operation. Therefore, the combined time complexity is `O(1)` per call in the absence of blocking.

- `dequeue`: Like enqueue, dequeue also has two semaphore operations and a popleft operation on the deque. Since deque's popleft is designed to be `O(1)` and semaphore operations are `O(1)` without blocking, the overall time complexity for dequeue is again `O(1)` per call in the absence of blocking.

- `size`: This simply returns the number of items in the deque, which is maintained internally and is thus an `O(1)` operation.

Space Complexity

- The space complexity revolves around the deque that stores elements. Since the capacity of the queue is fixed, the maximum space it will use is `O(capacity)`, corresponding to the maximum number of elements that can be enqueued at any given time.

In summary, except for the potential blocking on semaphores (which can't be quantified in standard complexity analysis), all fundamental operations (`__init__`, `enqueue`, `dequeue`, `size`) on the `BoundedBlockingQueue` class have a time complexity of `O(1)`. The space complexity is `O(capacity)` based on the fixed maximum size of the underlying deque.