2805. Custom Interval

Medium

Problem Description

The problem `customInterval` involves creating a scheduler that executes a function `fn` repeatedly with intervals that increase linearly over time. Each execution interval is determined by the formula `delay + period * count`, where `delay` is the initial delay before the first execution of `fn`, `period` is the increment factor which is added to the total delay for each subsequent execution, and `count` is a counter that keeps track of how many times `fn` has been executed. The function `customInterval` must return an identifier `id` that can be used to stop the scheduled executions.

`customClearInterval` is the function used to stop the execution of `fn`. It takes an `id` as input, which corresponds to the identifier returned by `customInterval`, and stops the scheduled execution that was associated with that `id`.

Intuition

The `customInterval` function needs to schedule and execute the given function `fn` at the incrementally increasing intervals. The approach to this is to set up a recursive timeout system, where after each execution of `fn`, a new timeout is scheduled with the updated interval based on the linear pattern. This ensures that `fn` is executed repeatedly at the correct intervals.

To facilitate starting the timeout, we define a `recursiveTimeout` function inside `customInterval`. This helper function uses JavaScript's `setTimeout` to schedule `fn`'s next execution, increases the `count` by 1 after each execution, and then calls itself to schedule the next execution.

An `intervalMap` is used to keep track of all scheduled timeouts using a unique identifier `id` which in this case is generated by `Date.now()`. This ensures each scheduled set of executions has a unique identifier, which is necessary for the `customClearInterval` function to properly identify and stop the correct execution.

The `customClearInterval` function's intuition is to provide a way to cancel the scheduled execution. It uses the unique identifier `id` to look up and clear the scheduled timeout from `intervalMap`. This prevents future calls of `fn` and also removes the entry from the map to avoid potential memory leaks from keeping unnecessary timeouts in the map.

Solution Approach

The solution to the problem uses a combination of a map for storing timeout references, recursion for scheduling future calls, and the `setTimeout` function for executing the provided function `fn` at increasing intervals.

Here's how the implementation breaks down step by step:

1. **Global Map for Timeout References (`intervalMap`):** We initialize `intervalMap`, a JavaScript Map object, to store and retrieve the timeout references using a unique number `id`. This `id` is used later to clear the timeout if needed.

2. **Custom Interval Scheduling (`customInterval`):**
   - We first initialize a local `count` variable to 0 inside `customInterval`. This variable tracks the number of times `fn` has been called.
   - We declare `recursiveTimeout`, a recursive function that will handle the scheduling of `fn` executions. Inside this function, the `setTimeout` method is called with a delay calculated using the formula `delay + period * count`. Each time `fn` is executed, `count` is incremented.
   - We use `Date.now()` to generate a unique `id` for the current scheduling of `fn`. `Date.now()` returns the current timestamp in milliseconds, so it has a high probability of being unique.
   - We store the result of `setTimeout` in the `intervalMap`. The key is the `id`, and the value is the timeout reference returned by `setTimeout`. This allows us to clear the specific timeout later.
   - We immediately call `recursiveTimeout` to start the interval process.

3. **Custom Interval Clearing (`customClearInterval`):**
   - This function accepts an `id` and checks if that `id` exists in the `intervalMap`.
   - If it exists, `clearTimeout` is called with the associated timeout reference, effectively stopping further executions of `fn`.
   - The `id` and its corresponding timeout reference are removed from `intervalMap` to free up memory and keep the map clean.

Recursion is a key pattern used in the solution for scheduling future calls. Each time `fn` executes, `recursiveTimeout` schedules the next execution. Instead of a standard interval where the delay between calls is constant, this pattern allows for the dynamic calculation of the delay based on the execution count, providing a linearly increasing interval between the calls.

Data structure usage (Map) allows us to efficiently keep track of and manage the different timeouts, providing a clean way to cancel them when necessary.

Example Walkthrough

Let's illustrate the solution approach with an example:

Suppose we want to schedule a function `fn` that simply logs "Hello, World!" to the console. We want the first execution to happen after an initial delay of 1000ms (1 second), and then each subsequent execution to be delayed by an additional 500ms.

We can declare `fn` as follows:

```
1 function fn() {
2     console.log("Hello, World!");
3 }
```

Now let's use the `customInterval` with `delay` of 1000ms and `period` of 500ms:

```
1 const id = customInterval(fn, 1000, 500);
```

Here's a step-by-step explanation of what happens next:

1. A unique `id` for this interval is generated. Let's assume the `id` returned by `Date.now()` is 1623249046531.

2. The `intervalMap` will hold this `id` as a key with the initial timeout reference as its value.

3. The `fn` function is scheduled to execute after 1000ms for the first time, as `count` is 0.

4. After 1000ms, `fn` executes and logs "Hello, World!" to the console for the first time. The `count` is then incremented to 1.

5. The `recursiveTimeout` function inside `customInterval` schedules the next execution of `fn` using a new timeout. The new delay is calculated as 1000 + (500 * 1) = 1500ms.

6. After 1500ms, `fn` executes for the second time, logging "Hello, World!" again, and `count` increases to 2.

7. This process continues with the delay increasing by 500ms after each execution.

If we want to stop this pattern, we can call `customClearInterval` and pass our `id` (1623249046531) as an argument:

```
1 customClearInterval(id);
```

Upon calling this function:

1. The `customClearInterval` function looks for the `id` in `intervalMap`.

2. If it finds the `id`, it calls `clearTimeout`, using the timeout reference, which stops future scheduled executions of `fn`.

3. The `id` is removed from `intervalMap`, cleaning up the reference and ensuring no memory leaks.

This example shows how the `customInterval` and `customClearInterval` functions manage the scheduled execution of `fn` with increasing intervals and how we can stop the schedule whenever needed.

Python Solution

```python
1  from threading import Timer
2  from time import time
3
4  # Dictionary to keep track of custom intervals
5  interval_map = {}
6
7
8  def custom_interval(fn, delay, period):
9      """
10     Creates a custom interval that mimics setInterval but allows for increasing delay durations.
11
12     :param fn: The function to execute after each delay period.
13     :param delay: The initial delay before the function execution in milliseconds.
14     :param period: The additional delay added after each execution in milliseconds.
15     :return: An ID that can be used to clear the interval.
16     """
17     def recursive_timeout(count):
18         """
19         Helper function that recursively sets timers.
20
21         :param count: The number of times the function has been called.
22         """
23         # Calculate new delay with added period for each interval
24         current_delay = delay + period * count
25
26         # Schedule the function to be called after the calculated delay
27         timer = Timer(current_delay / 1000, lambda: recursive_timeout(count + 1))
28         timer.start()  # Start the timer
29
30         fn()  # Execute the provided function
31
32         # Store the timer reference using the interval id
33         interval_map[interval_id] = timer
34
35     # Generate a unique ID for this interval
36     interval_id = int(time() * 1000)
37     recursive_timeout(0)  # Start the interval with 0 count #
38     return interval_id  # Return the unique ID
39
40
41 def custom_clear_interval(interval_id):
42     """
43     Clears a custom interval by ID.
44
45     :param interval_id: The ID of the interval to clear.
46     """
47     # Check if the interval id exists in the dictionary
48     if interval_id in interval_map:
49         # Get the timer and cancel it
50         interval_map[interval_id].cancel()
51
52         # Remove the interval ID from the dictionary
53         del interval_map[interval_id]
```

Java Solution

```java
1  import java.util.concurrent.*;
2  import java.util.Map;
3  import java.util.HashMap;
4
5  public class CustomInterval {
6      private Map<Long, ScheduledFuture<?>> intervalMap = new HashMap<>(); // Map to keep track of custom intervals
7      private ScheduledExecutorService executorService = Executors.newScheduledThreadPool(10);
8      // Assuming a maximum of 10 concurrent intervals. Adjust the handle_pool size as needed.
9
10     /**
11      * Creates a custom interval that mimics setInterval but allows for increasing delay durations.
12      *
13      * @param runnable The function (Runnable) to execute after each delay period.
14      * @param delay The initial delay before the function execution in ms.
15      * @param period The additional delay added after each execution in ms.
16      * @return An ID that can be used to clear the interval.
17      */
18     public long customInterval(Runnable runnable, long delay, long period) {
19         long[] count = new long[]{0}; // Counter to track the number of times the function has been called
20
21         // Helper class that recursively schedules the runnables
22         class RecursiveTask implements Runnable {
23             public void run() {
24                 // Execute the provided runnable
25                 runnable.run();
26                 // Increment the count
27                 count[0]++;
28                 // Reschedule with increased delay
29                 ScheduledFuture<?> scheduledFuture = executorService.schedule(this, delay + period * count[0], TimeUnit.MILLISECONDS);
30                 intervalMap.put(id, scheduledFuture);
31             }
32         }
33
34         // Generate a unique ID for this interval
35         long id = System.currentTimeMillis();
36         RecursiveTask task = new RecursiveTask();
37         // Schedule the task to start after the initial delay
38         ScheduledFuture<?> scheduledFuture = executorService.schedule(task, delay, TimeUnit.MILLISECONDS);
39         intervalMap.put(id, scheduledFuture);
40
41         return id; // Return the unique ID
42     }
43
44     /**
45      * Clears a custom interval by ID.
46      *
47      * @param id The ID of the interval to clear.
48      */
49     public void customClearInterval(long id) {
50         ScheduledFuture<?> scheduledFuture = intervalMap.get(id);
51         if (scheduledFuture != null) {
52             // Cancel the scheduled task
53             scheduledFuture.cancel(false);
54             // Remove the interval ID from the map
55             intervalMap.remove(id);
56         }
57     }
58 }
```

C++ Solution

```cpp
1  #include <functional>
2  #include <chrono>
3  #include <thread>
4  #include <map>
5  #include <mutex>
6
7  // A map to keep track of custom intervals
8  std::map<long long, std::thread> intervalMap;
9  std::mutex intervalMapMutex; // Mutex to protect access to the intervalMap
10
11 /**
12  * Creates a custom interval that mimics setInterval but allows for increasing delay durations.
13  *
14  * @param fn A function to execute after each delay period.
15  * @param delay The initial delay before the function execution in ms.
16  * @param period The additional delay added after each execution in ms.
17  * @return A unique ID that can be used to clear the interval.
18  */
19 long long customInterval(const std::function<void()>& fn, int delay, int period) {
20     // Counter to track the number of times the function has been called
21     int count = 0;
22
23     // Generate a unique ID for this interval
24     long long id = std::chrono::system_clock::now().time_since_epoch().count();
25
26     // Helper lambda function that recursively sets timeouts
27     std::function<void()> recursiveTimeout = [&fn, delay, period, &count, id]() {
28         std::this_thread::sleep_for(std::chrono::milliseconds(delay + period * count));
29         {
30             // Ensure exclusive access to the intervalMap before checking
31             std::lock_guard<std::mutex> lock(intervalMapMutex);
32             if (intervalMap.find(id) == intervalMap.end()) { // If the thread has stopped, break the recursion.
33                 return; // If the interval is not found, stop the recursion.
34             }
35         }
36         fn(); // Execute the provided function
37         count++; // Increment the count
38         recursiveTimeout(); // Set up the next interval
39     };
40
41     // Start the interval in a new thread
42     std::thread intervalThread(recursiveTimeout);
43     {
44         // Store the thread in the intervalMap
45         // Ensure exclusive access to the intervalMap before inserting
46         std::lock_guard<std::mutex> lock(intervalMapMutex);
47         intervalMap[id] = move(intervalThread);
48     }
49
50     // Return the unique ID
51     return id;
52 }
53
54 /**
55  * Clears a custom interval by ID.
56  *
57  * @param id The ID of the interval to clear.
58  */
59 void customClearInterval(long long id) {
60     // Check if the interval ID exists in the map
61     std::lock_guard<std::mutex> lock(intervalMapMutex);
62     auto it = intervalMap.find(id);
63     if (it != intervalMap.end()) {
64         // Detach the thread if running, join the thread before erasing it.
65         it->second.join();
66         // Remove the interval ID from the map
67         intervalMap.erase(it);
68     }
69 }
```

Typescript Solution

```typescript
1  const intervalMap = new Map<number, NodeJS.Timeout>(); // Map to keep track of custom intervals
2
3  /**
4   * Creates a custom interval that mimics setInterval but allows for increasing delay durations.
5   *
6   * @param {Function} fn - The function to execute after each delay period.
7   * @param {number} delay - The initial delay before the function execution in ms.
8   * @param {number} period - The additional delay added after each execution in ms.
9   * @returns {number} An ID that can be used to clear the interval.
10  */
11 function customInterval(fn: Function, delay: number, period: number): number {
12     let count = 0; // Counter to track the number of times the function has been called
13
14     // Helper function that recursively sets timeouts
15     function recursiveTimeout() {
16         // Schedule the function to be called after the calculated delay
17         const timeoutId = setTimeout(() => {
18             fn(); // Execute the provided function
19             count++; // Increment the count
20             recursiveTimeout(); // Set up the next interval
21         }, delay + period * count);
22
23         intervalMap.set(id, timeoutId); // Store the timeout reference in the map
24     }
25
26     // Generate a unique ID for this interval
27     const id = Date.now();
28     recursiveTimeout(); // Start the interval
29     return id; // Return the unique ID
30 }
31
32 /**
33  * Clears a custom interval by ID.
34  *
35  * @param {number} id - The ID of the interval to clear.
36  */
37 function customClearInterval(id: number) {
38     // Check if the interval ID exists in the map
39     if (intervalMap.has(id)) {
40         // If the thread has stopped, clear the timeout and
41         clearTimeout(intervalMap.get(id)!);
42         intervalMap.delete(id);
43     }
44 }
```

Time and Space Complexity

Time Complexity

The time complexity of the `customInterval` function primarily depends on the number of times the callback function `fn` is executed. However, since the scheduling of the function execution isn't a part of the actual computation, the time complexity for the JavaScript runtime to handle the execution of `customInterval` itself is O(1). The time complexity of `fn` will depend on the implementation of the function that is passed to `customInterval`.

For `customClearInterval`, the time complexity is O(1) since it performs a constant number of operations: checking if the `id` exists in the map and potentially clearing the timeout and removing the `id` from the map.

Space Complexity:

The space complexity of the `customInterval` function is O(n), where n is the number of active intervals. This is because it stores each interval's ID and corresponding timeout in the `intervalMap`.

Similarly, the space complexity for `customClearInterval` is O(1). It does not allocate additional space that grows with the size of the input, as it simply removes an element from `intervalMap`.