636. Exclusive Time of Functions

Medium Stack Array

Problem Description

In this problem, we're dealing with a single-threaded CPU that executes a series of function calls. Each function has a unique ID ranging from `0` to `n-1`, where `n` is the total number of functions. We have logs of when each function starts and ends, with their respective timestamps.

The logs are formatted with each entry looking like this: `"function_id:start_or_end:timestamp"`. An important detail is that functions may start and end multiple times, as recursion is possible in the execution.

The task is to calculate the exclusive time for each function, which is the total time that function spent executing in the CPU. By "exclusive," we mean that we're only interested in the time periods when that function was the one currently being executed by the CPU (top of the call stack). The outcome should be an array of integers where each index `i` corresponds to the function with ID `i` and the value is that function's exclusive time.

Intuition

To solve this problem, we can maintain the call stack of the CPU with a stack data structure. We iterate through the log entries, processing start and end signals. When a function starts, we add its ID to the stack. When it ends, we pop it from the stack. By keeping track of the current timestamp, we can calculate how much time has passed since we last encountered a log entry.

The solution involves a few key steps:

- Initialize an answer array to hold the exclusive times for each function, all set initially to zero.
- Use a stack to keep track of active function calls (function IDs) where the most recent call is at the top.
- Define a variable to keep track of the current time (`curr`). Initially, we set it to an invalid timestamp (e.g., `-1`) because we haven't started processing any function yet.
- Iteratively process each log entry:
  - Split the log into parts to extract the function ID, the action (`start` or `end`), and the timestamp.
  - If the action is `start`, update the exclusive time of the function at the top of the stack by adding the difference between the current timestamp and the previous `curr` value (if the stack is not empty). Then push the new function ID onto the stack and update `curr` to the new timestamp.
  - If the action is `end`, pop the function ID from the stack, add the time it took to execute to the corresponding index in the answer array (inclusive of start and end timestamp), and update `curr` to one more than the current timestamp to account for the passage of time.

By the end of processing, the answer array will contain the exclusive times for each function, fulfilling the requirements of the problem.

Solution Approach

The solution employs a stack data structure and a list to maintain the exclusive times for each function. The stack is used to simulate the behavior of a call stack in a single-threaded CPU, where only the top function is currently executing, and each nested function call pushes a new frame onto it.

Here's the step-by-step implementation walkthrough:

1. Initialize an array called `ans` with `n` zeros, where `n` is the total number of functions. This array will store the exclusive time for each function.
2. Create an empty list named `stk` to represent the call stack.
3. Set a variable `curr` to `-1`. This will be used to store the most recent timestamp we've processed.
4. Loop through each log entry in `logs`. For each `log`: a. Split the string using `log.split(':')` to get the function ID (`fid`), the type of log (`'start'` or `'end'`), and the timestamp (`ts`). b. Convert `fid` and `ts` to integers.
5. If the type of log is `'start'`: a. If the stack is not empty, update the top function's exclusive time by adding the difference between the current timestamp and the previous `curr` value to `ans[stk[-1]]`. The top function of the stack would have been executing so far, and we now have to pause its timer. b. Push the new function ID onto the stack. `stk.append(fid)` c. Update `curr` to the current timestamp `ts`.
6. If the type of log is `'end'`: a. Pop the last function ID off the stack with `fid = stk.pop()` since this function has finished executing. b. Add the difference between the current timestamp and the previous `curr` value, inclusive of the end timestamp, to `ans[fid]`. This adds the time for the function execution to `ans[fid]`. c. Update `curr` to `ts + 1` as the next function (if any) would start executing at the next timestamp.
7. After processing all the logs, return the `ans` array, which now contains the exclusive times for each function.

With this approach, we're able to simulate the execution of the functions on the CPU and calculate the exclusive times as needed. The use of the stack is crucial here because it correctly models the nested nature of function calls. Additionally, carefully managing the `curr` timestamp allows us to measure execution times correctly as we process the logs.

Example Walkthrough

Let's consider a small example to clearly illustrate the solution approach. Suppose we have `n = 2` functions, and the logs for their execution on the single-threaded CPU are given as follows:

```
logs = ["0:start:0", "1:start:2", "1:end:5", "0:end:6"]
```

To calculate the exclusive times of these functions, follow these steps:

1. Initialize `ans = [0, 0]` to store the exclusive times for the 2 functions.
2. Create an empty stack `stk`.
3. Set `curr` to `-1` as the initial timestamp before any functions have started.
4. Process the first log entry `"0:start:0"`.
   - Split the log into `fid = 0`, `type = 'start'`, and `ts = 0`.
   - Since the stack is empty, we don't update any function's exclusive time.
   - Push `fid` onto the stack, so `stk` becomes `[0]`.
   - Update `curr` to `0`.
5. Process the second log entry `"1:start:2"`.
   - Split the log into `fid = 1`, `type = 'start'`, and `ts = 2`.
   - Since the stack is not empty, update function 0's exclusive time. `ans[0] += 2 - 0`, so `ans` becomes `[2, 0]`.
   - Push `fid` onto the stack, so `stk` becomes `[0, 1]`.
   - Update `curr` to `2`.
6. Process the third log entry `"1:end:5"`.
   - Split the log into `fid = 1`, `type = 'end'`, and `ts = 5`.
   - Pop `fid` from the stack (`1` is popped, leaving `stk` as `[0]`).
   - Update function 1's exclusive time. `ans[1] += 5 - 2 + 1`, so `ans` becomes `[2, 4]`.
   - Update `curr` to `5 + 1`.
7. Process the fourth log entry `"0:end:6"`.
   - Split the log into `fid = 0`, `type = 'end'`, and `ts = 6`.
   - Pop `fid` from the stack (`0` is popped, leaving `stk` as `[]`).
   - Update function 0's exclusive time. `ans[0] += 6 - 5 + 1`, so `ans` becomes `[4, 4]`.
   - Update `curr` to `6 + 1`.

After processing all log entries, the `ans` array `[4, 4]` contains the exclusive execution times for functions `0` and `1`. Function `0` was active from time `0` to `2` and from `5` to `6` for a total of 4 units of time, and function `1` was active from time `2` to `5` for a total of 4 units of time as well. Therefore, the final answer is `[4, 4]`.

Solution Implementation

Python

```python
class Solution:
    def exclusiveTime(self, n: int, logs: List[str]) -> List[int]:
        # Initialize a list to hold the exclusive time for each function.
        exclusive_times = [0] * n

        # Use a stack to keep track of the function call hierarchy.
        call_stack = []

        # Store a pointer to the current time (initially set to an invalid time).
        current_time = -1

        # Process each log entry.
        for log in logs:
            # Split the string log into parts.
            parts = log.split(':')

            # The function id is the first part of the log, converted to an integer.
            function_id = int(parts[0])

            # The timestamp is the third part of the log, converted to an integer.
            timestamp = int(parts[2])

            # If the log entry indicates a start event:
            if parts[1] == 'start':
                # If there's an ongoing function, update its exclusive time.
                if call_stack:
                    exclusive_times[call_stack[-1]] += timestamp - current_time

                # Push the current function onto the stack.
                call_stack.append(function_id)

                # Update the current time to the new start time.
                current_time = timestamp
            else:
                # Otherwise, it's an end event.
                # Pop the last function from the stack.
                function_id = call_stack.pop()

                # Update the exclusive time for this function.
                exclusive_times[function_id] += timestamp - current_time + 1

                # Update the current time to the end time + 1 since the next time unit
                # will indicate the start of the next event.
                current_time = timestamp + 1

        # Return the list of calculated exclusive times.
        return exclusive_times
```

Java

```java
class Solution {
    public int[] exclusiveTime(int numFunctions, List<String> logs) {
        // This array holds the exclusive time for each function.
        int[] exclusiveTimes = new int[numFunctions];
        // Stack to keep track of the function calls.
        Deque<Integer> functionStack = new ArrayDeque<>();
        // Previous timestamp to calculate the duration.
        int prevTimestamp = -1;

        // Iterate over each log entry.
        for (String log : logs) {
            // Split the log into parts: functionId, event type, and timestamp.
            String[] parts = log.split(":");
            int functionId = Integer.parseInt(parts[0]);
            int timestamp = Integer.parseInt(parts[2]);

            // Check if the event type is start.
            if ("start".equals(parts[1])) {
                // If the stack is not empty, update the exclusive time of the function on the top.
                if (!functionStack.isEmpty()) {
                    exclusiveTimes[functionStack.peek()] += timestamp - prevTimestamp;
                }
                // Push the current function to the stack.
                functionStack.push(functionId);
                // Update the *prevTimestamp* with the current timestamp.
                prevTimestamp = timestamp;
            } else {
                // If the event type is 'end', pop the function from the stack as it ends.
                functionId = functionStack.pop();
                // Update the exclusive time for the popped function.
                exclusiveTimes[functionId] += timestamp - prevTimestamp + 1;
                // Move to the next timestamp as this event marks the end of the function.
                prevTimestamp = timestamp + 1;
            }
        }

        // Return the computed exclusive times for all functions.
        return exclusiveTimes;
    }
}
```

C++

```cpp
#include <vector>
#include <string>
#include <stack>
using namespace std;

class Solution {
public:
    vector<int> exclusiveTime(int n, vector<string>& logs) {
        vector<int> times(n, 0); // Vector to store exclusive times for each function
        stack<int> functionStack; // Stack to keep track of function calls
        int previousTime = 0; // The time of the last event

        for (string& log : logs) {
            char type[10]; // To store whether the event is a 'start' or 'end'
            int functionId, timestamp;
            // Parse the log string into function id:<start/end>:<timestamp>
            sscanf(log.c_str(), "%d:%[^:]:%d", &functionId, type, &timestamp);

            if (type[0] == 's') { // If the event is a function starting
                if (!functionStack.empty()) {
                    // Add the time since the last event to the currently running function
                    times[functionStack.top()] += timestamp - previousTime;
                }
                previousTime = timestamp; // Update the time of the last event
                functionStack.push(functionId); // Push functionId onto the stack
            } else { // If the event is function ending
                // Pop the top function, add the time since the last event to it,
                // and add one because an "end" logs the time at the end of the unit
                times[functionId] += timestamp - previousTime + 1;
                functionStack.pop();
                previousTime = timestamp + 1; // Update the time of the last event, moving past the end timestamp
            }
        }
        return times; // Return the vector with calculated exclusive times for each function
    }
};
```

TypeScript

```ts
function exclusiveTime(n: number, logs: string[]): number[] {
    const functionExecTimes = new Array(n).fill(0); // Array to store exclusive times for each function.
    const callStack: [number, number][] = []; // Stack to keep track of function calls.

    // Iterate through each log in the logs array.
    for (const log of logs) {
        // Splitting the log string into useful parts: function id, state ('start' or 'end'), and timestamp.
        const parts = log.split(':');
        const [functionId, state, timestamp] = [Number(parts[0]), parts[1], Number(parts[2])];

        if (state === 'start') {
            // If there is a function currently being executed, pause it and update its execution time.
            if (callStack.length !== 0) {
                const previousFunction = callStack[callStack.length - 1];
                functionExecTimes[previousFunction[0]] += timestamp - previousFunction[1];
            }
            // Start the new function by pushing it onto the stack with its start time.
            callStack.push([functionId, timestamp]);
        } else {
            // Function is ending, pop from stack to get its start time and calculate the time spent.
            const startedFunction = callStack.pop()!;
            functionExecTimes[startedFunction[0]] += timestamp - startedFunction[1] + 1;

            // If there's another function that was paused, resume its timer.
            if (callStack.length !== 0) {
                callStack[callStack.length - 1][1] = timestamp + 1;
            }
        }
    }

    // Return the array containing the exclusive times for each function.
    return functionExecTimes;
}
```

```python
class Solution:
    def exclusiveTime(self, n: int, logs: List[str]) -> List[int]:
        # Initialize a list to hold the exclusive time for each function.
        exclusive_times = [0] * n

        # Use a stack to keep track of the function call hierarchy.
        call_stack = []

        # Store a pointer to the current time (initially set to an invalid time).
        current_time = -1

        # Process each log entry.
        for log in logs:
            # Split the log into parts.
            parts = log.split(':')

            # The function id is the first part of the log, converted to an integer.
            function_id = int(parts[0])

            # The timestamp is the third part of the log, converted to an integer.
            timestamp = int(parts[2])

            # If the log entry indicates a start event:
            if parts[1] == 'start':
                # If there's an ongoing function, update its exclusive time.
                if call_stack:
                    exclusive_times[call_stack[-1]] += timestamp - current_time

                # Push the current function onto the stack.
                call_stack.append(function_id)

                # Update the current time to the new start time.
                current_time = timestamp
            else:
                # Otherwise, it's an end event.
                # Pop the last function from the stack.
                function_id = call_stack.pop()

                # Update the exclusive time for this function.
                exclusive_times[function_id] += timestamp - current_time + 1

                # Update the current time to the end time + 1 since the next time unit
                # will indicate the start of the next event.
                current_time = timestamp + 1

        # Return the list of calculated exclusive times.
        return exclusive_times
```

Time and Space Complexity

Time Complexity

The time complexity of the code is $O(N)$ where N is the number of logs. This is because the code iterates through the list of logs only once, and for each log, it does a constant amount of work (splitting the log into parts, accessing the last element from the stack, adding time to the answer list).

Space Complexity

The space complexity of the code is $O(N)$ as well. In the worst case, where all function calls are started before any is finished, the stack (named `stk` in the code) would grow to contain all N function ids. Additionally, the answer list (named `ans`) requires space for n integers, where n is the number of functions (which is different from N, the number of logs).