158. Read N Characters Given read4 II - Call Multiple Times

Hard Array Interactive Simulation

Leetcode Link

Problem Description

The problem provides a hypothetical file system and a function `read4` which can read 4 characters from a file at a time and store them into an array `buf4`. The main task is to implement a new method `read` which can read `n` characters from the file and store them in an array `buf`. This `read` function can be called multiple times, which adds complexity because it needs to handle subsequent reads correctly, taking into consideration what was read in previous calls.

Constraints are that you can't access the file directly, and your class should be stateful to remember its state between multiple calls to `read`. Moreover, the `buf` array provided as an argument to the `read` method has sufficient space to store `n` characters.

The description also underscores that the static/class variables maintained in your Solution class must be reset as they are persistent across multiple test cases. The function `read` returns the number of characters actually read and stored in `buf`.

Intuition

To solve this problem, we build around the functionality of the `read4` method, which is provided to us. Since we can only read the file by using `read4`, which reads up to 4 characters at a time, we have to call `read4` multiple times until we have read `n` characters or until we reach the end of the file.

However, since the `read` method can be called multiple times and should continue reading from where it left off, we need some stateful variables to store information between calls. This is necessary to keep track of any leftover characters from the last call to `read4` that were not used in the previous calls to `read`.

Here's where the variables `self.buf4`, `self.i`, and `self.size` come into play. The variable `self.buf4` is an array of 4 characters to store data from the `read4` calls. `self.i` keeps track of the current position in `buf4`, and `self.size` stores the number of characters read in the last call to `read4`.

The intuition behind the solution is to use a loop to fill `buf` with characters until we've read `n` characters or we've read all available characters in the file. Inside the loop, we check if we need to call `read4` to fill our buffer `buf4` (when `self.i == self.size`). If `read4` returns 0, it means we've reached the end of the file, and we can break out of the loop.

After filling `buf4`, we transfer characters from `buf4` to `buf` while incrementing both `self.i` and our counter `j` up to `n`, ensuring we don't read beyond the requested number of characters or beyond what's available in the buffer.

The method returns the number of characters written to `buf`, indicated by `j`.

Solution Approach

The solution involves calling the API `read4` and handling the buffer management manually. The two important aspects here are filling the buffer `buf` appropriately, and maintaining the state between multiple calls to `read`. The implementation can be described in the following steps:

1. Initialization: Create instance variables within the `Solution` class to keep track of the buffer `self.buf4` from `read4`, the current index `self.i` in `buf4`, and the number of characters `self.size` read in the last call to `read4`.
2. Iteration: Use a while-loop to continue reading characters until we have read `n` characters (the target amount) or until there are no more characters to read from the file.
3. Filling buf4: Inside the loop, check if `self.i` equals `self.size`, which means we have processed all characters in `buf4` from the previous `read4` call, or it's the first iteration. If true, call `read4(self.buf4)` to read the next chunk of characters from the file into `buf4`. The number of characters actually read is stored in `self.size`, and `self.i` is reset to 0.
4. Transferring to buf: After ensuring `buf4` is filled, enter another loop which runs as long as there are characters left to read (`j < n`) and there are characters available in `buf4` (`self.i < self.size`). During this loop, copy characters one by one from `self.buf4` to the target buffer `buf` using `buf[j] = self.buf4[self.i]`. Increment `self.i` and `j` with each character copied.
5. End of File Handling: If `read4` returns 0, it indicates the end of the file. At this point, break out of the loop since no more characters can be read.
6. Return Value: After filling `buf` or when the file end is reached, exit the loop and return `j`, the number of characters actually read and written into `buf`.

Using these steps, the algorithm ensures that the reading process can be paused and resumed across multiple calls to `read`. It correctly handles buffering of characters and maintains state between calls. Moreover, this approach does not depend on how many times `read` is called or the number of characters requested in each call; it always returns the correct number of characters read from the file.

Example Walkthrough

Let's consider a file containing the text "HelloWorld" and assume we want to read its contents using our method `read`. For demonstration, let's say we're making two calls to `read`: the first call to read 8 characters, and the second call to read 5 characters. Here's how our solution handles it:

1. **First `read` call to read 8 characters**:
   - We initialize `self.buf4` to `[]`, and set both `self.i` and `self.size` to 8.
   - Enter the while loop because the target `n` is 8, and our read count `j` is currently 8.
   - Since `self.i == self.size`, we call `read4` and it fills `self.buf4` with the first 4 characters, so `self.buf4 = ['H', 'e', 'l', 'l']` and `self.size = 4`.
   - We now copy from `self.buf4` to `buf`. Since our target `n` is 8, we continue to copy until `self.buf4` is exhausted or we reach the target `n`.
   - By the end of this iteration, `buf = ['H', 'e', 'l', 'l', self.i = 4`, and `j = 4`. We still need to read 4 more characters to meet our target.
   - We go through the loop again, call `read4` which now fills `self.buf4` with the next 4 characters: `['o', 'W', 'o', 'r']`, and `self.size = 4`.
   - We transfer these 4 characters to `buf`, resulting in `buf = ['H', 'e', 'l', 'l', 'o', 'W', 'o', 'r']` and updating `j` to 8, which meets our target. We exit the loop and return 8.
2. **Second `read` call to read 5 characters**:
   - We start again with `self.buf4` still having `['o', 'W', 'o', 'r']` but this time our initial `self.i` is 4 and `self.size` is 4 since we didn't reset them after the last call (as they're being used to maintain state).
   - As `self.i` is equal to `self.size`, we call `read4` again. It reads the final 2 characters of the file: `['l', 'd']`, and sets `self.size = 2`.
   - We copy `['l', 'd']` into `buf` to get `buf = ['l', 'd']`. Now `self.i` is 2, `self.size` is 2, and our read count `j` is 2.
   - Since we have reached the end of the file and `read4` cannot read more characters, the loop ends. We can't read 5 characters as requested because only 2 are left in the file. Thus we return 2.

So, over the course of two `read` calls asking for 8 and then 5 characters, we returned 8 characters the first time and 2 the second time, accurately reflecting the contents of "HelloWorld" and the stateful nature of consecutive reads.

Python Solution

```python
class Solution:
    def __init__(self):
        self.buf4_s = [''] * 4  # Buffer to store read characters from read4
        self.buf4_index = 0  # The next read position in buf4_s
        self.buf4_size = 0  # The number of characters read from read4

    def read(self, buf: List[str], n: int) -> int:
        """
        Reads n characters from the file using read4.

        :param buf: Destination buffer to store characters
        :param n: Number of characters to read
        :return: The number of characters actually read
        """

        # The total number of characters read so far
        total_read = 0
        # Loop until we read n characters or reach the end of the file
        while total_read < n:
            if self.buf4_index == self.buf4_size:  # All characters in buffer are read
                self.buf4_size = read4(self.buf4_4)  # Read next 4 characters
                self.buf4_index = 0  # Reset buffer index
                if self.buf4_size == 0:  # End of file reached
                    break

            # Transfer characters from buffer_4 to buf while we haven't read n characters
            while total_read < n and self.buf4_index < self.buf4_size:
                buf[total_read] = self.buf4_s[self.buf4_index]
                self.buf4_index += 1
                total_read += 1

        # Return the total number of characters read
        return total_read

```

Java Solution

```java
// Extends the functionality of Reader4 class by implementing a custom reader method.
public class Solution extends Reader4 {
    private char[] internalBuffer = new char[4]; // Buffer to hold read characters from read4.
    private int bufferIndex = 0; // Pointer to track the current position within internalBuffer.
    private int contentSize = 0; // Amount of valid characters in internalBuffer after a call to read4.

    /**
     * Reads n characters from the file into buf.
     *
     * @param buf Destination buffer to store the characters read from the file.
     * @param n Number of characters to read from the file.
     * @return The number of actual characters read, which could be less than n if EOF is reached.
     */
    public int read(char[] buf, int n) {
        int readCharsCount = 0; // Counter to keep track of the number of characters read into buf.

        // Continue reading until the specified number of characters (n) is read or until the end of file is reached.
        while (readCharsCount < n) {
            // If internalBuffer has been fully read, reload it with new data from read4.
            if (bufferIndex == contentSize) {
                contentSize = read4(internalBuffer); // Read 4 characters and save the number of characters read.
                bufferIndex = 0; // Reset buffer index.
                // If no characters are read, end of file has been reached, so break out of the loop.
                if (contentSize == 0) {
                    break;
                }
            }
            // Transfer characters from the internal buffer to buf until we either fill buf or exhaust internalBuffer.
            while (readCharsCount < n && bufferIndex < contentSize) {
                buf[readCharsCount++] = internalBuffer[bufferIndex++];
            }
        }

        // Return the total number of characters that were successfully read into buf.
        return readCharsCount;
    }
}
```

C++ Solution

```cpp
/**
 * The read4 API is defined in the parent class Reader4.
 *     int read4(char *buf4);
 */

class Solution {
public:
    /**
     * Reads n characters from the file and writes into buffer.
     * @param buffer Destination buffer
     * @param n Number of characters to read
     * @return The number of actual characters read
     */
    int read(char* buffer, int n) {
        int totalRead = 0; // Total number of characters read

        // Continue reading until we have read n characters or there is no more to read.
        while (totalRead < n) {
            // Refill the tempBuffer if it's empty
            if (bufferIndex == bufferSize) {
                bufferSize = read4(tempBuffer);
                bufferIndex = 0; // Reset buffer index
                // If no characters were read, we've reached the end of the file
                if (bufferSize == 0) break;
            }

            // Read from tempBuffer into buffer until we have read n characters
            // or the tempBuffer is exhausted.
            while (totalRead < n && bufferIndex < bufferSize) {
                buffer[totalRead++] = tempBuffer[bufferIndex++];
            }
        }

        return totalRead; // Return the total number of characters read
    }

private:
    char tempBuffer[4]; // Temporary buffer to store read4 results
    int bufferIndex = 0; // Index for the next read character in tempBuffer
    int bufferSize = 0; // Represents how many characters read4 last read into tempBuffer
};
```

Typescript Solution

```typescript
// Define global variables to keep track of the temporary buffer and indices.
let tempBuffer: string[] = new Array(4); // Temporary buffer to store read4 results
let bufferIndex: number = 0; // Index for the next read character in tempBuffer
let bufferSize: number = 0; // Represents how many characters read4 last read into tempBuffer

// Mocked read4 API function to match the context. This function should be replaced with the actual implementation.
function read4(buf4: string[]): number {
    // Implementation of read4 API should be provided.
    return 0;
}

/**
 * Reads 'n' characters from the file and writes into the buffer.
 * @param buffer Destination buffer which is a string array
 * @param n Number of characters to read
 * @return The number of actual characters read
 */
function read(buffer: string[], n: number): number {
    let totalRead: number = 0; // Total number of characters read

    // Continue reading until we have read 'n' characters or there is no more content to read.
    while (totalRead < n) {
        // Refill the tempBuffer if it's empty.
        if (bufferIndex === bufferSize) {
            bufferSize = read4(tempBuffer);
            bufferIndex = 0; // Reset buffer index

            // If no characters were read, we've reached the end of the file.
            if (bufferSize === 0) break;
        }

        // Read from tempBuffer into the buffer until we have read 'n' characters or tempBuffer is exhausted.
        while (totalRead < n && bufferIndex < bufferSize) {
            buffer[totalRead++] = tempBuffer[bufferIndex++];
        }
    }

    // Return the total number of characters read.
    return totalRead;
}
```

Time and Space Complexity

The time complexity of this `read` function is $O(n)$. Each call to `read4` reads at most 4 characters until `n` characters are read or there is no more content to read from the file. In the worst case, the function will call `read4` `ceil(n/4)` times to read `n` characters.

The space complexity of the solution is $O(1)$. The solution uses a constant amount of extra space, `buf4` of size 4, to store the read characters temporarily and a few integer variables (`self.i`, `self.size`, and `j`) to keep track of positions, which does not depend on the size of the input `n`.