Report for CS 307 (Operating Systems) Programming Assignment #4

Hagverdi Ibrahimli

30014

In this report, we will delve into the implementation of a program simulating the heap management processes such as allocating and de-allocating memory slots for different users. The program is written in C++ programming language and utilizes a single mutex lock as the synchronization mechanism to ensure atomicity of operations, preventing data race issues, thus allowing multiple threads effectively to run in parallel.

To begin with, here is a self-explanatory pseudocode of the program:

```
BEGIN

1. Create a class HeapManager to simulate a heap manager that supports malloc and free operations.

2. Initialize the HeapManager class with a constructor:
  - Initialize lock as PTHREAD_MUTEX_INITIALIZER.
  - Set heap to NULL.

3. Implement the initHeap method:
  - Accepts the size of the heap.
  - Create a new HeapNode with id -1, given size, and index 0.
  - Print "Memory initialized."
  - Acquire the lock.
  - Call print method.
  - Release the lock.
  - Return 1.

4. Implement myMalloc method:
  - Acquire the lock.
  - Check if the heap is initialized.
  - Iterate through the heap nodes:
    - If the node has id -1:
      - If the node size is greater than requested size:
        - Create a new HeapNode with the given id, requested size, and the current node's index.
        - Update the current node's index and size.
        - Insert the new node at the beginning or in the middle of the list.
        - Print "Allocated for thread [id]."
        - Call print method.
        - Release the lock.
        - Return the index of the new node.
      - If the node size is equal to the requested size:
        - Set the current node's id to the given id.
        - Print "Allocated for thread [id]."
        - Call print method.
        - Release the lock.
        - Return the current node's index.
    - Increment the position.
  - Print "Cannot allocate, requested size [size] for thread [id] is bigger than remaining size."
  - Call print method.
  - Release the lock.
  - Return -1.
```

```
5. Implement myFree method:
  - Acquire the lock.
  - Check if the heap is initialized.
  - Iterate through the heap nodes:
    - If the node has the given id and index:
      - Check if the left chunk is free, merge with the current node.
      - Set the current node's id to -1.
      - Check if the right chunk is free, merge with the current node.
      - Print "Freed for thread [id]."
      - Call print method.
      - Release the lock.
      - Return 1.
    - Increment the position.
  - Print "Cannot free, no chunk allocated for thread [id] at address [index]."
  - Call print method.
  - Release the lock.
  - Return -1.

6. Implement the print method:
  - Iterate through the heap nodes and print "[id][size][index]" for each node.
  - If a node has a next node, print "---" between nodes.

7. Define a HeapNode struct within the HeapManager class:
  - Contains id, size, index, and a pointer to the next node.

8. Declare a HeapNode pointer heap and a pthread_mutex_t lock in the private section of the HeapManager class.

END
```

To simulate the behavior of a heap management library, the program implements *HeapManager* class. In this class, we can point out the following:

1. Public methods
2. Private parameters

As private members for this class, we construct the *HeapNode* struct with the following values: *ID, Size, Index,* and the *next* pointer. This node structure will constitute our main list for keeping track of the allocated slots in the heap, which is pointed to by the *heap* pointer. Additionally, we declare a *pthread_mutex_t* type lock variable, to help synchronize the flow of operation in a multithreaded environment.

As can be observed from the pseudocode, we have some important mechanisms provided by the public methods of the class:

- *HeapManager* constructor to initialize the mutex lock as well as setting the heap pointer to NULL. As a reminder, this *heap* pointer is the head of our linked list which shall help us manage the allocated and free slots in the memory.

- *initHeap(int size)* function is responsible for setting the heap with the *size* parameter. After the setting, we print the memory layout of our heap with the *print()* method. Here, we encapsulate the *print()* method with the mutex lock to ensure its atomicity and prevent any issues related to data racing. Logically, this method shall be called by the main thread before any other secondary thread calls *myMalloc* or *myFree.* Nevertheless, we include it here to guarantee a valid multithreaded program.

- *print()* method traverses the heap list, printing each slot to the user console. The printed statement follows the ordering of *[id][size][index]---[id][size][index]*---etc. For the unallocated slots, the *id* field is provided as *-1.* The size field is simply the allocated size for the slot, followed by the *index* field which denotes the starting address of the slot.

- *myMalloc(int id, int size)* method tries to allocate *size* amount of space for the thread with the provided *id,* using the *first fit approach.* If there is available space in any of the slots, the space is allocated for the given thread. However, it is important to note that, to mitigate internal fragmentation we divide the slot into two (if slot space is more than the queried *size)* and assign the first slot to the thread, maintaining the second one unallocated. To ensure the atomicity of operations and prevent any data race conditions, we wrap critical section of the code (where we access the heap list and modify its values) with mutex *lock.* This ensures that only one thread at a time can modify the list, performing the operations atomically. Upon successful allocation of the memory, we return the starting address of the allocated slot to the caller thread.

- myFree(int id, int index) method attempts to free the allocated memory for the thread with the given id, starting from the index address. Once the provided arguments can be successfully mapped to our heap list, we perform the deletion operation. Importantly, whenever de-allocating the given slot for a thread, we also observe the slot's left and right neighbors such that we can coalesce with them (both, any, or none) if they are also de-allocated. This helps with the problem of external fragmentation. In parallel with the myMalloc method, we use the same synchronization techniques, utilizing mutex lock to wrap the critical sections of the code, again, to create a safe multithreading environment.

Importantly, the *mutex* lock mechanism employed in our implementation is carefully handled such that no deadlock shall occur throughout the lifetime of the program.

In summary, the careful integration of mutex-based synchronization mechanisms ensures that the heap manager operates seamlessly in a multi-threaded environment. The atomicity of operations, facilitated by mutex locks, addresses potential pitfalls associated with concurrent access, providing a robust foundation for memory allocation and deallocation.