Programming Assignment 4

Bilal Berkam Dertli 29267

In this Programming Assignment, I've written allocator.cpp file in order to form a heap simulator by using a linked list. Since multiple threads can access the linked list and try to allocate or deallocate memory at the same time, a mutex is used as a synchronization mechanism to prevent data races in the heap simulator linked list.

A general pseudocode for the program:

Firstly, I formed a single LinkedList implementation which actually includes the functions that the users can use: initHeap, myMalloc, myFree, print. It has a Node* head as usual, and Node is a struct including ID, SIZE, INDEX fields for thread id, size of the allocated/free space, starting index of the space.

When initHeap is called with an appropriate size (>0), **this action firstly locks the global mutex of the linked list,** creates a linked list of 1 head node which represents the initial free space for the heap, and at the end, **unlocks the mutex**.

When myMalloc(thread ID, size) is called with an appropriate size (>0), **this action firstly locks the global mutex of the linked list,** the caller thread starts traversing the linked list from the head pointer to find a free space (denoted with id -1) which has enough space (>= thread's requested size). If the thread finds the space, it splits the free space into 2 such that one of them is for the thread to use, and the other one is the remaining free space if it exists. If the requested space is the same as the first free space, then the space is fully allocated to the thread, leaving no node for a free space. After this action, the thread unlocks the mutex, or if the thread traverses the whole tree but cannot find a free space big enough to cover the request, again **unlocks the mutex**, and returns -1 to indicate that allocation has failed.

When myFree(thread ID, index) is called with an appropriate index (>=0**), this action firstly locks the global mutex of the linked list,** the caller thread starts traversing the linked list from the head pointer to find a node (simulating an allocated space) such that it has the starting index the same as the index parameter given to the function, and the space has been allocated by the currently running thread, which is checked by ID field of the node. If a node is found, firstly the region has been marked as free by changing the ID field of the node as -1, and then the thread looks for the previous and next nodes to see if they are also free. If none of them is free, then thread does nothing, since the current allocated space has been deallocated (by changing the ID field of the node to -1), if only the previous node is free, the size of it expands by newly deallocated node, and current node has been deleted, if only next one is free, size of newly deallocated node increases by next node's size, and next node gets deleted from the linked list, which simulates that the two nodes are merged and became one node with greater size. If both previous and next nodes are free, then a combination of the previously discussed methods is done, which merges the three nodes into one node, having the index of the previous node (since it is the first node to exist in order), and the size of current and next nodes are added to the previous node. Then, current and next nodes are deallocated, meaning that they have been removed from the heap simulator linked list. After deciding on what to do about merging, just before the thread exits myFree function, thread **unlocks the mutex of the linked list,** no matter if there was an allocated space by the running thread with starting index as given index parameter or not. Shortly, irrelevant of the deallocation being successful or not, the thread locks the linked list while entering the function body, and unlocks it just before returning from the function body.

When print() has been called, thread firstly **locks the mutex,** traverses through the linked list and prints the fields of the nodes in the requested format, **unlocks the mutex** and returns.

Locking Mechanism

As discussed in the previous section, I used a coarse-grained locking mechanism for achieving a synchronous linked list. Every thread entering any of the four function body firstly **locks the mutex of the linked list**, does what the function needs to do, **unlocks the mutex**, and returns. As an example:

```
int myMalloc(thread ID, size){
        managerLock.Lock()
        //Implementation goes here, as discussed above
        managerLock.unlock()
}
```

I used this mechanism because it is easy to implement and **provides mutual exclusion of any thread calling the functions of the HeapManager class,** meaning that whenever a thread calls the functions that does an action on the linked list in HeapManager, if **another thread** calls one of the functions described above, this new thread **waits for the other thread to finish its work on the linked list**. This way, the shared variable which is the linked list itself has been protected from any data race happening in the list, which may cause errors, unexpected outputs, and many other problems that are due to interleaving threads.

So, a pseudo code for the functions of HeapManager class:

```
Function begins
Lock the mutex of the linked list
Do whatever do function does: print the list, allocated/deallocate memory, initialize the heap
Unlock the mutex of the linked list
Function ends
```

This ensures that no two thread can work on the linked list at a given time.