Zeynep Türkmen - 29541

Programming Assignment 4

I used a **single mutex** in order to protect the atomicity of the class member list. It acts as a global mutex of the entire class aka the allocator library, thereby its able to protect all the methods of an instance. The class has a private member called *memory list*, for that I used the standard C++ lists and created a node struct that includes tid, start and size information.

Init heap is called only once by the logic of this assignment therefore I did not add any locks to that method.

myMalloc method needs to iterate through the entire list to find a free space for the requested size, for that I locked the method right before the iteration begins as the list may change while its iterating, leading to unexpected results. Since the entire method is based on modifying the structure of the memory list, all the method body needs to be protected by the lock. After the method iterates through the entire list, OR it finds a matching node that it can allocate with sufficient size, after doing all the modifications it releases the lock.

myFree method also need s to iterate through the entire list and do modifications on the structure of the list, so I also locked right before the iteration. The lock protects all the modifications and merge (freeing and merging with previous nodes) operations performed on the list. After all list was iterated or we managed to free and modify the list, the lock gets released.

printMe is a private function that is called only within the locked methods, hence even though it iterates through the list, it does not require a lock.

However, the regular print function requires a lock to be obtained, meaning it cannot print anything outside while other threads are doing some modifications to the list. So it gets the lock before iterating and releases is after printing is completed.

Another thing worth noting is that the functions do not only modify a single lock, but they also both perform a read operation on the entire list and they both modify more than one node at once. The functions job is to simply modify the values of the nodes, merge them if necessary and delete them in some cases. Thereby I chose the coarse-grained locking method and the atomicity of the memory list is always protected plus there is no unnecessary overhead like locking all nodes one by one.

Below is the Pseudocode of the entire program:

```
/*
class HeapManager
   private:
      Create a struct for the list to store threadID, sizeOfMemoryPiece and Starting point
      Create a mutex for atomicity
      Create a list for the class filled with the struct above (from list library)
```

```
    void printMe() {
        iterate through the list
        print the node information
    }


public:
    HeapManager() {
        initialize the mutex
    }

    int initHeap(int size){
        push a node to the private list with the given size, tid of -1 and starting point of 0
        call the print method of the class
        return 1
    }

    int myMalloc(int threadId, int size) {
        acquire the lock before traversing and modifying the class list
        iterate through all the elements in the list
            in the first occurence of a node (x) where its empty and of sufficient size
                create and insert the new node before x
                if remaining size is bigger than zero
                    modify size and start of node x
                else
                    remove x from the list
                print the messages and the updated list
                release the lock and return the newStarting value

        if we couldnt find a matching node, print the messages and the unchanged list
        release the lock
        return -1
    }

    int myFree(int threadId, int start) {
        acquire the lock before traversing and modifying the class list
        iterate through all the elements in the list
```

```
            if there is a matching node to free with the given start and tid value
                if this node is not the first node in the list
                    get the prev node
                    if prev node is free
                        increment prev size with the node to be deleted(x)
                        delete x from the list
                        set x to be the prev

                    mark the node as unoccupied aka set tid as -1

                    get the next value
                    if next value exists and its tid is -1
                        increment x's size with it and delete the next node
                    print the messages and the current list
                    release the lock
                    return 1

        if there was no matching node
        print the messages and the list
        release the lock
        return -1
        }
          void print() {
              get the lock
        iterate through the list
        print the node information
              release the lock
    }
};
*/
```