Doruk Benli
29182

PA4 Report

Introduction

in this programming assignment, we are asked to implement a memory allocator library for heap. We implement a class called HeapManager to mimic the behavior of free lists in heap. We are also asked to implement this class such that it works in a multithreaded manner.

Implementation

We have a struct called node which holds 4 fields, 3 integers corresponding to ID, size and index and a next field. Then our class, the HeapManager has 3 private fields, they are head ptr, for keeping the head of the free list, integer heapsize and a mutex for providing atomicity when working with multiple threads. Even though we have an initialization method, i have written constructor and a destructor which simply makes head node nullptr and destructor deletes all the nodes after we are done. Then we have the initHeap method, which just makes a new node for head with id -1. This function directly returns 1, as we always assume it succeeds. Then we have our own malloc method, we begin by using the lock method of mutex, because this method and our free method are critical sections of our class. myMalloc method, simply puts a current pointer starting from head and will find suitable space for requested amount of allocation, once it finds, it will split the list into 2 and create a new node for right hand side. We wrap our function body with lock and unlock methods, we call unlock right before the return. During our function body we traverse the list with for loop and check if retIndex which stands for return index != -1, which then we call the print function we also written for the class. We do this to print the allocation information. Similarly we also implement our own free method called myFree(), this method will first call lock to ensure safety when working with multiple threads, then it will iterate through linked list and it checks id for correct thread and checks blocks index to find the node to be freed/ deleted. After that it checks the next block, if it is also free, then it merges them together. Then it checks for the previous block to see if its also free to merge them and form even larger block. After we successfully freed and merged our blocks, it will return 1 and print relevant information, if we do not return 1 we in the end of function body will call unlock and just return -1 since we did not free.

Doruk Benli
29182

Pseudocodes

myMalloc(ID, size):

1. Acquire lock on mutex (mtx.lock())
2. Initialize pointer to head of the list
3. Initialize newBlock pointer to nullptr
4. Initialize return index to -1
5. Start a loop until the end of the list:
   - Check if the current block is free and large enough for the requested size:
     - If the block is exactly the size needed, assign it to the requesting thread and store the return index
     - If the block is larger, split it into two blocks: one for the request and the other remains free. Adjust pointers and update return index. Delete the old node.
   - Move to the next block in the list
6. If a block is allocated, print the allocation information and the state of the heap
7. Release the mutex lock (mtx.unlock())
8. Return the index of the allocated block (or -1 if no suitable block is found)

myFree(ID, index):

1. Acquire lock on mutex (mtx.lock())
2. Initialize pointers to head of the list and previous node
3. Start a loop until the end of the list:
   - Check if the current block matches the provided ID and index:
     - If matched, mark the block as free
     - If the next block is free, merge it with the current block (coalescing)
     - If the previous block is free, merge it with the current block (coalescing)
     - Print the freeing information and the state of the heap
     - Release the mutex lock (mtx.unlock())
     - Return 1 to indicate success
   - If not matched, move to the next block in the list
4. If no block is freed (no match found), release the mutex lock (mtx.unlock())
5. Return -1 to indicate failure

Our mutex in private field ensures atomicity by locking and unlocking in correct myMalloc and myFree, wrapping them. Other threads, who try to acquire the lock will have to wait for working thread to release the lock. I have also tried to put lock to print, but since we call it inside myMalloc and myFree, once we try to release the lock we cannot acquire it, hence we have a deadlock.

Doruk Benli
29182