CS307 PA4 Report

Alpay Naçar

31133

Implementation:

Created a class for the heap manager. Implemented my own linked list node structure and added a constructor for node creation. This heap manager is thread safe, meaning that it can work in a multithreaded environment.

InitHeap method creates a linked list with just one node, which has size of given parameter.

MyAlloc method tries to find an open space by iterating through the linked list and checking if id equals to -1 or not. When it finds a space, first creates another node if requested size is bigger than the found size. Then it changes the size and id of the found space.

MyFree method iterates through the linked list to find the node with given index, when it finds, checks if the id it has is the same as given id, if that's the case, the memory will be freed. When freeing the memory, there might occur consecutive free memories, to handle this, coalescing happens. First next node is checked and if it is free, its removed and its size is added to current node. Same thing happens for previous node as well. Also, for an edge case, if someone wants to remove a chunk from the middle, it is allowed. When the index is in the middle of a chunk, it frees half (half doesn't mean %50 of course) of the chunk, first half will stay as it is.

Print method writes all nodes of the linked list to the console. Added an extra semaphore just for print, so that if print is called separately from the

Synchronization:

I choose semaphores for synchronization. Every operation on the linked list locks the mutex inside semaphore and unlocks it after it finishes. Print is an exception, because I used print inside other methods, I didn't used semaphore inside print as well. If print is going to be used separately, the user should be aware of this fact.

Semaphore uses a queue in the background for each wait operation, it adds this operation to queue. When a method finishes, it posts the semaphore so that the next method waiting on the queue can continue its operation meaning that there is no one remaining who is doing operation on the linked list.

Atomicity Of Operations:

To ensure atomicity, semaphore is used for myAlloc and myFree operations. Semaphore covers the whole critical section, which includes traversing the linked list and doing operations on linked list.

Pseudocode:

```
function myMalloc(ID, size):
        sem_wait(&sem); // Acquire the semaphore
        // Critical Section
        sem_post(&sem); // Release the semaphore

function myFree(ID, index):
        sem_wait(&sem); // Acquire the semaphore
        // Critical Section
        sem_post(&sem); // Release the semaphore
```