**2021 – 2022 FALL CS 307 PA4**

**Elif Cemre Durgut - 26493**

Pseudocode of my algorithm:

mutex lock

int **initHeap**(){

lock the mutex

initialize the heap

print()

unlock the mutex

return

}

int **myMalloc**(){

lock the mutex

for i in list:

if i is free:

if i.size == wanted size:

allocate

unlock the mutex

return

else if i.size > wanted size:

divide i to two and allocate the first one

unlock the mutex

return

print()

unlock the mutex

return

}

int **myFree**(){

lock the mutex

for i in list:

if i.id == wanted id and i.index == wanted index:

check the previous and next nodes, if they are free, merge the free blocks

unlock the mutex

return

unlock the mutex

return

}

**Explanation:**

I have used the standard list library in C++, and I wrapped each method body with locks and unlocks as we have seen in the lecture as Attempt 1.

Graphical user interface, text

Description automatically generated (Concurrency.pptx)

In the lecture, we saw that this solution may be problematic if all the threads get stuck at a point but this is not a problem in my solution. Because I use for loops and break/return them right after finding the correct spot to add/delete a node. Moreover, a node to be updated cannot be deleted by another thread because it cannot own the lock.

My solution also provides atomicity because I use only one mutex for all methods so this ensures that a thread cannot use any of the methods while another thread uses a method. Using one mutex and unlocking the mutex before return prevent deadlocks.