CS307 2022 SPRING PA4 – Heap Management
UĞUR ÖZTUNÇ 28176

At first, I defined the struct `node_t` which will store id, size, index and next pointer information, and has a constructor for passing and assigning all values at once. Then the definition of `HeapManager` class comes. In addition to the member functions mentioned in document, there is a private pointer to `node_t` named head which points the head of the free list, a private `pthread_mutex_t` which will help Heap Manager to maintain the concurrency, a private function called merge which I will talk about later, and a default constructor which sets head pointer to `NULL`, and initializes the member mutex lock.

Free list

The free list is a dynamic linked list, all nodes are stored in actual heap and allocated with '`new`'. I did not use additional linked list class. All linked list operations are performed in member functions without using any other library.

`int initHeap(int size)`
Creates the first node of the free list with `ID` = -1, `size` = given size, `index` = 0 and `next` = `NULL`. Prints proper output and list, and returns 1.

`void print()`
A temporary pointer iterates the whole list starting from head, and prints the information of nodes with proper format.

`int myMalloc(int ID, int size)`
At the beginning of the function member mutex is locked, and necessary pointers (`node_t* ptr` and `node_t* prevNode`) are defined for iterating the list to find a suitable chunk to allocate with a while loop. Once a suitable chunk is found, the while loop breaks and `ptr` points that suitable chunk node. If `ptr` is `NULL` after while loop ends, it means that no suitable chunk is found, in this case proper output and the list is printed and function returns -1 after unlocking the mutex. If a suitable chunk is found, a new node is created with '`new`' which is pointed by `newAlloc` and this node's next pointer is `ptr` which is the chunk to be splitted. Then, an if condition checks whether the `newAlloc` must be head or not and ptr's size is decreased and index incremented. After these operations, if ptr's size becomes zero, this means requested size is exact size of the ptr node. In this case, newAlloc's next is set to ptr's next and ptr is deleted with '`delete`'. At last, proper output and linked list is printed and function returns newAlloc's index after unlocking the member mutex.

`int myFree(int ID, int index)`
Same operations as in `myMalloc` function is performed again, member mutex is locked and necessary pointers: `ptr` and `prevNode` is defined. A while loop iterates through list to find target chunk. After the while loop, if `ptr` is `NULL`, it means that target chunk does not exist, in this case proper output and the linked list is printed and function returns -1 after unlocking the member mutex. Otherwise, if the target chunk is found, `ptr` would point it. First target chunk node's id is set to -1. An if block checks for whether there is a need for coalescing or not, and if there is a need for coalescing how it must be performed (page 3, figure 2 in PA4 document). The coalescing operation is performed with `merge` private member function:

> **`void merge(node_t* node1, node_t* node2)`**
> This function merges two nodes by adding the size of second node to the first node's size and linking the first node to second node's next. Then deletes second node with '`delete`'.

-If freed chunk is surrounded by allocated chunks, no need to perform a coalescing.
-If the previous chunk of freed chunk is allocated, but the next chunk of its free, merge freed chunk with its next.
-If the previous chunk is free, but next chunk is allocated, merge previous chunk with freed chunk.
-If freed chunk is surrounded by free chunks in both sides, first merge freed chunk with next chunk and then merge previous chunk with it.

After the coalescing part, proper output and the linked list is printed, and function returns 1 after unlocking the member mutex.

Concurrency

For multiple threads to run concurrently, I have used the member mutex of the `HeapManager` class, which is kind of a global in member functions of the object. The mechanism is very simple: Since both `myMalloc` and `myFree` functions performs critical operations from beginning to end, locking the mutex at the very beginning of these two functions and unlocking the mutex just before returning in both functions was enough to manage concurrency problem.