CS307 - Operating Systems — 2021-2022 Fall

Programming Assignment 4 Report

Doğukan Yıldırım — 28364

The Heap Manager

(a) Pseudocode of Member Functions with Locking Mechanisms

```
int myMalloc(int ID, int size) {
    Create an iterator for linked list of nodes
    Lock the mutex
    For each node in the list:
        if the current node is a free node with enough space:
            Insert a new node before the current node with the given ID and size
            (Inserted new node gets the index of the current node)
            (New node represents the newly allocated space)

            Update the current free node, reduce its size and increase its index
            (Size is reduced by the given size, index increased by given size)
            (Current node represents the remaining free space)

            Print "Allocated for thread {ID}"
            Print all the nodes in the list

            Unlock the mutex
            Return the index of the newly allocated node

    Print "Can not allocate, requested size {size} for thread {ID} is bigger
        than remaining size"
    Print all the nodes in the list

    Unlock the mutex
    Return -1
}

int myFree(int ID, int index) {
    Create an iterator for linked list of nodes
    Lock the mutex
    For each node in the list:
        If the current node has the given ID and index:
            Set the current node's ID to -1 to make it free

            If the previous node has the ID of -1 (i.e. if it's free):
```

```
                Add the previous node's size to the current node's size
                Set the previous node's index to the current node's index
                Erase the previous node from the list of nodes

            If the next node has the ID of -1 (i.e. if it's free):
                Add the next node's size to the current node's size
                Erase the next node from the list of nodes

            Print "Freed for thread {ID}"
            Print all the nodes in the list

            Unlock the mutex
            Return 1

    Print "Can not free, requested node with ID {ID} and index {index} for thread
        {ID} does not exist"
    Print all the nodes in the list

    Unlock the mutex
    Return -1
}
```

(b) Complete Code of allocator.cpp

```
1  struct Node {
2      int ID;
3      int size;
4      int index;
5
6      Node(int myID, int mySize, int myIndex)
7      : ID(myID), size(mySize), index(myIndex) {}
8  };
9
10 ostream& operator << (ostream& os, const Node& myNode) {
11     os << "[" << myNode.ID << "][" << myNode.size << "][" << myNode.index << "]";
12     return os;
13 }
14
15 class HeapManager {
16     private:
17     list<Node> heap;
18     pthread_mutex_t mutex;
19
20     public:
21     int initHeap(int size) {
```

```
22          heap.push_back(Node(-1, size, 0));
23          pthread_mutex_init(&mutex, NULL);
24          print();
25          return 1;
26      }
27
28      int myMalloc(int ID, int size) {
29          list<Node> :: iterator it;
30
31          pthread_mutex_lock(&mutex);
32          for (it = heap.begin(); it != heap.end(); it++) {
33              if (it->ID == -1 && size < it->size) {
34                  heap.insert(it, Node(ID, size, it->index));
35                  it->size -= size; it->index += size;
36
37                  cout << "Allocated for thread " << ID << endl;
38                  print();
39
40                  pthread_mutex_unlock(&mutex);
41                  return it->index - size;
42              }
43          }
44
45          cout << "Can not allocate, requested size " << size << " for thread " << ID
46              << " is bigger than remaining size" << endl;
47          print();
48
49          pthread_mutex_unlock(&mutex);
50          return -1;
51      }
52
53      int myFree(int ID, int index) {
54          list<Node> :: iterator it;
55
56          pthread_mutex_lock(&mutex);
57          for (it = heap.begin(); it != heap.end(); it++) {
58              if (it->ID == ID && it->index == index) {
59                  it->ID = -1;
60
61                  if (prev(it)->ID == -1) {
62                      it->size += prev(it)->size;
63                      it->index = prev(it)->index;
64                      heap.erase(prev(it));
65                  }
66
67                  if (next(it)->ID == -1) {
68                      it->size += next(it)->size;
```

```
                    heap.erase(next(it));
                }

                cout << "Freed for thread " << ID << endl;
                print();

                pthread_mutex_unlock(&mutex);
                return 1;
            }
        }
        cout << "Can not free, requested node with ID " << ID << " and index "
            << index << " for thread " << ID << " does not exist" << endl;
        print();

        pthread_mutex_unlock(&mutex);
        return -1;
    }

    void print() {
        list<Node> :: iterator it;
        it = heap.begin();
        cout << *(it++);

        for (; it != heap.end(); it++) {
            cout << "---" << *it;
        }
        cout << endl;
    }
};
```

(c) Explanation

The class **HeapManager** has two private variables, a list of nodes referred to as list<Node> heap, which is using the linked list implementation of the the C++ <list.h>, and a pthread mutex, which is using the mutex implementation of the <pthread.h>. All the member functions of the **HeapManager** class are using and modifying these variables.

The **initHeap** operation creates a free node and inserts it to the linked list. The **print** operation displays the whole linked list. For the **myMalloc** and **myFree** operations of the **HeapManager** class, I have chosen to use a single pthread mutex as a locking mechanism to ensure atomic execution. Both operations access the list of nodes and modify it, therefore the code of both operations are between the pthread_mutex_lock(&mutex) and pthread_mutex_unlock(&mutex) to ensure atomicity.