Programming Assignment (PA) – 4
(Heap Management)

CS307- Operating Systems

25 May 2023

Solution Report

Ataollah Hosseinzadeh Fard
ID: 28610

```cpp
#include <iostream>
#include <mutex>

using namespace std;

struct Node{
    int id;
    int size;
    int index;
    Node* next;

    Node(int _id, int _size, int _index, Node* _next): id(_id), size(_size), index(_index), next(_next) {}
};

class HeapManager{
    public:
        int initHeap(int);
        int myMalloc(int, int);
        int myFree(int, int);
        void print(void);
    private:
        Node* LinkedList;
        mutex mtx;
};
```

Included needed libraries, in this solution only iostream needed for cout operations and mutex needed for critical sections of some parts. Defined the struct for node with a parametric constructor so that easily can initialize a new node with constructor. Also, defined template of “HeapManager” class, it has 4 public methods and 2 private variables. 4 methods are “initHeap()”, “myMalloc()”, “myFree()”, and “print()”. 2 private variables are “LinkedList” which is the main LinkedList that will simulate the heap and a “mtx” mutex that will be used for concurrency purposes.

```cpp
int HeapManager::initHeap(int size) {
    //cout << "Memory initialised\n";

    LinkedList = new Node(-1, size, 0, NULL);

    this->print();
    return 1;
}
```

int initHeap(int size)

“initHeap()”, initiates the LinkedList with a single free node, which has id of -1, with given size. Then calls “print()” to display view of LinkedList and returns 1. Since we assume that this function never fails, it always returns 1, an indicator of success of initialization.

```
int HeapManager::myMalloc(int id, int size) {
    this->mtx.lock();

    Node* ptr = this->LinkedList;
    Node* prev = NULL;
    while(ptr != NULL) {
        if(ptr->id == -1 && ptr->size > size) {
            cout << "Allocated for thread " << id << endl;

            int idx = ptr->index;
            ptr->size -= size;
            ptr->index += size;
            Node* newNode = new Node(id, size, idx, ptr);

            if(prev == NULL) {
                this->LinkedList = newNode;
            }
            else {
                prev->next = newNode;
            }

            this->print();
            this->mtx.unlock();
            return idx;
        }
        else if(ptr->id == -1 && ptr->size == size) {
            cout << "Allocated for thread " << id << endl;

            int idx = ptr->index;
            Node* newNode = new Node(id, size, idx, ptr->next);

            Node* toBeDeleted = ptr;
            if(prev == NULL) {
                this->LinkedList = newNode;
            }
            else {
                prev->next = newNode;
            }
            delete toBeDeleted;

            this->print();
            this->mtx.unlock();
            return idx;
        }
        prev = ptr;
        ptr = ptr->next;
    }

    cout << "Cannot allocate, requested size " << size << " for thread " << id << " is bigger than remaining size" << endl;

    this->print();
    this->mtx.unlock();
    return -1;
}
```

int myMalloc(int id, int size)

In “myMalloc()”, to avoid any concurrency issues, directly at beginning of function it locks the mutex. In this function, I iterated over LinkedList to find first occurring free node that has equal or larger size of given size input. If it cannot find such free node, it will fail to add the given thread to heap and it will prompt failure message and will not modify LinkedList and calls “print()” and will unlock mutex and return -1. However, if it finds such free node, there are two possibilities, first case when size of free node is larger that size of new node, so I add new node before that free node and increase index of free node by size of new node and decrease size of free node by size of new node. In second case when size of free node is equal to new node, at that time I delete the free node and add new node at same position with same index. At end of both cases, I call “print()” then unlock mutex and return index of new node.

```
90  int HeapManager::myFree(int id, int index) {
91      this->mtx.lock();
92
93      Node* ptr = this->LinkedList;
94      Node* prev = NULL;
95      while(ptr != NULL) {
96          if(ptr-> id == id && ptr->index == index) {
97              cout << "Freed for thread " << id << endl;
98
99              ptr->id = -1;
100
101             if(ptr->next != NULL && ptr->next->id == -1) {
102                 ptr->size += ptr->next->size;
103                 Node* toBeDeleted = ptr->next;
104                 ptr->next = ptr->next->next;
105                 delete toBeDeleted;
106             }
107
108             if(prev != NULL && prev->id == -1) {
109                 prev->size += ptr->size;
110                 Node* toBeDeleted = ptr;
111                 prev->next = ptr->next;
112                 delete toBeDeleted;
113             }
114
115             this->print();
116             this->mtx.unlock();
117             return 1;
118         }
119         prev = ptr;
120         ptr = ptr->next;
121     }
122
123     cout << "Cannot deallocate, there is no thread " << id << " at index of " << index << endl;
124
125     this->print();
126     this->mtx.unlock();
127     return -1;
128 }
```

int myFree(int id, int index)

In “myFree()”, again a same mutex usage of in “myMalloc()” is applied here as well. Then I iterate over LinkedList to find a node with id and index of given inputs. If such node not found, it prints error message and calls “print()” and then unlocks mutex and returns -1. However, if it finds such node, immediately changes that node’s id to -1 to flag it as a free node. Then I check its next and previous nodes to see if I need to merge free neighbour nodes. Then it calls “print()” and unlocks mutex and returns 1.

Important: In my solutions only in cases of “initHeap()” and adding a new thread to heap, which is possible in “myMalloc()”, I allocated new nodes but in other cases such as “myFree()”, I only modified already existed nodes.

```
130 void HeapManager::print() {
131     Node* ptr = this->LinkedList;
132     while(ptr != NULL) {
133         cout << "[" << to_string(ptr->id) << "][" + to_string(ptr->size) << "][" <<  to_string(ptr->index) << "]";
134         if(ptr->next != NULL)
135             cout << "---";
136         ptr = ptr->next;
137     }
138     cout << endl;
139 }
140
```

void print()

In “print()”, I iterate over LinkedList and print each node as following: [id][size][index].

Important: I assumed that “initHeap()” and “print()” calls will not have any concurrency problems since “initHeap()” only called before starting threads and “print()” called after threads have finished; Therefore, I did not use any mutex mechanism in these 2 functions. However, if I wanted to use, I could use “mtx” mutex in “initHeap()” but in “print()” I had to use another mutex since “myMalloc()” and “myFree()” functions lock “mtx” and then call “print()”, so if wanted to use “mtx” mutex in “print()” it would cause a deadlock case.