CS 307 OPERATING SYSTEMS PROGRAMMING ASSIGNMENT – 4 REPORT

Prepared by Pelinsu Sarac, 28820

Sabanci University

2023

1. Introduction

This report includes the implementation details of the solution for CS307 Programming Assignment 4. It includes the class definition, codes of 4 methods and the synchronization mechanism utilized.

2. Class Definition

```
struct Chunk {
    int threadID;
    int size;
    int startIndex;
};


class HeapManager {


    private:
        list<Chunk> heap;
        pthread_mutex_t heapLock = PTHREAD_MUTEX_INITIALIZER;
        void printPriv();


    public:
        int initHeap(int size);
        int myMalloc(int ID, int size);
        int myFree(int ID, int index);
        void print();
};
```

Here, the “Chunk” structure is to keep the metadata about the chunk in the heap. Thread ID is to show by which thread this chunk was allocated, or if it is not if this field is -1. Size is to show the size of the chunk. Finally, start index is to show the starting point of the chunk in memory.

Then comes the class definition of HeapManager. This class has two private properties: the heap, of type C++ standard linked list, and the associated mutex, which will be used to prevent data race. Last element in private section is a print method without any locking mechanism. Reason for it will be told later.

In the public section, all 4 methods that the assignment asks for can be seen.

3. Method Implementations

a. initHeap

```
int HeapManager::initHeap(int size) {


    Chunk firstChunk;

    firstChunk.threadID = -1;

    firstChunk.size = size;

    firstChunk.startIndex = 0;


    pthread_mutex_lock(&heapLock);


    heap.push_front(firstChunk);


    //cout << "Memory initialized\n";

    printPriv();
```

```
pthread_mutex_unlock(&heapLock);

return 1;
```

Here, initHeap takes a size for heap to be initialized with. What it does really is, creating an initial chunk which represents the initial stage of heap and place it in the list, again representing the heap. Index is 0 as memory starts from 0, and threadId is -1 as no threads allocated anything yet. Finally, heap list is printed to show initial stage of heap.

b. myMalloc

```
int retVal = -1;

list<Chunk>::iterator iter;

pthread_mutex_lock(&heapLock);

for (iter = heap.begin(); iter != heap.end(); ++iter) {

    if ((*iter).threadID == -1 && (*iter).size >= size) { //such chunk is found

        if ((*iter).size == size) { //chunk is exact size

            (*iter).threadID = ID;

            retVal = (*iter).startIndex;

        } else { //there will be some free space left from the chunk

            Chunk temp;

            temp.size = size;

            temp.startIndex = (*iter).startIndex;

            temp.threadID = ID;


            (*iter).size -= size;

            (*iter).startIndex += size;


            iter = heap.insert(iter, temp);
```

```
                retVal = (*iter).startIndex;
            }
            break;
        }
    }

    if (retVal != -1) {
        cout << "Allocated for thread " << ID << endl;
    } else {
        cout << "Can not allocate, requested size " << size << " for thread " << ID << " is
bigger than remaining size\n";
    }
    printPriv();
    pthread_mutex_unlock(&heapLock);

    return retVal;
```

Here, method first starts with iterating over the list with an iterator object to find a free chunk with enough size. To do that, it both checks sizes of chunks and the threadId fields. In case of finding such chunk, there are two cases: size exactly matches, or some space remaining from chunk. In the first case, method only updates the threadId value of the chunk. Otherwise, it first creates chunk that will represent the allocated space, then it will decrease the amount of size of the remaining chunk and move the index further. Finally, allocated chunk is inserted and starting index of it is set as return value. In case of allocation, loop end in the middle and method prints a success message. Otherwise, loop iteration ends and return value Is returned with its initial value of -1, which is for unsuccessful allocations, along with a message indicating it.

c. myFree

```
int HeapManager::myFree(int ID, int index) {
   int retVal = -1;
   list<Chunk>::iterator iter;
   pthread_mutex_lock(&heapLock);


   for (iter = heap.begin(); iter != heap.end(); ++iter) {
      if ((*iter).threadID == ID && (*iter).startIndex == index) {
         retVal = 1;
         (*iter).threadID = -1;
         list<Chunk>::iterator rightIter = iter;
         ++rightIter;
         if (rightIter != heap.end() && (*rightIter).threadID == -1) {
            (*iter).size += (*rightIter).size;
            heap.erase(rightIter);
         }
         if (iter != heap.begin()) {
            list<Chunk>::iterator leftIter = iter;
            --leftIter;


            if ((*leftIter).threadID == -1) {
               (*iter).size += (*leftIter).size;
               (*iter).startIndex = (*leftIter).startIndex;
               heap.erase(leftIter);
```

```
                }
            }
            break;
        }
    }
    if (retVal == 1) {
        cout << "Freed for thread " << ID << endl;
    } else {
        cout << "Can not free, no allocation such that ID: " << ID << " and starting index:
" << index << " was found\n";
    }
    printPriv();
    pthread_mutex_unlock(&heapLock);

    return retVal;
```

Here, method iterates over the list to find the matching chunk with the given threadId and starting index. If such chunk is found, first threadId is updated to -1, to indicate the deallocation, along with the return value to indicate success. After that, method does the controls for possible coalescing. First, it checks the right of the chunk. If it is not the end of list (which means there is no element further this point) and if it is free (threadId = -1), then method merges them by increasing the size of the current chunk and deleting the nearing chunk. Next, it checks the left side. If the current chunk is not the first (because then there is no preceding element) and the lefthand side chunk is free, method merges them. To do that, it increases the size of the current chunk using other chunk's size field and the index is updated to the lefthand chunk's index as it comes before. Finally, lefthand chunk is deleted. Thus, there is only one chunk remaining to show the merge.

d. Print

```
void HeapManager::print() {
    pthread_mutex_lock(&heapLock);
    printPriv();
    pthread_mutex_unlock(&heapLock);
}


void HeapManager::printPriv() {
    list<Chunk>::iterator iter;
    iter = heap.begin();
    cout << "[" << (*iter).threadID << "][" << (*iter).size << "][" << (*iter).startIndex
<< "]";
    iter++;

    for (; iter != heap.end(); ++iter) {
        cout << "---[" << (*iter).threadID << "][" << (*iter).size << "][" <<
(*iter).startIndex << "]";
    }
    cout << endl;
```

There are two print function, but only difference between them is the locks. A private print function is created so that the other methods can call it without any deadlocks. If there was only one with locks, then other methods would have to wait for the lock to print, which they already acquired, i.e., they would have a cyclic dependency on themselves. Private print function first prints the first element of the list, as there will always be one after initialization to represent heap. Then, if there are multiple chunks, they are also printed according to the given format with an iteration over the heap list. In the public version, private print method is wrapped with locks to enable synchronization.

4. Synchronization

Object that is the reason for possible data race is the list that represents the heap. Thus, within the class definition, a mutex is initialized that is associated with the list. In the methods, any operation that is about the heap manipulation or reading it is wrapped with the lock-unlock of this mutex to assure the atomicity of each of the methods, which can be seen in the implementations. In other words, only a single thread can manipulate the thread at a time, as only one of them would have the lock of the list. Basically, synchronization in the code works as follows:

```
pthread_mutex_lock(&heapLock);

//operations on heap list

pthread_mutex_unlock(&heapLock);
```