Programming Assignment (PA) - 3
Heap Management

CS307 - Operating Systems (Summer 2021)

8 August 2021
DEADLINE: 15 August 2021, 23:55

1 Introduction

Heap is a segment in a process' virtual address space. Processes use the heap to keep variable sized objects during their execution. The allocator (heap management) library implements and manages the heap of a process. It allocates a portion of the physical address space (main memory) for implementing the heap. As threads of a process request and return space from the heap, the allocator library serves them through its API.

Basics of the allocator library is described in our main textbook OSTEP, Chapter 17[1]. Here, chunks of free memory are kept in a linked-list called free-list. Moreover, the list itself is implemented inside the heap. So, metadata like next pointers occupy space inside heap as well.

In this PA, you are expected to implement a simple allocator library. You are expected to keep a linked-list as well. However, your linked-list will not occupy space inside the heap. You can safely assume that it is disjoint with the heap. Moreover, your linked list will not just track free chunks but also occupied spaces inside the heap. Details will be provided in the following sections.

Operations of your allocator library will be called concurrently by different threads. Therefore, you need to be careful about data races. However, you are advised to implement a sequential heap manager first i.e., it works correctly when processes have single thread. Then, you add synchronization mechanism to prevent data races. The way to get the best score possible from this PA is to follow the grading section during the implementation such that you implement an item after completing its preconditions.

2 Heap Management Library

The library provides the following API:

- **int initHeap(int size)**: This operation takes the size of the heap in terms of bytes as input and initializes the heap with the requested size. As you will see later, you do not really allocate memory for the heap in your implementation. You just initialize

[1] see `https://pages.cs.wisc.edu/~remzi/OSTEP/vm-freespace.pdf` for more details.

the linked-list that represents your heap layout. Therefore, we can safely assume that **initHeap** always succeeds for any positive size input and returns 1 after initializing the heap. Lastly, it prints the initial heap layout (see sample runs at the end).

- **int myMalloc(int ID, int size)**: This operation takes ID of the thread issuing the operation and the requested size in bytes as input and tries to allocate heap space of requested size. If there is a free chunk in the heap that can accommodate the request (that has enough space), this operation must succeed and return the beginning address of the newly allocated space. Otherwise, it fails and returns -1. In both cases, it prints whether the operation succeeds or fails and the final state of the heap (see sample runs at the end).
- **int myFree(int ID, int index)**: This operation takes a thread ID and the starting address of the chunk to be freed as input. If this thread has really allocated this chunk, it frees the chunk and returns 1. Otherwise, it returns -1. In both cases, it prints whether the operation succeeds or fails and the final state of the heap (see sample runs at the end).
- **void print()**: Prints the memory layout (see sample runs at the end).

2.1 State: Linked-List

State of the library consists only of a linked list. The list keeps both free and allocated heap locations. The list is initialized with a single node that represents a free chunk of the size of the heap when the **initHeap** operation is called. The list should be updated with **myMalloc** and **myFree** operations.

The only object library keeps is the list. **This means that you will not really allocate memory for the heap itself.** This PA simulates the heap. Threads using this library just allocates and frees spaces from the heap. You can assume that they will not really access these locations and perform writes and reads. Therefore, allocating space for the actual heap is not needed.

Nodes of the list has three fields as seen in Figure 1: *ID*, *SIZE* and *INDEX*. The first field is the thread ID that allocated the chunk. Each thread has a unique non-negative ID given to them as input during its generation. If the chunk is free, this field is -1. *SIZE* and *INDEX* fields represent the size and the starting address of the chunk, respectively.

ID	SIZE	INDEX

Figure 1: A linked list node

Given the state representation above, the API operations should be implemented as follows:

- **int initHeap(int size)**: Initializes the list with a single free node with given input size and start index 0. This operation always succeeds and returns 1. You can assume that the input size will be always positive. Before returning, it prints the linked list (see sample runs at the end).
- **int myMalloc(int ID, int size)**: Your allocator library will implement the first fit approach: It will allocate space from the first free node that has enough space. Here, the

first node means the one closest to the head of the list. Once the candidate free node is found, it will be divided into two nodes if it has more space than the requested size. The first node will represent the newly allocated space. The second one will represent the remaining free space. If there is no such candidate node, it will return -1. Otherwise, it will return the start index of the newly allocated node. Before returning, it prints the success info of the operation and the linked list (see sample runs at the end).

- **int myFree(int ID, int index)**: This operation traverses the list for finding a node with the given ID and index. There can be at most one such a node in the list. If it exists, this node is turned into a free node, it merges with the neighbors if they are free as well (see Figure 2 for details of coalescing) and returns 1. Otherwise, the list does not change and returns -1. Before returning, it prints the success info of the operation and the linked list (see sample runs at the end).

- **void print()**: This operation just prints the list in the format seen in sample runs.

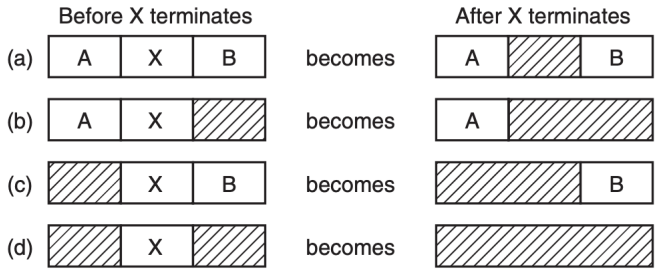

Figure 2: Coalescing. The left column shows the allocated chunk X and its immediate neighbors and the right column shows the same after freeing X and coalescing. A and B denote allocated chunks and shaded regions denote free chunks. In (a) both neighbors are allocated, so freeing X does not need coalescing. In (b) (resp., (c)), the right (left) neighbor or next (previous) node is free as well. So, after freeing X, both nodes become free and they need to be merged into a bigger single node. In (d), both neighbors of X, previous and next nodes in the list, are free. Therefore, after freeing X, a single free node replaces the previous three nodes.

2.2 Taming Concurrency

So far, we have described operations of the allocator library as if they execute atomically in isolation. However, multiple threads might call **myMalloc** and **myFree** operations concurrently. We want your implementation to be correct under concurrency as well.

In order to ensure atomicity of your operations under concurrency, you must use synchronization mechanisms like locks or semaphores. You have to make sure that there are no race conditions. However, you have to be careful about deadlocks. Using synchronization mechanisms should never introduce deadlocks to your implementation.

3 Implementation Details

You are expected to implement your allocator library as a C++ class.

Your implementation will be tested with C++ clients. Your submission should not contain any main method. Our test clients will have their main method that will create different threads which execute **myMalloc** and **myFree** operations. We will use PThread library for creating and managing threads. We also provide sample clients in the PA package so that you can test your implementation before submitting.

You can use C++ Standard Library linked list class or implement your own linked list. Please note that the standard library list is not thread safe and does not ensure atomicity of its operations. Therefore, you need to utilize synchronization mechanisms.

Submission

You are expected to submit a zip file named *<YourSUUserName>_PA4.zip* until 15 July 2021, 23h55.

The content of the zip file is as follows:

- **report.pdf**: In your report, you must present your locking algorithm as a pseudo code. You discuss which locking mechanism you have chosen, how you adapted it to suit your needs and provide formal arguments on why it satisfies the atomicity requirement.
- **allocator.cpp**: Your heap management class implementation in C++.

Grading

Some parts of the grading will be automated. If automated tests fail, your code will not be manually inspected for partial points. Some students might be randomly called for an oral exam to explain their implementations and reports.

Submissions will be graded over 100 points:

1. **Compilation** (10 pts): Sample test clients provided to you compiles and links with your `HeapManager` class implementation without producing any errors.
2. **Sequential Operations** (50 pts): Your implementation passes various automated tests in which single thread calls **myMalloc** and **myFree** operations.
3. **Concurrency** (20 pts): Your allocator library operations work as if they are atomic when there are more than one thread calling its operations concurrently. Moreover, your implementation is deadlock free.
4. **Report** (20 pts): Your report clearly explains your implementation, synchronization mechanisms used and how they ensure atomicity of operations.

Item 1 is a precondition of 2 and 4; and 2 is a precondition of 3.

Sample Output

Sample Run 1

This is an example of the very basic case. A thread is allocating 40 Bytes of memory and the execution is finished without using the free function.

```
[-1][100][0]
Allocated for thread 0
[0][40][0]---[-1][60][40]
Execution is done
[0][40][0]---[-1][60][40]
```

Sample Run 2

Addition to the first example, now thread uses a freeMemory function.

```
Memory initialized
[-1][100][0]
Allocated for thread 0
[0][40][0]---[-1][60][40]
Freed for thread 0
[-1][100][0]
Execution is done
[-1][100][0]
```

Sample Run 3

Now there are two threads, first allocating 40 Bytes of memory and uses free. Then allocates 20 Bytes of memory and uses free.

```
[-1][100][0]
Allocated for thread 0
[0][40][0]---[-1][60][40]
Freed for thread 0
[-1][100][0]
Allocated for thread 0
[0][20][0]---[-1][80][20]
Freed for thread 0
[-1][100][0]
Allocated for thread 1
[1][40][0]---[-1][60][40]
Freed for thread 1
[-1][100][0]
Allocated for thread 1
[1][20][0]---[-1][80][20]
Freed for thread 1
[-1][100][0]
Execution is done
[-1][100][0]
```

Sample Run 4

Let’s consider more realistic case. There are 5 threads running at the same time. All of them will try to request a memory between 1 and 30 Bytes. If the allocation is successfully done, thread will sleep random amount of time. When the memory allocated for a thread, It should spend some time to finish the desired job, we can use sleep function to simulate such behaviour. After passing the above allocation request, each thread will create another request for requesting a memory between 1 and 30 Bytes. After the allocation and memory usage is done, successfully allocated threads will free the memory.

Note that, thread with ID 3 creates a request with 11 Bytes size. However, 4 Bytes available and the allocation is not successful. And the state of the memory is not changed. Also consider the corner cases for the hole management. Check how the holes are merged after the free operations of threads.

```
Memory initialized
[-1][100][0]
Allocated for thread 0
[0][14][0]---[-1][86][14]
Allocated for thread 1
[0][14][0]---[1][28][14]---[-1][58][42]
Allocated for thread 4
[0][14][0]---[1][28][14]---[4][24][42]---[-1][34][66]
Allocated for thread 2
[0][14][0]---[1][28][14]---[4][24][42]---[2][17][66]---[-1][17][83]
Allocated for thread 3
[0][14][0]---[1][28][14]---[4][24][42]---[2][17][66]---[3][10][83]---[-1][7][93]
Allocated for thread 2
[0][14][0]---[1][28][14]---[4][24][42]---[2][17][66]---[3][10][83]---[2][3][93]
---[-1][4][96]
Can not allocate, requested size 11 for thread 3 is bigger than remaining size
[0][14][0]---[1][28][14]---[4][24][42]---[2][17][66]---[3][10][83]---[2][3][93]
---[-1][4][96]
Freed for thread 3
[0][14][0]---[1][28][14]---[4][24][42]---[2][17][66]---[-1][10][83]---[2][3][93]
---[-1][4][96]
Allocated for thread 0
[0][14][0]---[1][28][14]---[4][24][42]---[2][17][66]---[0][4][83]---[-1][6][87]
---[2][3][93]---[-1][4][96]
Freed for thread 2
[0][14][0]---[1][28][14]---[4][24][42]---[-1][17][66]---[0][4][83]---[-1][6][87]
---[2][3][93]---[-1][4][96]
Freed for thread 2
[0][14][0]---[1][28][14]---[4][24][42]---[-1][17][66]---[0][4][83]---[-1][13][87]
Allocated for thread 1
[0][14][0]---[1][28][14]---[4][24][42]---[1][1][66]---[-1][16][67]---[0][4][83]
---[-1][13][87]
Allocated for thread 4
[0][14][0]---[1][28][14]---[4][24][42]---[1][1][66]---[4][13][67]---[-1][3][80]
---[0][4][83]---[-1][13][87]
Freed for thread 0
[-1][14][0]---[1][28][14]---[4][24][42]---[1][1][66]---[4][13][67]---[-1][3][80]
---[0][4][83]---[-1][13][87]
Freed for thread 0
[-1][14][0]---[1][28][14]---[4][24][42]---[1][1][66]---[4][13][67]---[-1][20][80]
```

```
Freed for thread 1
[-1][42][0]---[4][24][42]---[1][1][66]---[4][13][67]---[-1][20][80]
Freed for thread 1
[-1][42][0]---[4][24][42]---[-1][1][66]---[4][13][67]---[-1][20][80]
Freed for thread 4
[-1][67][0]---[4][13][67]---[-1][20][80]
Freed for thread 4
[-1][100][0]
Execution is done
[-1][100][0]
```