PA4 Report

Heap Management

CS307 - Operating Systems

Mert Ekici 26772
ekicimert@sabanciuniv.edu

I. Introduction

Heap is a segment in the virtual address space of a process. Processes use the heap for dynamic memory allocation where they keep variable-sized objects during their execution. Allocator or **Heap Management Library** manages the heap allocation and segmentation of a process. It allocates a portion of the physical address space for the heap. As threads of a process request or return space from the heap, Heap Management Library serves through API. Chunks of free memory are kept in a **linked-list** called **free-list**. In this Programming Assignment, a simple allocator library should be implemented. Our linked-list does not necessarily occupy space inside the heap (not mandatory). Created linked-list representatively follows the heap information in its **nodes' fields**. Moreover, it does not only track **free chunks** but also **occupied spaces** inside the heap. Operations of the allocator library will be called by different **threads concurrently**. Therefore, **data races** should be followed and handled. Various **synchronization mechanisms** can be used for this purpose such as **mutex, semaphore or barrier**.

II. Heap Management Library

Heap Management Library is implemented inside **allocator.cpp**. By implementing **structs**, **classes** and the **functions** that are used by these **objects**; Heap Management functionality will be provided. Based on the descriptions given in the PA4 instructions document, created aspects (structs, classes and functions) will be explained in the following sections with respect to the concept of **Object Oriented Programming**.

a) Structs

Instructions document describes the **linked-list node** with three main **features**: **ID, size, index**.

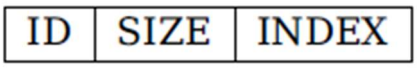


Figure 1: A linked list node

There are two **structs created to be used in the utility** classes of Heap Manager Library. Since we are going to use a ***"doubly linked-list structure"*** (it is decided as the best data structure for implementing the **coalescing** part)**,** first step is to create a **node** struct. Node struct contains node pointers **next, prev** (double linked-list) and the instance of the other struct we created **data_t info**. Data struct holds the information about each heap segment that is shown in **Figure 1** which are **ID, SIZE** and **INDEX**. All three are **integer variables**.

```
typedef struct data{
	int id;
	int size;
	int index;
} data  ;
```

```
struct node{
	node* next;
	node* prev;
	data_t info;
};
```

b) Classes

Classes in the Heap Management Library are **LinkedList** and **HeapManager**. LinkedList class is a simple double linked-list class that only contains a **public** variable **node pointer head** that points the head of the linked-list and **public default constructor function** that initializes **head** as **NULL**. Reason for using them as **public** is that they will be used in **HeapManager Class** and they should be accessible from outer classes. Next, a class named **HeapManager** is implemented which will be used in **samplerun.cpp files**. Instances of these classes are created as "**HeapManager m;"** and used to reach its special methods: **initHeap, myMalloc, myFree, print**. Besides these **public** functions, HeapManager class also has a **private variable LinkedList heapList**. It is a variable of previously mentioned class (LinkedList) and used to access linked-list functionality with the **head pointer** of the list. Aim of each function and their high-level structure from the description document will be explained under the following *Functions* title. Detailed implementation will be pointed out in the **Implementation** part.

c) Functions

In **LinkedList** class, only function is public default constructor function that assigns a **null pointer** to the head of the linked list.

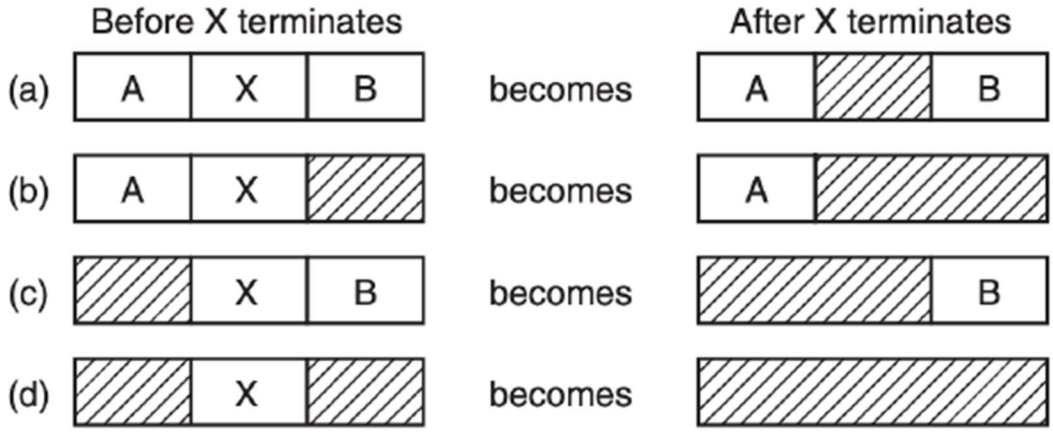
```
class LinkedList
{
public:
	node* head;

	//constructor
	LinkedList(){
		head = NULL;
	}
};
```

HeapManager class on the other hand, has four public functions that are required to provide Heap Management functionality with the clearly indicated segmentation strategies. The library provides the following API:

- **int initHeap(int size):** This operation takes the size of the heap (bytes) as input and initializes the heap. Initializes the list with a single **free node (ID is -1) with start index 0**. As mentioned in the **Introduction** part, we do not really allocate memory but we keep track of the information to symbolize the allocation. Therefore, we can safely assume that **initHeap** always succeeds for any positive size input and returns 1 after initialization. Lastly it prints "Memory initialized" and the initial heap layout.

- **int myMalloc(int ID, int size):** This operation takes **ID of the thread** issuing the operation and the requested size (bytes) as input. Function tries to allocate heap space of requested size. If there is a **free chunk in the heap that has enough space**, this free space node will be divided into two nodes: the first node is for the newly allocated space and the second one represents the remaining space (if left). When operation succeeds, function returns the beginning address of the newly allocated space. Otherwise, it fails and returns -1 (if there is no such candidate node). Before returning, it prints the success info ("**Allocated for thread #**") and the linked list.

- **int myFree(int ID, int index):** This operation traverses the list to find a node with the **given ID and index** which can be at most one instance suits these parameters. If exists, this node turns into a free node and it **merges with the adjacent nodes** if they are free as well (previous and next nodes). The advantage of doubly linked-list in this case is that we can **easily reach the previous node** and move freely in the linked-list. In success, it returns 1 and otherwise, list does not change and returns -1. Before the return, it prints success info ("**Freed for thread #**") and the updated state of the linked list. **Figure 2** below shows each possible case for **coalescing**.



- **void print():** Prints the memory layout (the linked-list) in the following format;

[0][14][0]---[1][28][14] ---[4][24][42] ---[2][17][66] ---[1][10][83] ---[-1][7][93]

III. Implementation

Using the guidelines above, **functions of HeapManager class** were implemented to **mimic** heap allocation behaviour of the modern operating systems. Each function will be explained in detail (sequentially) to provide a better understanding of how a **doubly linked-list** can achieve similar memory modifications with **malloc()** and **free()**.

initHeap

In initHeap, an **initial node** that has **NULL** pointers for **next** and **prev** is created. After that, info field of this initial node is filled accordingly to the given instructions. Since this node represents the **starting segment of the heap memory**; its index is 0, size equals to the given size (parameter) and **ID is -1 which indicates that the memory space is free**. Initial node will be assigned as the **head of heapList of HeapManager** instance that we are currently using. Finally, **"Memory initialized"** and the current state of the linked list will be printed.

myMalloc

For the myMalloc function, linked-list will be traversed to find a free memory segment which has enough space for the memory allocation. We first create a **curr** node for the traversal. This current node goes one by one on each node to check if the **node is free and has a space that is greater or equal to the size entered as parameter**. If a node with a greater size is found, two nodes are created **newFNode** and **newSNode**. After they are created, newFNode is connected to the newSNode with **next** and **curr->prev** as its prev value. The important part here is not to forget checking the **previous node of the curr**. If it is not null it should be linked with the newFNode as its next (two-way connection). Otherwise, newFNode will be assigned as the new head of the linked-list. Since newFNode is the newly allocated memory, id and size will be the parameters of this function and index value will remain the same. When we come to the newSNode, necessary links should be built between the adjacent nodes similar to the newFNode. Next of newSNode is **curr->next** and prev is **newFNode** obviously. Now, we should check the next of curr and if it is not null it should be connected to **newSNode as its prev**. Info field of the newSNode is a bit different than the newFNode. As you remember, newSNode represents the remaining space after the allocation and these attributes should be filled accordingly. Index of newSNode is going to be **current node's index plus the size of the**

desired allocation (starting index of the memory left). The size of the newSNode is the remaining size which is **curr->info.size minus the entered size**. And finally, **id should be -1** since the node is free. Newly created nodes are added to the linked-list and old node should be freed which is handled by **free(curr)**. At the end of this case (greater space than required is found), **"Allocated for thread ID#"** and the linked-list state will be printed. As a return value, **index** of newFNode is returned. If the size is equal to the entered parameter size, then only update to be done is **curr->info.id = ID** (we updated the **node from free to allocated by ID** thread) and the rest goes the same. If no suitable space is found at the end of the traverse **"Can not allocate requested size for thread ID is bigger than remaining size"** will be printed with the list and -1 is returned.

myFree

In myFree function, we are looking for a node with the given thread ID and index by traversing the linked-list. There can exist only one node with these attributes. If the node is found, first thing to do is update its id as -1 (updating it as free). Then two conditions will be checked: "Is the next node is free?" and "Is the previous node is free?". If the next node is free, its size will be **added** to the current node's size and curr's **next** will be the next of the next node which is going to be **absorbed** by the current node (**curr**). Again, we should check for the next node of the next node and if it is **not null** it should be linked to the **curr** by its **prev**. Then, we should check if the previous node is free. In the case which the previous node is free, **same approach will be applied** but now instead of curr pointer we use the **pointer of curr's previous node (curr->prev).** We can simplify the steps as;

A --- PREV --- CURR --- NEXT --- B

1. If NEXT is free, CURR absorbs NEXT.

A --- PREV --- (CURR + NEXT) --- B

2. If PREV is free, PREV absorbs CURR.

A --- (PREV + CURR + NEXT) --- B

3. Then, one single free memory is obtained by combining the adjacent ones.

A --- (FREE) --- B

If free process successed, **"Freed for thread#"** is printed with the current state of the linked-list and 1 is returned. If no match found, **again linked-list is printed but -1 is returned.**

print

As the last function, **print()** prints the whole linked-list (**heap memory**) with the given format. Again, the list is traversed using **curr node pointer** and all of the elements of the linked-list is printed. See the example below.

[0][14][0]---[1][28][14] ---[4][24][42] ---[2][17][66] ---[1][10][83] ---[-1][7][93]

IV. Concurrency

As suggested in the PA4 description, I coded this library with the assumption of the **sequential usage**. However, multiple threads might call **myMalloc** and **myFree** operations **concurrently**. To keep my implementation **correct under concurrency**, I used **synchronization tools** where **data races for the critical points** might exist. In order to ensure **atomicity** of these critical points, I added **mutex locks** to my implementation. This implementation is double-checked to avoid **deadlocks** and other possible problematic cases. Two **pthread mutex locks** are used in the allocator.cpp (Heap Manager Library):

- pthread_mutex_t **lock**
- pthread_mutex_t **printLock**

First, **lock** is used in every scenario where a modification or update in linked-list happens. Thus, when a thread wants to modify the linked-list, it needs to acquire the ***mutex lock*** and when it is done, it releases the **lock**. Important note, functions with **early return** possibilities such as **myMalloc** and **myFree** should also **release the lock before returning** to **prevent deadlocks.** Other mutex lock that is used in the implementation is **printLock**. This lock is specifically used in the **print function** and can be thought of as a **read-lock** (in that case **lock** appears as a **write-lock**). When any of the threads try to read the linked-list, this lock should be obtained and after the printing process it is released.