Memory Manager

Omkar, Tenghoit

Purpose

Our purpose is to make a memory management system. This will allow processes to request a memory allocation of their specified amount and return it once they are done using it. These methods will be implemented through a MemoryManager, which will handle the unlying code that makes these functions possible.

Specifications

- Utilize a custom data structure
- Ability to handle memory allocation requests of different sizes
- Utilize a first-fit scheme when allocating memory allocation
- Defragment existing free memory blocks back into one allocation

Design Overview

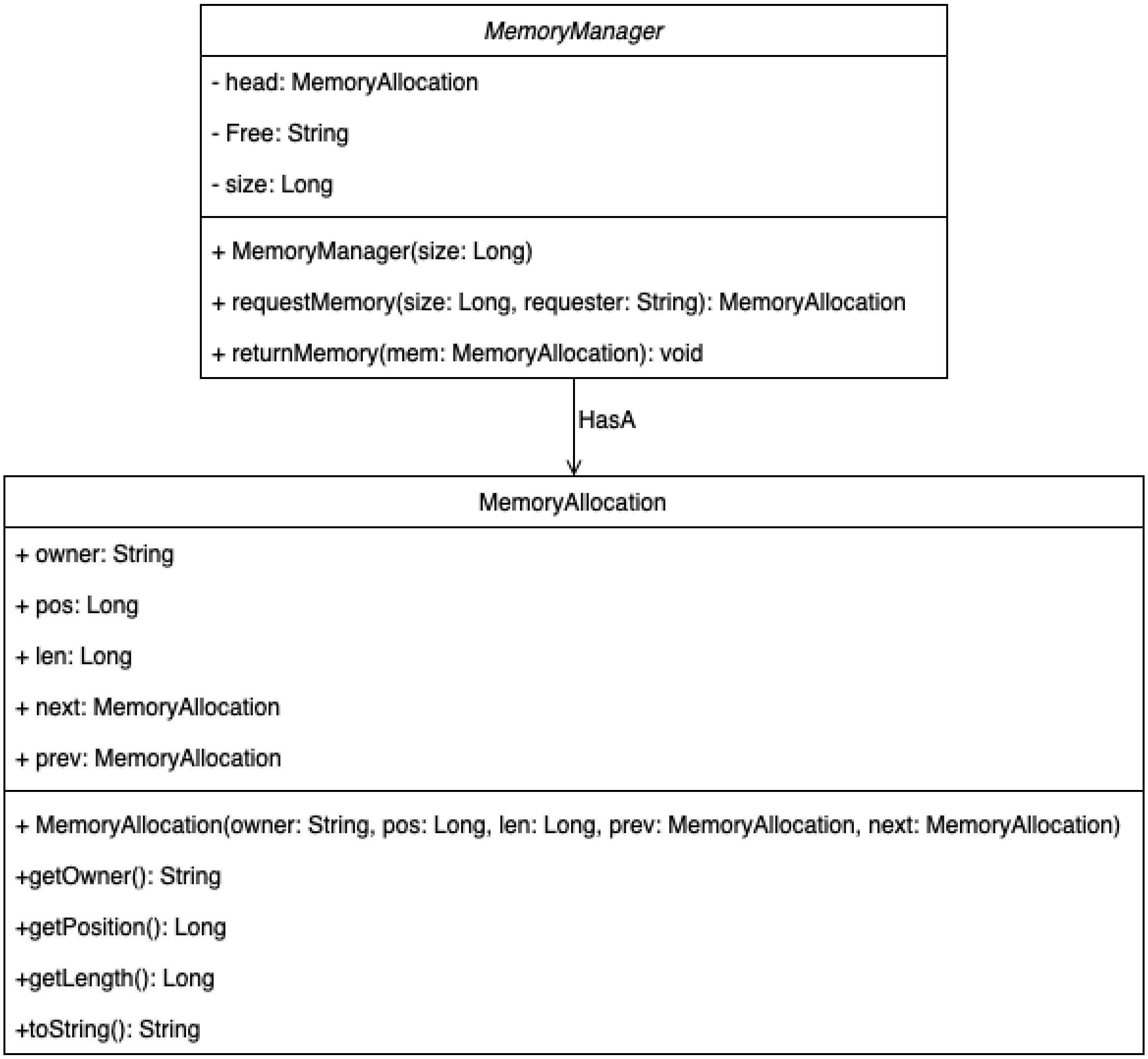

Figure 1 shows the UML diagram for MemoryManager and MemoryAllocation.

MemoryAllocation

Our main data type for this project is the `MemoryAllocation` class. Figure 1 shows the attributes that `MemoryAllocation` contains: a string `owner`, which is the name of the process that owns it; an int `pos`, corresponding to its position in the MemoryManager; and finally an int `len`, which is the size of the `MemoryAllocation`.

In addition to storing the allocation information, `MemoryAllocation` also acts as a node in our doubly-linked list, as shown by the `next` and `prev` attributes being another

`MemoryAllocation`. Using a doubly linked list is useful in this problem because it makes creating and removing nodes much more efficient since the reference to the previous node is made available to us. In addition, our `releaseMemory()` will heavily use this ability in its defragmentation step.

As for the methods, `MemoryAllocation` has the usual getters for its attributes and a constructor. The only noticeable function is the `toString()` method which will allow us to compare between `MemoryAllocations` by converting its information into a string format.

MemoryManager

`MemoryManager` is the class where the different tasks are allocated memory linking together like a list. We create different `MemoryAllocations` for different tasks which will act like a node and linked with previous and next memory allocation.

We can check if space or memory allocation is free or not by comparing its owner as string, if it is “free”.

When we initialize a memory manager it will create a big empty `MemoryAllocation` which will fill the entire memory manager which is free. We then make fragments of the big `MemoryAllocation` every task we are adding.

There are two main methods in the `MemoryManager` class, as explained shown in Figure 2 and Figure 3.

```
public MemoryAllocation requestMemory(String owner, Long size){

    find a free space >= size

    create new MemoryAllocation mem before the free space

    subtracts & reposition free space by size

    return mem

}
```

Figure 2 shows the pseudocode for the requestMemory() method.

requestMemory()

`requestMemory()` method creates a new `memoryAllocatio`n. Its inputs are : task's owner as a string for identification usage and the size of the task to specify how much memory it needs.

`requestMemory()` method will then check for free spaces in the `MemoryAllocation` by going through every next `MemoryAllocation` and checking if it's free. If it is free it will check if the size of that free space is big enough to fit the new task.

If we have any space equal or more than we need we allocate our new task there.If free space is bigger than what we needed we fragment it in two smaller parts: one for our task and other just free.

When we fragment a `MemoryAllocation`, We make sure that the neighbors, previous and next, are changed appropriately. The task will be at the earliest position and the free space will follow it.

```
public void releaseMemory(MemoryAllocation curr){

    set curr to free

    if(left neighbor is free) { combine with left neighbor }
    if(right neighbor is free) { combine with right neighbor }

}

private MemoryAllocation combine(MemoryAllocation curr, MemoryAllocation neighbor){

    adds neighor length to curr
    reposition curr

    set new prev and next for curr
    delete neighbor

    return curr
}
```

Figure 3 shows the pseudocode for the releaseMemory() method.

returnMemory()

Our `returnMemory()` method will do two things: it will free the `MemoryAllocation` that is passed through and it will also combine neighboring free `MemoryAllocation` into one block if possible. This combination step is necessary as without it, it may lead to a situation where we have enough memory to allocate to a process, but there isn't any individual `MemoryAllocation` that is big enough to accommodate the request.

As shown in Figure 3, the process of freeing a `MemoryAllocation` involves changing its `owner` attribute. `MemoryManager` has an attribute for a free `MemoryAllocation`, which is the `Free` variable. By changing the `owner` to this variable, we are signaling to `MemoryManager` that this block is free to use.

The next step involves combining neighboring blocks. This involves checking the `prev` and `next` of `MemoryAllocation` of the allocation that is being passed throughed. If any of these statements are true then we combine with that other `MemoryAllocation` with the use of the `combine()` function.

The `combine()` function takes two `MemoryAllocations` and returns a `MemoryAllocation`. It first determines which of these allocations is closer to the start and makes that the primary `MemoryAllocation`. The `len` of this primary `MemoryAllocation` will then be increased by the `len` of the other `MemoryAllocation`. Finally, we will change the connections of the primary `MemoryAllocation` so that it will skip over the other `MemoryAllocation` as well as change that next `MemoryAllocation` so that its `prev` points to our primary `MemoryAllocation`.