LP Assignment - I

CSL316 - Assignment 1 - Semester 6 (July-Nov) 2024
Programmer : Aditi Mukund Dhenge - BT21CSE077
Section : R4
TA :
Date due : January 24, 2024
Purpose : In this assignment, we are simulating memory management as is done in interpreted languages like Python or Java. This is done using two linked lists (free list and used list). When a program requests a block of memory dynamically, the system allocates memory from free list and moves the block to used list. When a program terminates or frees memory previously requested, the system deallocates the memory by moving the block to free list

---

Function: compactMemory

Use: Performs memory compaction by merging adjacent free memory blocks.

Arguments: 1.freeList - MemoryList pointer representing the list of free memory blocks.
2.usedList - MemoryList pointer representing the list of used memory blocks.
3.memoryArray - MemoryBlock pointer representing the initial memory block.
4.blockName - char pointer representing the name of the memory block being compacted.

Returns: void

Notes: The function performs memory compaction by merging adjacent free memory blocks. It uses two nested loops to find pairs of compatible free blocks and merges them into a single larger block.
***/

```
void compactMemory(MemoryList* freeList, MemoryList* usedList, MemoryBlock*
memoryArray,char* blockName) {
   MemoryBlock* head1;
   MemoryBlock* prev1;
   MemoryBlock* prev = NULL;
   MemoryBlock* head = freeList->head;
   MemoryBlock* first_node = freeList->head;

   while (head != NULL) {
      head1 = head->next;
      prev1 = head;

      while (head1 != NULL) {
         if (head->start_address + head->size == head1->start_address || head1-
>start_address + head1->size == head->start_address) {
            break;
         }
```

```
            prev1 = head1;
            head1 = head1->next;
        }

        if (head1 != NULL) {
            prev1->next = head1->next;

            MemoryBlock* newBlock = (MemoryBlock*)malloc(sizeof(MemoryBlock));
            char* allocatedName = (char*)malloc(strlen(blockName) + 1);
            strcpy(allocatedName, blockName);
            newBlock->name = allocatedName;
            newBlock->start_address = (head->start_address<head1->start_address) ?  head-
>start_address : head1->start_address;
            newBlock->size = head->size + head1->size;

            if (prev) {
                prev->next = head->next;
            } else {
                first_node = head->next;
            }

            newBlock->next = first_node;
            first_node = newBlock;
            freeList->head = first_node;

            head = first_node;
            continue;
        }

        prev = head;
        head = head->next;
    }
}
```

/***
Function: allocateMemory

Use: Allocates memory of specified size from the free memory list.

Arguments: 1.freeList - MemoryList pointer representing the list of free memory blocks.
2.usedList - MemoryList pointer representing the list of used memory blocks.
3.size - integer representing the size of memory to be allocated.
4.memoryArray - MemoryBlock pointer representing the initial memory block.
5.blockName - char pointer representing the name of the memory block being allocated.

Returns: int - the start address of the allocated memory block or -1 if allocation fails.

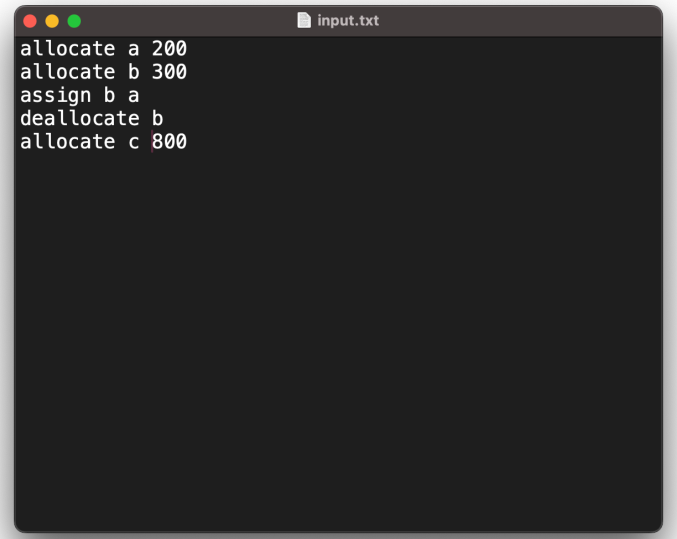
Notes:
• For allocating a memory block first traversed the freelist and searched for the block of size either greater than equal to required size.

• If the block is found the required size memory is taken and added the remaining again at the start of freelist and the required block to the usedlist.
• If the block is not found compaction is done and then allocate.

```
**********************************************************************************************************/

int allocateMemory(MemoryList* freeList, MemoryList* usedList, int size, MemoryBlock*
memoryArray,char* blockName) {

    MemoryBlock* current = freeList->head;
    MemoryBlock* prev = NULL;

    while (current != NULL) {

        if (current->size >= size) {

            // Allocate memory from the current free block
            int allocatedAddress = current->start_address;

            MemoryBlock* newBlock = (MemoryBlock*)malloc(sizeof(MemoryBlock));

            char* allocatedName = (char*)malloc(strlen(blockName) + 1);
            strcpy(allocatedName, blockName);
            newBlock->name = allocatedName;

            newBlock->ref_count = 1;
            newBlock->size = size;
            newBlock->start_address = allocatedAddress;
            newBlock->next = NULL;
            current->start_address += size;
            current->size -= size;

            // Move the allocated block to the used list
            addMemoryBlock(usedList, newBlock);

            return allocatedAddress;
        }

        prev = current;
        current = current->next;
    }

    compactMemory(freeList, usedList, memoryArray,blockName);

    // Retry allocation after compaction
    current = freeList->head;
    prev = NULL;

    while (current != NULL) {

        if (current->size >= size) {

            // Allocate memory from the current free block
```

```c
            int allocatedAddress = current->start_address;

            MemoryBlock* newBlock = (MemoryBlock*)malloc(sizeof(MemoryBlock));

            char* allocatedName = (char*)malloc(strlen(blockName) + 1);
            strcpy(allocatedName, blockName);
            newBlock->name = allocatedName;

            newBlock->ref_count = 1;
            newBlock->size = size;
            newBlock->start_address = allocatedAddress;
            newBlock->next = NULL;
            current->start_address += size;
            current->size -= size;
            addMemoryBlock(usedList, newBlock);

            return allocatedAddress;
        }

        prev = current;
        current = current->next;
    }

    printf("Error: Not enough free memory\n");
    return -1;
}


/*************************************************************************************************************
Function: deallocateMemory

Use: Deallocates memory by moving the block to the free memory list when reference
count becomes zero.

Arguments: 1.freeList - MemoryList pointer representing the list of free memory blocks.
2.usedList - MemoryList pointer representing the list of used memory blocks.
3.blockName - char pointer representing the name of the memory block to be
deallocated.

Returns: void

Notes: Decreases the reference count of a memory block. If the reference count becomes
zero, it moves the block from the used list to the free list. This is essentially deallocating
memory when the block is no longer in use.
*************************************************************************************************************/
void deallocateMemory(MemoryList* freeList, MemoryList* usedList, char* blockName) {
    MemoryBlock* block = usedList->head;
    MemoryBlock* prev = NULL;

    // Find the block with the specified name
    while (block != NULL && strcmp(block->name, blockName) != 0) {
        block = block->next;
    }

    if (block) {
        block->ref_count--;

        // Deallocation only happens when the reference count goes to 0
        if (block->ref_count == 0) {
            // Move the block back to the free list
            removeMemoryBlock(usedList, block);
            addMemoryBlock(freeList, block);
            printf("Deallocated memory for %s\n", blockName);
            return;
        }
    }

    printf("Error: Invalid deallocation\n");
}
```

input.txt file :

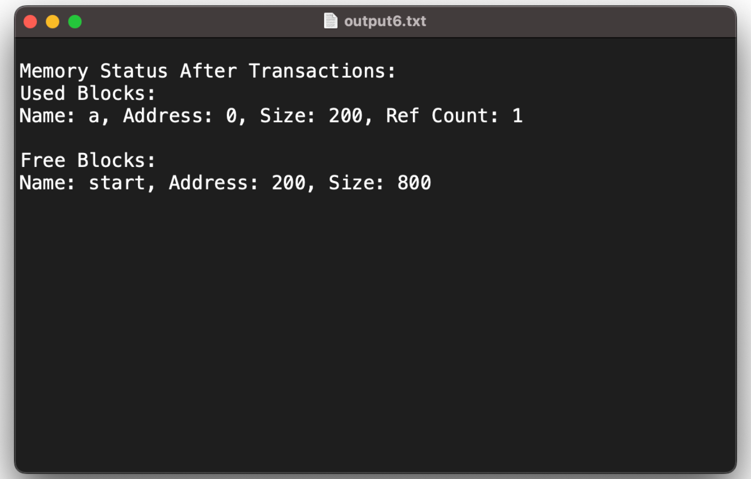


input.txt

```
allocate a 200
allocate b 300
assign b a
deallocate b
allocate c 800
```

Output.txt after each step :

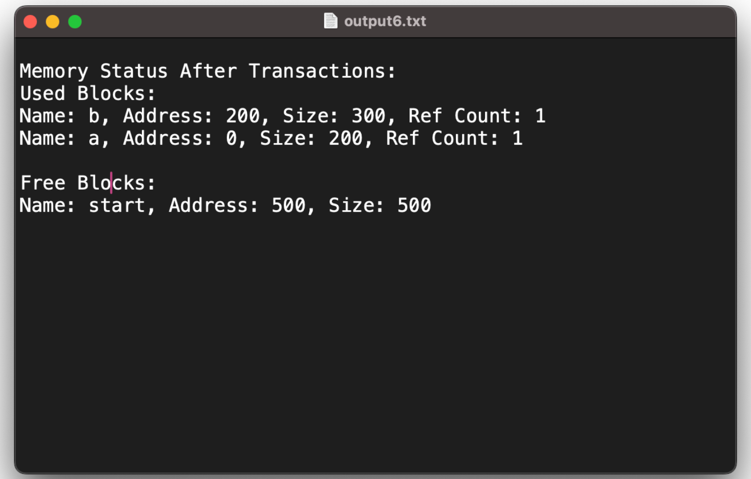
After allocate a 200 :



output6.txt

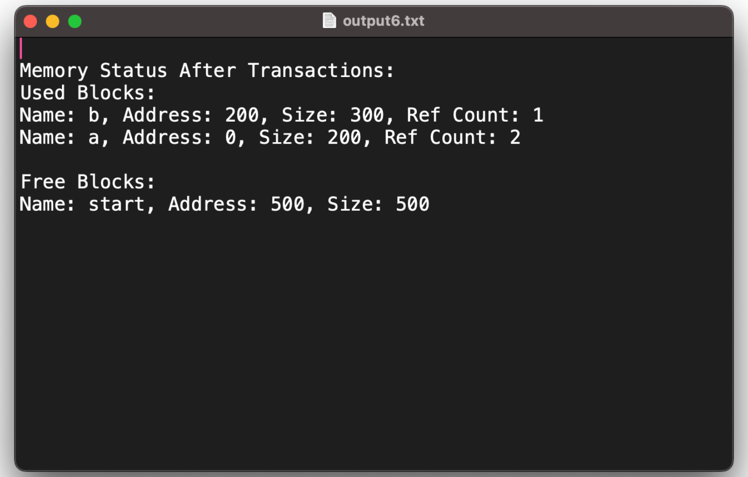
```
Memory Status After Transactions:
Used Blocks:
Name: a, Address: 0, Size: 200, Ref Count: 1

Free Blocks:
Name: start, Address: 200, Size: 800
```

After allocate b 300 :



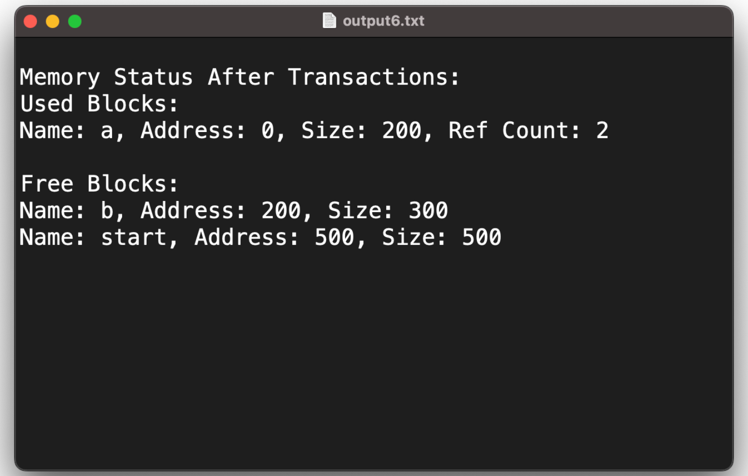
```
Memory Status After Transactions:
Used Blocks:
Name: b, Address: 200, Size: 300, Ref Count: 1
Name: a, Address: 0, Size: 200, Ref Count: 1

Free Blocks:
Name: start, Address: 500, Size: 500
```

After assign b a :



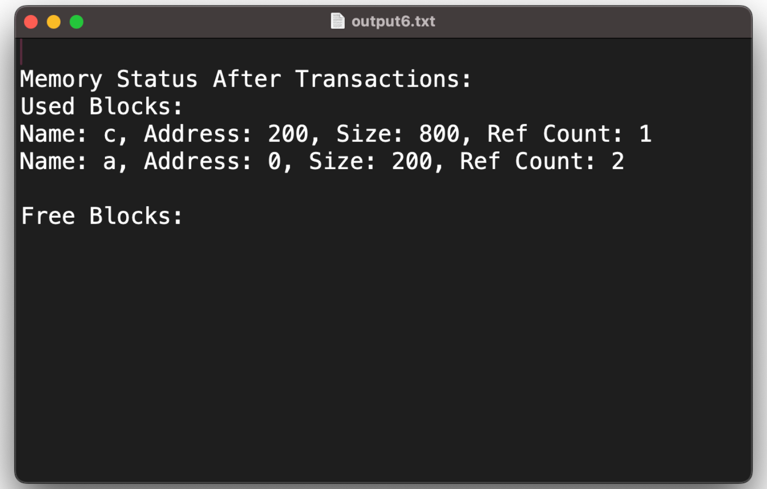
```
Memory Status After Transactions:
Used Blocks:
Name: b, Address: 200, Size: 300, Ref Count: 1
Name: a, Address: 0, Size: 200, Ref Count: 2

Free Blocks:
Name: start, Address: 500, Size: 500
```

After deallocate b :



```
Memory Status After Transactions:
Used Blocks:
Name: a, Address: 0, Size: 200, Ref Count: 2

Free Blocks:
Name: b, Address: 200, Size: 300
Name: start, Address: 500, Size: 500
```

After allocate c 800 :


output6.txt

```
Memory Status After Transactions:
Used Blocks:
Name: c, Address: 200, Size: 800, Ref Count: 1
Name: a, Address: 0, Size: 200, Ref Count: 2

Free Blocks:
```