**CS2106 Introduction to Operating Systems**

**Lab 4**

**Contiguous Memory Allocation**

**Answer Book**

|  |  |  |
| --- | --- | --- |
| **Name:**  **Ling Guan Yu** | **Student ID:**  **A0308292A** | **Lab Group Number:**  **B20** |
| **Name:**  **NIL** | **Student ID:**  **NIL** | **Lab Group Number:**  **NIL** |

Question 3.1a (1 mark)

8 bytes (64 units to allocate [64 / 1] which corresponds to 64 bits and hence 8 bytes [64 / 8] as 1 byte is 8 bits)

Question 3.1b (1 mark)

1088 bytes (64 units to allocate [64 / 1] in the worst case which corresponds to 64 nodes and hence 1088 bytes [64 \* (4 + 4 + 4 + 4 + 1)])

Question 3.1c (1 mark)

Advantage: Lesser memory is used for the same number of units as each unit is represented by a single bit.

Disadvantage: Cannot contain more information than free/not free per unit as each unit is only represented by a single bit.

Question 3.2 (1 mark)

No, it would not make a difference as both (signed) char and unsigned char are 1 byte (8 bits) in size and the difference between the range of values they can represent does not affect anything.

Besides, choosing to use char (signed or unsigned) in the 1st place is probably just because it is the smallest primitive data type in C and hence using it results in space savings. In reality, using other data types that can represent the 2 states of allocated and freed will work too.

Question 3.3 (1 mark)

Yes, it needs to know how many bytes of memory need to be freed.

To get this information, I used the linked list library to create a static global linked list for keeping track of the length of each allocated block of memory using its “start” as the key. To get “start” in the myfree routine, I used the get\_index routine provided.

Question 4.1 (1 mark)

char isTaken; //No primitive bool :(

size\_t length;

Question 4.2 (1 mark)

I did printf("%zu\n", sizeof(TNode)); which led to “64” being printed to the screen so 64 bytes per node.

Best Case Storage Requirement: 64 bytes (1 node in linked list partitions the heap so 64 \* 1 bytes)

Worst Case Storage Requirement: 4 MiB (64K nodes in linked list partition the heap since 64K / 1 units to allocate so 64 \* 1024 \* 64 bytes which is 4 \* 1024 \* 1024 bytes and hence 4 MiB)

Question 4.3 (1 mark)

It incurs an overhead which grows quickly with the total size of memory since it is relatively small compared to total memory size. The overhead is largely the huge amount of information needed to be kept to ensure proper functioning of the dynamic memory allocation.

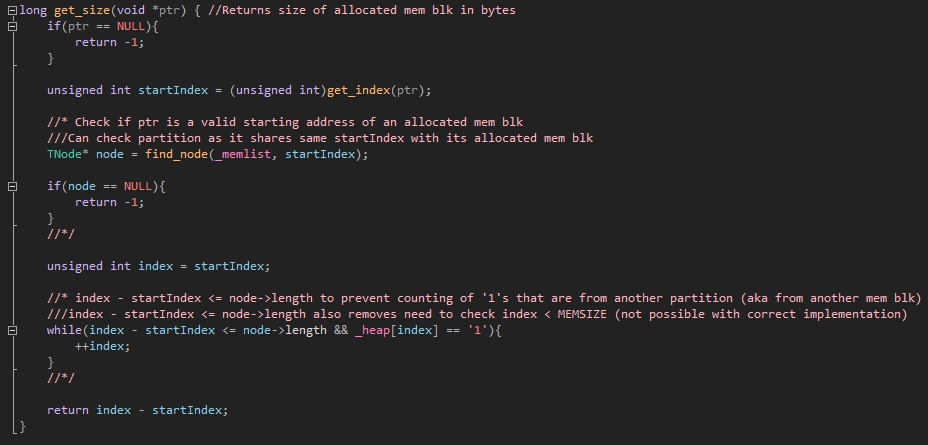
Question 4.4 (1 mark)

Sensible minimum allocation unit size is 64 bytes (matches size of node in linked list).

Question 5.1a (1 mark)

**Explanation:** \_memlist keeps track of all allocated and free partitions in the buddy system, using their starting indices as keys. Starting address = \_heap + start index so get\_index(ptr) yields start index which is used to find the corresponding node in \_memlist. \_memlist does not store size of the allocated memory block and hence the number of ‘1’s that belong in the partition represented by the node found from \_memlist was counted to obtain it.

**IDE ver (clearer to see imo):**



**Text ver (comments removed so not so messy):**

long get\_size(void \*ptr) { //Returns size of allocated mem blk in bytes

if(ptr == NULL){

return -1;

}

unsigned int startIndex = (unsigned int)get\_index(ptr);

TNode\* node = find\_node(\_memlist, startIndex);

if(node == NULL){

return -1;

}

unsigned int index = startIndex;

while(index - startIndex <= node->length && \_heap[index] == '1'){

++index;

}

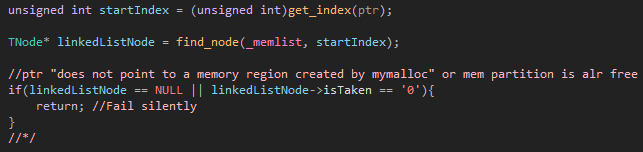
return index - startIndex;

}

Question 5.1b (1 mark)

In myfree(), start index can be obtained by doing get\_index(ptr), which will then be used as a key to a node in \_memlist. As mentioned before, since \_memlist keeps track of all allocated and free partitions in the buddy system, the node found represents a block (partition) in the buddy system. No node found means the pointer given was not a valid starting address and linkedListNode->isTaken == ‘0’ implies that the partition is already free and hence there is no need to free the memory block with the starting address of “ptr”.

**IDE ver:**



**Text ver:**

unsigned int startIndex = (unsigned int)get\_index(ptr);

TNode\* linkedListNode = find\_node(\_memlist, startIndex);

if(linkedListNode == NULL || linkedListNode->isTaken == '0'){

return; //Fail silently

}

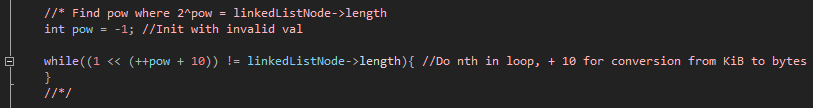
Question 5.2 (1 mark)

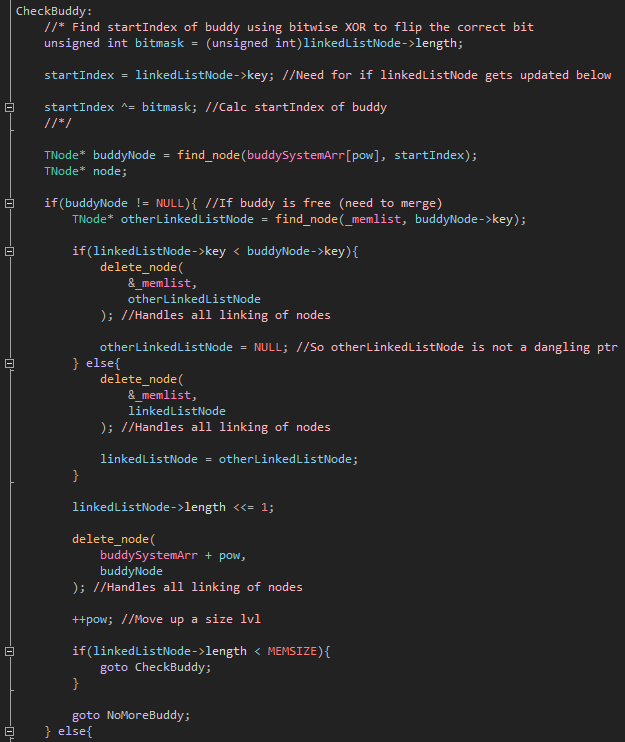
The bit corresponding to the length of the block (partition) in the starting address is inverted to get the starting address of the buddy. This can be done with a combination of a suitable bitmask (= block length) and bitwise XOR (for flipping the bit).

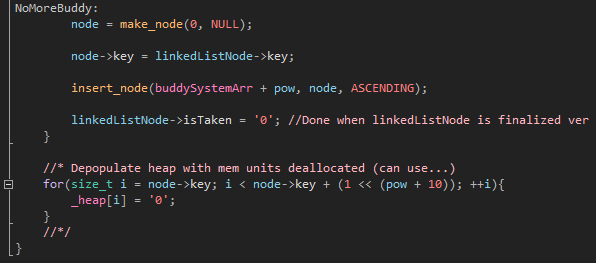
Question 5.3 (2 marks)

The index to the correct linked list to search for a free buddy is found using the pow variable. With that, the pointer to the buddy node (“buddyNode”) can be found. As mentioned in 5.2, a combination of bit masking and bitwise XOR can be used to translate the starting address of the memory to be freed to the starting address of its buddy. Since buddyNode is a pointer to a node in 1 of the linked lists in buddySystemArr, it represents a free buddy since its value is the return value of find\_node(buddySystemArr[pow], startIndex), where the left parameter ensures the pointer found is a node that represents a free memory block while the right parameter ensures the pointer found is a node adjacent to the node corresponding to the memory to be freed and hence is its buddy. For merging, the linked list node corresponding to its buddy (“otherLinkedListNode”) is found using the key from buddyNode (buddyNode->key). Then, the key from the linked list node corresponding to the memory to be freed is compared with buddyNode->key to determine which linked list node should be deleted and which should be retained to represent the merger of both nodes. The length of the retained link list node is doubled to reflect this. After that, the buddyNode is removed to prepare for merging of it and its buddy. The pow variable is then incremented by 1 before control is handed back upwards to check for a free buddy again or handed back downwards to insert the newly merged node into the correct linked list in buddySystemArr and mark the finalized merged linked list node as free.

**IDE ver (3 pics? GG for the text ver):**







**Text ver (spam incoming):**

int pow = -1; //Init with invalid val

while((1 << (++pow + 10)) != linkedListNode->length){

}

CheckBuddy:

unsigned int bitmask = (unsigned int)linkedListNode->length;

startIndex = linkedListNode->key;

startIndex ^= bitmask;

TNode\* buddyNode = find\_node(buddySystemArr[pow], startIndex);

TNode\* node;

if(buddyNode != NULL){

TNode\* otherLinkedListNode = find\_node(\_memlist, buddyNode->key);

if(linkedListNode->key < buddyNode->key){

delete\_node(

&\_memlist,

otherLinkedListNode

);

otherLinkedListNode = NULL;

} else{

delete\_node(

&\_memlist,

linkedListNode

);

linkedListNode = otherLinkedListNode;

}

linkedListNode->length <<= 1;

delete\_node(

buddySystemArr + pow,

buddyNode

);

++pow;

if(linkedListNode->length < MEMSIZE){

goto CheckBuddy;

}

goto NoMoreBuddy;

} else{

NoMoreBuddy:

node = make\_node(0, NULL);

node->key = linkedListNode->key;

insert\_node(buddySystemArr + pow, node, ASCENDING);

linkedListNode->isTaken = '0';

}

**TOTAL: \_\_\_\_\_\_\_\_\_\_\_ / 14**