ECE-350 L1 Report

Tony, Shen, t48shen
Moyez Farook, Mansoor, mfmansoo
Anthony Natale, De Luca, andeluca
Anson, Wan, a23wan

DATA STRUCTURES AND ALGORITHMS

For our buddy system implementation the overall goal was to achieve an O(h) (h is the height of the binary tree) time complexity for memory allocation and deallocation. To do this we employed the following data structures:

- Circular Linked List
- Binary Tree

Circular Linked List

We used a circular linked list to implement our free list. Without a free list, to find a free node you may have to traverse the entire binary tree. This would not allow us to stay inline with the goal of having both operations happen in O(h) time.

For allocation the free list allows us to find a free node in O(h) time. This is because the operation of finding a used node consists of potentially traversing through every level of the tree O(h). In addition the maximum number of splits you will have to do is O(h). Every split itself is O(1). Removing a node from the free list and setting it as used are O(1), which makes the overall runtime for allocation O(h).

With respect to deallocation, the free list does not offer much benefit, but it also does not add much run time. The free lists' only concern is how expensive it is to add a node to the free list. In our implementation it is O(1), so this extra memory usage is a worthwhile tradeoff for O(h) allocation.

Binary Tree

Since we used a buddy system implementation for our data blocks it made the most sense to store our blocks in a binary tree as blocks would always be split in 2, naturally creating a binary tree structure. The binary tree structure allows us to implement deallocation in O(h) time as it keeps track of which data blocks are paired together which allows us to coalesce paired free blocks in at most O(h) steps if blocks need to be coalesced from the bottom of the tree all the way to the top. The binary tree structure does not affect the implementation of allocation by much as it only assists the free lists in finding a starting height in which to search for a free block.

Allocation Algorithm

Our allocation algorithm works by first calculating the height in the binary tree in which a data block must be allocated based on size of the request, then we move on to looking through the free list at that height and checking if there are any free blocks that we can allocate. If we find a free

block then we simply allocate it, but if we cannot find a block at our required height we then move up a layer in the binary tree and check that height for free blocks and continue doing so until we come across a free block. If there is no free block then we do not have space for the requested allocation; if we do find a free block we will commence splitting the blocks until we have created a block at our required height in the tree. We do this by marking the block we are going to split as used, removing it from the free lists, and then adding two child blocks to the free lists and moving down a layer to repeat the process until completion. In a worst case this will run in O(h) time since we might have to traverse the entire tree twice, with splitting and allocating being done in constant time.

Deallocation Algorithm

_____Our deallocation algorithm works by first finding which index in the binary tree the pointer belongs to. Because there are many pointers that correspond to the same index we need to traverse the binary tree to find the index lowest in the tree that is marked as used. Once we have the index we can mark that index as free. We then start to coalesce blocks until the buddy of the most recent freed/coalased block is no longer free. Coalescing consists of finding a pair of buddies who are both free, removing them from the free list and then adding their parent node to the free list instead. Overall the runtime for deallocation is O(h) because we need to traverse the tree to find the used index and we can potentially coalesce from the bottom of the tree to the top.

TEST SUITE DESCRIPTIONS

Test Suite 1 - ae_mem1_G10.c

In this Test Suite we test that our program successfully coalesces free buddy regions together, reuses these newly-freed blocks, and returns the correct number/size of free blocks, all while not leaking memory. The Test Suite takes the following steps, with 10 Test Cases included at various points. This Test Suite is meant to test basic functionality as Test Cases are close together to emulate simple checkpoints in code execution.

START

1. mem_alloc p[0] 0x2000
    a. TEST CASE 1: verify there are 2 available blocks left (0x2000, 0x4000)
2. mem_alloc p[1] 0x2000-2
3. mem_alloc p[2] 0x2000
    a. TEST CASE 2: verify there is 1 available block left (0x2000), no blocks the same
4. mem_alloc p[3] 0x2000-2 (NOW FULL)
    a. TEST CASE 3: verify there are 0 available blocks left, no blocks the same
5. mem_dealloc p[0]
6. mem_dealloc p[1]
    a. TEST CASE 4: verify there is 1 available (coalesced) block left (0x4000)
7. mem_alloc p[0] 0x4000 (NOW FULL)

    a. TEST CASE 5: verify that size 0x4000 (coalesced) block allocated correctly
    b. TEST CASE 6: verify there are 0 available blocks left
8. mem_dealloc p[2]
9. mem_alloc p[1] 0x1000
    a. TEST CASE 7: verify there is 1 available block left (0x1000), no blocks the same
10. mem_dealloc p[1]
11. mem_dealloc p[3]
12. mem_alloc p[1] 0x4000-2 (NOW FULL)
13. mem_alloc p[2] 0x1000-2
    a. TEST CASE 8: verify there are 0 available blocks left, only p[1] is allocated
14. mem_dealloc p[0]
    a. TEST CASE 9: verify there is 1 available block left (0x4000)
15. mem_dealloc p[2] (NOW EMPTY)
    a. TEST CASE 10: verify there is 1 available block left (0x8000)

Test Suite 2 - ae_mem2_G10.c

This test suite stress tests that the RTOS can allocate all the memory available and can correctly utilize the buddy algorithm to split the memory into appropriate blocks.
This suite also tests that the program returns the correct number of free memory blocks by filling the memory fully and also by deallocating selected blocks to see if the returned number of free blocks changes.

There are various tests in this suite which cover basic to extreme/stress scenarios:

- Test 1: Allocates 1 block of the total 32K bytes of memory
- Test 2: Allocates 2 blocks of 16K bytes of memory
- Test 3: Allocates 8 blocks of 4K bytes of memory
- Test 4: Allocates 128 blocks of 256 bytes of memory
- Test 5: Allocates 256 blocks of 128 bytes of memory
- Test 6: Allocates 512 blocks of 64 bytes of memory
- Test 7: Allocates 1024 blocks of 32 bytes of memory
- Test 8: Deallocates the last 32 byte block and checks that mem_dump() returns the correct number of free blocks
- Test 9: Deallocates a 32 byte block elsewhere and checks that mem_dump() returns the correct number of free blocks
- Test 10: Deallocates the buddy block next to a previously deallocated block (thus the two blocks should coalesce into one block) and checks that mem_dump() returns the correct number of free blocks
- Test 11: Deallocates a block that is next to a previously deallocated block but is not the buddy of that block (thus they should NOT coalesce into one block), then checks that mem_dump() returns the correct number of free blocks
- Test 12: Allocates new memory blocks that fit into the previously deallocated blocks from Test 8,9,10,11 and checks that mem_dump() returns the correct number of free blocks afterwards

- Test 13: Deallocate all 1024 32 byte blocks that were allocated, then checks that mem_dump() returns the correct number of free blocks

Test Suite 3 - ae_mem3_G10.c

In this Test Suite, we test that our program successfully returns the correct number of free memory blocks, coalesces free buddy regions together, reuses these newly-freed blocks, and returns the correct number/size of free blocks, all while not leaking memory. This is achieved by splitting the Test Suite into separate Test Cases, each of which tests all parts of our program with operations similar to those we imagine would be required of the allocator in future projects. This Test Suite includes 7 Test Cases, each of which contain a large set of consecutive operations with outputs to be verified. Test Cases 1-3 are connected and emulate a sequence of possible required operations, while Test Cases 4-7 individually test memory conservation and internal/external fragmentation.

TEST CASE 1:

Allocate increasing amounts of memory (leaving gaps in between), then call mem_dump() along the way to verify that the correct number of available blocks + sizes are displayed (when filling the available blocks). This Test Case checks that memory is always allocated in the first available block with sufficient space.

1. mem_alloc(p[0]) 0x400
2. mem_dump()
    a. Number of free blocks = 5 (0x400, 0x800, 0x1000, 0x2000, 0x4000)
3. mem_alloc(p[2]) 0x800
4. mem_dump()
    a. Number of free blocks = 4 (0x400, 0x1000, 0x2000, 0x4000)
5. mem_alloc(p[4]) 0x2000note
6. mem_dump()
    a. Number of free blocks = 3 (0x400, 0x1000, 0x4000)

Test Case 1 will pass if each mem_dump() call returns the correct number of free blocks.

TEST CASE 2:

Fill the entire memory block by allocating into the available blocks/gaps, in decreasing order: 0x4000 -> 0x1000 -> 0x400. Use mem_dump() along the way to verify they are slotted correctly. This Test Case checks that memory is able to be allocated into spaces in which they would fit exactly, as well as makes sure by allocating the largest block first that we are able to quickly find a large enough free block, and that memory is never overwritten. This case fills the entire memory space.

1. mem_alloc(p[5]) 0x4000
2. mem_dump()
    a. Number of free blocks = 2 (0x400, 0x1000)
3. mem_alloc(p[3]) 0x1000
4. mem_dump()
    a. Number of free blocks = 1 (0x400)
5. mem_alloc(p[1]) 0x400
6. mem_dump()
    a. Number of free blocks = 0

Test Case 2 will pass if each mem_dump() call returns the correct number of free blocks, and the first two equally sized blocks (p[0] and p[1]) are not the same, as an additional check.

TEST CASE 3:

Deallocate all memory blocks in increasing order (size), verifying along the way that each freed block is coalesced with its buddy. At each step, there should only be one available memory block, which increases in size. Once all memory is deallocated, allocate + deallocate a full block (0x8000) to verify that the entire freed memory block is reusable. This case empties the entire memory space.

1. mem_dealloc(p[0])
2. mem_dump()
    a. Number of free blocks = 1 (0x400)
3. mem_dealloc(p[1])
4. mem_dump()
    a. Number of free blocks = 1 (0x800)
5. mem_dealloc(p[2])
6. mem_dump()
    a. Number of free blocks = 1 (0x1000)
7. mem_dealloc(p[3])
8. mem_dump()
    a. Number of free blocks = 1 (0x2000)
9. mem_dealloc(p[4])
10. mem_dump()
    a. Number of free blocks = 1 (0x4000)
11. mem_dealloc(p[5])
12. mem_dump()
    a. Number of free blocks = 1 (0x8000)
13. mem_alloc p[0] 0x8000
14. mem_dump()
    a. Number of free blocks = 0
15. mem_dealloc p[0]
16. mem_dump()
    a. Number of free blocks = 1 (0x8000)

Test Case 3 will pass if each mem_dump() call returns the correct number of free blocks, and the final allocation of 0x8000 into p[0] is successful (p[0] != NULL), as an additional check.

TEST CASE 4:

This test checks to see that no extra memory appears and that no memory gets lost after repeatedly allocating and deallocating memory. The sum of free memory and allocated memory should always be constant.

The way this test checks that no extra memory appears is by allocating all blocks and deallocating all blocks twice, then allocating another 1024 32 byte blocks. If at this point the test did not fail, that means no memory was lost as 1024 x 32 bytes = 32768 bytes so there is still the correct maximum number of bytes available.

The way this test checks that no extra memory appears is by trying to allocate another 32 byte block after fully allocating all 1024 32 bytes blocks. If the pointer that is returned by the mem_alloc()

function is NULL, that means the allocation was unsuccessful and if mem_dump() also returns 0 that means there is no free block, thus no extra memory appears.

TEST CASE 5:

This simple test checks to see internal fragmentation and that the correct number of free blocks is correctly returned using mem_dump().

This test allocates a large irregular (non power of 2) value of 17000 bytes in memory and since 17000 is larger than 16384, that means the 32K byte block cannot be split, thus the 17000 byte allocation takes up the entirety of the 32K bytes of memory. Then the test checks that mem_dump() returns the correct number of free/allocatable blocks which in this case should be 0. Then also tries to allocate blocks of 3000 bytes which should technically fit as there is still about 15000 bytes available, but since the allocated 17000 bytes takes up the whole 32k bytes of memory the allocations should fail and the test checks that those pointers are NULL. Thus testing the internal fragmentation issues that exist with the buddy method/algorithm.

TEST CASE 6:

This is a more advanced test to see internal fragmentation and that the correct number of free blocks is correctly returned using mem_dump().

This test allocates many smaller irregular (non power of 2) values of memory. In this case 9000, 1500, 1500, 4100, and 2050 bytes are allocated in that order, then 32 bytes are attempted to be allocated. Although there are much more than 32 bytes that are not being used, they are all within allocated blocks (internal fragmentation) so this allocation should fail and thus the pointer is checked to be NULL. Also mem_dump() should return zero to show that no blocks are free/allocatable.

TEST CASE 7:

This test checks to see external fragmentation by allocating various blocks of large and small sizes then until there are no more free/allocatable memory blocks. In this case that is done by allocating 9000, 1500, 1500, 4100, and 2050 byte blocks which results in no free blocks. Then, the 9000 and 4100 blocks are deallocated and thus there should be about 24576 bytes of unused memory, but since those two blocks are not buddys, they cannot be merged and thus external fragmentation exists and attempting to allocate 24000 bytes of memory results in a failure so the pointer is checked as NULL, showing that the allocation failed. Also, mem_dump() is checked to return 2 since there are two free blocks.