University of Waterloo

ECE 350 Lab1 Report

Group 12

Members: Geoffrey Qin, Yiru Wu,

Rebecca Zhou, Kehan Xing

Group ID: 12

Lab 1 Kernel Memory Management

Data Structure

We have two structures created for this lab, one is for each node in the free list and the other one is for the overall free list that links all the free memory together in each level of the binary tree. Since we are creating a doubly linked list, each node is constructed of two pointers – one points to the previous node (`dnode *prev`) and one points to the next node (`dnode *next`). Each node represents a section of free memory that has been created. The free list consists of a head (`dnode *head`) and a tail pointer (`dnode *tail`) that points to the head node and tail node respectively.

Other than the two data structures, we also have bit tree which is an array of bits and a free list array. They are initialized during memory pool initialization (`k_mpool_create`). To create the bit array, we use an array of `unsigned char` where each `char` contains eight bits. To calculate the length of the array, we simply take the ceiling function of the division of total memory length by eight. We then initialize each bit with 0. The array of free list has the length of the maximum number of levels and index 0 (`ram_array[0]`) points to the starting address of the memory pool.

Algorithm

Helper Functions

a. `set_bit (bit_tree [], index)`

   This function takes in an array of `unsigned char` and an index (in terms of bit) as inputs. The purpose of this function is to set the value in that index (in terms of bit) in the array to 1. If the bit in that index is already set to 1, the value will remain the same.

b. `clear_bit (bit_tree [], index)`

   This function takes in an array of `unsigned char` and an index (in terms of bit) as inputs. The purpose of this function is to clear the value in that index (in terms of bit) in the array to 0. If the bit in that index is already cleared, the value will remain as 0.

c. `check_bit (bit_tree [], index)`

   This function takes in an array of `unsigned char` and an index (in terms of bit) as inputs. The purpose of this function is to check the value in that index (in terms of bit) in that array. It will return 1 if the value is set to 1 and return 0 if the value is cleared or never assigned.

d. `print_bit_tree (bit_tree [], array_length)`

This function takes in an array of `unsigned char` and the length of that array as inputs. This function will print the value of every bit inside of the array from index (in terms of bit) 0 to the end of that array.

e. `LOG2(U32 val)`

This function takes an unsigned int variable and calculates its binary logarithm. Because the function is supposed to take the size of memory and help to determine which level it should be placed in the tree representation, the smallest return value will be MIN_BLK_SIZE_LOG2.

f. `POWER2 (U32 power)`

This function takes an unsigned int variable and calculates the power of 2 of the input. The function is mainly used to calculate the address of a memory block.

g. get_address(U32 curr_location, U32 index, U32 level)

This function takes the *curr_location* (address), *index*, and *level* values of the current block that is being freed to compute the starting address of the block that needs to be coalesced. It starts with getting the reminder of *index*/2 to know whether the block is left(reminder = 0) or right(reminder =1) children node. Then it multiplies the reminder value with the size value of the block in that level to get the offset address, and subtract the offset address from *curr_location* to get the starting address of the block that needs to be coalesced.

Allocation

The k_mpool_alloc function takes the size of memory and calculates the level best fits the size in the tree representation of buddy system. We recorded the level value in a variable called *level* and created another variable called *curr_level* which has the same initial value but is used when traversing through the free list.

If there's no free block on the calculated level according to the free list, our code will check its upper level repeatedly using a while loop until a free block is found. The variable *curr_level* is decreased by 1 each time to track the level we're checking. If we still can't find a free block after we reach the top level, the function will return NULL.

After finding a free block, we want to split it to best fit the input size. To do that, we split the block into two children just like the tree representation. The original block will then be removed from the free list, and its children will be inserted into the free list instead. We can then increase the value of *curr_level* and do the same operation to one of its children until *curr_level* equals to *level* which means we are back to the desired level.

Now we can find two free blocks in the free list of the level which can best fit the input size, so we can simply allocate memory to the first free block, return its address and remove it from the free list.

Deallocation

The deallocation takes the memory pool id *mpid* and pointer to the block *ptr* to free the block in the memory pool. We will first verify if the *ptr* or the *mpid* inputs are valid, and return RTX_ERR if the inputs are invalid. Once the *pt*r address is verified belonging to the memory pool specified by the *mpid* input, the function starts computing the level and *x* index value of the block that needs to be freed.

The function starts with setting *curr_level* variable to the lowest level of the tree and calculate *x* variable by dividing the offset between the *ptr* address and the starting address of the memory pool with MIN_BLK_SIZE according to the algorithm on lecture slide 24. Then, the function will keep and checking the bit_tree entry for the block specified by the *curr_level* and *x* variable values and updating *curr_level* and x variable values to move upward from the block until it finds the block that is allocated and needs to be freed or reached the root of the tree.

If the block that needs to be freed was found in non-root level, the function will set the block's allocation status in *bit_tree* to 0(free), check its buddy's allocation status and coalesce both blocks if they are all free. The function calls *get_address* function with the current block's address, *x*, and *curr_level* value to get the starting address of the block that needs to be coalesced and store it in *curr_address* variable. Then, it will remove one DNODE from the *free_list[curr_level]* and update the head node for the *free_list[curr_level]* as the buddy node is now coalesced. Subsequently, the program will update the *curr_level* and x values to the parent block of the current block and create a new free DNODE at *curr_address* in the *free_list[curr_level]* which points to the parent block level to complete coalescence. The function will keep setting block's allocation status to 0(free), checking buddy's allocation status and performing coalescence moving upward in the tree until its buddy's allocation status is not free. The function will return RTX_OK once all deallocation and coalescence are completed.

If the block that needs to be freed was found in root level, the function will set the block's allocation status in bit_tree to 0(free) and create a new free DNODE at block's address in the *free_list[curr_level]* which points to the root level to finish deallocation. The function will return RTX_OK once the deallocation for the root node is completed.

If the block address cannot be found in any level of the tree, the function will return RTX_ERR. If the *ptr* address does not belong to the memory pool specified by *mpid*, it will set *errno* to EFAULT and return RTX_ERR. If the memory pool id *mpid* is invalid, it will set *errno* to EINVAL and return RTX_ERR. If the *ptr* is null, it will return RTX_ERR with no operation.

Test Cases

ae_mem1_G12.c

Overall: This test suit is used to test allocation issues and dump function.

Allocation has following properties:

1. Allocate required size memory to pointer.
2. Found the appropriate block based on the specified size.
3. Split memory block if needed.
4. Minimum block size is 0x20.
5. No extra memory allowed.

Dump function has following properties:

- Return number of free blocks.
- Print address of free blocks.

1) Test #1
    a) Initial state: 1 0x8000 block is free.
    b) Actions: Ask for size of 0x4000 memory, and assign the address to p[0]
    c) Main idea: Allocation should find the appropriate size of 0x4000 is 0x4000. Then the 0x8000 block should split to 2 0x4000 blocks, one is assigned to p[0] and the other one is free.
    d) Expected final state: 1 0x4000 block is free.
    e) Test purposes: This case tests whether allocation could allocate memory to the pointer.

2) Test #2
    a) Initial state: 1 0x4000 block is free.
    b) Actions: Ask for size of (0x1000-4) memory, and assign the address to p[1]
    c) Main idea: Allocation should find the appropriate size of (0x1000-4) is 0x1000. Then the 0x4000 block should split to 2 0x1000 blocks and 1 0x2000 blocks. One 0x1000 block is assigned to p[1] and the others are free.
    d) Expected final state: 1 0x1000 block and 1 0x2000 block are free.
    e) Test purposes: This case tests whether the 0x8000 block will split when we asked for 0x4000 block in test #1. If the (0x1000-4) memory could be allocated, it means that the allocation could find the appropriate size and split to small pieces if needed.

3) Test #3
   a) Initial state: 1 0x1000 block and 1 0x2000 block are free.
   b) Actions:
      Ask for size of (0x1000+2) memory, appropriate size is 0x2000.(1 0x1000 free block left)
      Ask for size of (0x800) memory, appropriate size is 0x800. The 0x1000 block split to two 0x800 blocks. (1 0x800 free block left)
      Ask for size of (0x400-2) memory, appropriate size is 0x400. The 0x800 block split to two 0x400 blocks. (1 0x400 free block left)
      Ask for size of (0x10) memory, due to min size is 0x20. The 0x400 block should split until the minimum size of block is 0x20. (1 0x20, 1 0x40, 1 0x80, 1 0x100, 1 0x200 free blocks left)
      Ask for size of (0x180) memory, appropriate size is 0x200. (1 0x20, 1 0x40, 1 0x80, 1 0x100 free blocks left)
   c) Main idea: Allocation should find the appropriate size of these memory and split as needed.
   d) Expected final state: 1 0x20, 1 0x40, 1 0x80, 1 0x100 blocks are free.
   e) Test purposes: This case tests whether the allocation could find appropriate block for required memory size and split if needed. Also, this case is a prepare for following test on minimum size block limit.

4) Test #4
   a) Initial state: 1 0x20, 1 0x40, 1 0x80, 1 0x100 blocks are free.
   b) Actions: mem_dump()
   c) Main idea: The function mem_dump() should return the correct number of free blocks, which should be 4.
   d) Expected final state: 1 0x20, 1 0x40, 1 0x80, 1 0x100 blocks are free.
   e) Test purposes:

      This case is used to test whether the dump function could check the free-list and return correct number of free blocks.

      Also, this case tests if there is a minimum block size limit. If the return value is 4, then the appropriate size for 0x10 memory is 0x20, which means the minimum size limit is 0x20. If the return value is 5, then it means there is no minimum block size limit.

5) Test #5
   a) Initial state: 1 0x40, 1 0x100 blocks are free.
   b) Actions: Ask for size of (0x101) memory, appropriate size is 0x200.
   c) Main idea: Since there is no free block has size more than 0x100, the 0x101 memory should not be allocated.
   d) Expected final state: 1 0x40, 1 0x100 blocks are free.

   e) Test purposes: This case tests whether there is extra memory exist. I tried to allocate a memory larger than the largest existing block size. If it cannot succeed, it means that there is no extra memory.

6) Test #6
   a) Initial state: 1 0x40, 1 0x100 blocks are free.
   b) Actions: mem_dump()
   c) Main idea: The function mem_dump() should return the correct number of free blocks, which should be 2.
   d) Expected final state: 1 0x40, 1 0x100 blocks are free.
   e) Test purposes: This case tests whether allocated an invalid memory will change the number of free blocks.

7) Test #7
   a) Initial state: no free block.
   b) Actions: Ask for size of (0x10) memory, appropriate size is 0x20.
   c) Main idea: The 0x10 memory could not be allocated because there is no free blocks
   d) Expected final state: no free block.
   e) Test purposes: This case tests whether there are illegal gaps, and to confirm the minimum block size limit again. If the 0x10 memory could allocated, there are two possible reasons. One is that there are small memory gaps, the other is that the previous allocated 0x10 memory occupies a 0x10 block and left a free 0x10 block.

8) Test #8
   a) Initial state: no free block.
   b) Actions: mem_dump()
   c) Main idea: The function mem_dump() should return the correct number of free blocks, which should be 0.
   d) Expected final state: no free block.
   e) Test purposes: Finally use dump function to check there is no free blocks.

If all test cases are pass, it should print:
G12-TS1: 8/8 tests PASSED
G12-TS1: 0/8 tests FAILED
G12-TS1: END

<u>ae_mem2_G12.c</u>

Overall: This test suit is used to test deallocation issues, dump function, and allocation issues.

Deallocation has following properties:

1. Find the pointed address and deallocate the whole block.
2. Double deallocation will bring RTX_ERR message.

3. If the adjacent blocks are also free, combine to make a bigger free block.
4. Repeated allocating and deallocating is available.
5. No extra memory exists.
6. No memory lost.

Dump function has following properties:

- Return number of free blocks.
- Print address of free blocks.

1) Test #1
    a. Initial state: 1 0x8000 block is free.
    b. Actions:
       Ask for size of (0x2000+100) memory, appropriate size is 0x4000. (1 0x4000 free block left)
       Ask for size of (0x2000-2) memory, appropriate size is 0x2000. (1 0x2000 free block left)
    c. Main idea: Allocation should find the appropriate size require memory and split if needed.
    d. Expected final state: 1 0x2000 block is free.
    e. Test purposes: This case tests whether allocation could find appropriate block for required memory and split as needed. This step is used to confirm the blocks are allocated and then we could continuous following deallocation test.

2) Test #2
    a. Initial state: 1 0x2000 block is free.
    b. Actions: Deallocate a block with size of 0x2000 and using mem_dump to check.
    c. Main idea: After the 0x2000 block is deallocated, there should have 2 0x2000 free blocks. Deallocation should detect they are adjacent free blocks and combine them to form a 0x4000 free block. Then mem_dump should return 1 since there is only one 0x4000 free block.
    d. Expected final state: 1 0x4000 block is free.
    e. Test purposes: This case tests whether the deallocation could work, and also blocks should combine together when there are two adjacent free blocks.

3) Test #3
    a. Initial state: 1 0x4000 block is free.
    b. Actions: Deallocate the address to a null pointer
    c. Main idea: When trying to deallocate a null pointer, deallocation should detect the pointer is null and return RTX_ERR
    d. Expected final state: 1 0x4000 block is free.
    e. Test purposes: This case tests null pointer deallocation error and mem_dealloc() function return value.

4) Test #4
    a. Initial state: 1 0x4000 block is free.
    b. Actions: Ask for size of (0x1000-4) memory, appropriate size is 0x1000. (1 0x1000, 1 0x2000 free blocks left) Then using mem_dump() to check.
    c. Main idea: Allocation should find the appropriate size require memory and split if needed.
    d. Expected final state: 1 0x1000 block and 1 0x2000 block are free.
    e. Test purposes: mem_dump() should return correct number of free blocks to prove repeatedly allocating and deallocating could work on same pointer.

5) Test #5
    a. Initial state: 1 0x1000 block and 1 0x2000 block are free.
    b. Actions:
       Ask for size of (0x1000+2) memory, appropriate size is 0x2000. (1 0x1000 free block left)
       Ask for size of (0x1000+2) memory, appropriate size is 0x2000, could not be allocated. (1 0x1000 free block left)
    c. Main idea: Since there is no free block has size more than 0x1000, the (0x1000+2) memory should not be allocated.
    d. Expected final state: 1 0x1000 block is free.
    e. Test purposes: This case tests whether there is extra memory exist. I tried to allocate a memory larger than the largest existing block size. If it cannot succeed, it means that there is no extra memory.

6) Test #6
    a. Initial state: 1 0x2000 block is free.
    b. Actions:
       Ask for size of (0x2000) memory, appropriate size is 0x2000. (no free block left)
    c. Main idea: The two 0x1000 blocks should combine to form one 0x2000 free block, and the 0x2000 memory should be allocated.
    d. Expected final state: no free block.
    e. Test purposes: This case tests whether two free adjacent blocks could combine and no memory leak in this process. If the 0x2000 memory could not be allocated, one possible reason is the two free blocks do not combine, another possible reason is memory leak.

7) Test #7
    a. Initial state: 1 0x8000 block is free.
    b. Actions: Ask for size of (0x8000+4) memory, which exceed the maximum available memory size.
    c. Main idea: Allocation should fail to allocate a (0x8000+4) memory because this memory size exceeds the maximum available memory size.
    d. Expected final state: 1 0x8000 block is free.

   e. Test purposes: This case tests whether there is extra memory exist. I tried to allocate a memory larger than the maximum block size limit. If it cannot succeed, it means that there is no extra memory.

8) Test #8
   a. Initial state: 1 0x8000 block is free.
   b. Actions: Ask for size of (0x4000+4) memory, appropriate size is 0x8000. (No free block left)
   c. Main idea: Allocation should allocate the whole 0x8000 memory block.
   d. Expected final state: no free block.
   e. Test purposes: This case tests the root level allocation. Also, this case tests whether the small pieces free blocks will combine to a whole one.

9) Test #9
   a. Initial state: no free block.
   b. Actions: Deallocate the 0x8000 block and ask for size of 0x8000 memory.
   c. Main idea: After deallocating 0x8000 block, another 0x8000 memory should be allocated successfully since there is enough memory.
   d. Expected final state: no free block.
   e. Test purposes: I tried to repeatedly allocate and deallocate the whole memory block to make sure there is no memory leak.

If all test cases are pass, it should print:

```
G12-TS2: 9/9 tests PASSED
G12-TS2: 0/9 tests FAILED
G12-TS2: END
```