350 Lab 1 Report

Description of Data Structures and Algorithms

The algorithm works by representing the memory storage as linked list. The purpose of the linked list is to allow persistent storage of information such as how large in use memory regions are and whether they are free, or not.

This allows us to implement a first fit strategy where the first free memory region with large enough size to accommodate the request found in the linked list will be returned to the user.

A malloc call will traverse the linked list, finding the first suitable node. Once found, one of two cases will occur:

1. The node's size can accomodate the users request just barely and thus the entire region is returned to the user. This does not create any new nodes.
2. The node's size can accomodate the users request while leaving space for a new node to be created. This will create a new node, effectively splitting the region such that a region just large enough for the user is returned, while the rest of the region is still marked free in another node.

For example, calling

1. Malloc(20) -> p0
2. Malloc(20) -> p1
3. Malloc(20) -> p2

will result in our memory layout looking like

| (node) (20 bytes) | (node) (20 bytes) | node (20 bytes) | node (free region) |

Dealloc works by traversing the region looking for the requested pointer to free. Dealloc will then mark the region as free, and then look for neighbouring nodes. If other neighbouring nodes are free, it will defragment the memory layout by merging the two consecutive free nodes into a singular free node, with size equal to the sum of the two nodes.

Calling: Dealloc(p1) will result in
| (node) (20 bytes) | (node) (free) | node (20 bytes) | node (free region) |

Calling Dealloc(p2) will result in
| (node) (20 bytes) | (node) (free) |

And calling Dealloc(p0) will result in
| (node) (free) |

Testing Scenario Descriptions

Coalescing free regions

- We test for the scenario where after a dealloc, adjacent free regions are merged together.
- Example:
- Call malloc 3 times to create 3 used regions, and 1 free region
  `used, used, used, free`
  - 4 regions total
- Dealloc the 2nd region to create:
  `used, free, used, free`
  - 4 regions total
- Dealloc the first region to trigger the coalescing of the two adjacent free regions
  `free, free, used, free`
  - the first two regions are free and adjacent, therefore they are merged
    `free, used, free`
  - 3 regions total
- Code sample for coalescing free regions test case:

```
if (countNodes()==1){
        result |= BIT(0);
}
p[0] = mem_alloc(12);
p[1] = mem_alloc(24);
p[2] = mem_alloc(48);

if (countNodes()==4){
        result |= BIT(1);
}

mem_dealloc(p[1]);
if (countNodes()==4){
        result |= BIT(2);
}

mem_dealloc(p[0]);
if (countNodes()==3){
        result |= BIT(3);
}

mem_dealloc(p[2]);
if (countNodes()==1){
        result |= BIT(4);
}```
```

Reusing newly-freed regions

- These test cases call first call malloc multiple times, then dealloc, and malloc regions of different size in order to test the different scenarios.

- Reusing a newly-freed region with no new node created
    - Example 1:
        - This example checks the scenario where the malloc size is less than or equal to the free node region size. However, no new node is created since the free node region size is less than the sum of the malloc size and node struct size.
        - malloc(12), malloc(24), malloc(48)
          `| (node) (12 bytes) | (node) (24 bytes) | node (48 bytes) | node (FREE region) |`
        - dealloc(p0)
          `| (node) (FREE) | (node) (24 bytes) | node (48 bytes) | node (FREE region) |`

        - malloc(12) will allocate 12 bytes in the first region, no new node is necessary
          `| (node) (12 bytes) | (node) (24 bytes) | node (48 bytes) | node (FREE region) |`

- Reusing a newly-freed region requiring a new node be created
    - Example 2:
        - This example checks the scenario where the free node region size is larger than or equal to the sum of the malloc size and size of the node struct. This means a new node will be created.
        - malloc(12), malloc(60), malloc(48)
          `| (node) (12 bytes) | (node) (60 bytes) | node (48 bytes) | node (FREE region) |`
        - dealloc(p1)
          `| (node) (12 bytes) | (node) (FREE) | node (48 bytes) | node (FREE region) |`
        - malloc(12) will take the 60 bytes free region and allocate the first twelve bytes to be used. The remaining region will be occupied by a new node with a free space of 36 bytes.
          `| (node) (12 bytes) | (node) (12 bytes) | (node) (FREE) | node (48 bytes) | node (FREE region) |`
        - dealloc(p3)
          `| (node) (12 bytes) | (node) (60 bytes) | node (48 bytes) | node (FREE region) |`

Not leaking memory (size of heap should not increase or decrease)

- We check for memory leaks by calling the `memLeakCheck()` function
- The `memLeakCheck()` function parses the linked list and compares the size of the original heap to the size of the memory regions and the node structs. This checks that the size of the heap remains unchanged.

```
int memLeakCheck(){
    unsigned int howMuchMem = 0;
    Node* curNode = HEAD;
    while(curNode != NULL){
        howMuchMem += curNode->size + sizeof(Node);
        curNode = curNode->next;
    }
    unsigned int end_addr = (unsigned int) &Image$ZI_DATA$ZI$Limit;
    unsigned int totalSize = 0xBFFFFFFF - end_addr;
    return howMuchMem == totalSize;
}
```

- Each integration test will deallocate any remaining used regions and check for memory leaks by calling `memLeakCheck()`
- A basic test for memory leaks involves checking the heap size before alloc, after alloc, and after dealloc. In all cases, the heap size should remain the same.

```
int test_mem_leak(){
    void *p[4];
    U32 result = 0;
    if(memLeakCheck()==1){
        result |= BIT(0);
    }

    p[0] = mem_alloc(12);

    if(memLeakCheck()==1){
        result |= BIT(1);
    }

    mem_dealloc(p[0]);

    if(memLeakCheck()==1){
        result |= BIT(2);
    }

    return result == 7;
}
```

Utilizing memory with minimum overheads

- We malloc random numbers until we max out the heap to check how much unallocated space is leftover.

Returning correct number of externally-fragmented regions

- We test the `mem_count_extfrag()` function by allocating memory several times, and deallocating memory to create pockets of free regions. We then call `mem_count_extfrag()` to check whether the number of free regions of size X are as expected.
- Example sample code:

```
    p[0] = mem_alloc(12);
    p[1] = mem_alloc(12);
    p[2] = mem_alloc(12);
    p[3] = mem_alloc(12);
    p[4] = mem_alloc(16);
    p[5] = mem_alloc(12);
    p[6] = mem_alloc(18);

    if (countNodes() == 8){
    result |= BIT(1);
}

    mem_dealloc(p[2]);
    p[7] = mem_alloc(15);

    if (countNodes() == 9){
    result |= BIT(2);
}

    mem_dealloc(p[4]);
    p[8] = mem_alloc(18);

    print_list();

    if (countNodes() == 10){
    result |= BIT(3);
}

    if (mem_count_extfrag(12+12) == 0){
            result |= BIT(4);
    }

    if (mem_count_extfrag(13+12) == 1){
            result |= BIT(5);
    }

    if (mem_count_extfrag(17+12) == 2){
            result |= BIT(6);
    }
```

Testing Results

Throughput:

200 runs per test

- test_mem_leak:
  - 5 seconds for 200 tests
  - 2 alloc/dealloc for 1 test
  - 80.00 inst/sec
- test_coales:
  - 17 seconds for 200 tests
  - 6 alloc/dealloc for 1 test
  - 70.59 inst/sec
- test_reuse_freed:
  - 23 seconds for 200 tests
  - 8 alloc/dealloc for 1 test
  - 69.57 inst/sec
- test_reuse_freed_2:
  - 23 seconds for 200 tests
  - 8 alloc/dealloc for 1 test
  - 69.57 inst/sec
- test_malloc_new_node:
  - 23 seconds for 200 tests
  - 8 alloc/dealloc for 1 test
  - 69.57 inst/sec
- test_extfrag:
  - 62 seconds for 200 tests
  - 18 alloc/dealloc for 1 test
  - 58.06 inst/sec

ONLY RAN ONCE

- test_utilization:
  - 15 seconds for 1 test
  - 1020 alloc/dealloc for 1 test
  - 68 inst/sec

Heap Utilization at peak

Heap size given our OS image: 1070585247

- test_mem_leak:
  - mem_alloc peak bytes: 12
  - 1.12E-6 % utilization
- test_coales:
  - mem_alloc peak bytes: 84
  - 7.85E-6 % utilization
- test_reuse_freed:
  - mem_alloc peak bytes: 84
  - 7.85E-6 % utilization
- test_reuse_freed_2:
  - mem_alloc peak bytes: 84
  - 7.85E-6 % utilization
- test_malloc_new_node:
  - mem_alloc peak bytes: 120
  - 1.12E-7 % utilization
- test_extfrag:
  - mem_alloc peak bytes: 102
  - 9.53E-6 % utilization
- test_utilization:
  - mem_alloc peak bytes: 1069547520
  - 99.57% utilization