시스템 프로그래밍 과제 3 malloclab 레포트

Malloc Lab: Writing a Dynamic Storage Allocator

2019-13674
양현서
바이오시스템소재학부

Contents

1 Introduction

수업 시간에 virtual memory와 dynamic memory allocation에 대해 배웠다. 효율적으로 dynamic memory 할당과 해제를 관리하려면 여러 가지 연구가 필요한데, 이번 랩에서는 이러한 연구를 하여 `mm_malloc`과 `mm_free`, `mm_realloc`을 효율적으로 처리할 수 있는 라이브러리를 직접 만들어 보았다.

2 Step 1: 구상

동적 메모리 할당을 구현할 때 고려해야 할 것은 힙의 사용량과 할당 속도이다. 이것의 조화를 잘 맞추어야 한다. 힙의 사용 효율을 최대로 늘리기 위해서는 할당받고자 하는 메모리의 크기에 가장 잘 들어맞는 빈 공간을 찾아 할당해 주는 것이 이상적이다. 또한 할당 속도를 늘리기 위해서는 최대한 먼저 발견하는 공간을 할당해 주거나, 빈 공간의 탐색 알고리즘 효율이 좋아야 한다.

2.1 트리를 이용한 구현

그래서 처음에는 항상 정렬되어 있고 탐색 속도도 $O(\log n)$ 정도인 이진 탐색 트리를 이용하고자 마음먹었다. 이진 탐색 트리는 항상 정렬된 순서로 정보를 저장할 수 있다는 장점이 있어 이진 탐색 트리를 사용하고자 하였는데, 항상 balance를 적절히 유지해 주는 red-black tree는 구현의 복잡성을 우려하여 사용하지 않았고, 보통 이진 탐색 트리를 이용하여 구현하기로 하였다. 기본 생각은 할당받고자 하는 메모리의 크기를 갖는 비할당 노드를 빠르게 찾는 것이다. 따라서 할당받고자 하는, 또는 할당받은 메모리의 크기를 key로 하는 이진 탐색 트리를 만들기로 하였다.

2.2 리스트를 이용한 구현

트리를 이용한 구현은 생각보다 buggy하여 segmentation fault가 많이 발생하였다. 많은 할당과 해제를 반복하는 경우 트리의 balance가 깨져 성능 저하가 발생하는데 위 구현에서는 처리를 해주지 않았으므로, 구현이 간단하고 버그 해결이 쉬운 리스트 방식으로 해도 별 차이가 없을 것이라 생각하였다.

기본적으로 first fit을 이용하여 탐색을 하게 되는데, 작은 메모리 공간을 할당 받을 때 큰 빈 공간을 활용하게 되는 경우, 남는 공간을 할당 가능한 노드로 표시함으로써 internal fragmentation을 막고자 했다.

3 Step 2: 트리를 이용한 구현

트리를 구현하는 방법은 여러 가지가 있지만, 이번에는 포인터를 이용하여 구현하였다.

```
35 typedef struct memhdr_tree_node_t{
36         size_t size;
37         struct memhdr_tree_node_t *parent;
38         struct memhdr_tree_node_t *left;
39         struct memhdr_tree_node_t *right;
40 } __attribute__((aligned(ALIGNMENT))) memhdr_tree_node;
```

메모리 블록의 사이즈와 할당 여부를 표시하는 size 변수와, left와 right, 그리고 parent 변수가 있다.

3.1 malloc의 구현

`malloc`에서 트리 구조를 이용하여 메모리를 관리하기로 하였으므로, 이것을 적극 이용한다. malloc의 실제 구현은 `find_fit`에서 담당한다. 트리의 적절한 위치에 메모리 블록을 만들기 위해, 재귀적 구조를 사용했다.

```
94  static char * find_fit(memhdr_tree_node ** parent, size_t size) {
95          //printf("%p\n", *parent);
96          // printf("Find_fit %d\n", size);
97          size_t total_size = MAKE_TOTAL_SIZE(size);
98          // printf("Total_size %d\n", total_size);
99          if(*parent == NULL) { // new node created
100                 *parent = (memhdr_tree_node *) mem_sbrk(total_size);
101                 if(*parent == (void *)-1)
102                         return NULL;
103                 (*parent) -> size = total_size | 0x1; // Allocated
104                 (*parent) -> left = NULL;
105                 (*parent) -> right = NULL;
106 //                PHS_NEXT_HDR(*parent)-> size=0;
107                 writeFooter(*parent);
108                 return HDR2PTR(*parent);
```

`*parent` 가 `NULL`일 경우는 이 노드가 새로이 생성되는 경우이므로 새로 메모리를 할당한 후, 자식들을 초기화한다. 이 노드의 parent 변수 설정은 호출한 측에서 한다.

```
109         } else if(!GET_ALLOC(*parent)) {
110                 if(GET_SIZE(*parent) >= total_size) {
111                         (*parent) -> size = total_size | 0x1;
112                         writeFooter(*parent);
113                         return HDR2PTR(*parent);
114                 }  else {
115
116                         //free, but not enough space
117                 }
118         }
```

그렇지 않고 해당 노드가 할당되지 않은 상태일 경우 해당 노드에 할당 표시를 하고, 리턴한다.

```
119         // already allocated
120         char * result; // = *parent; // test for *
121         // MAGIC
122         //(*parent)->left = NULL;
123         //(*parent)->right = NULL;
124         if((*parent)->right &&  GET_SIZE((*parent)->right) == 0) {
```

```
125                 (*parent)-> right = NULL;
126         }
127         if((*parent)->left && GET_SIZE((*parent)->left) == 0) {
128                 (*parent)-> left = NULL;
129         }
130         if(total_size > GET_SIZE(*parent)) {
131                 result = find_fit(&((*parent)->right), size); // allocate new and
                    ↪  set as right if this node did not have data
132         } else {
133                 result = find_fit(&((*parent)->left), size);
134         }
135         writeFooter(*parent);
136         //HDRP(result) -> left = NULL;
137         //HDRP(result) -> right = NULL;
138         return result;
```

해당 노드가 비어 있지 않다면, 그 노드의 자식 노드들에게 이 할당을 맡긴다. 만약 해당 자식이 NULL일 경우 그 자식이 새로운 노드가 되고, 그렇지 않을 경우 새로운 노드는 그 자식의 자식이 될 것이다.
보고서를 쓰며 생각을 해 보니 할당되지 않은 노드를 찾은 경우 바로 그곳에 메모리를 할당받기보다는, 그 노드의 자식들이 NULL이 아닌지 검사하고, 더 tight한 자식의 노드에 메모리를 할당받는 편이 더 효율적이었을 것 같다는 생각이 든다.

3.2 free의 구현

```
359 void mm_free(void *ptr)
360 {
361         memhdr_tree_node * header = HDRP(ptr);
362         printf("Free %p hdr %p\n", ptr, header);
363         SET_FREE(header);
364         writeFooter(header);
365         // Coerce right
366         //memhdr_tree_node * phy_next = PHS_NEXT_HDR(header);
367         right_coerce(header);
```

free block이라고 표시하고, 그 블록의 다음 블록들과 coalescing을 시도한다.

3.3 realloc의 구현

```
399 void *mm_realloc(void *ptr, size_t size)
400 {
401 //        printf("realloc");
402         if(ptr == NULL)
403                 return mm_malloc(size);
404         if(size == 0) {
405                 mm_free(ptr);
406                 return NULL;
407         }
408
409     void *oldptr = ptr;
410     void *newptr;
411
412     size_t oldSize = GET_SIZE(HDRP(ptr));
413     size_t newSize = MAKE_TOTAL_SIZE(size);
414         size_t copySize;
415
416     if(oldSize >= newSize) {
417             HDRP(ptr)->size = newSize | 0x1;
418             writeFooter(HDRP(ptr));
419                 return ptr;
420         }
```

```
421
422     right_coerce(HDRP(ptr));
423     if(GET_SIZE(HDRP(ptr)) >= newSize) {
424             printf("Coerce success : %d\n", newSize - GET_SIZE(HDRP(ptr)));
425             HDRP(ptr)->size = newSize | 0x1;
426             writeFooter(HDRP(ptr));
427                 return ptr;
428         }
429     newptr = mm_malloc(size);
430     if (newptr == NULL)
431       return NULL;
432     copySize = GET_SIZE(HDRP(ptr)) - 2* sizeof(memhdr_tree_node);//*(size_t
        ↪  *)((char *)oldptr - 2*sizeof(memhdr_tree_node));
433     if (size < copySize)
434       copySize = size;
435     if(newptr != oldptr)
436             memcpy(newptr, oldptr, copySize);
437     mm_free(oldptr);
438     return newptr;
439 }
```

realloc의 내용은 복잡한 내용 없이 해당 노드를 해제하고 새로 메모리를 할당받되, NULL을 realloc 하거나 새로운 size가 0인 상황만 따로 처리해 준다. 또한 기존에 할당한 크기가 새로운 크기보다 이미 큰 경우, 헤더에 사이즈를 다시 표시한 후 리턴한다. 만약 적절한 공간을 찾지 못했을 경우, malloc을 이용하여 새로운 공간을 할당받은 후, 데이터를 복사하고, 기존 포인터를 free 하고 리턴하게 된다.

3.4 coalescing의 구현

```
102 void right_coerce(memhdr_tree_node * hdr) {
103         printf("right_coerce: %p\n", hdr);
104 //        printf("Coerce before: %d\n", GET_SIZE(hdr));
105         memhdr_tree_node * nextNode =  PHS_NEXT_HDR(hdr);
106         // no next node
107         if(!nextNode)
108                 return;
109         // Cannot coerce if reserved
110         if(GET_ALLOC(nextNode))
111                 return;
112         // while nextnode is  free
113         while(nextNode && !GET_ALLOC(nextNode)) {
114                 hdr->size = GET_SIZE(hdr) + GET_SIZE(nextNode); //coerce blocks
115                 nextNode -> size = 1; // mark as allocated
116                 // update parent & l & r
117                 int isInLeft = (nextNode -> parent -> left) == nextNode;
118                 if(nextNode -> left && nextNode -> right) {
119                         if(isInLeft) {
120                                 nextNode->parent->left = nextNode -> left;
121                                 nextNode->parent->left->right = nextNode->right;
122                                 nextNode->left->parent = nextNode->parent;
123                                 nextNode->right->parent = nextNode->parent->left;
124
125                         } else {
126                                 nextNode->parent->right = nextNode -> left;
127                                 nextNode->parent->right->right = nextNode->right;
128                                 nextNode->left->parent = nextNode -> parent;
129                                 nextNode->right->parent = nextNode->
                                    ↪  parent->right;
130                         }
131                 } else if(nextNode -> left) {
132                         if(isInLeft) {
133                                 nextNode->parent->left = nextNode->left;
```

```
                                nextNode->left->parent = nextNode->parent;
                        } else {
                                nextNode->parent->right = nextNode -> left;
                                nextNode->left->parent = nextNode->parent;
                        }
                } else if(nextNode -> right) {
                        if(isInLeft) {
                                nextNode->parent->left = nextNode->right;
                                nextNode->right->parent = nextNode->parent;
                        } else {
                                nextNode->parent->right = nextNode -> right;
                                nextNode->right->parent = nextNode->parent;
                        }
                } else {

                }
                // deleteNode(nextNode);
                writeFooter(nextNode->parent);
                writeFooter(nextNode);
                nextNode = PHS_NEXT_HDR(nextNode);
        }
//        printf("Coerce result: %d\n", GET_SIZE(hdr));
        writeFooter(hdr);
        printf("Coerce finished\n");
}
```

기본적으로 다음 메모리 블록을 검사하여 free 상태라면 앞의 메모리 블록과 합치고, 해당하는 노드들의 포인터들을 전부 제대로 업데이트해준다.

3.5 mm_check의 구현

과제의 handout에서 mm_check를 이용하면 디버깅이 편리하다고 쓰여 있는 것을 발견하였다. 따라서 mm_check함수를 구현하였다. mm_check_sub에서 루트 이외의 노드들을 검사하고, 이 노드들의 메모리 주소와 left, right, parent의 유효성, footer의 유효성 등을 판단한다. 중간에 노드들에 대한 자세한 정보들을 출력하여 버그를 잡기 쉽게 만들어 두었다.

```
int mm_check() {
        int err = 0;
        if(root == NULL) {
                printf("Root is NULL\n");
                return 0;
        }
        if(root > mem_heap_hi() || root < mem_heap_lo()) {
                printf("Root out of range :%p\n", root);
                return 0;
        }
        if(root->left != NULL) {
                if(!mm_check_sub(root->left)) {
                        printf("Check for left node %p failed\n", root->left);
                        err = 1;
                }
        }
        if(root->right != NULL) {
                if(!mm_check_sub(root->right)) {
                        printf("Check for right node %p failed\n", root->right);
                        err =1;
                }
        }
        if(root->parent!=NULL && root->parent != root ) {
                printf("Root parent is wrong: %p, root: %p\n", root->parent,
                ↪  root);
```

```
465                 err = 1;
466         }
467         if(err)
468                 return 0;
469         return 1;
470 }
471
472 int mm_check_sub(memhdr_tree_node *node) {
473         int err = 0;
474         printf("Checking node %p ===============\n", node);
475         printf("Heap lo: %p, Heap hi: %p\n", mem_heap_lo(), mem_heap_hi());
476         if(node == NULL) {
477                 printf("Checker is 0: node is NULL\n");
478                 return 0;
479         }
480         if(node > mem_heap_hi() || node < mem_heap_lo()) {
481                 printf("Node out of range: %p\n");
482                 return 0;
483         }
484         if(GET_ALLOC(node)) {
485                 printf("The node %p is allocated: size = %d\n", node
                    ↪  ,GET_SIZE(node));
486         } else {
487                 printf("The node %p is not allocated: size = %d\n", node,
                    ↪  GET_SIZE(node));
488         }
489
490         if(node->parent == NULL) {
491                 printf("parent of %p is NULL\n", node);
492                 return 0;
493         }
494         if(node->parent > mem_heap_hi() || node->parent < mem_heap_lo()) {
495                 printf("invalid parent %p for node %p\n", node->parent, node);
496                 return 0;
497         }
498         if(node->parent->left== node) {
499                 printf("The node %p is a left node of parent %p\n", node,
                    ↪  node->parent);
500         }
501         if(node->parent->right==node) {
502                 printf("The node %p is a right node of parent %p\n", node,
                    ↪  node->parent);
503         }
504
505         if(node->left != NULL) {
506                 if(node->left > mem_heap_hi() || node->left < mem_heap_lo()) {
507                         printf("The node %p has wrong childs: left: %p right:
                            ↪  %p\n", node, node->left, node->right);
508                         return 0;
509                 }
510                 printf("The node %p has left: %p, size = %d\n", node, node->left,
                    ↪  GET_SIZE(node->left));
511                 if(!mm_check_sub(node->left)) {
512                         printf("Check for left node %p failed\n", node->left);
513                         err = 1;
514                 }
515         }
516         if(node->right != NULL) {
517                 if(node->right > mem_heap_hi() || node->right < mem_heap_lo()) {
```

```
518                         printf("The node %p has wrong childs: left: %p right:
                             ↪  %p\n", node, node->left, node->right);
519                         return 0;
520                 }
521                 printf("The node %p has right: %p, size = %d\n", node,
                     ↪  node->right, GET_SIZE(node->right));
522                 if(!mm_check_sub(node->right)) {
523                         printf("Check for right node %p failed\n", node->right);
524                         err = 1;
525                 }
526         }
527         memhdr_tree_node * footer = HDR2FTR(node);
528         if(memcmp(node, footer, sizeof(memhdr_tree_node))) {
529                 printf("node %p and footer %p differ\n", node, footer);
530                 for(int i=0;i<sizeof(memhdr_tree_node); i++) {
531                         if(((char*)node)[i] != ((char*)footer)[i]) {
532                                 printf("node[%d]:%x footer[%d]:%x\n", i,
                                     ↪  (int)((char*)node)[i] , i,
                                     ↪  (int)((char*)footer)[i]);
533                         }
534                 }
535                 return 0;
536         }
537
538         printf("=====End for node %p======\n", node);
539         if(err)
540                 return 0;
541         return 1;
542 }
```

4 Step 3: 리스트를 이용한 단순한 구현

위와 같이 Step 2에서 구현한 방법은 높은 효율을 보여주었지만, 자동으로 균형을 맞추는 기능이 없기 때문에 사용할수록 탐색 효율이 $O(n)$으로 가는 것을 알 수 있었다. 게다가 이 구현은 은근히 복잡하고 탐색을 포인터에 의존하기 때문에 invalid한 포인터를 만나면 걷잡을 수 없이 버그가 생기게 된다. 이러한 버그를 찾기 위해 백방으로 노력해 보았지만, 아무리 찾아봐도 왜 segmentation fault가 나는지를 알아낼 수가 없어서, 하는 수 없이 구현도 간단하고 관리도 쉬운 리스트 방식으로 구현하기로 마음먹었다. Step 2의 트리 구현은 한 노드당 3개의 포인터를 이용하는 등 여러 정보를 담고 있어서 `struct`를 이용하였지만, 이번 구현은 할당된 크기와 할당 여부를 나타내는 `size_t`형 변수 하나만 이용하면 되어서 `struct`를 이용할 필요가 없었다. 대신 매크로 함수를 많이 이용하였다.

4.1 매크로 함수들

```
27  #define ALIGNMENT 8
28
29  /* rounds up to the nearest multiple of ALIGNMENT */
30  #define ALIGN(size) (((size) + (ALIGNMENT-1)) & ~0x7)
31
32
33  #define SIZE_T_SIZE (ALIGN(sizeof(size_t)))
34
35  #define MAKE_SIZE(size) (ALIGN(SIZE_T_SIZE*2+size))
36
37  #define GET(p) (*(unsigned int *)(p))
38  #define SET(p, value) (*(unsigned int*)(p) = value)
39  #define HDRP(p) (p-SIZE_T_SIZE)
40  #define GET_ALLOC(hdr) (GET(hdr) & 0x7)
41  #define SET_ALLOC(hdr) (*(unsigned int *)hdr |= 0x1)
42
43  #define GET_SIZE(hdr) (GET(hdr) & ~0x7)
```

```
#define FTRP(hdr) (hdr+GET_SIZE(hdr)-SIZE_T_SIZE)
#define NEXT_HDR(hdr) (hdr+GET_SIZE(hdr))
#define PREV_HDR(hdr) (hdr-GET_SIZE(hdr-SIZE_T_SIZE))
#define HDR2PTR(hdr) (hdr+SIZE_T_SIZE)
#define HDR2FTR(hdr) (hdr+ GET_SIZE_FROM_HDR(hdr) - SIZE_T_SIZE)
#define SET_SIZE(hdr, size) SET(hdr, (size) & ~0x7)
#define SET_FREE(hdr) (*(unsigned int *) hdr &= ~0x7)
```

이 라이브러리의 메모리 정렬은 8로 한다. 이를 통하여 하위 3비트를 할당 여부를 나타내는 비트 필드로 이용할 수 있다. 즉, 헤더는 size_t의 크기의 작은 변수인 것이다. 또한 size는 그 메모리 블록의 헤더와 footer를 포함한 전체 크기를 나타내도록 정하였다. 이 규칙을 가지고 매크로 함수들을 만들었다. 매크로 함수의 매개변수 이름을 p와 hdr를 혼용하여 버그가 자꾸 발생하였기 때문에, 대부분의 매크로의 입력 매개변수를 header의 포인터로 정하고, 그에 맞추어 작성하였다.

```
static void writeFooter(char * header) {
        char * ftrp = FTRP(header);
        memcpy(ftrp, header, SIZE_T_SIZE);
}
```

writeFooter 함수는 헤더의 내용을 footer에 복사해주는 함수이다.

4.2 malloc의 구현

뼈대 코드의 malloc은 단순히 mem_sbrk함수를 호출하여 힙 크기를 늘리고, 그 공간을 리턴하는 구현이었다. 그런데 이렇게 하면 free 된 공간을 이용할 수 없기 때문에, 힙의 맨 앞에서부터 하나하나 블록을 탐색하는 것보다 공간 효율이 떨어질 수밖에 없다. 따라서 앞에서부터 블록을 탐색하다 필요 크기보다 크기가 크거나 같은 빈 블록을 만나면, 그 자리에 할당하게 하였다. 그런데 만약 필요 크기보다 지나치게 큰 빈 블록을 발견하여 전부를 사용하게 될 경우, 불필요하게 많은 공간을 점유하게 된다. 따라서 그러한 블록을 발견하여 할당받을 경우, 필요량 이상의 나머지 블록 공간은 새로운 빈 블록으로 할당하여 다음 malloc시 이용할 수 있게 처리하였다.

```
    if(search_start <= (void *)0) {
//            printf("Initializing search_start\n");
            search_start = mem_sbrk(newSize);
            if (search_start == (void *)-1)
                    return NULL;
                SET(search_start, newSize);
                SET_ALLOC(search_start);
                writeFooter(search_start);
//                printf("Initialized search_start: %p with size %d\n",
    ↪  search_start, size);
                return HDR2PTR(search_start);
        }
```

처음 할당 시 메모리 블록의 맨 처음을 초기화하는 코드이다.

```
        char * iterator = search_start;
        char * max_heap = mem_heap_hi();
        while(iterator <= max_heap-SIZE_T_SIZE*2 - newSize) {
                if(!GET_ALLOC(iterator)) {
                        size_t freeSize = GET_SIZE(iterator);
                        int leftSize = freeSize - newSize;
                        if(leftSize >= 0) {
//                                printf("Found free space from %p to %p size %d,
    ↪  and %d needed\n", iterator, iterator+freeSize, freeSize,  newSize);
                                if(leftSize > 2 * SIZE_T_SIZE) { // allocate
                                ↪  leftover as free
//                                        printf("Allocated leftover: %d \n",
    ↪  newSize);
                                        SET(iterator, newSize);
                                        SET_ALLOC(iterator);
                                        writeFooter(iterator);
```

```
95                                          char * nextHdr = NEXT_HDR(iterator);
96                                          SET(nextHdr, leftSize);
97                                          writeFooter(nextHdr);
98                                  } else {
99  //                                        printf("Allocated the full space\n");
100                                         SET(iterator, freeSize);
101                                         SET_ALLOC(iterator);
102                                         writeFooter(iterator);
103                                 }
104                                 return HDR2PTR(iterator);
105                         }
106                 }
107                 iterator = NEXT_HDR(iterator);
108         }
```

적당한 메모리 블록을 탐색하여 할당하는 코드이다.

```
109         iterator = mem_sbrk(newSize);
110 //        printf("Could not find free space. Created: %p\n", iterator);
111     if (iterator == (void *)-1)
112                 return NULL;
113         SET(iterator, newSize);
114         SET_ALLOC(iterator);
115         writeFooter(iterator);
116         return HDR2PTR(iterator);
```

만약 앞에서 빈 공간을 찾지 못하였다면 힙의 새로운 공간을 할당받는다.

4.3 free의 구현

free를 구현하지 않으면, 실제로는 사용되지 않는 메모리가 사용중이라고 인식되어 빈공간을 찾는 데 방해가 된다. 따라서 free함수가 호출되었을 때, 해당하는 메모리 블록의 할당 여부에 0을 넣음으로써 그 메모리 블록이 비어 있음을 표시한다. 이와 더불어 coalescing을 수행하여 주변 빈 메모리 블록과 합침으로써 연속된 빈 공간을 만드는데 기여하게 하였다.

```
168 void mm_free(void *ptr)
169 {
170 //        printf("Free %p called \n", ptr );
171         SET_FREE(HDRP(ptr));
172         writeFooter(HDRP(ptr));
173         coalesce_right(HDRP(ptr));
174         coalesce_left(HDRP(ptr));
175 //        printf("Free finished\n");
176 }
```

해당 메모리 블록을 free로 마크하고 left coalesce와 right coalesce를 수행한다.

4.4 realloc의 구현

realloc의 기존 구현은 malloc후 해당 메모리를 복사하고 free하는 내용이었다. 이렇게 하면 기존에 할당해 둔 메모리의 이점을 무시하고 바로 처음부터 다시 빈 공간을 탐색하게 되는 것이므로 효율이 떨어질 수 있다. 그러므로 right coalescing를 시행하여 해당 메모리 블록을 키워 보고, 원하는 만큼 키워졌을 시 정보를 업데이트하고 리턴한다. 만약 right coalescing를 한 후에도 크기가 충분하지 않다면 left coalescing도 수행하여 다시 한번 시도해본다. 충분한 크기의 메모리 블록이 생성되었다면 해당 메모리 블록으로 기존 메모리 내용을 복사한 후, 리턴한다. 만약 앞의 과정에서 충분한 메모리 블록을 할당받지 못하였다면, malloc을 호출하여 새로운 공간에서 메모리를 할당받고, 원래 데이터를 복사한 뒤 리턴하게 구현하였다.

```
181 void *mm_realloc(void *ptr, size_t size)
182 {
183 //        printf("Realloc %p size %d\n", ptr, size);
184     void *oldptr = ptr;
```

```
185     void *newptr;
186     size_t copySize;
187     char * hdr = HDRP(ptr);
188     int leftSize= 0;
189 
190     size_t realSize=  MAKE_SIZE(size);
191 
192         size_t curSize = GET_SIZE(hdr);
193         leftSize = curSize - realSize;
194         if(leftSize >=0) {
195 //                printf("First try tried %p\n", ptr);
196                 if(leftSize >= 2*SIZE_T_SIZE) { // use left space
197 //                        printf("First try success %p\n", ptr);
198                         SET(hdr, realSize);
199                         SET_ALLOC(hdr);
200                         writeFooter(hdr);
201                         char * nextHdr = NEXT_HDR(hdr);
202                         SET(nextHdr, leftSize);
203                         writeFooter(nextHdr);
204                         return ptr;
205                 } else {
206 //                        printf("First try no leftover %p\n", ptr);
207                         return ptr; // Nothing to do
208                 }
209         }
210 
211         coalesce_right(hdr);
212 
213     curSize = GET_SIZE(hdr);
214     leftSize= curSize - realSize;
215     if(leftSize >= 0){
216             if(leftSize >= 2*SIZE_T_SIZE) {
217                     SET(hdr, realSize);
218                         SET_ALLOC(hdr);
219                         writeFooter(hdr);
220                         char * nextHdr = NEXT_HDR(hdr);
221                         SET(nextHdr, leftSize);
222                         writeFooter(nextHdr);
223                         return ptr;
224                 } else {
225                         return ptr;
226                 }
227         }
228 
229         copySize = GET_SIZE(hdr) - 2*SIZE_T_SIZE;
230         if(copySize > size)
231                 copySize = size;
232 
233         newptr = coalesce_left(hdr);
234 
235         if(newptr != hdr) {
236                 size_t newSize = GET_SIZE(newptr);
237                 leftSize = newSize - realSize;
238                 if(leftSize >= 0){
239                     if(leftSize >= 2*SIZE_T_SIZE) {
240                             SET(newptr, realSize);
241                                 SET_ALLOC(newptr);
242                                 writeFooter(newptr);
243                                 memmove(HDR2PTR(newptr), ptr, copySize);
244                                 char * nextHdr = NEXT_HDR(newptr);
```

```
245                                     SET(nextHdr, leftSize);
246                                     writeFooter(nextHdr);
247                                     return HDR2PTR(newptr);
248                             } else {
249                                     SET(newptr, newSize);
250                                     SET_ALLOC(newptr);
251                                     writeFooter(newptr);
252                                     memmove(HDR2PTR(newptr), ptr, copySize);
253                                     return HDR2PTR(newptr);
254                             }
255                     }
256             }
257
258         newptr = mm_malloc(size);
259         if (newptr == NULL)
260           return NULL;
261 //      copySize = *(size_t *)((char *)oldptr - SIZE_T_SIZE);
262 //      if (size < copySize)
263 //        copySize = size;
264         memcpy(newptr, oldptr, copySize);
265         mm_free(oldptr);
266         return newptr;
267 }
```

4.5 coalescing의 구현

coalescing을 시행하면, 인접한 메모리 블록들이 서로 합쳐져, 빠른 시간 내에 충분히 큰 메모리 블록을 찾는 데 도움을 줄 것이다. 따라서 free를 할 때 주변 빈 블록들을 탐색하여 연속된 빈 공간을 만드는데 기여하기 위하여 coalesce기능을 구현하였다. 우선 `coalece_right`의 구현이 `coalesce_left`의 구현보다 체감상 쉬웠다. 계속 물리적으로 인접한 블록들을 하나씩 방문하여 할당된 블록을 발견하기 전까지 최초의 블록의 사이즈에 해당 블록의 사이즈를 더하고, footer를 적당히 업데이트 해주면 되기 때문이었다.

```
119 void coalesce_right(char * header) {
120 //        printf("Coalesce right %p\n", header);
121         int alloc = GET_ALLOC(header);
122         char * max_heap = mem_heap_hi();
123         char * iterator = NEXT_HDR(header);
124         while(iterator<=max_heap-SIZE_T_SIZE*2 && !GET_ALLOC(iterator)) {
125                 size_t size = GET_SIZE(iterator);
126                 SET_SIZE(header, GET_SIZE(header) + size);
127                 if(alloc)
128                         SET_ALLOC(header);
129                 iterator = NEXT_HDR(iterator);
130 //                printf("right Iterator: %p\n", iterator);
131         }
132         writeFooter(header);
133 //        printf("Coerce right finished\n");
134 }
```

반면 `coalesce_left`를 구현할 때는 크기를 저장하는 헤더가 계속 앞으로 이동하기 때문에 그것을 추적하는 것이 약간 까다로웠다.

여기서 잠깐 문제가 생겼었는데, coalescing을 수행하기 위하여 차근차근 다음 블록들을 탐색하던 중, 크기가 0인 노드를 발견하면 무한 루프가 발생한다는 것이었다. 그래서 크기가 0인 노드를 발견하면 루프를 빠져나올 수 있게 하였다.

4.6 `mm_check`의 구현

```
269 int mm_check() {
270         char * iterator = search_start;
271         char * max_heap = mem_heap_hi();
272         while(iterator<=max_heap-SIZE_T_SIZE*2) {
```

```
273                 // printf("Checking block %p.. Alloc: %d, Size: %d\n", iterator,
                    ↪  GET_ALLOC(iterator), GET_SIZE(iterator));
274                 if(memcmp(iterator, FTRP(iterator), SIZE_T_SIZE)) {
275                         printf("node %p and footer %p differ\n", iterator,
                            ↪  FTRP(iterator));
276                         for(int i=0;i<SIZE_T_SIZE; i++) {
277                                 if(iterator[i] != FTRP(iterator)[i]) {
278                                         printf("node[%d]:%x footer[%d]:%x\n", i,
                                            ↪  (int)iterator[i] , i,
                                            ↪  (int)FTRP(iterator)[i]);
279                                 }
280                         }
281                         return 0;
282                 }
283                 iterator = NEXT_HDR(iterator);
284         }
285         printf("Last iterator: %p\n", iterator);
286         return 1;
287 }
```

이번 리스트를 이용한 구현에서는 각 메모리 블록들이 별로 복잡하지 않고 변수 하나만을 헤더로 가지기 때문에 `mm_check` 함수의 내용도 간단하다.

5 Conclusion

5.1 어려웠던 점

디버깅

포인터 연산을 매우 많이 사용하는 과제였는데, 이 때문에 포인터 관련하여 문제가 매우 많이 발생했다. 포인터가 스택이 아니라 힙에 생성되고, 할당 영역을 정확히 표시하지 않아서 이 포인터가 유효한지 유효하지 않은지 판단하는데 어려움을 겪었다. 최대한 각 노드들을 생성할 때 포인터들을 초기화하려 하기는 했지만 coalescing 할 때 주변 블록들을 탐색하는 과정에서 가짜 포인터들을 많이 만났다. 이러한 것들 때문에 segmentation fault 오류가 매우 많이 발생했다. 그래서 이러한 것들을 디버깅해야 했는데, 물론 gdb 사용법은 아직 잘 모르지만, gdb가 있으면 따라가면서 분석할 수 있어서 좋을 것 같았지만, 안타깝게도 gdb는 없었다. 그래서 다른 방법으로 디버깅을 시도하였는데, 오류 메시지가 segmentation fault(core dumped) 밖에 없어서 디버깅하기가 어려웠다. 이것을 해결한 방법은 `dmesg` 명령을 이용하는 것이었다. 이것으로 해당 segmentation fault가 발생한 정확한 주소를 알아낸 후, objdump로 비교하여 정보를 알아내려 하였다. 그런데 이렇게 하면 번거롭고 시간도 많이 걸려서 다른 정보를 찾아낸 결과, addr2line이라는 것을 사용해 보게 되었다. 이것을 사용하기 위해 `Makefile`의 컴파일 옵션에 `-g`를 추가하였다. 그 후 addr2line 유틸리티를 이용하여 해당 c 파일의 몇째 줄에서 오류가 발생하였는지 좀 더 빠르게 알아낼 수 있었다. 간혹 포인터 연산이 전혀 없는 곳에서 segmentation fault가 발생한 건이 있는데, 이 때는 objdump도 같이 이용하는 수밖에 없었다.
위와 같이 오류 위치를 찾는 것도 쉽지 않았지만, 더 어려운 것은 미리 미리 포인터들을 초기화하는 것이었다. 포인터들을 최대한 미리 초기화 해 두지 않으면 다음 coerce할 때 문제가 생긴다. 왜 포인터들에 이런 쓰레기 값이 들어가 있는지 원인을 찾아 수정하는 것이었다.
아직도 call 인스트럭션에서 왜 segmentation fault가 나는지는 잘 모르겠다.

5.2 놀라웠던 점

$O(n)$ 은 그렇게 나쁘지 않았다

원래는 빈 공간 탐색 속도가 빠른, 평균적으로 $O(logn)$이 기대되는 red-black tree 자료구조를 이용하여 구현하려 했었는데, balance를 맞추는 red-black tree를 구현하는 것은 오래 걸릴 것이라고 생각했고, struct를 함부로 사용하기 어려울 것 같아서 간단한 tree를 이용하여 구현하려고 하였다. 그런데 이렇게 그냥 tree를 이용하여 구현하다가 포인터들에 쓰레기 값이 들어가는 등의 문제로 인하여 어쩔 수 없이 tree를 이용한 구현을 폐기하고, 일단 디버깅과 구현이 상대적으로 간단한 list 방식으로 구현하게 되었다.