System Programming Project 3

담당 교수 : 김영재

이름 : 임종환

학번 : 20171683

1. 개발 목표

Project3의 개발 목표는 My Memory Allocator(**mm_malloc**, **mm_free**, **mm_realloc**)를 구현하여 libc에서 제공하는 **malloc**, **free**, **realloc**과 같이 작동하도록 만드는 것이다. **Segregated free list**를 통해 free block을 관리함으로써 Memory Allocator의 performance를 결정하는 Throughput과 Memory Utilization을 최대화한다.

2. 개발 범위 및 내용

① **mm_init() 함수**

Application program이 실행될 때 처음 호출되는 함수. Memory의 heap영역과 segregated list, class를 initialize한다.

② **mm_malloc(size) 함수**

heap영역에 **size** byte 이상 크기의 block을 allocate하는 함수. libc에서 제공하는 **malloc**이 8-byte aligned된 block의 pointer를 return하기 때문에 이 함수도 8-byte로 align하여 block을 allocate한다. 따라서 **size**를 8의 배수로 round-up하고, 할당하려는 block의 size 와 allocate flag를 나타내기 위한 header와 footer영역 8byte를 더하여 새로 allocate할 block의 size를 결정한다.

그런 다음 **find_fit** 함수를 호출하여 allocate할 수 있는 free block을 찾는다. Segregated list에서 allocate할 free block을 찾지 못한 경우, **extend_heap** 함수를 통해 heap영역을 늘려 allocate할 free block을 찾는다.

place 함수를 호출하여 찾은 free block에 memory를 allocate한 후 해당 block의 pointer를 return한다.

③ **mm_free(ptr) 함수**

ptr이 가리키는 allocate된 block을 free시키는 함수. 해당 block의 header와 footer의 allocate flag를 unallocated로 바꾸고 **coalesce** 함수를 호출한다.

free시킨 block의 이전 block, 또는 다음 block이 free block이라면 coalescing하여 하나의 free block으로 변경한다.

④ **mm_realloc(ptr, size) 함수**

ptr이 가리키는 allocate된 block의 크기를 변경하는 함수. **mm_malloc**과 size

를 double word로 align하여 reallocate할 block의 size인 **newsize**를 결정한다. 기존 ptr의 size를 **oldsize,** ptr의 다음 block의 size를 **nextsize**, 다음 block의 allocation 여부를 **next_alloc,** double word를 **DSIZE**고 할 때, **mm_realloc** 함수는 4가지 Case로 분류하여 작동한다.

Case1: oldsize – newsize >= 2*DSIZE

새로 변경하는 block의 size가 기존 block의 size보다 작아 split할 수 있는 경우이다. 조건에 **2*DSIZE**가 붙은 이유는 free block에 저장되는 정보가 header, footer, next free block의 주소, prev free block의 주소 총 4 words가 필요하기 때문이다.

이 경우 block을 **newsize**의 allocate block으로 바꾸고, 남은 부분은 split하여 **insert_list** 함수를 통해 free list에 삽입한다.

Case2: newsize + DSIZE == oldsize || newsize == oldsize

새로 변경하는 block의 size가 기존 block의 size와 같거나 혹은 새로 변경하는 block의 size가 기존 block의 size보다 **DSIZE**만큼만 작은 경우이다. 이 경우 block을 split할 수 없기 때문에 기존 block을 그대로 사용한다.

Case3: !next_alloc && (nextsize + oldsize – newsize >= 2*DSIZE)

Case3와 Case4는 새로 변경하는 block의 size가 기존 block의 size보다 큰 경우이다. Case3는 기존 block의 다음 block을 활용하여 기존 block을 움직이지 않고 size만 늘리는 경우이다. 다음 block이 free block이고, **nextsize + oldsize**가 **newsize** + 2***DSIZE** 보다 큰 경우가 case3에 해당한다.

다음 block을 split하여 일부는 기존 block으로 사용하고, 나머지는 free block으로 사용한다.

Case4: 그 외

기존 block을 다른 공간으로 옮겨야 하는 경우이다. **mm_malloc**을 통해 **newsize** 크기의 block을 allocate하고, 기존 block의 내용을 새로 allocate한 block으로 옮긴 후 **mm_free**를 통해 기존 block을 free시킨다.

3. 개발 방법

- **2.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 전역변수 등의 구현이나 수정을 서술)**

① **Global Variable**

segregated_list[CLASS]: free block을 관리하기 위한 pointer array이다. Macro에서 정의한 **CLASS** 개수만큼 pointer 배열을 생성한다. 각 class의 pointer는 class의 header block의 주소를 가리킨다.

② **extend_heap(words) 함수**

mm_malloc에서 **find_fit** 함수가 실패할 때마다 호출되어 heap영역의 size를 **words** 크기만큼 증가시키는 함수이다. **mem_sbrk**함수를 호출하여 heap영역을 늘리고, 새로 확장한 영역의 시작 주소를 return 받아 free block을 생성한다. 그 후 **coalesce**함수를 호출하여 Heap 영역을 확장하기 전 heap의 마지막 block이 free block인 경우 새로 확장한 block과 coalescing하여 block의 pointer를 return한다.

③ **coalesce(bp) 함수**

mm_free, extend_heap 함수에 의해 free되는 block이 생성될 때마다 호출되는 함수이다. Free된 block **bp**의 이전 block을 **prev_block**이라 하고, 다음 block을 **next_block**이라 할 때 4가지 Case가 있다.

Case1: prev_block allocated, **next_block** allocated

이 경우 이전 block, 다음 block과 coalescing할 수 없다. **insert_list** 함수를 호출하여 **bp**를 free list에 삽입한다.

Case2: prev_block allocated, **next_block** free

이 경우 **bp**와 **next_block**을 coalescing한다. 우선 **delete_list** 함수를 호출하여 **next_block**을 free list에서 제거하고, **bp**와 **next_block**을 합쳐 하나의 block으로 만든 뒤 **insert_list** 함수를 호출하여 합친 block을 free list에 삽입한다.

Case3: prev_block free, **next_block** allocated

Case2의 과정에서 **next_block**을 **prev_block**으로 변경한다.

Case4: prev_block free, **next_block** free

delete_list 함수를 호출하여 **prev_block, next_block**을 모두 free list에서 제거하고, **bp**와 **prev_block**, **next_block**을 모두 합쳐 하나의 block으로 만든 뒤 **insert_list** 함수를 호출하여 합친 block을 free list에 삽입한다.

④ **find_fit(asize) 함수**

mm_malloc이 실행될 때마다 호출되는 함수로, free list에서 **asize**크기의 block을 할당할 수 있는 free block을 찾는다. 낮은 class의 segregated list 부터 순차적으로 탐색한다. **asize**가 해당 class에 속할 수 있는 block의 최대 크기인 **class_size**보다 큰 경우 다음 class를 확인한다. **asize**가 **class_size**보다 작은 경우 해당 class의 list를 탐색한다. 탐색하는 list에 **asize**를 할당할 수 있는 free block을 찾은 경우, 해당 free block의 pointer를 return한다. 모든 list를 탐색하고도 block을 할당할 수 있는 free block을 찾지 못한 경우 실패를 의미하는 NULL을 return한다.

⑤ **place(bp, asize) 함수**

find_fit 함수를 통해 찾은 free block **bp**에 **asize**크기만큼 allocate하는 함수이다. **delete_list** 함수를 호출하여 **bp**를 free list에서 제거한다. **bp**의 크기를 **csize**라고 할 때, **csize – asize >= 2*DSIZE**인 경우 block을 split한다. **mm_realloc**에서와 마찬가지로, free block의 크기는 4 words 이상이어야 하기 때문에 **csize – asize >= 2*DSIZE** 인 경우에만 split한다. 이 경우 **asize**만큼의 공간은 allocate하고, 남은 공간은 free block으로 사용할 수 있기 때문에 **insert_list** 함수를 호출하여 free list에 삽입한다.

⑥ **insert_list(bp, asize) 함수**

segregated_list에 **asize**크기의 **bp**를 삽입하는 함수이다. **bp**를 삽입할 적절한 class를 찾은 후, 해당 class의 header에 **bp**를 삽입한다.(LIFO) 삽입은 doubly linked list와 같은 방식으로 작동한다.

⑦ **delete_list(bp, class) 함수**

segregated_list에서 **bp**를 제거하는 함수이다. 함수 호출 단계에서 bp의 **class**를 알 수 있는 경우 parameter로 넘겨주고, 그렇지 않은 경우 -1을 넘긴다. **class**의 값이 -1로 넘어온 경우 **bp**의 size를 통해 저장되어 있을

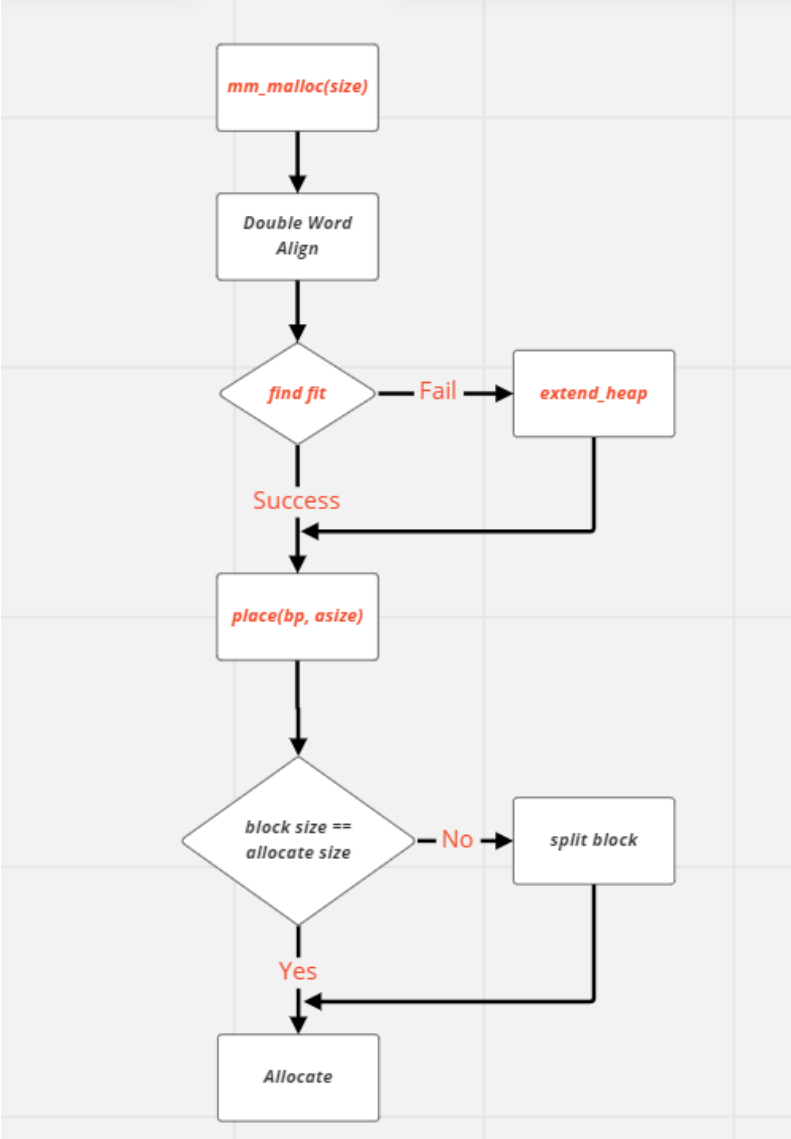
bp가 저장되어 있을 list의 **class**를 찾는다.

해당 **class**의 list에서 **bp**를 제거한다. 제거는 doubly linked list와 같은 방식으로 작동한다.

4. 구현 결과

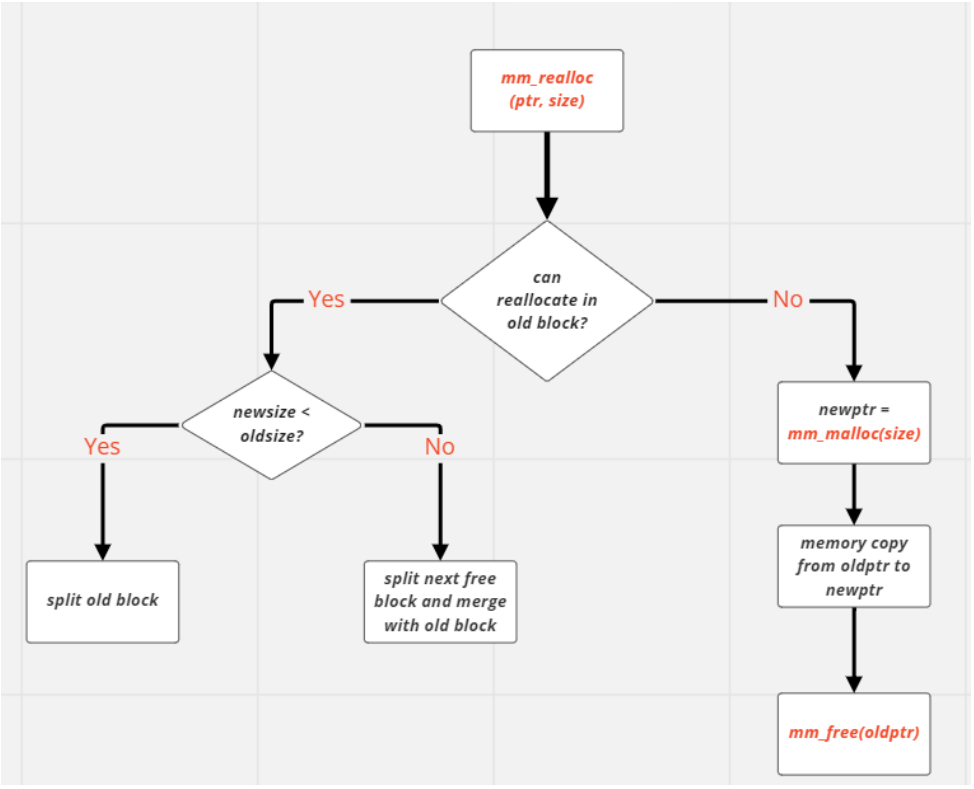
A. Flow Chart

1. mm_malloc(size)

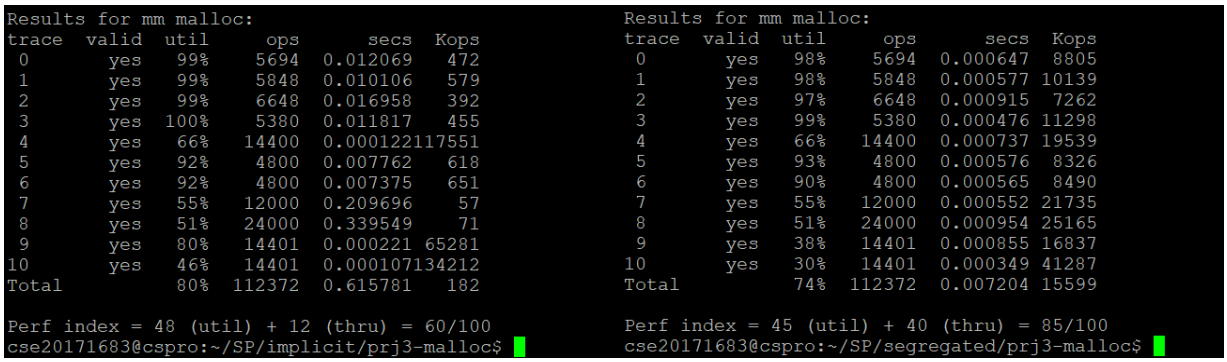


2. mm_free(ptr)

3. mm_realloc(ptr, size)



B. 실행 성능 분석(./mdriver -V)



```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.012069   472
 1       yes   99%    5848  0.010106   579
 2       yes   99%    6648  0.016958   392
 3       yes  100%    5380  0.011817   455
 4       yes   66%   14400  0.000122117551
 5       yes   92%    4800  0.007762   618
 6       yes   92%    4800  0.007375   651
 7       yes   55%   12000  0.209696    57
 8       yes   51%   24000  0.339549    71
 9       yes   80%   14401  0.000221 65281
10       yes   46%   14401  0.000107134212
Total          80%  112372  0.615781   182

Perf index = 48 (util) + 12 (thru) = 60/100
cse20171683@cspro:~/SP/implicit/prj3-malloc$
```

```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   98%    5694  0.000647  8805
 1       yes   98%    5848  0.000577 10139
 2       yes   97%    6648  0.000915  7262
 3       yes   99%    5380  0.000476 11298
 4       yes   66%   14400  0.000737 19539
 5       yes   93%    4800  0.000576  8326
 6       yes   90%    4800  0.000565  8490
 7       yes   55%   12000  0.000552 21735
 8       yes   51%   24000  0.000954 25165
 9       yes   38%   14401  0.000855 16837
10       yes   30%   14401  0.000349 41287
Total          74%  112372  0.007204 15599

Perf index = 45 (util) + 40 (thru) = 85/100
cse20171683@cspro:~/SP/segregated/prj3-malloc$
```

왼쪽의 화면은 **first-fit implicit list**로 구현한 program의 실행 화면이고, 오른쪽의 화면은 **LIFO segregated free list**로 구현한 program의 실행 화면이다.

first-fit implicit list는 free block 뿐만 아니라 allocate된 block까지 모두 하나의 list에 연결되어 있다. 또한 memory를 allocate할 때 free block을 찾기 위해 list를 맨 앞에서부터 탐색을 하기 때문에 memory가 이미 많이 할당된 경우 free block을 찾는 시간이 증가한다. 그에 따라 throughput이 감소하게 되고 performance가 그다지 좋지 않다.

Segregated free list는 allocate된 block은 따로 list에 연결하지 않고, free block만 list로 연결한다. 또한 각 free block의 size에 따라 class를 구분하여 memory를 allocate하려는 size에 맞는 class만 탐색함으로써 빠른 시간에 free block을 찾을 수 있다.

실행 결과를 보면 **implicit list**로 실행한 결과보다 throughput이 훨씬 높은 것을 확인할 수 있다.

하지만 memory utilization은 오히려 더 낮게 나왔다. **Segregated free list**는 first-fit search를 해도 best-fit과 비슷한 memory utilization이 나타나는 것으로 알려져 있는데, 내가 구현한 **mm_realloc**의 비효율성 때문에 이런 결과가 나타나는 것으로 보인다.

5. 추가 구현(Heap Consistency Checker)

My dynamic memory allocator의 heap consistency checking을 하는 함수 **mm_check()**를 작성했다. 이 함수는 다음 세 가지를 check한다.

① **Is every block in the free list marked as free?**

② **Are there any contiguous free blocks that somehow escaped coalescing?**

③ **Is every free block actually in the free list?**

첫번째 조건은 free list에 연결되어 있는 모든 block의 allocation flag가 모두 0인지 확인한다. **Segregated list**의 모든 class의 list를 완전 탐색하여 allocation flag를 check하고, 만약 연결된 block의 allocation flag가 1이라면 오류 메시지를 출력하고 fail을 의미하는 0을 return한다. 첫번째 조건을 check하는 부분의 코드는 아래와 같다.

```
/* Is every block in the free list marked as free? */
for(int i=0; i<CLASS; i++){
    bp = segregated_list[i];          // i class list의 header
    while(bp != NULL){
        if(GET_ALLOC(HDRP(bp))){      // allocation flag가 1인 경우
            printf("block in the free list not marked as free\n");
            return 0;                 // 오류
        }
        bp = *(char**)NEXT_FREE_BLKP(bp);    // next free block
    }
}
```

두번째 조건은 free block이 정상적으로 coalescing되는지 확인한다. 즉, 연속되는 분리된 free block이 존재하는지 확인한다. 세번째 조건은 free block이 정상적으로 free list에 삽입되는지 확인한다. 두번째 조건과 세번째 조건은 heap의 모든 block을 탐색하면서 같이 check한다.

mem_heap_lo() ~ mem_heap_hi() 사이의 block을 완전 탐색한다. 탐색 도중 free block을 발견한 경우, 해당 block의 이전 block 또는 다음 block이 free block인지 확인하여 2번째 조건을 check한다. 만약 free block의 이전 block 또는 다음 block이 free block이라면 coalescing이 정상적으로 수행되지 않은 것이므로 오류 메시지를 출력하고 fail을 의미하는 0을 return한다.

또한 발견한 free block이 free list에 존재하는지 확인하여 3번째 조건을 check한다. 3번째 조건은 free block **bp**가 free list에 존재하는지 확인하는 **find_free_bp**

함수를 작성하여 check했다. **find_free_bp** 함수는 **bp**의 size를 통해 bp가 속해있을 class를 찾은 후, 해당 class의 list를 완전 탐색하면서 **bp**와 일치하는 block이 존재하는지 찾는다. 만약 **bp**와 일치하는 block을 찾지 못한 경우 오류 메시지를 출력하고 fail을 의미하는 0을 return한다. 두번째 조건, 세번째 조건을 check하는 부분의 code는 아래와 같다.

```
    bp = mem_heap_lo() + DSIZE;         // heap의 시작 block
    while(bp != mem_heap_hi() + 1){     // heap의 마지막 block전 까지 check
        /* bp가 free block인 경우 */
        if(!GET_ALLOC(HDRP(bp))){
            /* Are there any contiguous free blocks that somehow escaped coalescing? */
            /* bp의 이전 block과 다음 block이 free인지 확인 */
            if(!GET_ALLOC(HDRP(PREV_BLKP(bp))) || !GET_ALLOC(HDRP(NEXT_BLKP(bp)))){
                printf("There is a free block that escaped coalescing\n");
                return 0;
            }
            /* Is every free block actually in the free list? */
            /* Segregated list에 bp가 point하는 block 존재하는지 확인 */
            if(!find_free_bp(bp)){
                printf("free block not in the free list\n");
                return 0;
            }
        }
        bp = NEXT_BLKP(bp);     // check next block
    }

    /* Heap is Consistent */
    return 1;
}
/* bp가 가리키는 free block이 free list에 존재하는지 check */
static int find_free_bp(void* bp){
    size_t size = GET_SIZE(HDRP(bp));
    size_t class_size = CLASS_ZERO_SIZE;
    int class = CLASS-1;
    /* find class of bp */
    for(int i=0; i<CLASS-1; i++){
        if(size <= class_size){
            class = i;
            break;
        }
        class_size <<= 1;
    }
    /* search class */
    void* ptr = segregated_list[class];
    while(ptr != NULL){
        /* bp가 list에 존재하는 경우 */
        if((char*)ptr == (char*)bp)
            return 1;
        ptr = (char*)(char**)NEXT_FREE_BLKP(bp);
    }
    /* bp가 list에 존재하지 않는 경우 */
    return 0;
}
```

세 가지 조건을 모두 만족하면 success를 의미하는 1을 return한다. Program의 heap consistency를 check하려면 **mm_malloc, mm_free, mm_realloc**에 주석 처리되어 있는 **mm_check()** 함수의 주석을 해제하여 check할 수 있다.