Malloc Lab

수리과학부 2019-16022 박채연

1. 실행 결과

로컬에서 테스트한 결과는 아래와 같다. 11개의 traces files에 대해 정확하게 동작하는 모습을 살펴볼 수 있다. 또한, Perf index는 92/100인데, 그 중 메모리 공간 활용의 효율성을 의미하는 util 점수가 52점, 단위 초 당 수행할 수 있는 malloc/free 명령어의 개수를 의미하는 throughput 점수가 40점이다.

```
chaeyeon@LAPTOP-4V7LQOS5:/mnt/c/users/kids1/lab1/SNU-System-Programming/malloclab/src$ ./mdriver -V
Using default tracefiles in ./traces/
Measuring performance with gettimeofday().

Testing mm malloc
Reading tracefile: amptjp-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: cccp-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: cp-decl-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: expr-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: coalescing-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: random-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: random2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: binary-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: binary2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: realloc-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: realloc2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.000303 18811
 1       yes   99%    5848  0.000263 22210
 2       yes  100%    6648  0.000268 24824
 3       yes  100%    5380  0.000249 21624
 4       yes   66%   14400  0.000127113744
 5       yes   96%    4800  0.005456   880
 6       yes   95%    4800  0.005299   906
 7       yes   96%   12000  0.015640   767
 8       yes   88%   24000  0.016635  1443
 9       yes   40%   14401  0.000229 62777
10       yes   69%   14401  0.000108133219
Total          86%  112372  0.044577  2521

Perf index = 52 (util) + 40 (thru) = 92/100
```

아래의 두 개의 스크린 샷은 sp05 서버에서 테스트를 수행한 결과이다. 로컬에서 테스트 수행 시 고정적으로 92점이 나왔지만 서버에서는 Perf Index의 변동폭이 매우 컸다. 대체적으로 Perf index가 80점에서 92점 사이로 나왔다. util 점수는 로컬에서와 마찬가지로 52점으로 고정적이었지만 수행 시간을 의미하는 throughput 점수가 binary-bal.rep과 binary2-bal.rep 테스트 수행 시간에 크게 영향을 받았다.

```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.002875  1981
 1       yes   99%    5848  0.000394 14850
 2       yes  100%    6648  0.000501 13262
 3       yes  100%    5380  0.000476 11298
 4       yes   66%   14400  0.000344 41897
 5       yes   96%    4800  0.025726   187
 6       yes   95%    4800  0.016626   289
 7       yes   96%   12000  0.134001    90
 8       yes   88%   24000  0.047561   505
 9       yes   40%   14401  0.002975  4841
10       yes   69%   14401  0.000269 53476
Total          86%  112372  0.231749   485

Perf index = 52 (util) + 32 (thru) = 84/100
```

```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.000512 11112
 1       yes   99%    5848  0.000391 14953
 2       yes  100%    6648  0.000507 13102
 3       yes  100%    5380  0.001642  3276
 4       yes   66%   14400  0.000358 40223
 5       yes   96%    4800  0.013484   356
 6       yes   95%    4800  0.018042   266
 7       yes   96%   12000  0.111822   107
 8       yes   88%   24000  0.105044   228
 9       yes   40%   14401  0.000572 25194
10       yes   69%   14401  0.000273 52770
Total          86%  112372  0.252647   445

Perf index = 52 (util) + 30 (thru) = 81/100
```

2. 어떻게 구현했는지

주교재인 CSAPP에서는 Allocator를 Implicit Free List으로 구현한 기본적인 코드가 제공되는데, 해당 코드를 그대로 이용할 경우 Perf Index가 낮게 나온다. 따라서 해당 코드를 Explicit Free List으로 바꾸는 작업을 거쳤다. 각 매크로 및 함수들에 대한 설명은 아래와 같다.

(0) 매크로 및 heap_listp, free_list

```c
/* single word (4) or double word (8) alignment */
#define ALIGNMENT 8

/* rounds up to the nearest multiple of ALIGNMENT */
#define ALIGN(size) (((size) + (ALIGNMENT-1)) & ~0x7)

#define SIZE_T_SIZE (ALIGN(sizeof(size_t)))

/* Basic constants and macros */
#define WSIZE 4
#define DSIZE 8
#define CHUNKSIZE (1<<12)

/* Return Maximum and Minimum value */
#define MAX(x, y) ((x) > (y) ? (x) : (y))
#define MIN(x, y) ((x) > (y) ? (y) : (x))

/* Pack a size and allocated bit into a word */
#define PACK(size, alloc) ((size) | (alloc))

/* Read or write the value */
#define GET(p) (*(unsigned int *)(p))
#define PUT(p, val) (*(unsigned int *)(p) = (val))

/* Read the size and allocated fields from address p */
#define GET_SIZE(p) (GET(p) & ~0x7)
#define GET_ALLOC(p) (GET(p) & 0x1)

/* Given a block ptr bp, compute adress of its header and footer */
#define HDRP(bp) ((char *)(bp) - WSIZE)
#define FTRP(bp) ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)

/* Given block ptr bp, compute adress of net and previous blocks */
#define NEXT_BLKP(bp) ((char *)(bp) + GET_SIZE(((char *)(bp) - WSIZE)))
#define PREV_BLKP(bp) ((char *)(bp) - GET_SIZE(((char *)(bp) - DSIZE)))

/* Basic Macros for Explicit Free List Implementation */
#define PREV_PTR(bp) ((char *)(bp))
#define NEXT_PTR(bp) (((char *)(bp)) + (WSIZE))
#define PRED(bp) (*(char **)(bp))
#define SUCC(bp) (*(char **)(NEXT_PTR(bp)))

void *heap_listp = NULL;
void *free_list = NULL;
```

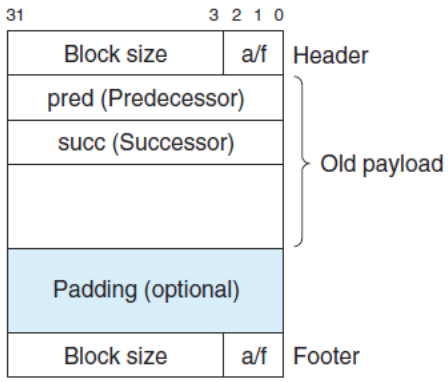
본 랩에서는 double word alignment (8 바이트)를 사용하므로 ALIGNMENT을 8로 지정하였다. 매크로 함수인 ALIGN에서 이를 이용하여 주어진 값보다 크거나 같은 8의 배수 가운데 가장 작은 값을 반환하도록 하였다. 워드 크기를 의미하는 WSIZE는 4로 지정하였고, 그 두 배 크기를 의미하는 DSIZE를 8로 지정하였다. 또한, Heap 공간이 부족할 경우 extend_heap() 함수를 호출하여 heap을 확장하는데, 확장하고자 하는 default byte 수를 CHUNKSIZE로 지정하였다.

매크로 함수인 MAX와 MIN은 각각 주어진 두개의 값 가운데 큰 값과 작은 값을 반환한다. PACK은 크기와 할당 여부를 워드 크기로 동시에 묶어주는 역할을 하는데, 32비트 가운데 첫 29비트가 크기 정보를 저장하고 그 뒤의 3비트가 할당 여부를 저장한다.

GET과 PUT은 각각 주어진 메모리 위치로부터 값을 읽어들이거나 해당 위치에 원하는 값을 저장하는 역할을 한다. 한편, void 형 포인터는 직접적으로 참조하는 것이 불가능하므로 unsigned int 형 포인터로 casting을 하고 나서 참조하였다. GET_SIZE와 GET_ALLOC은 내부에서 GET을 사용하여 각각 주어진 메모리 공간의 크기와 할당 여부를 반환한다.

block pointer인 bp는 header 바로 다음의 메모리 공간을 가리킨다. 즉, 이 공간은 할당된 메모리 공간일 경우 payload의 첫번째 부분과 일치한다. HDRP는 주어진 block pointer가 가리키는 메모리 block의 header 주소를 반환하고, FTRP는 footer 주소를 반환한다.

PREV_BLKP와 NEXT_BLKP는 각각 주어진 block pointer가 가리키는 메모리 block의 이전과 다음 메모리 block의 block pointer를 반환한다. 이때, 반환하는 메모리 공간이 할당된 상태일 수도 있고 할당되지 않은 상태일 수도 있다.



추가적으로 CSAPP 교재에서 제공하지 않는 매크로 함수인 PREV_PTR, NEXT_PTR, PRED, SUCC을 정의하였는데, 이는 Explicit Free List 구현을 위한 함수들이다. PREV_PTR과 NEXT_PTR은 free 상태인 메모리 block의 block pointer를 인자로 받는데, 해당 block 이전과 다음의 free 상태인 메모리 block의 block pointer를 반환한다. 또한, PRED와 SUCC 역시 free 상태인 메모리 block의 block pointer를 인자로 받아서 해당 block 이전과 다음의 free 상태인 메모리 block을 반환한다. 이때, 위의 이미지와 같이 Explicit Free List는 헤더 바로 다음의 워드에 해당 메모리 block 이전의 free 상태인 메모리 block 주소를 저장하고, 그 다음의 워드에 해당 메모리 block 다음의 free 상태인 메모리 block 주소를 저장한다는 점을 이용하였다.

마지막으로 heap_listp는 heap의 prologue block를 가리키고, free_list는 free 상태인 메모리 block 가운데 가장 마지막을 가리킨다.

(1) mm_init

```
int mm_init(void)
{
    /* Initialize free_list with NULL */
    free_list = NULL;

    /* Create the initial empty heap */
    if ((heap_listp = mem_sbrk(4*WSIZE))==(void *)-1)
        return -1;

    PUT(heap_listp, 0); /* Alignment padding */
    PUT(heap_listp + (1*WSIZE), PACK(DSIZE, 1)); /* Prologue header */
    PUT(heap_listp + (2*WSIZE), PACK(DSIZE, 1)); /* Prologue footer */
    PUT(heap_listp + (3*WSIZE), PACK(0, 1)); /* Epilogue header */
    heap_listp += (2*WSIZE);

    /* Extend the empty heap with a free block of CHUNKSIZE bytes */
    if (extend_heap(CHUNKSIZE/WSIZE) == NULL)
        return -1;
    return 0;
}
```

malloc package를 초기화하는 단계이다. 4개의 워드 공간 만큼 heap을 할당 받고 heap의 가장 처음 워드에 NULL을 지정한다. 다음으로 이어지는 세 개의 워드에 순서대로 Prologue header/footer 및 Epilogue header를 설정해준다. 이때, Prologue block의 크기는 8이고 할당된 상태이다. 또한, Epilogue block에는 footer가 없으며 Epilogue block의 크기는 0이고 할당된 상태이다. heap_listp를 2*WSIZE 만큼 증가시켜 prologue block을 가리키게 하였다. 마지막으로 extend_heap 함수를 호출하여 default size로 지정했던 CHUNKSIZE 바이트만큼 힙 공간을 확장하였다.

(2) mm_malloc

```
void *mm_malloc(size_t size)
{
    size_t asize; /* Adjusted block size */
    size_t extendsize; /* Amount to extend heap if no fit */
    void *bp = free_list;

    /* Ignore spurious requests */
    if (size == 0)
        return NULL;

    /* Adjust block size to include overhead and alignment reqs */
    asize = ALIGN(size + DSIZE);
    /* Search the free list for a fit */
    while(bp && GET_SIZE(HDRP(bp))<asize)
        bp = PRED(bp);

    /* Appropriate memory block found */
    if (bp != NULL){
        bp = place(bp, asize);
        return bp;
    }
```

```
    /* No fit found. Get more memory and place the block */
    extendsize = MAX(asize, CHUNKSIZE);
    if ((bp = extend_heap(extendsize/WSIZE))==NULL)
        return NULL;
    bp = place(bp, asize);
    //mm_check();
    return bp;
}
```

우선 동적 할당 요청된 메모리 block의 크기가 0이면 곧바로 NULL을 반환한다. 한편, 본 랩에서는 각 메모리 block마다 header/footer가 존재하고 double word alignment를 사용하기 때문에 size 바이트 크기만큼 할당 요청하더라도 size+DSIZE 보다 크거나 같은 8의 배수 가운데 가장 작은 수만큼의 바이트를 할당하게 된다. 이 크기를 asize라고 하자.

우선 while 문을 통해 free_list 내에서 asize보다 크거나 같은 free 상태의 메모리 block이 있는지 확인한다. 이때, Explicit Free List를 사용하므로 Implicit Free List와 다르게 모든 메모리 block을 탐색할 필요가 없어 효율적이다. 만약 asize보다 크거나 같은 크기의 block을 찾으면 place함수를 호출하여 해당 block에 asize만큼 메모리 block을 할당받는다. 그러나 만약 free_list 내에 해당 조건을 만족하는 block이 없다면 extend_heap을 호출하여 heap을 확장하고 나서 place 함수를 통해 확장된 공간에 asize만큼 메모리 block을 할당받는다.

(3) mm_free

```
void mm_free(void *ptr)
{
    //Get the size of the given memory block
    size_t size = GET_SIZE(HDRP(ptr));

    //Update the header and the footer of the given block pointer
    PUT(HDRP(ptr), PACK(size, 0));
    PUT(FTRP(ptr), PACK(size, 0));

    //Coalesce the memory blocks if the previous block or the next block is freed.
    coalesce(ptr);
}
```

mm_free는 주어진 block pointer가 가리키는 메모리 block의 header와 footer에 접근하여 할당 여부를 0으로 지정한다. 만약 방금 할당 해제한 메모리 block의 이전 메모리 block이나 다음 메모리 block이 free 상태이면 이들을 한 개의 free 상태인 메모리 block으로 합쳐주어야 한다. 따라서 최종적으로 coalesce(ptr)을 호출하였다.

(4) mm_realloc

```c
void *mm_realloc(void *ptr, size_t size)
{
    void *nextptr, *prevptr, *newptr;
    size_t nextsize, cursize, prevsize;
    size_t asize;

    // If ptr is NULL the call is equivalent to mm_malloc(size)
    if (ptr == NULL)
        return mm_malloc(size);
    // If size is equal to zero, the call is equivalent to mm_free(ptr)
    else if (size == 0){
        mm_free(ptr);
        return NULL;
    }
```

mm_realloc 함수의 parameter는 void형 포인터인 ptr과 재할당받고자 하는 크기인 size이다. 만약 ptr이 NULL이라면 이는 mm_malloc(size)와 같은 역할을 수행해야 한다. 또한, 만약 size가 0이라면 이는 mm_free(ptr)과 같은 역할을 수행하고 NULL을 반환해야 한다.

```c
// The call to mm_realloc changes the size of the memory block pointed to by ptr to size bytes
else {
    asize = ALIGN(size + DSIZE);
    cursize = GET_SIZE(HDRP(ptr));

    // If the requested size is less than or equal to the current size of the block
    if (asize <= cursize){
        // If the remaining size of the memory block is too small
        if (cursize-asize < 2*DSIZE){
            PUT(HDRP(ptr), PACK(cursize,1));
            PUT(FTRP(ptr), PACK(cursize,1));
        }
        // Otherwise, split the block for the efficiency of memory use
        else{
            PUT(HDRP(ptr), PACK(asize,1));
            PUT(FTRP(ptr), PACK(asize,1));
            PUT(HDRP(NEXT_BLKP(ptr)), PACK(cursize-asize,0));
            PUT(FTRP(NEXT_BLKP(ptr)), PACK(cursize-asize,0));
            insert_free(NEXT_BLKP(ptr));
        }
        return ptr;
    }
```

이제 위의 두 가지 경우에 해당하지 않는 경우를 살펴보자. 본 랩에서는 각 메모리 block마다 header/footer가 존재하고 double word alignment를 사용하기 때문에 size 바이트 크기만큼 할당 요청하더라도 size+DSIZE 보다 크거나 같은 8의 배수 가운데 가장 작은 수만큼의 바이트를 할당하게 된다. 이 크기를 asize라고 하자. 또한 ptr이 가리키는 메모리 block의 크기를 cursize라고 하자. 만약 asize가 cursize보다 작거나 같다면 ptr이 가리키는 메모리 block 내에서 재할당이 가능함을 의미한다.

한편, prologue block을 제외한 임의의 할당된 메모리 공간은 header와 footer가 각각 1개의 워드를 차지하고 payload가 적어도 1개의 워드를 차지한다. 따라서 double word alignment에 의해 prologue block을 제외한 임의의 할당된 메모리 공간은 적어도 4개의 워드, 즉 16 바이트를 차지한다. 만약 asize만큼 재할당하고 남은 메모리 block의 크기인 csize-asize가 2*DSIZE, 즉 4개의 워드 만큼의 크기보다 작다면 해당 공간은 어차피 필요없으므로 시간 효율성을 위해 split 하지 않고 csize만큼 할당한다. 그렇지 않다면 메모리 사용의 효율성을 위해 해당 block을 split 하고 남

은 메모리 공간을 free_list에 추가하였다.

```
//If the previous block is allocated and the next block is freed
if (GET_ALLOC(HDRP(prevptr))&&!GET_ALLOC(HDRP(nextptr))){
    tot = nextsize + cursize;

    // If the sum of these two blocks' sizes is greater than or equal to the requested size
    if (tot >= asize){
        delete_free(nextptr);
        //If the remaining size of the memory block is too small
        if (tot - asize < 2*DSIZE){
            PUT(HDRP(ptr), PACK(tot, 1));
            PUT(FTRP(ptr), PACK(tot, 1));
        }
        // Otherwise, split the block for the efficiency of the memory use
        else{
            PUT(HDRP(ptr), PACK(asize, 1));
            PUT(FTRP(ptr), PACK(asize, 1));
            PUT(HDRP(NEXT_BLKP(ptr)), PACK(tot-asize,0));
            PUT(FTRP(NEXT_BLKP(ptr)), PACK(tot-asize,0));
            insert_free(NEXT_BLKP(ptr));
        }
        return ptr;
    }
}
```

```
// If the previous block is freed and the next block is allocated
 else if (!GET_ALLOC(HDRP(prevptr))&&GET_ALLOC(HDRP(nextptr))){
     tot = prevsize + cursize;
     // If the sum of these two blocks' sizes is greater than or equal to the requested size
     if (tot >= asize){
         delete_free(prevptr);
         // Copy the payload of the block
         memmove(prevptr, ptr, cursize-DSIZE);
         // If the remaining size of the memory block is too small
         if (tot - asize < 2*DSIZE){
             PUT(HDRP(prevptr), PACK(tot, 1));
             PUT(FTRP(prevptr), PACK(tot, 1));
         }
         // Otherwise, split the block for the efficiency of the memory use
         else{
             PUT(HDRP(prevptr), PACK(asize, 1));
             PUT(FTRP(prevptr), PACK(asize, 1));
             PUT(HDRP(NEXT_BLKP(prevptr)), PACK(tot-asize,0));
             PUT(FTRP(NEXT_BLKP(prevptr)), PACK(tot-asize,0));
             insert_free(NEXT_BLKP(prevptr));
         }
         return prevptr;
     }
 }
```

```
// If both the previous block and the next block is freed
else if (!GET_ALLOC(HDRP(prevptr))&&!GET_ALLOC(HDRP(nextptr))){
    tot = prevsize + cursize + nextsize;
    // If the sum of these three blocks' sizes is greater than or equal to the requested size
    if (tot >= asize){
        delete_free(nextptr);
        delete_free(prevptr);
        //Copy the payload of the block
        memmove(prevptr, ptr, cursize-DSIZE);
        // If the remaining size of the memory block is too small
        if (tot - asize < 2*DSIZE){
            PUT(HDRP(prevptr), PACK(tot, 1));
            PUT(FTRP(prevptr), PACK(tot, 1));
        }
        // Otherwise, split the block for the efficiency of the memory use
        else{
            PUT(HDRP(prevptr), PACK(asize, 1));
            PUT(FTRP(prevptr), PACK(asize, 1));
            PUT(HDRP(NEXT_BLKP(prevptr)), PACK(tot-asize,0));
            PUT(FTRP(NEXT_BLKP(prevptr)), PACK(tot-asize,0));
            insert_free(NEXT_BLKP(prevptr));
        }
        return prevptr;
    }
}
```

만약 asize가 cursize보다 크다면 ptr이 가리키는 메모리 block 내에서 재할당이 불가능하다. 따라서 ptr이 가리키는 메모리 block의 이전 block과 다음 block이 free 상태인지, 만약 두 메모리 block 가운데 free 상태인 block이 있을 경우 해당 freed 메모리 block 들의 크기 합이 asize보다 크거나 같은지 살펴보아야 한다. 만약 이전 block 혹은 다음 block이 free 상태이고 {ptr이 가리키는 메모리 block 크기 + (이전 block과 다음 block 가운데 free 상태인 block 의 크기)}가 asize보다 크거나 같다면 mm_malloc을 호출하지 않고 해당 공간을 재사용하였다.

```
// Otherwise, we should allocate a new memory block
newptr = mm_malloc(size);
memcpy(newptr, ptr, cursize-DSIZE);
mm_free(ptr);
return newptr;
```

만약 위의 모든 경우에 해당하지 않는다면 mm_malloc함수를 호출하여 heap의 다른 적절한 공간에 새로운 메모리 block을 할당받았다. memcpy를 통해 ptr이 가리키던 메모리 block의 payload를 새로운 block에 복사하고 mm_free(ptr)을 통해 ptr이 가리키던 메모리 block을 할당 해제하였다.

(5) extend_heap

```
static void *extend_heap(size_t words){
    void *bp;
    size_t size;

    /* Allocate an even number of words to maintain alignment */
    size = (words % 2) ? (words + 1) * WSIZE : words * WSIZE;
    if ((long)(bp = mem_sbrk(size)) == -1)
        return NULL;

    /* Initialize free block header/footer and the epilogue header */
    PUT(HDRP(bp), PACK(size, 0)); /* Free block header */
    PUT(FTRP(bp), PACK(size, 0)); /* Free block footer */
    PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1)); /* New epilogue header */

    /* Coalesce if the previous block was free */
    return coalesce(bp);
}
```

extend_heap(word)는 heap 메모리 공간이 부족할 때 heap을 확장시켜 주는 함수이다. 본 랩에서는 double word alignment을 사용하므로 주어진 워드의 개수가 홀수라면 (주어진 워드 수 + 1)개의 워드 크기 만큼 heap 공간을 확장한다.

이때, bp=mem_sbrk(size)의 호출로 포인터 bp는 새롭게 확장된 heap 메모리 공간의 시작 부분을 가리키고 있다. 새롭게 확장된 메모리 block의 header와 footer에 block의 크기와 할당 여부(=0)을 지정하고 bp가 가리키는 메모리 block의 다음 block의 header을 새로운 epilogue block으로 지정한다.

한편, 새로운 heap 공간을 확장하기 전에 마지막에 있던 메모리 block이 free 이었을 경우 freed memory block이 두 개 연속되는 문제가 발생하므로 coalesce(bp)를 호출하여 해당 문제를

해결하였다.

(6) coalesce

```
static void *coalesce(void *bp){
    size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
    size_t size = GET_SIZE(HDRP(bp));

    // Case 1: The previous block and the next block are allocated
    if (prev_alloc && next_alloc){
        insert_free(bp);
    }
    // Case 2: The previous block is allocated and the next block is freed
    else if (prev_alloc && !next_alloc){
        delete_free(NEXT_BLKP(bp));

        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size, 0));

        insert_free(bp);
    }
    // Case 3: The previous block is freed and the next block is allocated
    else if (!prev_alloc && next_alloc){
        delete_free(PREV_BLKP(bp));

        size += GET_SIZE(HDRP(PREV_BLKP(bp)));
        PUT(FTRP(bp), PACK(size, 0));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);

        insert_free(bp);
    }
    // Case 4: The previous block and the next block is freed
    else{
        delete_free(PREV_BLKP(bp));
        delete_free(NEXT_BLKP(bp));

        size += GET_SIZE(HDRP(PREV_BLKP(bp))) + GET_SIZE(FTRP(NEXT_BLKP(bp)));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);

        insert_free(bp);
    }
    return bp;
}
```

coalesce(ptr) 함수는 block pointer인 ptr이 가리키는 메모리 block을 free하고 나서 해당 메모리 block의 이전 block과 다음 block을 살핀다. 만약 이전 block 혹은 다음 block이 free 상태일 경우 이를 free 상태인 하나의 메모리 block으로 합쳐주어야 한다.

coalesce(ptr)이 호출될 때 총 4가지의 경우의 수가 존재한다. ptr이 가리키는 메모리 block의 이전 block과 다음 block이 모두 할당된 상태일 수 있고, 이전 block만 할당되어 있고 다음 block은 free 상태일 수 있다. 또한, 이전 block이 free 상태이고 다음 block은 할당된 상태일 수도 있으며, 마지막으로 두 block 모두 free 상태일 수 있다.

1) ptr이 가리키는 메모리 공간의 이전 block과 다음 block이 모두 할당된 상태일 경우 insert_free(ptr)을 호출한다. 뒤에 이 함수에 대한 구체적인 설명이 제시되어 있는데, 간단히 설명하자면 free list에 ptr을 추가하는 것이다.

2) ptr이 가리키는 메모리 공간의 이전 block이 할당된 상태이고 다음 block이 free 상태일 경우 우선 delete_free(NEXT_BLKP(bp))를 통해 다음 block을 free_list로부터 제거한다. 두 개의 block을 합쳐 하나의 block으로 만들고 header와 footer에 크기 및 할당 여부를 업데이트한다. 마지막으로 해당 block을 insert_free()를 통해 free_list에 추가하며 해당 block의 block pointer를 반환한다.

3) ptr이 가리키는 메모리 공간의 이전 block이 free 상태이고 다음 block이 할당된 상태일 경우 우선 delete(PREV_PTR(bp))를 통해 이전 block을 free_list로부터 제거한다. 두 개의 block을 합쳐 하나의 block으로 만들고 header와 footer에 크기 및 할당 여부를 업데이트한다. 마지막으로 해당 block을 insert_free()를 통해 free_list에 추가하며 해당 block의 block pointer를 반환한다.

4) ptr이 가리키는 메모리 공간의 이전 block과 다음 block이 free 상태일 경우 delete(PREV_PTR(bp))와 delete(NEXT_PTR(bp))를 통해 이전 block과 다음 block을 free_list로부터 제거한다. 세 개의 block을 합쳐 하나의 block으로 만들고 header와 footer에 크기 및 할당 여부를 업데이트한다. 마지막으로 해당 block을 insert_free()를 통해 free_list에 추가하며 해당 block의 block poitner를 반환한다.

(7) place

```
static void *place(void *bp, size_t asize){
    size_t csize = GET_SIZE(HDRP(bp));
    delete_free(bp);

    // If the remaining size is smaller than 2*DSIZE
    if ((csize-asize) < (2*DSIZE)){
        PUT(HDRP(bp), PACK(csize, 1));
        PUT(FTRP(bp), PACK(csize, 1));
        return bp;
    }
    // If the size of the block that is allocated is bigger than 100 bytes
    else if (asize>100){
        PUT(HDRP(bp), PACK(asize, 1));
        PUT(FTRP(bp), PACK(asize, 1));
        PUT(HDRP(NEXT_BLKP(bp)), PACK(csize-asize, 0));
        PUT(FTRP(NEXT_BLKP(bp)), PACK(csize-asize, 0));
        insert_free(NEXT_BLKP(bp));
        return bp;
    }
    // If the size of the block that is allocated is not bigger than 100 bytes
    else {
        PUT(HDRP(bp), PACK(csize-asize, 0));
        PUT(FTRP(bp), PACK(csize-asize, 0));
        PUT(HDRP(NEXT_BLKP(bp)), PACK(asize, 1));
        PUT(FTRP(NEXT_BLKP(bp)), PACK(asize, 1));
        coalesce(bp);
        return NEXT_BLKP(bp);
    }
}
```

place(bp, size)는 block pointer가 가리키는 free 상태인 메모리 block에 size 만큼의 block을 할당하는 함수이다. 우선 delete_free 함수를 호출하여 bp가 가리키는 메모리 block을 free_list로부터 제거하였다.

한편, prologue block을 제외한 임의의 할당된 메모리 block을 고려하자. header와 footer가 각각 1개의 워드를 차지하고 payload가 적어도 1개의 워드를 차지한다. 따라서 double word alignment

에 의해 prologue block을 제외한 임의의 할당된 메모리 공간은 적어도 4개의 워드를 차지한다는 사실을 알 수 있다.

만약 block pointer인 bp가 가리키는 free 상태의 메모리 block이 csize 바이트를 차지하고 할당하고자 하는 메모리 block의 크기가 asize라면 csize-asize를 계산한다. 만약 이 값이 2*DSIZE, 즉 4개의 워드 만큼의 크기보다 작다면 할당하고 남은 공간은 어차피 필요없으므로 시간 효율성을 위해 split 하지 않고 csize만큼 할당한다.

그러나 만약 csize-asize 값이 2*DSIZE 보다 크거나 같다면 메모리 공간 활용의 효율성을 위해 해당 block을 split 해야 한다. 만약 할당받고자 하는 크기가 100(=threshold)보다 크다면 bp가 가리키는 메모리 공간의 앞쪽에 asize만큼의 메모리 block을 할당 받고, 작다면 bp가 가리키는 메모리 공간의 뒤쪽에 메모리 block을 할당받았다. 마지막으로 insert_free를 통해 split 한 free 상태의 block을 free_list에 추가하였다. 실험 결과 threshold를 80에서 100 사이의 값으로 지정했을 때 Perf index가 가장 높았다.

(8) insert_free

```
static void insert_free(void *bp){
    void *head = free_list;
    void *tail = NULL;
    size_t size = GET_SIZE(HDRP(bp));

    // Freed blocks are sorted according to their size
    while (head && (size > GET_SIZE(HDRP(head)))){
        tail = head;
        head = PRED(head);
    }

    // If there is a block whose size is greater than or equal to that of the block that 'bp' points to
    if (head != NULL){
        // If tail is not NULL, then the block that 'bp' points to should be placed between head and tail
        if (tail != NULL){
            PUT(PREV_PTR(bp), (unsigned int)head);
            PUT(NEXT_PTR(head), (unsigned int)bp);
            PUT(NEXT_PTR(bp), (unsigned int)tail);
            PUT(PREV_PTR(tail), (unsigned int)bp);
        }
        // If tail is NULL, it implies that the block that 'bp' points to is the smallest freed block.
        else {
            PUT(PREV_PTR(bp), (unsigned int)head);
            PUT(NEXT_PTR(head), (unsigned int)bp);
            PUT(NEXT_PTR(bp), (unsigned int)NULL);
            free_list = bp;
        }
    }
    // If there doesn't exist any block whose size is greater than or equal to that of the block that 'bp' points to
    else{
        // If tail is not NULL, it implies that s block that bp points to is the biggest freed block.
        if (tail != NULL){
            PUT(PREV_PTR(bp), (unsigned int)NULL);
            PUT(NEXT_PTR(bp), (unsigned int)tail);
            PUT(PREV_PTR(tail), (unsigned int)bp);
        }
        // If tail is NULL, it implies that there doesn't exist any freed blocks except for the one that 'bp' points to
        else {
            PUT(PREV_PTR(bp), (unsigned int)NULL);
            PUT(NEXT_PTR(bp), (unsigned int)NULL);
            free_list = bp;
        }
    }
}
```

insert_free는 Explicit Free List 구현을 위해 필수적인 함수이다. 어떤 메모리 block을 free 하고나서 insert_free 함수를 호출함으로써 free_list에 free 된 메모리 block을 추가한다. 이때, 효율성을 위해 free_list는 freed block의 크기 순으로 정렬되어 있다.

while문을 통해 free_list를 차례로 탐색하면서 bp가 가리키는 메모리 block의 크기보다 크거나 같은 메모리 공간을 찾는다. 이때, head는 현재 비교하는 메모리 block, tail은 바로 직전에 비교했던 메모리 block을 의미한다.

1) 만약 head가 NULL이 아니라면 free list에 bp가 가리키는 메모리 block보다 크거나 같은 메모리 block이 존재한다는 것을 의미한다. 이 때, tail이 NULL이 아니라면 bp가 가리키는 메모리 block의 크기는 head가 가리키는 메모리 block보다 작거나 같고 tail이 가리키는 메모리 block보다 크다는 것을 의미하므로 그 중간에 연결한다. 만약 tail이 NULL이라면 bp가 가리키는 메모리 block이 가장 작은 크기의 freed block임을 의미한다. 따라서 head와 bp를 연결한다.

2) 한편, head가 NULL이라면 free_list에서 bp가 가리키는 메모리 block보다 크거나 같은 메모리 block을 찾지 못했다는 것을 의미한다. 이때 만약 tail이 NULL이 아니라면 bp가 free_list 내의 메모리 block 가운데 사이즈가 가장 크다는 것을 의미하고, bp와 tail을 연결한다. 그러나 만약 tail이 NULL이라면 free_list에 어떠한 메모리 block도 존재하지 않는다는 것을 의미하므로 free_list에 bp만 있게 된다.

(9) delete_free

```
static void delete_free(void *bp){
    // If there exists previous freed block
    if (PRED(bp) != NULL){
        // If there exists next freed block
        if (SUCC(bp) != NULL){
            PUT(NEXT_PTR(PRED(bp)), (unsigned int)SUCC(bp));
            PUT(PREV_PTR(SUCC(bp)), (unsigned int)PRED(bp));
        }
        //If there doesn't exist next freed blcok
        else{
            PUT(NEXT_PTR(PRED(bp)), (unsigned int)NULL);
            free_list = PRED(bp);
        }
    }
    // If there does't exist previous freed block
    else{
        // If there exists next freed block
        if (SUCC(bp) != NULL)
            PUT(PREV_PTR(SUCC(bp)), (unsigned int)NULL);
        // If there doesn't exist any freed block
        else
            free_list = NULL;
    }
}
```

delete_free 역시 Explicit Free List 구현에 필수적인 함수이다. free 상태인 어떤 메모리 block이 mm_malloc() 호출 등으로 인해 더 이상 free 상태가 아니면 free_list로부터 제거해주어야 한다.

1) 만약 bp가 가리키는 메모리 block의 이전 freed block과 다음 freed block이 모두 존재한다면 이전 freed block과 다음 freed block을 연결함으로써 bp가 가리키는 메모리 block을 free_list로부터 제거한다.

2) 만약 bp가 가리키는 메모리 block의 이전 freed block이 존재하지만 다음 freed block이 존재하지 않는다면 이전 freed block의 다음 freed block에 대한 포인터를 NULL로 지정하여 bp가 가리키는 메모리 block을 free_list로부터 제거한다.

3) 만약 bp가 가리키는 메모리 block의 이전 freed block은 존재하지 않고 다음 freed_ block만 존재한다면 다음 freed block의 이전 freed block에 대한 포인터를 NULL로 지정하여 bp가 가리키는 메모리 block을 free_list로부터 제거한다.

4) 만약 bp가 가리키는 메모리 block의 이전 freed block과 다음 freed block이 모두 존재하지 않는다면 free_list 내에 오직 bp가 가리키는 메모리 block만이 존재했다는 것을 의미한다. 따라서 free_list를 NULL로 지정함으로써 bp가 가리키는 메모리 block를 free_list에서 제거한다.

(10) mm_check

```
int mm_check(void){

    void *bp = free_list;
    while (bp){
        /* Is every block in the free list marked as free? */
        if (GET_ALLOC(HDRP(bp))){
            fprintf(stderr, "Every block int he free list is not marked as free.\n");
        }
        /* Are there any contiguous free blocks that somehow escaped coalescing? */
        else if (PRED(bp) == PREV_BLKP(bp)){
            fprintf(stderr,"There are contiguous free blocks.\n");
        }
        bp = PRED(bp);
    }
    bp = heap_listp;
    while (GET_SIZE(HDRP(bp))){
        /* Do the pointers in a heap block point to valid heap adresses? */
        if (bp < heap_listp || bp >= mem_sbrk(0)){
            fprintf(stderr,"Pointers in a healp block do not point to valid heap adress\n");
        }
        bp = NEXT_BLKP(bp);
    }
    return 0;
}
```

mm_check 함수에서는 free_list의 모든 block이 실제로 헤더에 free 상태로 지정되어 있는지, 연속된 두 개의 메모리 block이 free 상태는 아닌지, 힙 공간 내의 포인터들이 유효한 주소를 가리키는지 점검한다. 이때, 만약 구현한 프로그램에 치명적인 문제가 존재한다면 관련 메시지를 출력하도록 하였다. 다만 가능한 많은 문제점을 알아내기 위한 목적으로 문제 발생시 exit() 하지 않음으로써 프로그램이 강제적으로 종료되지 않는 한 에러 메시지를 계속 출력하도록 하였다.

3. 어려웠던 점

구현을 어느정도 마무리하고 나서 ./mdriver -V 를 통해 테스트를 수행할 때 결과가 출력되지

않고 tracefile을 읽어들이는 중이라는 문구가 출력된 후 더 이상 아무런 변화가 없었다. 무한 루프에 걸린 것인지 혹은 구현한 프로그램 자체의 성능이 좋지 않아서 그런 것인지 정확한 원인을 알지 못해 어려움을 겪었다. 특히 각 함수별로 아무리 코드를 반복적으로 읽어봐도 잘못된 부분은 딱히 보이지 않았다. printf() 함수를 각 함수 내에 적절히 배치하여 어떤 부분이 문제인지 확인해 본 결과 프로그램이 mm_malloc() 함수 내에 존재하는 while 문을 빠져나오지 못한다는 것을 알게 되었다. 해당 while 문은 free_list를 탐색하면서 할당하고자 하는 크기보다 크거나 같은 메모리 block을 찾는 역할을 수행하는데, free_list를 NULL로 초기화하지 않아서 해당 while 문을 빠져나오지 못한다는 것을 발견하였다. mm_init 함수 내에서 free_list=NULL로 초기화하여 해당 문제를 해결할 수 있었다.

GET_SIZE 함수를 사용할 때 block pointer가 아닌 메모리 block의 헤더를 argument로 제공해야 하는데 실수로 block pointer 자체를 argument로 제공하여 valid 테스트를 모두 통과하지 못했었다. 코드가 길어지면서 디버깅을 하기가 굉장히 까다로워져서 어려움을 겪었다.

또한, Explicit Free List 를 구현하고 나서 ./mdriver -V 를 통해 테스트를 수행해보니 binary2-bal.rep, binary-bal.rep와 realloc-bal.rep, realloc2-bal.rep 에 대한 테스트 수행시 util 점수가 비교적 낮게 나왔다. throughput 점수를 크게 떨어뜨리지 않으면서 util 점수를 향상시킬 수 있는 방법을 찾는데 어려움을 겪었다.

4. 새롭게 배운 점

mm_realloc을 보다 효율적으로 구현하는 방법에 대해 배웠다. 주교재인 CSAPP는 mm_init, mm_malloc, mm_free를 어떻게 구현해야 하는지에 대해 관련 코드까지 제공하는 등 설명이 자세하다. 그러나 mm_realloc을 어떻게 구현해야 하는지에 대한 설명은 아예 없어서 구현에 어려움을 겪었다. malloclab-handout.pdf 파일에서는 mm_realloc 구현시 ptr이 NULL이면 mm_malloc(size)와 동일한 역할을 수행해야하고, size가 0이면 mm_free(ptr)과 동일한 역할을 수행해야 한다고 제시한다. 만약 이 두 가지 경우에 해당하지 않는다면 구현하는 방식에 따라 새로 할당하는 메모리 공간의 주소가 이전 메모리 공간의 주소와 다를 수도 있고 같을 수도 있다고 제시되어 있었다.

처음에는 매우 단순하게 ptr 이 NULL 이 아니고 size 도 0 이 아닌 경우 mm_malloc() 함수를 호출하여 무조건 새로운 메모리 공간을 할당받도록 하였다. 테스트를 수행해보니 valid 항목은 모두 yes 가 떴지만 realloc-bal.rep 과 realloc2-bal.rep 에 대한 util 점수가 매우 낮은 것을 확인하였다. 보다 효율적인 메모리 공간 사용을 위해 고민하던 중 재할당하고자 하는 메모리 크기가 ptr 이 가리키는 메모리 block 의 크기보다 작거나 같은 경우 해당 메모리 공간을 재사용하고, 그렇지 않을 경우 {(ptr 이 가리키는 메모리 block 의 이전 block 과 다음 block 가운데 free 상태인 메모리 block 의 크기 합) + ptr 이 가리키는 메모리 block 의 크기} 를 계산하여 이 값이 재할당하고자 하는 메모리 크기보다 클 경우 해당 메모리 block 들을 재사용하도록 구현해보았다. 앞의 모든 경우에 해당하지 않는 경우 최후의 선택으로 mm_malloc() 함수를

호출하여 메모리 공간의 효율성을 높이고자 했는데 실제로 테스트를 수행해보니 이전보다 훨씬 높은 수준으로 util 점수가 향상되는 것을 확인할 수 있었다.

Standard C 라이브러리에서 제공하는 malloc 관련 함수들이 어떤 원리로 동작하는지 정확히 이해하게 되었다. 본 실습을 수행하기 전에는 malloc 관련 함수들이 어떤 원리로 동작하는지 크게 관심이 없었고, 단지 Standard C 라이브러리에서 제공하는 해당 함수들을 사용하기만 할 뿐이었다. 본 랩 과제를 수행하면서 malloc package 의 성능을 향상시키기 위해 Implicit Free List 와 Explicit Free List 등 다양한 구현 방법에 대해 이해하는 것이 필수였는데, 과거의 많은 컴퓨터 공학도들이 동적 메모리 할당의 효율성 향상을 위해 고민해온 흔적을 느낄 수 있었다.