**CSED211: Malloc Lab Report**

20200952 / Chaeyeon Jang

## 1. Overview

Malloc Lab 은 C program 에서 dynamic storage allocator, 즉 malloc, realloc, free, initialize routines 를 구현하는 lab 이다. 구현된 dynamic memory allocator 를 평가하는 기준은 총 두 가지로, throughput 과 memory utilization 이 있다. Throughput 은 단위시간 당 완료되는 작업의 수로, 이번 lab 에서는 average number of operations completed per second 로 계산된다. Memory utilization 은 aggregate payload(aggregate amount of memory used by the driver)와 size of the heap 의 비율로, optimal memory utilization 은 1 이다. Throughput 과 memory utilization 의 weighted sum 을 통해, 아래 수식과 같이 최종 performance index $P$를 얻어 이를 memory allocator evaluation 에 사용한다.

$$P = wU + (1-w)\min\left(1, \frac{T}{T_{libc}}\right)$$

## 2. Type of Memory Allocator

이번 lab 에서 구현한 memory allocator 의 종류는 아래와 같다.

| | |
|---|---|
| Free block management schemes | Implicit free list |
| Placement Policy | Next fit |

## 3. Function Implementation

### (0) Macros

Dynamic memory allocator function 들을 구현하기 위해 필요한 macro 들을 아래와 같이 선언해주었다. 교과서에 있는 macro 정의 code 를 참고하여 작성하였다.

```
/* 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)))

/*****newly defined macros******/
#define WSIZE 4
#define DSIZE 8
#define CHUNKSIZE (1<<12) /*Extend heap by this amount (bytes)*/
#define MAX(x,y) (x > y? x: y)

/*pack a size and allocated bit into a word*/
```

```
#define PACK(size, alloc) (size | alloc)

/*read and write a word at address p*/
#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 block ptr bp, compute address 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 address of next 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))

static char *heap_listp = NULL;
static char *next = NULL;
/*****head of user defined function*****/
static void *extend_heap(size_t words);
static void *coalesce(void *bp);
static void *find_fit(size_t asize);
static void place(void *bp, size_t asize);
```

**(1) mm_init**

mm_init 을 위한 코드는 아래와 같다. 교과서에 있는 code 를 참고하여 작성하였다.

```
int mm_init(void)
{
    /*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);
    next = heap_listp;

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

    return 0;
}
```

Memlib.h 에 정의되어 있는 mem_sbrk 함수를 사용하여 heap 영역을 4 * WSIZE(16bytes)만큼 확장하고, global void pointer variable 인 heap_listp 를 업데이트한다. 그리고 memory 의 특정 위치에 데이터를 write 하는 PUT macro 함수를 이용하여 heap 의 prologue, epilogue 를 생성한다. 이때 prologue 와 epilogue 를 생성하는 이유는 heap 의 경계를 정의하기 위해서이고, prologue 와 epilogue 는 항상 할당된 상태(allocation status == 1)로 유지된다. heap_list pointer 변수에 2 * WSIZE 를 더하고, next fit 에서 다음으로 탐색할 block 위치를 저장하는 pointer 인 next 는 heap_listp pointer 와 같은 곳을 가르키도록 한다.
그 후 extend_heap 함수를 사용하여 CHUNKSIZE 만큼 heap 을 확장한다. extend_heap 함수는 아래와 같다.

```
static void *extend_heap(size_t words){
    char *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));
    PUT(FTRP(bp), PACK(size, 0));
    PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1));

    return coalesce(bp);
}
```

extend_heap 함수에서는 alignment 를 유지하기 위해 짝수 개의 words 를 할당하고, header, footer, next block header 를 각각 size | 0, size | 0, 0 | 1 로 initialize 해준다.

### (2) mm_malloc

mm_malloc 을 위한 코드는 아래와 같다.

```
void *mm_malloc(size_t size)
{
    size_t asize; /*adjusted block size*/
    size_t extendize; /*amount to extend heap if no fit*/
    char *bp;

    if(size == 0) return NULL;

    if (size <= DSIZE) asize = 2 * DSIZE;
    else asize = DSIZE * ((size + DSIZE + (DSIZE - 1)) / DSIZE);

    if((bp = find_fit(asize)) != NULL){
        place(bp, asize);
        return bp;
    }

    /*if no fit found, get more memory and place the block*/
    extendize = MAX(asize, CHUNKSIZE);
    if((bp = extend_heap(extendize/WSIZE)) == NULL){
        return NULL;
    }
    place(bp, asize);
    return bp;

}
```

우선 size 가 0 일 경우, NULL 포인터를 반환하고, size 가 DSIZE 보다 작거나 같을 경우 adjusted block size(실제로 malloc 을 통해 할당될 size)인 asize 변수에 2 * DSIZE 를 저장한다, 그 외의 경우 asize 변수에 DSIZE * ((size + DSIZE + DSIZE - 1) / DSIZE)를 저장한다. 그리고 next fit policy 를 통해 block 이 allocate 될 위치를 찾아 pointer 를 반환하는 함수인 find_fit 을 사용하여

allocation 위치에 대한 pointer 를 bp 변수에 저장하고, 만약 반환된 pointer 가 NULL 이 아니라면 정해진 위치에 block 을 allocate 시키는 place 함수를 사용하여 asize 만큼 bp 위치에 block 을 할당시킨다.
만약 next fit policy 를 통해 적합한 위치가 찾아지지 못했다면, extend_heap 함수를 통해 MAX(assize, CHUNKSIZE)만큼 heap 을 확장시키고 반환된 위치를 bp 에 저장한 후, place 함수를 통해 bp 위치에 asize 만큼 block 을 할당시킨다.

아래는 next fit policy 를 사용하는 find_fit 함수에 대한 코드이다.

```
static void *find_fit(size_t asize){ /*use next fit strategy*/
    char *bp;

    /*search starts from next pointer(where last search ends)*/
    for(bp = next; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKP(bp)){
        if(!GET_ALLOC(HDRP(bp)) && (GET_SIZE(HDRP(bp)) >= asize)){
            next = bp;
            return bp;
        }
    }

    /*if cannot find fit block through next search,
    then start search from very first block in an implicit list*/
    for(bp = heap_listp; bp < next; bp = NEXT_BLKP(bp)){
        if(!GET_ALLOC(HDRP(bp)) && (GET_SIZE(HDRP(bp)) >= asize)){
            next = bp;
            return bp;
        }
    }

    return NULL; /*if no fit*/
}
```

먼저 next fit search 를 시작할 위치인 next pointer 에서부터, 그 위치의 header 에 저장된 size 가 0 보다 클 때에 한해(list 의 끝까지 도달할 때까지) 다음 block 들을 순차적으로 탐색해가며 allocation status 가 0 이고(block 이 free 상태이고), 할당할 asize 보다 크기가 큰 block 이 있다면 그 block 을 allocation 할 위치로 지정한다. 그 후 해당 위치에 대한 pointer 를 반환한다.

만약 next 부터 list 의 끝까지 search 를 진행했을 때 적합한 allocation 위치를 찾지 못했다면, list 의 처음부터 next 위치 전까지 search 를 진행하며 적합한 allocation 위치를 찾는다. 이때 적합한 block 이 있다면 그 block 을 allocation 할 위치로 지정한 후 해당 위치에 대한 pointer 를 반환한다.
만약 위 두 과정을 거쳤음에도 적합한 block 를 찾지 못했다면 NULL pointer 를 반환한다.

아래는 place 함수에 대한 코드이다.

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

    if((size - asize) >= 2 * DSIZE){ /*split the block after allocation*/
        PUT(HDRP(bp), PACK(asize, 1));
        PUT(FTRP(bp), PACK(asize, 1));
        bp = NEXT_BLKP(bp);
        PUT(HDRP(bp), PACK((size - asize), 0));
        PUT(FTRP(bp), PACK((size - asize), 0));

        // next = bp;
    }else{ /*need not splitting*/
        PUT(HDRP(bp), PACK(size, 1));
        PUT(FTRP(bp), PACK(size, 1));
    }
}
```

Size(현재 pointer 가 가르키는 block header 에 저장된 크기)와 asize(할당할 block 크기)의 차가 2 * DSIZE 보다 크거나 같다면, split 을 한다. 그 외의 경우 split 을 하지 않는다.

### (3) mm_free

mm_free 를 위한 코드는 아래와 같다.

```
void mm_free(void *ptr)
{
    size_t size = GET_SIZE(HDRP(ptr));
    PUT(HDRP(ptr), PACK(size, 0));
    PUT(FTRP(ptr), PACK(size, 0));

    coalesce(ptr);
}
```

인자로 받은 pointer 가 가르키는 위치의 block header 에 저장된 크기를 size 변수에 저장하고, 그 block 의 header 와 footer 를 size | 0 으로 업데이트한다. 즉 allocation status 를 0 으로 하여 free block 으로 업데이트하는 것이다. 그리고 coalesce 함수를 사용하여 previous block 이나 next block 이 free 일 경우 coalescing 을 진행한다.

Coalesce 함수는 아래와 같다.

```
static void *coalesce(void *bp){
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp))); //get allocation status of the next block
    size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp))); //get allocation status of the previous block
    size_t cur_size = GET_SIZE(HDRP(bp)); // get size of the current block

    /*case 1*/
    if(prev_alloc && next_alloc){
        return bp;
    }
    /*case 2: free prev block, allocated next block*/
    if(!prev_alloc && next_alloc){
        cur_size += GET_SIZE(FTRP(PREV_BLKP(bp)));
        PUT(FTRP(bp), PACK(cur_size, 0));
        PUT(HDRP(PREV_BLKP(bp)), PACK(cur_size, 0));
        bp = PREV_BLKP(bp);
    }
    /*case 3: allocated prev block, free next block*/
    else if(prev_alloc & !next_alloc){
        cur_size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(bp), PACK(cur_size, 0));
        PUT(FTRP(bp), PACK(cur_size, 0));
    }
    /*case 4: both are free*/
    else{
        cur_size += GET_SIZE(HDRP(NEXT_BLKP(bp))) + GET_SIZE(FTRP(PREV_BLKP(bp)));
        // PUT(FTRP(NEXT_BLKP(bp)), PACK(cur_size, 0));
        // PUT(HDRP(PREV_BLKP(bp)), PACK(cur_size, 0));
        PUT(HDRP(PREV_BLKP(bp)), PACK(cur_size, 0));
        PUT(FTRP(NEXT_BLKP(bp)), PACK(cur_size, 0));
        bp = PREV_BLKP(bp);
```

```
    }

    if((next > (char *)bp) && (next < NEXT_BLKP(bp))) next = bp;

    return bp;

}
```

Coalesce 함수는 총 4 가지의 case 에 대해 각 case 에 맞는 operation 을 진행한다.
Case 1 은 previous block 과 next block 이 모두 allocate 된 상태이다. 이 경우 아무 operation 을 하지 않고 해당 위치의 포인터를 그대로 반환한다.
Case 2 는 previous block 은 free 상태, next block 은 allocated 상태인 경우이다. 이 경우, 현재 block 의 size 가 저장되어 있는 cur_size 변수에 previous block 의 size 를 더하고, 현재 block 의 footer 의 size 와 allocation status 를 각각 cur_size, 0 으로 업데이트 해준다. 그리고 previous block 의 header 의 size 와 allocation status 를 각각 cur_size, 0 으로 업데이트 해준다. 그 후 현재 위치에 대한 pointer 인 bp 를 previous block 을 가리키도록 업데이트 해준다.
Case 3 는 previous block 은 allocated 상태, next block 은 free 상태인 경우이다. 이 경우, 현재 block 의 size 가 저장되어 있는 cur_size 변수에 next block 의 size 를 더하고, 현재 block 의 header 의 size 와 allocation status 를 각각 cur_size, 0 으로 업데이트 해준다. 그리고 현재 block footer 의 size 와 allocation status 를 각각 cur_size, 0 으로 업데이트 해준다.
Case 4 는 previous block 과 next block 이 모두 free 인 상태이다. 이 경우, previous block size 와 next block size 를 cur_size 에 더해주고, previous block 의 header 의 size 와 allocation status 를 각각 cur_size, 0 으로 업데이트 해준다. 그리고 next block 의 footer 의 size 와 allocation status 를 각각 cur_size, 0 으로 업데이트 해준다. 그 후 현재 위치에 대한 pointer 인 bp 를 previous block 을 가르키도록 업데이트 해준다.

마지막으로 next pointer 가 bp 와 NEXT_BLKP(bp) (next block 에 대한 위치) 사이에 있다면, next pointer 의 위치를 bp 로 조정해준다.

### (4) mm_realloc

mm_realloc 를 위한 코드는 아래와 같다.

```
void *mm_realloc(void *ptr, size_t size)
{
    void *oldptr = ptr;
    void *newptr;
    void *nextptr;
    size_t oldsize = GET_SIZE(HDRP(ptr)) - DSIZE;
```

```
size_t newsize = ALIGN(size);
size_t nextsize;

if(oldptr == NULL) return mm_malloc(size);
if(size == 0){
    mm_free(oldptr);
    return NULL;
}
if(newsize == oldsize) return oldptr;

/*if size of to be allocated block(newsize) is smaller than old one(oldsize)*/
if(newsize < oldsize){
    newptr = oldptr;
    if((oldsize - newsize) >= (2 * DSIZE)){ //split the block after allocation
        PUT(HDRP(newptr), PACK((newsize + DSIZE), 1));
        PUT(FTRP(newptr), PACK((newsize + DSIZE), 1));
        newptr = NEXT_BLKP(oldptr);
        PUT(HDRP(newptr), PACK((oldsize - newsize), 0));
        PUT(FTRP(newptr), PACK((oldsize - newsize), 0));

        coalesce(newptr);
    }
    return oldptr;
}

/*if next block is free && size of current block + size of next block is larger than newsize*/
nextptr = NEXT_BLKP(oldptr);
if(nextptr != NULL && !GET_ALLOC(HDRP(nextptr))){
    nextsize = GET_SIZE(HDRP(nextptr)) - DSIZE;
    if(oldsize + nextsize > newsize){
        newptr = oldptr;

        if((oldsize + nextsize - newsize) >= (2 * DSIZE)){
            PUT(HDRP(newptr), PACK((newsize + DSIZE), 1));
            PUT(FTRP(newptr), PACK((newsize + DSIZE), 1));
            newptr = NEXT_BLKP(oldptr);
```

```
            PUT(HDRP(newptr), PACK((oldsize + nextsize + DSIZE - newsize), 0)); //(oldSize + DSIZE + nextSize +
DSIZE) - (newSize + DSIZE)
            PUT(FTRP(newptr), PACK((oldsize + nextsize + DSIZE - newsize), 0));

            coalesce(newptr);
        }else{
            PUT(HDRP(newptr), PACK((oldsize + DSIZE + nextsize + DSIZE), 1));
            PUT(FTRP(newptr), PACK((oldsize + DSIZE + nextsize + DSIZE), 1));
        }

        return oldptr;
    }
  }

  /*neither next block is free nor newsize is smaller than oldsize*/
  newptr = mm_malloc(size);
  if(newptr == NULL) return NULL;
  memcpy(newptr, oldptr, oldsize);
  mm_free(oldptr);
  return newptr;
}
```

우선, 현재 위치에 대한 pointer 인 oldptr 가 NULL 이면, mm_malloc 함수로 malloc 을 진행하고 malloc 으로부터 반환된 pointer 를 return 한다. 만약 인자로 받은 size(realloc 을 진행할 size)가 0 일 경우, mm_free 함수로 free 를 진행하고 NULL pointer 를 return 한다. 만약 oldsize(현재 위치 block 의 size – DSIZE)와 newsize(ALIGN(size))가 같다면, 아무 operation 을 수행하지 않고 oldptr 을 return 한다.

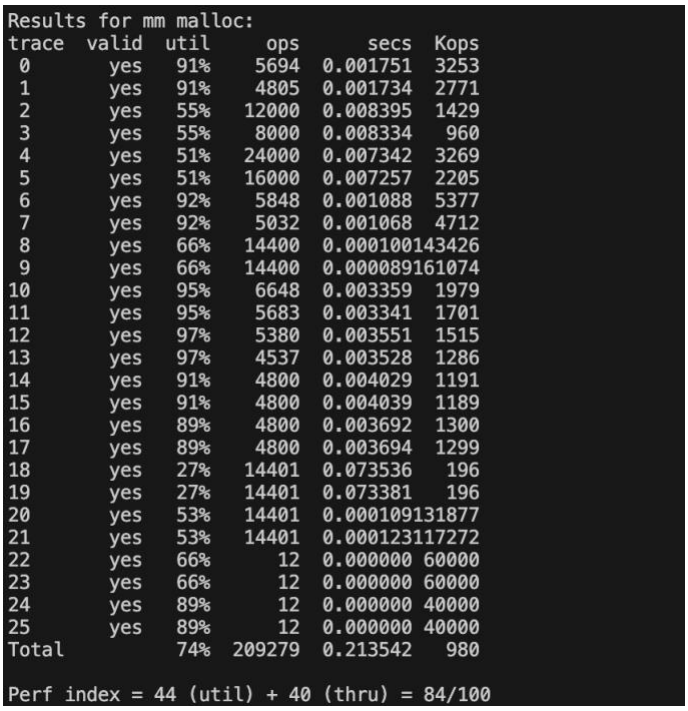
만약 newsize 가 oldsize 보다 작다면, newptr 을 oldptr 로 업데이트 해준다(현재 ptr 에서 그대로 realloc 을 진행한다). 만약 두 사이즈의 차가 2 * DSIZE 보다 크거나 같다면 split 을 진행하여 allocate 되지 않은 split 된 부분(split 된 두 부분 중 뒷부분)을 free 상태로 업데이트해준다. 그리고 coalesce 함수로 next block 이 free 상태라면 coalescing 을 진행한다.

만약 newsize 가 oldsize 보다 작지 않지만, 현재 block 의 next block 이 free 상태이고 oldsize + next block size 가 newsize 보다 커서 newsize 를 할당할 수 있다면, newptr 을 oldptr 로 업데이트해준다(현재 ptr 에서 그대로 realloc 을 진행한다). 마찬가지로 split 조건을 만족할 때 split 및 coalsecing 을 진행하고 split 조건을 만족하지 않으면 split 을 하지 않는다.

만약 next block 이 free 하지도 않고 oldsize + nextsize 가 newsize 보다 크지 않다면, mm_malloc 함수로 새로 malloc 을 진행하고, 이 과정에서 반환된 pointer 를 newptr 에 저장한다. 만약 malloc 후 반환된 pointer 가 NULL 이면 최종적으로 NULL 을 반환하고, NULL 이 아니라면 oldptr 는 free 시키고 newptr 을 반환한다.

### 4. Evaluation Result

구현한 dynamic memory allocator 에 대한 evaluation 결과는 아래와 같다.



```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   91%    5694  0.001751  3253
 1       yes   91%    4805  0.001734  2771
 2       yes   55%   12000  0.008395  1429
 3       yes   55%    8000  0.008334   960
 4       yes   51%   24000  0.007342  3269
 5       yes   51%   16000  0.007257  2205
 6       yes   92%    5848  0.001088  5377
 7       yes   92%    5032  0.001068  4712
 8       yes   66%   14400  0.000100143426
 9       yes   66%   14400  0.000089161074
10       yes   95%    6648  0.003359  1979
11       yes   95%    5683  0.003341  1701
12       yes   97%    5380  0.003551  1515
13       yes   97%    4537  0.003528  1286
14       yes   91%    4800  0.004029  1191
15       yes   91%    4800  0.004039  1189
16       yes   89%    4800  0.003692  1300
17       yes   89%    4800  0.003694  1299
18       yes   27%   14401  0.073536   196
19       yes   27%   14401  0.073381   196
20       yes   53%   14401  0.000109131877
21       yes   53%   14401  0.000123117272
22       yes   66%      12  0.000000 60000
23       yes   66%      12  0.000000 60000
24       yes   89%      12  0.000000 40000
25       yes   89%      12  0.000000 40000
Total          74%  209279  0.213542   980

Perf index = 44 (util) + 40 (thru) = 84/100
```

위와 같이 throughput, memory utilization score 가 도출된 이유에 대해 분석해보았다.

#### (1) Throughput

구현한 dynamic memory allocator 는 implicit free list + next fit 인데, 우선 implicit free list 는 모든 block 들(free block + allocated block)을 모두 순회하며 적절한 allocation 위치를 탐색하기 때문에 explicit free list, segregated free list 에 비해서 탐색 시간이 길다. 그렇기 때문에 단위 시간 당 처리하는 task 수인 throughput 이 다른 free list 유형에 비해 낮을 수 밖에 없다.
그래도 next fit policy 는 first fit policy 와 best fit policy 에 비해 allocation 위치 탐색 속도가 더 빠르다. 다음 탐색 위치에서부터 탐색을 시작하고 높은 확률로 fit block 을 찾아내기 때문에 re-scanning 을 방지하여 latency 를 줄일 수 있는 것이다. 그렇기 때문에 implicit free list 의 속도적인 단점을 next fit policy 를 사용함으로써 보완하였고, 낮지 않은 throughput 을 이끌어 낼 수 있었다.

#### (2) Memory Utilization

Implicit free list 는 fragmentation 에 의해 memory utilization efficiency 가 다른 explicit free list(address-ordered), segregated free list 에 비해 낮다. 더불어, next fit strategy 도 first fit, best

fit 보다 worse fragmentation 을 가지고 있어 utilization 이 아주 높게 나오지는 못했다. 이는 best fit 방식의 사용, 혹은 segregated list + first fit 방식의 사용을 통해 보완될 수 있을 것이라고 생각하며, 기회가 된다면 구현해보면 좋을 것 같다.