CSED211: Malloc Lab Report

20230499/KimJaeHwan

1. Overview

이번 랩에서는 C언어를 이용해 힙 영역의 메모리를 런타임 시간에 사용할 수 있도록 할당기를 만드는 것을 목표로 하며, 구체적으로 mm_init(), mm_malloc(), mm_free(), mm_realloc(), mm_check 다섯 함수를 구현해야 한다. 이번 랩에서 함수를 구현을 위해 교과서의 코드를 적극적으로 참고하였으며, Implicit free method를 사용하였다.

2. memlib 라이브러리 분석.

처음에는 직접 메모리 구조에서 힙 영역을 할당받아야 하는 것으로 이해했으나, memlib 사용자 라이브러리를 이용하여 MAXHEAP 만큼의 메모리를 받아와, 이를 전체 힙 영역으로 사용한다. 즉, 우선 받아온 메모리 영역을 통해 힙, 동적 할당을 시뮬레이션하는 느낌으로 작성하였다.

memlib 라이브러리에서 직접 사용하는 함수는 mem_sbrk() 함수로, 기존 sbrk() 함수와 같이 힙 메모리를 추가로 받아오는 함수이다. 매개변수로 정수를 받아, 받은 정수만큼 힘 영역을 증가시킨 후 힙의 첫 주소를 반환한다. 이번 랩에서 힙을 줄이지 않는다.

3. 유용한 매크로 함수들

코드의 작성을 더 쉽게 하도록 다음과 같은 매크로 함수를 사용한다.

```
/***** Generel settings from Textbook (Figure 9.41, 9.43 from textbook) *****/
/* Basic constants and macros */
#define WSIZE 4 /* Word and header/footer size (bytes) */
#define DSIZE 8 /* Double word size (bytes) */
#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)))
/***** End of general settings *****/
```

WSIZE / DSIZE: 각각 워드 크기(4바이트)와 더블 워드 크기(8바이트)를 정의하였다.

CHUNKSIZE는 힙을 확장하는 단위이다.

PACK(size, alloc) 매크로 함수는 각각 bit or 연산을 통해 하나로 결합하며, 이는 정렬되었을 때 free 블록의 하위 비트가 항상 0이 되는 것을 이용하여 해당 자리에 할당 여부를 표시하기 때문에 이 자리를 이용한다.

GET_SIZE(), GET_ALLOC() 매크로 함수를 이용해 서로를 분리해서 얻을 수 있다.

GET(), PUT() 매크로 함수를 이용하여 포인터의 주소의 값을 읽고 쓸수 있도록 하였다.

HDRP(), FTRP() 매크로 함수를 사용하여 블록 포인터를 통해 헤더와 푸터의 위치를 계산하였다.

NEXT_BLKP(), PREV_BLKP() 매크로 함수를 정의하여 이전, 다음 블록으로 쉽게 이동할 수 있도록 하였다.

4. mm_init() 구현

```
int mm_init(void);
/* pointer to the first block */
static char *heap_listp =0;
/*
 * mm_init - initialize the malloc package.
 */
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);
    /* Extend the empty heap with a free block of CHUNKSIZE bytes */
    if (extend_heap(CHUNKSIZE/WSIZE) == NULL)
        return -1;
    return 0;
}
```

mm_init() 함수는 프로그램 맨 처음에 실행되어, 이후 mm_malloc() 함수와 mm_free() 함수가 잘 동작할 수 있도록 초기 설정하는 함수이다. 초기 힙을 설정하기 위해 먼저 mem_sbrk를 통해 4개의 워드를 확장하고, 정렬 패딩, 프롤로그 블록, 에필로그 헤더를 초기화한다. 초기화 후 extend_heap을 호출하여 힙을 기본 크기(CHUNKSIZE)만큼 확장하고, 실패 시 -1을 반환한다. 성공적으로 초기화되면 프롤로그와 에필로그가 포함된 힙 구조가 생성된다.

5. mm_extend()

```
/*
 * extend_heap - Extend heap with free block and return its block pointer
 */
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)); /* 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);
}
```

원하는 크기만큼 매개변수로 받아, 그만큼 힙을 늘려주는 함수이다. 정렬을 위해 짝수만큼의 바이트 크기로 변환한 후, 늘린 힙 영역을 free block으로 변환하고 늘어난 만큼 epilogue 헤더를 이동한다. 그리고 이전 블록이 비어있는 경우 free block을 병합해야 하므로, coalesce() 함수를 호출해 주었다. coalesce() 함수는 추후 mm_free() 함수와 함께 설명하였다.

6. mm_malloc()

```
/*
 * mm_malloc - Allocate a block by incrementing the brk pointer.
 *     Always allocate a block whose size is a multiple of the alignment.
 */
void *mm_malloc(size_t size)
{
    size = size == 112 ? 128 : size; // To reduce internal fragment
    size = size == 448 ? 512 : size; // To reduce internal fragment
    size_t asize; /* Adjusted block size */
    size_t extendsize; /* Amount to extend heap if no fit */
    char *bp;
    /* Ignore spurious requests */
    if (size == 0)
        return NULL;
    /* Adjust block size to include overhead and alignment reqs. */
    if (size <= DSIZE)
        asize = 2*DSIZE;
    else
        asize = DSIZE * ((size + (DSIZE) + (DSIZE-1)) / DSIZE);
    /* Search the free list for a fit */
    if ((bp = find_fit(asize)) !=NULL) {
        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;
    place(bp, asize);
    return bp;
}
```

함수 맨 앞의 굵은 글씨의 두 줄은 최적화를 위한 코드이며, 나머지가 기본적인 동작이다. 먼저 크기가 0인 경우 예외처리해 주고, 정렬 기준에 따라 크기를 알맞게 조절한다. 이후 해당 크기만큼 find_fit() 함수를 통해 가능한 자리를 탐색하고, 적절한 자리가 있는 경우 place() 함수를 통해 원하는 크기의 allocated block을 배치한다. 만약 알맞은 자리가 없다면, extend_heap 함수를 통해 힙을 늘린 후 앞에 배치한다. 이 경우, mdriver 도구를 통해 검사를 한 결과, binary*.rep trace 파일에서 특히 메모리 활용률이 낮게 나타났는데, 해당 trace 파일들을 확인한 결과 112, 448과 같이 애매한 크기의 할당으로 인해 Internal fragmentation이 많이 발생한다고 예상되어, 해당 케이스에 맞춰 최적화하였다.

7. find_fit(), place()

```
/*
 * find_fit - Find a fit for a block with asize bytes
 */
static void *find_fit(size_t asize)
{
    /* First-fit search */
    void *bp;
    for (bp = heap_listp; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKP(bp)) {
        if (!GET_ALLOC(HDRP(bp)) && (asize <= GET_SIZE(HDRP(bp)))) {
            return bp;
        }
    }
    /* Best-fit search */
    // void *bp;
    // void *best_bp = NULL;
    // for (bp = heap_listp; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKP(bp)) {
    //     if (!GET_ALLOC(HDRP(bp)) && (asize <= GET_SIZE(HDRP(bp)))) {
    //         if (best_bp == NULL || GET_SIZE(HDRP(bp)) < GET_SIZE(HDRP(best_bp))) {
    //             best_bp = bp;
    //         }
    //     }
    // }
    // if (best_bp != NULL) return best_bp;
    return NULL; /* No fit */
}
/*
 * place - Place block of asize bytes at start of free block bp
 */
static void place(void *bp, size_t asize)
{
    size_t csize = GET_SIZE(HDRP(bp));
    if ((csize - asize) >= (2*DSIZE)) {
        PUT(HDRP(bp), PACK(asize, 1));
        PUT(FTRP(bp), PACK(asize, 1));
        bp = NEXT_BLKP(bp);
        PUT(HDRP(bp), PACK(csize-asize, 0));
        PUT(FTRP(bp), PACK(csize-asize, 0));
    }
    else {
        PUT(HDRP(bp), PACK(csize, 1));
        PUT(FTRP(bp), PACK(csize, 1));
    }
}
```

현재 implicit list 방법을 사용하고 있으므로, 대표적인 placement policy 중 first fit과 best fit을 구현하였다. first fit의 경우, 프롤로그 블록부터 에필로그 블록까지 차례대로 순회하면서 가능한 블록(free 블록이면서 필요한 크기보다 큰 블록) 이 있는 경우 바로 할당 및 리턴한다. best fit의 경우 항상 프롤로그 블록부터 에필로그 불록까지 모두 순회한 뒤, 가능한 블록 중에서 가장 작은 블록에 할당한다. first fit의 경우 속도를, best fit의 경우 메모리 활용률을 중시한 방법이며, 둘의 결과는 마지막 result에 나타내었다. Next fit의 경우 알 수 없는 payload 오류로 인해 구현하지 못했다.

place 함수의 경우, 배치할 위치의 블록이 배치하고자 하는 블록보다 크기가 더욱 큰 경우(4 words 만큼, 즉 새로운 헤더와 푸터를 배치할 여유가 있는 경우) 배치할 위치의 블록을 나눈 후 배치하며, 그렇지 않은 경우 그대로 배치하도록 구현하였다.

8. mm_free(), coalesce()

```
/*
 * mm_free - Free a block, then coalesce it with adjacent free blocks
 */
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);
}
/*
 * coalesce - coalesce adjacent free blocks
 */
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 */
    if (prev_alloc && next_alloc) {
        return bp;
    }
    /* Case 2 */
    else if (prev_alloc &&!next_alloc) {
        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size,0));
    }
    /* Case 3 */
    else if (!prev_alloc && next_alloc) {
        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);
    }
    /* Case 4 */
    else {
        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);
    }
    return bp;
}
```

mm_free() 함수의 동작은 매우 단순하다. 현재 블록의 헤더 및 푸터에 현재 할당 상태를 free로 표시한 후, coalesce() 함수를 호출한다.

coalesce() 함수는 현재 블록과 이전, 이후 블록의 free 상태를 확인해 하나로 합쳐주는 함수이다. 가장 먼저 이전, 이후 블록의 할당 상태와 free 블록의 크기를 저장한 후, 네가지 경우에 따라 알맞게 작동한다.

1. 이전, 이후 블록이 모두 할당된 경우, 병합할 수 없으므로 그대로 리턴한다.
2. 이후 블록만 free인 경우, 블록 크기에 이후 블록의 크기를 더하고, 헤더와 푸터를 재설정한다.
3. 이전 블록만 free인 경우, 블록 크기에 이전 블록의 크기를 더하고, 헤더와 푸터를 재설정한다. 리턴

할 포인터를 이전 블록으로 옮긴다.

4. 이전, 이후 블록이 둘 다 free인 경우, 블록 크기를 전 블록, 현재 블록, 이후 블록의 합으로 바꾼 후 전체에 헤더와 푸터를 설정, 이전 블록을 반환한다.

9. mm_realloc()

```
/*
 * mm_realloc - Implemented simply in terms of mm_malloc and mm_free
 */
void *mm_realloc(void *ptr, size_t size)
{
    /* If size == 0 then this is just free, and we return NULL. */
    if (size ==0) {
        mm_free(ptr);
        return 0;
    }
    /* If oldptr is NULL, then this is just malloc. */
    if (ptr ==NULL) {
        return mm_malloc(size);
    }
    /* If size is not 0 and old ptr is not NULL, change the size of memory */
    size_t oldsize = GET_SIZE(HDRP(ptr));
    size_t newsize = ALIGN(size + DSIZE);
    void *newptr;
    /* If the size of the block is already enough, return the pointer */
    if (GET_SIZE(HDRP(ptr)) >= ALIGN(size + DSIZE)) {
        /* If the size of the block is too large, split the block */
        if (GET_SIZE(HDRP(ptr)) >= ALIGN(size + DSIZE + DSIZE)) {
            PUT(HDRP(ptr), PACK(newsize, 1));
            PUT(FTRP(ptr), PACK(newsize, 1));
            PUT(HDRP(NEXT_BLKP(ptr)), PACK(oldsize - newsize, 0));
            PUT(FTRP(NEXT_BLKP(ptr)), PACK(oldsize - newsize, 0));
        }
        return ptr;
    }
    /* If the next block is free and the size of the block is enough, combine the blocks */
    void *next_block = NEXT_BLKP(ptr);
    if (!GET_ALLOC(HDRP(next_block)) && (oldsize + GET_SIZE(HDRP(next_block)) >= newsize)) {
        size_t combined_size = oldsize + GET_SIZE(HDRP(next_block));
        PUT(HDRP(ptr), PACK(combined_size, 1));
        PUT(FTRP(ptr), PACK(combined_size, 1));
        return ptr;
    }
    newptr = mm_malloc(size);
    /* If realloc() fails the original block is left untouched  */
    if (!newptr) {
        return 0;
    }
    /* Copy the old data. */
    if (size < oldsize) oldsize =size;
    memcpy(newptr, ptr, oldsize);
    /* Free the old block. */
    mm_free(ptr);
    return newptr;
}
```

mm_realloc() 함수는 기존의 동적 메모리 할당의 크기를 변경하는 함수이다. 기존 블록의 크기를 늘리거나 줄일 수 있으며, 내부적으로 mm_malloc() 함수와 mm_free() 함수를 이용한다.

1. 가장 먼저 size가 0인 경우, free() 함수로 간주하여 free() 함수를 실행한다.

2. ptr가 NULL인 경우, mm_malloc() 함수로 간주, 실행한다.

3. 만약 prt가 NULL이 아니고, 크기 변경을 요청한 경우, 새로운 포인터에 새로운 크기의 공간을 할당

받은 후, memcpy() 함수를 통해 데이터를 복사한 후, 기존의 포인터는 free()함수를 통해 할당을 해제하고 새로운 포인터를 반환하였다. 그러나 이때 모든 케이스에 대해서 free() 및 malloc()을 실행하므로, 몇 가지 예외를 처리하여 성능을 최적화하였다.

1) 먼저, 기존 크기보다 적은 크기를 요청, 즉 기존 공간을 줄이려고 시도하는 경우이다. 만약 블록을 나누고 새로운 헤더와 푸터가 들어갈 공간이 충분한 경우, 블록을 나누고, 이전 블록은 줄어든 블록, 이후 블록은 새로운 free() 블록이 된다. 새로운 헤더와 푸터가 들어갈 공간이 적은 경우 기존 포인터를 그대로 반환한다.
2) 요청한 크기가 기존의 크기보다 큰 경우, 만약 현재 다음 블록이 비어있으면서, 두 블록을 합쳤을 때 요청받은 크기를 만족할 수 있다면, 복사 없이 블록을 병합하여 크기를 조정한다.
3) 요청한 크기가 기존의 크기보다 크면서 이후 블록을 합치는 방법이 통하지 않을 경우에만 memcpy() 함수를 사용하여 적절한 위치로 복사하였다.
4) 또한 예외 처리로서, 재 할당에 실패하였을 때 기존 블록은 변경되지 않고 NULL을 반환한다.

10. mm_check()

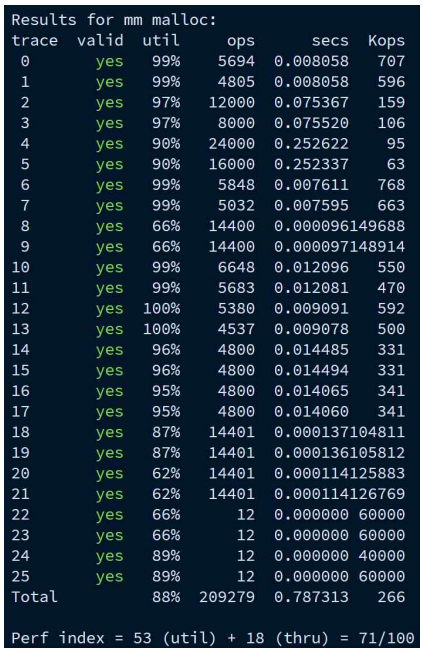
```
/*
 * mm_check - Check the heap for correctness
 */
void mm_check(int verbose)
{
    checkheap(verbose);
}
/*
 * checkheap - Minimal check of the heap for consistency
 */
void checkheap(int verbose)
{
    char *bp = heap_listp;
    if (verbose)
        printf("Heap (%p):\n", heap_listp);
    if ((GET_SIZE(HDRP(heap_listp)) != DSIZE) ||!GET_ALLOC(HDRP(heap_listp)))
        printf("Bad prologue header\n");
    checkblock(heap_listp);
    for (bp = heap_listp; GET_SIZE(HDRP(bp)) >0; bp = NEXT_BLKP(bp)) {
        if (verbose)
            printblock(bp);
        checkblock(bp);
    }
    if (verbose)
        printblock(bp);
    if ((GET_SIZE(HDRP(bp)) !=0) ||!(GET_ALLOC(HDRP(bp))))
        printf("Bad epilogue header\n");
}
static void printblock(void *bp)
{
    size_t hsize, halloc, fsize, falloc;
    checkheap(0);
    hsize = GET_SIZE(HDRP(bp));
    halloc = GET_ALLOC(HDRP(bp));
    fsize = GET_SIZE(FTRP(bp));
    falloc = GET_ALLOC(FTRP(bp));
    if (hsize ==0) {
        printf("%p: EOL\n", bp);
        return;
    }
    printf("%p: header: [%ld:%c] footer: [%ld:%c]\n", bp,
           hsize, (halloc ? 'a' : 'f'),
           fsize, (falloc ? 'a' : 'f'));
}
static void checkblock(void *bp)
{
    if ((size_t)bp % 8)
        printf("Error: %p is not doubleword aligned\n", bp);
    if (GET(HDRP(bp)) != GET(FTRP(bp)))
        printf("Error: header does not match footer\n");
}
```

mm_check() 함수는 디버깅을 위해 메모리 상태를 출력하는 함수이다. mm_check() 함수로 힙 검사를 시작하며, 내부적으로 checkheap() 함수를 호출해 힙 전체를 확인한다. checkheap() 함수는 힙의 시작점인 프로로그 블록에서부터 각 블록을 순회하며 검사하고, 마지막으로 에필로그 블록까지 확인한다. 각 블

록은 checkblock() 함수를 통해 개별적으로 점검되며, 블록 포인터의 정렬 상태와 헤더와 푸터 값의 일치 여부를 확인한다. 만약 오류가 발견되면 적절한 메시지를 출력한다. 검사 과정에서 verbose 옵션이 활성화되어 있으면, printblock() 함수를 통해 블록의 크기와 할당 상태 등의 상세 정보를 출력한다. 프롤로그와 에필로그 블록의 크기와 할당 상태도 검사하여 올바른지 확인한다.

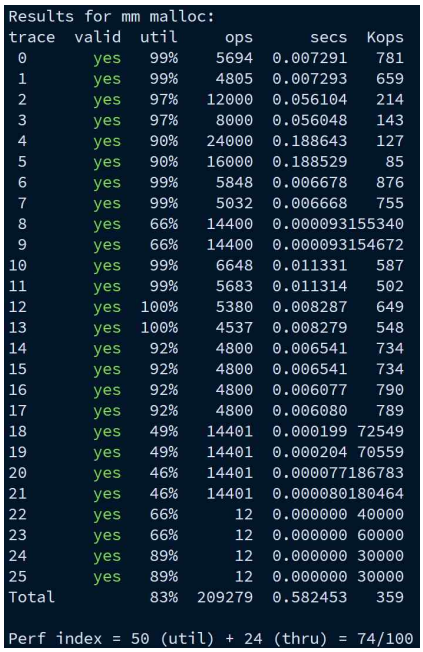
11. Result

Implicit list 방법을 사용하여 최적화하여, 다음과 같은 결과를 얻었다.



```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.008058   707
 1       yes   99%    4805  0.008058   596
 2       yes   97%   12000  0.075367   159
 3       yes   97%    8000  0.075520   106
 4       yes   90%   24000  0.252622    95
 5       yes   90%   16000  0.252337    63
 6       yes   99%    5848  0.007611   768
 7       yes   99%    5032  0.007595   663
 8       yes   66%   14400  0.000096149688
 9       yes   66%   14400  0.000097148914
10       yes   99%    6648  0.012096   550
11       yes   99%    5683  0.012081   470
12       yes  100%    5380  0.009091   592
13       yes  100%    4537  0.009078   500
14       yes   96%    4800  0.014485   331
15       yes   96%    4800  0.014494   331
16       yes   95%    4800  0.014065   341
17       yes   95%    4800  0.014060   341
18       yes   87%   14401  0.000137104811
19       yes   87%   14401  0.000136105812
20       yes   62%   14401  0.000114125883
21       yes   62%   14401  0.000114126769
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 60000
Total          88%  209279  0.787313   266

Perf index = 53 (util) + 18 (thru) = 71/100
```



```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.007291   781
 1       yes   99%    4805  0.007293   659
 2       yes   97%   12000  0.056104   214
 3       yes   97%    8000  0.056048   143
 4       yes   90%   24000  0.188643   127
 5       yes   90%   16000  0.188529    85
 6       yes   99%    5848  0.006678   876
 7       yes   99%    5032  0.006668   755
 8       yes   66%   14400  0.000093155340
 9       yes   66%   14400  0.000093154672
10       yes   99%    6648  0.011331   587
11       yes   99%    5683  0.011314   502
12       yes  100%    5380  0.008287   649
13       yes  100%    4537  0.008279   548
14       yes   92%    4800  0.006541   734
15       yes   92%    4800  0.006541   734
16       yes   92%    4800  0.006077   790
17       yes   92%    4800  0.006080   789
18       yes   49%   14401  0.000199 72549
19       yes   49%   14401  0.000204 70559
20       yes   46%   14401  0.000077186783
21       yes   46%   14401  0.000080180464
22       yes   66%      12  0.000000 40000
23       yes   66%      12  0.000000 60000
24       yes   89%      12  0.000000 30000
25       yes   89%      12  0.000000 30000
Total          83%  209279  0.582453   359

Perf index = 50 (util) + 24 (thru) = 74/100
```

Best Fit　　First Fit

Placement policy로 first fit과 best fit을 사용했는데, best fit의 경우 메모리 활용률이 더 높지만, 속도가 느리게 측정되었다. 반면 fisrt fit의 경우 메모리 활용률은 best fit보다 나쁘지만, 성능 폭이 더욱 큰 것을 확인할 수 있었다. 종합적으로 점수를 비교했을 때, first fit 방법이 더 좋게 나타났다.

Perf. Index	<50	50~59	60~69	70~79	80~89	90~100
Score	10 pts	20 pts	25 pts	30 pts	35 pts	40 pts

주어진 점수 Table에 따라, 코드 40점 중 30점을 획득하였다.

** 위 점수는 AVG_LIBC_THRUPUT을 수정하지 않은 상태에서, 포스텍 프로그래밍 서버를 이용해 확인한 점수이다.

12. Reference

[1] Computer Systems: A Programmer's Perspective, 3/E (CS:APP3e), Randal E. Bryant and David R. O'Hallaron, Carnegie Mellon University.

[2] [CSED211 2024 Fall] Lecture 17 Dynamic Memory Allocation - Basic Concepts, Prof. Jisung Park.

[3] [CSED211 2024 Fall] Lecture 18 Dynamic Memory Allocation – Advanced, Prof. Jisung Park

[4] [Writeup] Malloc Lab: Writing a Dynamic Storage Allocator, Prof. Jisung Park.