CSED211 LAB12 REPORT

20220778 표승현

변수, 함수 및 매크로 선언

```
// define macro
#define WSIZE 4 // word size
#define DSIZE 8 // double word size
#define CHUNKSIZE (1<<12) // 4KiB size of heap to extend

#define MAX(x,y) ((x)>(y)? (x) : (y)) // find bigger one

#define PACK(size,alloc) ((size)| (alloc)) // pack size and alloc bit in a block
```

WSIZE: 워드 크기를 나타낸다.

DSIZE: 더블 워드 크기를 나타낸다.

CHUNKSIZE: 힙을 확장할 때 사용되는 기본적인 크기인 4KiB 를 나타낸다.

MAX(x, y): 두 값을 비교하여 더 큰 값을 반환한다.

PACK(size, alloc): 블록의 크기와 할당 비트를 조합하여 블록의 헤더와 풋터에 저장한다.

```
// from address of p, read and write a word
#define GET(p) (*(unsigned int*)(p)) // read a word
#define PUT(p,val) (*(unsigned int*)(p)=(val)) // write

// from address of p, read a size and allocate some feild
#define GET_SIZE(p) (GET(p) & ~0x7) // read size
#define GET_ALLOC(p) (GET(p) & 0x1) // read alloc

// from address of bp, compute position of header and footer
#define HDRP(bp) ((char*)(bp) - WSIZE) //
#define FTRP(bp) ((char*)(bp) + GET_SIZE(HDRP(bp)) - DSIZE) //

// from address of bp, compute position from next and previous block
#define NEXT_BLKP(bp) ((char*)(bp) + GET_SIZE(((char*)(bp) - WSIZE))) //
#define PREV_BLKP(bp) ((char*)(bp) - GET_SIZE(((char*)(bp) - DSIZE))) //
```

GET(p): 메모리 주소 p 에서 워드를 읽는다.

PUT(p, val): 메모리 주소 p 에 워드 val 을 덮어 쓴다.

GET_SIZE(p): 블록의 헤더에서 크기를 읽는다.

GET_ALLOC(p): 블록의 헤더에서 할당 비트를 읽는다.

HDRP(bp): 블록 포인터 bp 에서 헤더의 위치를 계산한다.

FTRP(bp): 블록 포인터 bp 에서 풋터의 위치를 계산한다.

NEXT_BLKP(bp): 현재 블록 다음 블록의 위치를 계산한다.

PREV_BLKP(bp): 현재 블록 이전 블록의 위치를 계산한다.

```
// Manage heap
static char *heap_listp; //heap pointer

// define functions
static void *extend_heap(size_t size);
static void *coalesce(void* bp);
static void *find_fit(size_t size);
static void place(void *bp, size_t asize);
```

static char *heap_listp: 힙의 시작 주소를 나타낸다.

static void *extend_heap(size_t size): 힙을 확장하는 함수로, 새로운 힙 블록을 할당하고 초기화한다.

static void *coalesce(void *bp): 주어진 블록 주소 bp 를 받아 인접한 빈 블록들을 병합한다.

static void *find_fit(size_t size): 주어진 크기에 가장 적절한 빈 블록을 찾는다.

static void place(void *bp, size_t asize): 할당된 블록의 크기를 조정하고, 남은 부분을 적절히 처리한다.

mm_init()

```
int mm_init(void)
{
    if((heap_listp = mem_sbrk(4*WSIZE)) == (void*)-1){
        return -1;
    }
    PUT(heap_listp,0);
    PUT(heap_listp + (1*WSIZE), PACK(DSIZE,1)); // make prologue header
    PUT(heap_listp + (2*WSIZE), PACK(DSIZE,1)); // make prologue footer
    PUT(heap_listp + (3*WSIZE), PACK(0,1)); // make epilogue block header
    heap_listp += (2*WSIZE); // move pointer between the header and footer

    if (extend_heap(CHUNKSIZE/WSIZE)==NULL) { // extend heap
        return -1;
    }

    return 0;
}
```

mm_init 함수는 메모리 할당기를 초기화하는 역할을 한다. 이 함수는 메모리를 할당하고 초기 힙 구조를 설정하여 할당기를 사용할 준비를 한다. mem_sbrk(4 * WSIZE)를 호출하여 초기 힙을 설정한다. 4 워드의 메모리를 요청한다. 이후 힙 리스트 포인터 heap_listp 가 초기 힙의 시작 지점을 가리키도록 설정된다.

초기 힙에는 프롤로그 블록이라고 불리는 특수한 블록을 생성한다. 이 블록은 힙의 시작 부분에 위치하며, 영역을 표시하는 헤더와 풋터로 이루어져 있다. 초기 힙의 끝에는 에필로그 블록이라고 불리는 특수한 블록이 생성된다. 이 블록은 힙의 끝을 표시하는 헤더로만 이루어져 있다.

extend_heap 함수를 호출하여 초기 힙의 크기를 CHUNKSIZE 로 확장한다. 확장된 힙에는 추가적인 프리 블록이 생성되어, 할당할 메모리를 추가적으로 확보한다. 초기화가 성공하면 0 을 반환하고, 실패하면 -1 을 반환한다.

mm_malloc()

```c
void *mm_malloc(size_t size)
{
    size_t asize;   // Adjusted block size
    char *bp;       // Block pointer

    // Check for exception
    if (size == 0) {
        return NULL;
    }

    // Calculate the adjusted block
    if (size <= DSIZE) {
        // If the requested size is small, allocate a minimum block size
        asize = 2 * DSIZE;
    } else {
        // Align the block size to ensure proper alignment and add the header and footer sizes
        asize = ALIGN(size + DSIZE);
    }

    // Find a free block that best fits the adjusted block size
    bp = find_fit(asize);

    // If no suitable free block is found, extend the heap and allocate a new block
    if (bp == NULL) {
        if ((bp = extend_heap(asize / WSIZE)) == NULL) {
            // Return NULL if extending the heap fails
            return NULL;
        }
        // Place the block in the newly allocated space
        place(bp, asize);
    } else {
        // If a suitable free block is found, place the block in that space
        place(bp, asize);
    }

    // Return the pointer to the allocated block
    return bp;
}
```

mm_malloc 함수는 메모리 할당을 수행하는 역할을 한다. 주어진 size 에 따라 mm_malloc 함수는 메모리 할당 작업을 수행한다. 먼저, 예외 처리를 통해 size 가 0 일 경우, NULL 을 반환하여 예외 상황을 처리한다. 그 후, 요청된 크기를 조정하여 최종 할당할 블록의 크기 asize 를 계산한다. 만약 size 가 DSIZE 보다 작거나 같으면 최소 블록 크기인 2 * DSIZE 로 설정하고, 그렇지 않으면 메모리 정렬을 위해 ALIGN 매크로를 이용하여 크기를 조정한다.

다음으로, 조정된 블록 크기에 맞는 가장 적절한 빈 블록을 찾기 위해 find_fit 함수를 호출한다. 적절한 빈 블록이 없다면, extend_heap 함수를 통해 힙을 확장하고 새로운 블록을 할당한다. 이후, 해당 블록을 찾거나 새로 할당한 블록에 삽입하여 메모리를 할당한다. 마지막으로, 할당된 블록의 포인터를 반환한다.

mm_free()

```c
void mm_free(void *ptr)
{
    size_t size = GET_SIZE(HDRP(ptr));  // Get the size

    PUT(HDRP(ptr), PACK(size, 0));  // Set header for the free block
    PUT(FTRP(ptr), PACK(size, 0));  // Set footer for the free block

    // Coalesce the free block with adjacent free blocks
    coalesce(ptr);
}
```

mm_free 함수는 주어진 포인터 ptr 이 가리키는 블록을 해제하고, 이를 통해 메모리를 반환하는 역할을 한다. 먼저, 블록의 크기를 GET_SIZE 매크로를 이용하여 얻어온다. 그 후, 해당 블록의 헤더와 푸터를 새롭게 할당되지 않은 상태로 설정하기 위해 PUT 매크로를 사용한다. 이렇게 하면 해당 블록이 더 이상 할당되지 않음을 나타내게 된다.

이후, coalesce 함수를 호출하여 인접한 빈 블록들과 현재 해제된 블록을 병합한다. 이를 통해 연속된 여러 빈 블록들이 하나로 합쳐질 수 있고, 메모리의 효율성이 증가하게 된다.

mm_realloc()

```
void *mm_realloc(void *ptr, size_t size)
{
    void *oldptr = ptr;
    void *newptr;
    size_t copySize;

    newptr = mm_malloc(size);
    if (newptr == NULL)
      return NULL;
    copySize = GET_SIZE(HDRP(oldptr));
    if (size < copySize)
      copySize = size;
    memcpy(newptr, oldptr, copySize);
    mm_free(oldptr);
    return newptr;
}
```

mm_realloc 함수는 주어진 포인터 ptr 이 가리키는 블록의 크기를 새로운 크기 size 로 조정한다. 우선, 기존의 블록을 보존하기 위해 oldptr 에 주어진 포인터 ptr 을 복사한다. 그리고 새로운 크기 size 에 맞게 mm_malloc 함수를 호출하여 메모리를 할당받는다. 만약 메모리 할당에 실패하면 NULL 을 반환한다.

다음으로, 기존 블록의 크기를 얻기 위해 GET_SIZE 매크로를 사용하여 copySize 에 저장한다. 새로운 크기 size 가 copySize 보다 작다면, copySize 를 size 로 조정한다. 그리고 memcpy 함수를 사용하여 기존 블록에서 새로운 블록으로 데이터를 복사한다.

마지막으로, 기존 블록을 해제하기 위해 mm_free 함수를 호출하고, 새로 할당받은 블록을 가리키는 포인터 newptr 을 반환한다.

extend_heap()

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

    if((words%2)==0){ // when words size is even
        size = WSIZE*words;
    }
    else{ // case for odd
        size = WSIZE*(words+1);
    }

    if((bp = mem_sbrk(size))==(void*)-1){ // when there is an error extending heap, return NULL
        return NULL;
    }

    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

    return coalesce(bp);
}
```

힙을 확장하는 extend_heap 함수는 주어진 워드 수 words 에 따라 힙을 증가시키는데 사용된다. 먼저, 주어진 words 의 홀수 여부를 확인하여 적절한 크기의 메모리를 할당한다. 만약 words 가 짝수라면 size 에 WSIZE * words 를 할당하고, 홀수라면 size 에 WSIZE * (words + 1)를 할당하여 align 을 맞춰준다.

그 후, mem_sbrk 함수를 호출하여 힙을 확장하고, 확장된 힙의 시작 지점인 블록 포인터 bp 를 얻는다. 만약 힙을 확장하는 과정에서 오류가 발생하면 (mem_sbrk 가 -1 을 반환) NULL 을 반환하여 오류를 나타낸다.

PUT(HDRP(bp), PACK(size, 0)): 확장된 블록의 헤더에 크기와 할당 여부 정보를 free 로 설정한다.

PUT(FTRP(bp), PACK(size, 0)): 확장된 블록의 풋터에도 크기와 할당 여부 정보를 free 로 설정한다.

PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1)): 새로운 에필로그 블록의 헤더를 설정하여 힙의 끝을 나타낸다.

마지막으로, coalesce 함수를 호출하여 새로 할당된 블록과 인접한 빈 블록들을 병합한다. 마지막으로 coalesce 함수에서 반환된 포인터를 extend_heap 함수가 반환한다.

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));

    if (prev_alloc && next_alloc) { /* Case 1: Both previous and next blocks are allocated */
    // No merging is required, return the current block pointer
    return bp;
    }
    else if (prev_alloc && !next_alloc) { /* Case 2: Only the next block is free */
    // Merge the current block with the next free block
    size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
    PUT(HDRP(bp), PACK(size, 0));
    PUT(FTRP(bp), PACK(size, 0));
    }
    else if (!prev_alloc && next_alloc) { /* Case 3: Only the previous block is free */
    // Merge the current block with the previous free block
    size += GET_SIZE(HDRP(PREV_BLKP(bp)));
    PUT(FTRP(bp), PACK(size, 0));
    PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
    // Update the block pointer to the start of the merged block
    bp = PREV_BLKP(bp);
    }
    else { /* Case 4: Both previous and next blocks are free */
    // Merge the current block with both the previous and next free blocks
    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));
    // Update the block pointer to the start of the merged block
    bp = PREV_BLKP(bp);
    }

    // Return the block pointer after potential merging
    return bp;
}
```

주어진 블록 bp 을 받아와 주변의 빈 블록들과의 상태를 확인하고, 필요한 경우 이들을 병합한다. 먼저, prev_alloc 과 next_alloc 변수를 통해 이전 블록과 다음 블록의 할당 여부를 확인한다. 그리고 현재 블록의 크기를 size 에 저장한다. 다음으로, 4 가지 경우에 따라 병합을 수행한다.

Case 1 (prev_alloc && next_alloc): 이전 블록과 다음 블록이 모두 할당된 경우에는 추가적인 병합이 필요하지 않으므로 현재 블록 포인터 bp 를 그대로 반환한다.

Case 2 (!prev_alloc && !next_alloc): 이전 블록이 빈 블록이고, 다음 블록이 빈 블록인 경우에는 현재 블록과 다음 블록을 병합한다. 크기를 업데이트하고, 헤더와 풋터를 갱신한다.

Case 3 (!prev_alloc && next_alloc): 이전 블록이 빈 블록이고, 다음 블록이 빈 블록이 아닌 경우에는 현재 블록과 이전 블록을 병합한다. 크기를 업데이트하고, 헤더와 풋터를 갱신하며, 블록 포인터 bp 를 병합된 블록의 시작으로 업데이트한다.

Case 4 (prev_alloc && !next_alloc): 이전 블록이 할당된 상태이고, 다음 블록이 빈 블록인 경우에는 현재 블록과 다음 블록을 병합한다. 크기를 업데이트하고, 헤더와 풋터를 갱신하며, 블록 포인터 bp 를 병합된 블록의 시작으로 업데이트한다.

최종적으로, coalesce 함수는 병합을 수행한 뒤 병합된 블록의 시작 포인터를 반환하다.

find_fit()

```
static void *find_fit(size_t size)
{
    void *bp;      // Pointer to traverse the heap
    void *best = NULL;  // Pointer to the best fit block found so far
    int flag = 0;   // Flag to indicate whether a suitable block has been found

    // Traverse the heap to find the best fit block
    for (bp = heap_listp; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKP(bp)) {
        // Skip allocated blocks
        if (GET_ALLOC(HDRP(bp)))
            continue;

        // If no suitable block has been found yet
        if (flag == 0) {
            // Check if the current block is large enough
            if (size <= GET_SIZE(HDRP(bp))) {
                best = bp;
                flag++;     // Suitable block has been found
            }
        }
        else {
            // Choose the block with the smallest size that still fits the requested size
            if ((GET_SIZE(HDRP(best)) > GET_SIZE(HDRP(bp))) && (size <= GET_SIZE(HDRP(bp)))) {
                best = bp;  // Update the best fit block
            }
        }
    }

    // Return the pointer to the best fit block or NULL
    return best;
}
```

find_fit 함수는 주어진 크기에 맞는 최적의 빈 블록을 찾는 역할을 한다. find fit method 중 best fit method 를 차용했다. 주어진 크기에 맞는 최적의 빈 블록을 힙을 순회하면서 검색한다. 먼저, bp 가 힙을 끝까지 도달할 때까지 반복문을 수행한다. 각 반복에서 현재 블록이 이미 할당되어 있는지 여부를 확인하고, 할당되어 있으면 건너뛰고 다음 블록으로 이동한다.

최초로 적절한 블록을 찾기 위해 flag 변수를 사용하는데, flag 가 0 인 경우는 아직 적절한 블록을 찾지 못한 상태를 나타낸다. 이때, 현재 블록이 요청한 크기 이상인지 확인하고, 만약 그렇다면 best 에 현재 블록의 포인터를 저장하고 flag 를 증가시켜 적절한 블록을 찾은 것으로 표시한다. 그 이후의 반복에서는 이미 flag 가 1 인 상태로, best 에 저장된 블록보다 더 작은 크기의 블록 중에서 가장 작은 크기의 블록을 선택한다. 이때, 선택된 블록의 크기가 요청한 크기보다 크거나 같아야 한다.

순회를 마치면, best 에는 최적의 빈 블록의 포인터가 저장되어 있다. 이 포인터를 반환하여 최적의 빈 블록을 가리키거나, 적절한 블록을 찾지 못했을 경우 NULL 을 반환한다.

place()

```c
static void place(void *bp, size_t size){
    size_t csize = GET_SIZE(HDRP(bp));  // Get the size of the free block

    // Check if there is enough space to split the block
    if ((csize - size) >= (2 * DSIZE)) {
        // Split the block
        PUT(HDRP(bp), PACK(size, 1));   // Set header
        PUT(FTRP(bp), PACK(size, 1));   // Set footer
        bp = NEXT_BLKP(bp);             // Move next block
        PUT(HDRP(bp), PACK(csize - size, 0));  // Set header for remaining free block
        PUT(FTRP(bp), PACK(csize - size, 0));  // Set footer for remaining free block
    } else {
        // Allocate the entire block
        PUT(HDRP(bp), PACK(csize, 1));   // Set header
        PUT(FTRP(bp), PACK(csize, 1));   // Set footer
    }
}
```

메모리 할당을 위한 place 함수는 주어진 블록에 대해 적절한 크기의 할당 또는 분할을 수행한다. 먼저, 현재 블록의 크기를 csize 에 저장하여 빈 블록의 크기를 확인한다.

그 후, 현재 블록을 분할할 수 있는 여유 공간이 있는지 확인한다. 만약 (csize - size)가 최소 분할 크기 2 * DSIZE 보다 크거나 같다면, 현재 블록을 분할한다. 먼저, 할당된 부분에 대한 헤더와 풋터를 설정하고, 다음 블록으로 이동한다. 나머지 부분에 대한 헤더와 풋터를 설정하여 남은 공간에 대한 빈 블록을 만든다.

만약 현재 블록을 분할할 공간이 부족하다면, 현재 블록 전체를 할당한다. 이를 위해 전체 블록에 대한 헤더와 풋터를 설정하여 해당 블록을 할당 상태로 표시한다.

첫번째 결과

```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.008010   711
 1       yes   99%    4805  0.007995   601
 2       yes   55%   12000  0.085061   141
 3       yes   55%    8000  0.084475    95
 4       yes   51%   24000  0.271699    88
 5       yes   51%   16000  0.272018    59
 6       yes  100%    5848  0.007468   783
 7       yes  100%    5032  0.007457   675
 8       yes   66%   14400  0.000104138462
 9       yes   66%   14400  0.000104138462
10       yes   99%    6648  0.012074   551
11       yes   99%    5683  0.012082   470
12       yes  100%    5380  0.009049   595
13       yes  100%    4537  0.009063   501
14       yes   95%    4800  0.013864   346
15       yes   95%    4800  0.013875   346
16       yes   95%    4800  0.013252   362
17       yes   95%    4800  0.013274   362
18       yes   29%   14401  0.072349   199
19       yes   29%   14401  0.072387   199
20       yes   30%   14401  0.002384  6041
21       yes   30%   14401  0.002383  6042
22       yes   66%      12  0.000000 40000
23       yes   66%      12  0.000000 60000
24       yes   90%      12  0.000000 40000
25       yes   90%      12  0.000000 60000
Total          75%  209279  0.990427   211

Perf index = 45 (util) + 14 (thru) = 59/100
```

best fit 기법을 차용하여 malloc 을 진행한 결과이다. best fit 을 찾아 공간을 효율적으로 활용하기 때문에 util 점수에서 높은 점수를 받았으나, 적절한 블록을 찾는 과정에서 많은 시간이 소요되어 비교적 낮은 thru 점수를 받은 것으로 판단된다.

코드를 추가적으로 개선하기 위해 함수를 살펴보던 중 realloc 함수가 공간을 비효율적으로 활용하고 있다는 사실을 알아내었다.

개선된 mm_realloc()

```
void *oldptr = ptr; // Store the old pointer
void *newptr; // Declare a new pointer for the reallocated block
size_t newSize = size + DSIZE; // Calculate the new size required (considering the overhead for headers and footers)
size_t oldSize = GET_SIZE(HDRP(oldptr)); // Get the current size of the block pointed by oldptr
size_t addsize = oldSize; // Variable to store the cumulative size of contiguous free blocks
int flag = 0;

// If the requested size is less than or equal to the current size, return the old pointer
if (newSize <= oldSize) {
    return oldptr;
}

// Search for contiguous free blocks starting from the oldptr
void *temp = oldptr;
for (temp = oldptr; GET_ALLOC(HDRP(NEXT_BLKP(temp))) == 0; temp = NEXT_BLKP(temp)) {
    // Accumulate the size of contiguous free blocks
    addsize += GET_SIZE(HDRP(NEXT_BLKP(temp)));

    if(newSize <= addsize){ // Break if find sufficient free block
        flag = 1;
        break;
    }
}

// When size is sufficient
if (flag) {
    PUT(HDRP(oldptr), PACK(addsize, 1));
    PUT(FTRP(oldptr), PACK(addsize, 1));
    return oldptr;
}
else{
    // Allocate a new block with the requested size
newptr = mm_malloc(newSize);
if (newptr == NULL) {
    // If mm_malloc fails, return NULL
    return NULL;
}

// Copy the data from the old block to the new block
memcpy(newptr, oldptr, newSize);
mm_free(oldptr);
}

// Return the new pointer
return newptr;
```

먼저, oldptr 에 주어진 포인터 ptr 를 저장하고, 필요한 새로운 크기 newSize 를 계산한다. 이때, 헤더와 풋터의 오버헤드를 고려하여 크기를 조정한다. 현재 블록의 크기는 oldSize 에 저장하고, 추가로 블록과 이어진 연속된 빈 블록의 크기를 누적한 값을 addsize 에 저장한다.

새로운 크기가 현재 크기 이하인 경우에는 더 이상의 작업이 필요 없으므로 oldptr 를 그대로 반환한다. 그 다음, temp 포인터를 사용하여 현재 블록부터 시작하여 연속된 빈 블록을 검색하며, 이들의 크기를 addsize 에 누적한다. 이때, GET_ALLOC(HDRP(NEXT_BLKP(temp)))가 0 인 경우, 즉 다음 블록이 빈 블록인 경우에만 누적한다.

그 후, 만약 다음 블록이 빈 블록이고, 요청한 새로운 크기가 addsize 이하이면, 현재 블록을 확장하여 새로운 크기로 할당한다. 헤더와 풋터를 갱신하고, oldptr 를 반환하여 작업을 마친다.

만약 위의 경우가 아니라면, mm_malloc 함수를 호출하여 새로운 크기에 맞는 블록을 할당받고, memcpy 함수를 사용하여 기존 데이터를 새로운 블록으로 복사한다. 그리고 mm_free 함수를 호출하여 이전에 할당되었던 블록을 해제한다.
마지막으로, 새로운 블록을 가리키는 포인터를 반환하여 메모리 재할당 작업을 완료한다.

기존의 mm_realloc 은 항상 mm_malloc 을 이용하여 메모리를 재할당한다. 이렇게 되면 적절한 빈 블록을 탐색하는 시간이 매 실행마다 소요된다. 더불어 기존 블록과 근접한 빈 블록을 고려하지 않고 새로운 메모리를 할당하기 때문에 최악의 경우 불필요한 heap extention 을 수행하여 메모리 낭비가 발생한다. 따라서 기존의 블록과 인접한 빈 블록을 고려하였을 때, 요청한 사이즈의 메모리를 충분히 할당할 수 있는 경우 해당 위치에 곧바로 메모리를 할당하도록 하였다.

두번째 결과

```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.007994   712
 1       yes   99%    4805  0.007996   601
 2       yes   55%   12000  0.084445   142
 3       yes   55%    8000  0.084273    95
 4       yes   51%   24000  0.269741    89
 5       yes   51%   16000  0.269065    59
 6       yes  100%    5848  0.007476   782
 7       yes  100%    5032  0.007471   674
 8       yes   66%   14400  0.000104138196
 9       yes   66%   14400  0.000104137931
10       yes   99%    6648  0.012114   549
11       yes   99%    5683  0.012056   471
12       yes  100%    5380  0.009036   595
13       yes  100%    4537  0.009050   501
14       yes   95%    4800  0.013709   350
15       yes   95%    4800  0.013706   350
16       yes   95%    4800  0.013304   361
17       yes   95%    4800  0.013291   361
18       yes   39%   14401  0.000206 69908
19       yes   39%   14401  0.000206 70078
20       yes   67%   14401  0.000083173089
21       yes   67%   14401  0.000084170830
22       yes   66%      12  0.000000 60000
23       yes   66%      12  0.000000 60000
24       yes   90%      12  0.000000 40000
25       yes   90%      12  0.000000 40000
Total          79%  209279  0.835516   250

Perf index = 47 (util) + 17 (thru) = 64/100
```

mm_realloc 을 수정한 후 실행 결과이다. 먼저 realloc 과정에서 발생할 수 있는 불필요한 힙 영역 확장을 방지하고, 기존 블록 인근의 빈 블록을 활용한 결과 util 점수의 소폭 증가를 확인할 수 있었다. 또한 realloc 과정에서 불필요한 find_fit 호출을 방지하여 thru 점수에서도 소폭 증가를 확인할 수 있었다.

next_fit method

best fit method 로는 thru 점수에 있어 한계가 있는 것으로 판단하여 next fit method 를 시도해보았다.

```
static char *next_p; //next pointer
```

다음과 같이 마지막으로 탐색했던 블록의 주소를 저장할 next pointer 를 전역변수로 선언한다.

```
void *bp; // Pointer to traverse the heap
void *next = NULL;  // Pointer to the best fit block found so far

if(next_p == NULL){
    next_p = heap_listp;
}

for (bp = next_p; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKP(bp)) {
    // Skip allocated blocks
    if (GET_ALLOC(HDRP(bp)))
        continue;

    if (size <= GET_SIZE(HDRP(bp))){
        next_p = bp;
        next = bp;
        break;
    }
}

return next;
```

find_fit 함수를 next fit method 에 맞게 수정하였다. heap 의 처음부터 탐색하는 것이 아닌 next_p 부터 탐색하여 탐색 시간을 줄인다. 그러나 find fit 만을 수정하면 오류가 발생하게 된다. 블록을 합치거나 새로 할당하는 과정에서 next_p 가 가리키고 있는 블록이 다른 새 블록의 payload 영역을 가리키게 될 수 있다. 이 경우 payload overlap 오류가 발생하게 된다. 따라서 이러한 위험이 발생할 수 있는 모든 경우에 next_p 를 새로 갱신하는 코드를 삽입해야 한다.

```
// Search for contiguous free blocks starting from the oldptr
void *temp = oldptr;
for (temp = oldptr; GET_ALLOC(HDRP(NEXT_BLKP(temp))) == 0; temp = NEXT_BLKP(temp)) {
    // Accumulate the size of contiguous free blocks
    addsize += GET_SIZE(HDRP(NEXT_BLKP(temp)));
    if(temp == next_p){
        flag_over = 1;
    }

    if(newSize <= addsize){ // Break if find sufficient free block
        flag = 1;
        break;
    }
}

// When size is sufficient
if (flag) {
    PUT(HDRP(oldptr), PACK(addsize, 1));
    PUT(FTRP(oldptr), PACK(addsize, 1));
    if(flag_over==1){
        next_p = oldptr;
    }
    return oldptr;
}
```

먼저 mm_realloc 함수이다. oldptr 의 근접한 빈 블록을 찾아 활용하는 과정에서 next_p 가 가리키고 있던 블록이 새로운 블록에 덮어 씌워질 수 있다. 따라서 이 경우 flag 를 설정해주고 next_p 를 oldptr 로 갱신한다.

```c
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));

    if (prev_alloc && next_alloc) { /* Case 1: Both previous and next blocks are allocated */
    // No merging is required, return the current block pointer
    next_p=bp;
    return bp;
    }
    else if (prev_alloc && !next_alloc) { /* Case 2: Only the next block is free */
    // Merge the current block with the next free block
    size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
    PUT(HDRP(bp), PACK(size, 0));
    PUT(FTRP(bp), PACK(size, 0));
    }
    else if (!prev_alloc && next_alloc) { /* Case 3: Only the previous block is free */
    // Merge the current block with the previous free block
    size += GET_SIZE(HDRP(PREV_BLKP(bp)));
    PUT(FTRP(bp), PACK(size, 0));
    PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
    // Update the block pointer to the start of the merged block
    bp = PREV_BLKP(bp);
    }
    else { /* Case 4: Both previous and next blocks are free */
    // Merge the current block with both the previous and next free blocks
    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));
    // Update the block pointer to the start of the merged block
    bp = PREV_BLKP(bp);
    }

    next_p = bp;
    // Return the block pointer after potential merging
    return bp;
}
```

mm_coalesce 함수도 수정을 해야한다. 인접한 빈 블록을 합치는 과정에서 next_p 가 가리키고 있는 블록이 다른 블록에 의해 덮어 씌워질 수 있다. 따라서 pointer 를 반환하기 전에 next_p 를 유효한 주소를 담을 수 있도록 갱신하는 코드를 삽입한다.

```c
static void place(void *bp, size_t size){
    size_t csize = GET_SIZE(HDRP(bp));  // Get the size of the free block

    // Check if there is enough space to split the block
    if ((csize - size) >= (2 * DSIZE)) {
        // Split the block
        PUT(HDRP(bp), PACK(size, 1));   // Set header
        PUT(FTRP(bp), PACK(size, 1));   // Set footer
        bp = NEXT_BLKP(bp);             // Move next block
        next_p = bp;
        PUT(HDRP(bp), PACK(csize - size, 0));  // Set header for remaining free block
        PUT(FTRP(bp), PACK(csize - size, 0));  // Set footer for remaining free block
    } else {
        next_p = bp;
        // Allocate the entire block
        PUT(HDRP(bp), PACK(csize, 1));   // Set header
        PUT(FTRP(bp), PACK(csize, 1));   // Set footer
    }
}
```

마지막으로 place 함수이다. 빈 블록이 split 되는 과정에서 next_p 이 자연스럽게 다음 빈 블록을 가리킬 수 있도록 갱신하는 코드를 수정한다.

세번째 결과

```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   86%    5694  0.000074 76842
 1       yes   86%    4805  0.000060 79950
 2       yes   55%   12000  0.000145 82702
 3       yes   55%    8000  0.000102 78355
 4       yes   51%   24000  0.000271 88561
 5       yes   51%   16000  0.000184 87193
 6       yes   90%    5848  0.000078 74497
 7       yes   90%    5032  0.000066 76474
 8       yes   66%   14400  0.000089161980
 9       yes   66%   14400  0.000088163823
10       yes   94%    6648  0.000094 71026
11       yes   94%    5683  0.000080 70684
12       yes   95%    5380  0.000079 68448
13       yes   95%    4537  0.000067 67515
14       yes   84%    4800  0.000739  6494
15       yes   84%    4800  0.000740  6489
16       yes   82%    4800  0.000752  6380
17       yes   82%    4800  0.000751  6390
18       yes   45%   14401  0.000250 57535
19       yes   45%   14401  0.000251 57306
20       yes   53%   14401  0.000071201695
21       yes   53%   14401  0.000072200014
22       yes   66%      12  0.000000 40000
23       yes   66%      12  0.000000 60000
24       yes   90%      12  0.000000 60000
25       yes   90%      12  0.000000 40000
Total          74%  209279  0.005105 40993

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

util 점수는 소폭 줄어들었으나 thru 점수에서 상당한 개선을 보였다. best fit method 는 항상 모든 블록을 검사해야 하기 때문에 search 하는 과정에서 시간이 많이 소요되었기 때문에 이런 결과를 보인 것으로 판단된다.