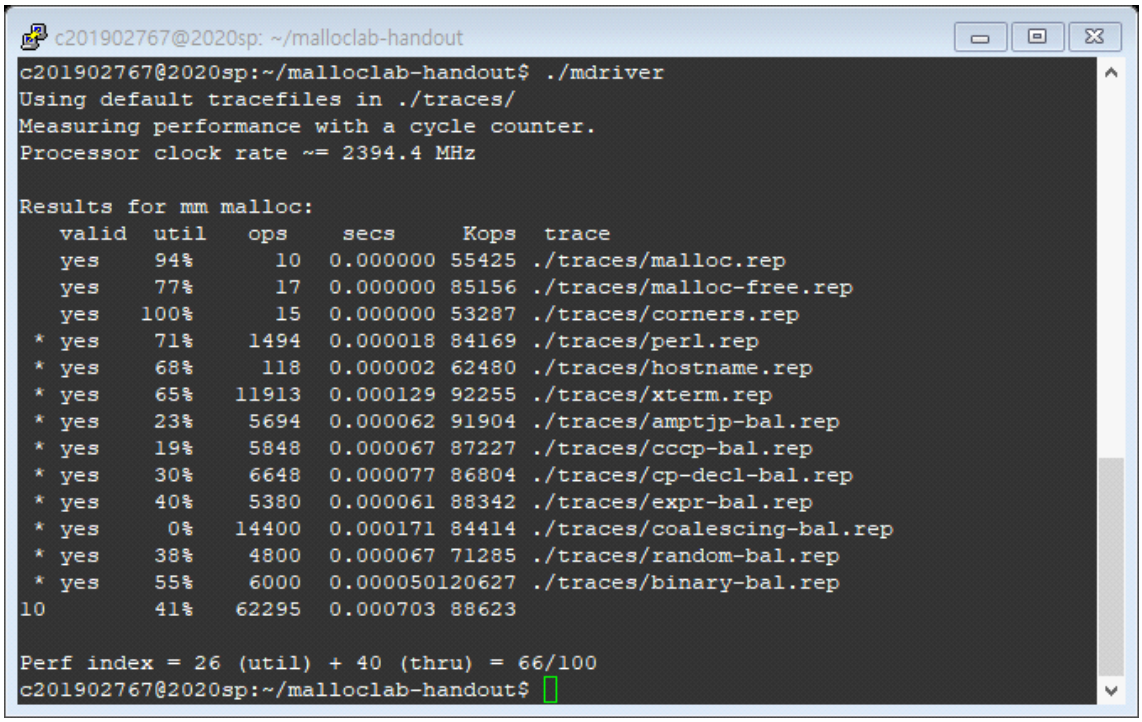
# 2020 시스템 프로그래밍
# - Malloc Lab -

| 제출일자 | 2020.12.16 |
| --- | --- |
| 분 반 | 02 |
| 이 름 | 한민수 |
| 학 번 | 201902767 |

## Naive




## 매크로 및 함수 설명

```
42  /* single word (4) or double word (8) alignment */
43  #define ALIGNMENT 8
44
45  /* rounds up to the nearest multiple of ALIGNMENT */
46  #define ALIGN(size) (((size) + (ALIGNMENT-1)) & ~0x7)
47
48
49  #define SIZE_T_SIZE (ALIGN(sizeof(size_t)))
50
51  #define SIZE_PTR(p)  ((size_t*)(((char*)(p)) - SIZE_T_SIZE))
```

- ALIGNMENT : 아래 ALIGN(size) 함수에서 할당할 크기인 size를 8의 배수로 맞춰서 할당하도록 하기 위한 매크로로 해석하였다.

- ALIGN(size) : 할당할 크기인 size를 보고 8의 배수 크기로 할당하기 위해 size를 다시 align하는 작업을 한다. 만약 size가 4이면 (4+8-1) = 11 = 0000 1011 이고 이를 ~0x7 = 1111 1000 과 AND 연산하면 0000 1000 = 8 이 되므로 적당한 8의 배수 크기로 align할 수 있음을 확인할 수 있다.

- SIZE_T_SIZE : long형 size_t의 크기만큼 align하여 8을 나타내는 매크로로 해석하였다.

- SIZE_PTR(p) : 할당된 크기를 얻기 위한 매크로. 할당된 블록의 맨 앞부분에 접근하여 크기 정보를 얻어낸다. p를 포인터처럼 이동시키기 위해 (char*)로 캐스팅해 준 다음 SIZE_T_SIZE 만큼 앞으로 이동시켜서 얻은 크기를 (size_t*)로 캐스팅하여 반환한다.

> mm_init

```
56  int mm_init(void)
57  {
58    return 0;
59  }
```

mm_init은 초기 heap을 구현하는 함수이지만 naive에서는 특별한 구현 없이 0만을 리턴하고 있다.

> malloc

```
65  void *malloc(size_t size)
66  {
67    int newsize = ALIGN(size + SIZE_T_SIZE);
68    unsigned char *p = mem_sbrk(newsize);
69    //dbg_printf("malloc %u => %p\n", size, p);
70
71    if ((long)p < 0)
72      return NULL;
73    else {
74      p += SIZE_T_SIZE;
75      *SIZE_PTR(p) = size;
76      return p;
77    }
78  }
```

파라미터 size 크기의 메모리를 할당하는 함수이다. newsize 변수에 ALIGN을 이용하여 size + SIZE_T_SIZE 만큼의 크기를 저장한다. SIZE_T_SIZE를 더한 이유는 블록의 앞에 할당된 크기를 기록하는 헤더를 위한 공간을 추가로 할당하기 위해서이다. 그 사이즈가 8byte이기 때문에 SIZE_T_SIZE를 사용하였다. mem_sbrk 함수를 사용하여 newsize 만큼을 힙에 할당한 다음 이전의 mem_brk를 가리키는 포인터를 *p에 저장한다.

그리고 나서 p의 사이즈가 0보다 작으면 할당되지 않았으므로 NULL을 리턴하고, 할당되었으면 p의 위치를 8만큼 옮겨서 할당한 블록의 헤더 위치에 할당한 크기인 size를 기록한 다음 p를 리턴한다.

> free

```
84  void free(void *ptr)
85  {
86  }
```

naive에서는 할당한 블록을 다시 가용 블록으로 만드는 free 함수를 구현하지 않는다. 따라서 naive는 할당은 하지만 다시 반환하지 않으므로 처리율(throughput)은 좋지만 메모리 공간 사용률(utilization)은 효율이 나쁠 것이다.

> realloc

```
93  void *realloc(void *oldptr, size_t size)
94  {
95    size_t oldsize;
96    void *newptr;
97
98    /* If size == 0 then this is just free, and we return NULL. */
99    if(size == 0) {
100     free(oldptr);
101     return 0;
102   }
```

```
104   /* If oldptr is NULL, then this is just malloc. */
105   if(oldptr == NULL) {
106     return malloc(size);
107   }
108
109   newptr = malloc(size);
110
111   /* If realloc() fails the original block is left untouched  */
112   if(!newptr) {
113     return 0;
114   }
115
116   /* Copy the old data. */
117   oldsize = *SIZE_PTR(oldptr);
118   if(size < oldsize) oldsize = size;
119   memcpy(newptr, oldptr, oldsize);
120
121   /* Free the old block. */
122   free(oldptr);
123
124   return newptr;
125 }
```

realloc 함수는 이미 할당된 메모리 블록의 크기를 다시 변경하는 함수이다. oldptr 파라미터는 크기를 변경할 할당되어 있는 블록의 포인터이고, size는 변경할 크기이다. size_t 타입의 oldsize와 void형 newptr 포인터 변수를 선언한다.

그리고 나서, 인자에 따라 조건문을 수행한다. 첫째로, size가 0 이면 할당을 하지 않겠다는 의미이므로 oldptr가 가리키는 블록을 가용 블록으로 만들어주고 0을 리턴한다. 둘째로, oldptr가 NULL이면 해당 공간에 malloc(size)를 호출하여 할당 작업을 한다. 이미 할당되어 있으면 line 109에서와 같이 malloc(size)를 호출한 다음 제대로 할당이 되었다면 oldsize에 할당되어 있던 블록의 크기를 저장하고, 만약 oldsize가 할당하려 하는 size보다 크다면 size를 oldsize에 저장한 다음(크지 않으면 oldsize는 변하지 않는다) memcpy 함수로 newptr 블록에 oldptr 블록의 내용을 oldsize만큼 복사한다. 그리고 나서 oldptr가 가리키는 블록을 fr 함수로 가용 블록으로 만든 다음 새롭게 할당한 블록을 가리키는 포인터 newptr를 리턴한다.

> calloc

```
130 void *calloc (size_t nmemb, size_t size)
131 {
132   size_t bytes = nmemb * size;
133   void *newptr;
134
135   newptr = malloc(bytes);
136   memset(newptr, 0, bytes);
137
138   return newptr;
139 }
```

calloc 함수는 malloc과 비슷하지만 할당한 공간의 값을 모두 0으로 초기화하는 역할을 한다. bytes 변수에 nmemb와 size를 곱한 값을 저장한 다음, malloc 함수를 사용하여 bytes만큼의 크기를 할당하여 newptr에 할당한 공간의 포인터를 저장한다. 그다음, memset 함수를 사용하여 newptr부터 bytes 만큼의 공간을 0으로 세팅한 후 newptr를 리턴한다.

> mem_sbrk

```
51  void *mem_sbrk(int incr)
52  {
53      char *old_brk = mem_brk;
54
55      if ( (incr < 0) || ((mem_brk + incr) > mem_max_addr)) {
56      errno = ENOMEM;
57      fprintf(stderr, "ERROR: mem_sbrk failed. Ran out of memory...\n");
58      return (void *)-1;
59      }
60      mem_brk += incr;
61      return (void *)old_brk;
62  }
```

힙 메모리를 더 할당받기 위해 사용하는 mem_sbrk 함수는 memlib.c에 구현되어 있다. 파라미터 incr로부터 인자를 받는다. 먼저 현재 힙 메모리의 마지막 주소를 가리키는 mem_brk를 old_brk에 저장한다. 그리고 나서 mem_brk + incr 값이 힙 메모리의 최대 주소인 mem_max_addr를 넘어가지 않으면 mem_brk를 incr만큼 증가시키고 원래 mem_brk 주소였던 old_brk를 리턴한다.

## Implicit

```
c201902767@2020sp: ~/malloclab-handout
c201902767@2020sp:~/malloclab-handout$ ./mdriver
Using default tracefiles in ./traces/
Measuring performance with a cycle counter.
Processor clock rate ~= 2394.4 MHz

Results for mm malloc:
   valid  util    ops     secs      Kops  trace
   yes     34%       10  0.000000 20677 ./traces/malloc.rep
   yes     28%       17  0.000000 35334 ./traces/malloc-free.rep
   yes     96%       15  0.000001 22088 ./traces/corners.rep
 * yes     86%     1494  0.002263   660 ./traces/perl.rep
 * yes     75%      118  0.000023  5112 ./traces/hostname.rep
 * yes     91%    11913  0.124396    96 ./traces/xterm.rep
 * yes     99%     5694  0.011998   475 ./traces/amptjp-bal.rep
 * yes     99%     5848  0.011135   525 ./traces/cccp-bal.rep
 * yes     99%     6648  0.017926   371 ./traces/cp-decl-bal.rep
 * yes    100%     5380  0.013285   405 ./traces/expr-bal.rep
 * yes     66%    14400  0.000322 44788 ./traces/coalescing-bal.rep
 * yes     93%     4800  0.010660   450 ./traces/random-bal.rep
 * yes     55%     6000  0.027818   216 ./traces/binary-bal.rep
10         86%    62295  0.219826   283

Perf index = 56 (util) + 11 (thru) = 67/100
c201902767@2020sp:~/malloclab-handout$
```

## 구현 방법

```
38  /* single word (4) or double word (8) alignment */
39  #define ALIGNMENT 8
40
41  /* rounds up to the nearest multiple of ALIGNMENT */
42  #define ALIGN(p) (((size_t)(p) + (ALIGNMENT-1)) & ~0x7)
43
44  /* Basic constants and macros */
45  #define WSIZE       4       /* word와header/footer의사이즈*/
46  #define DSIZE       8       /* double word 사이즈*/
47  #define CHUNKSIZE   (1<<12) /* 힙의 확장 사이즈 */
48
49  #define MAX(x, y)   ((x) > (y)? (x) : (y))  /* x, y 중 큰 값을 구하는 매크로함수 */
50
51  /* Pack a size and allocated bit into a word */
52  #define PACK(size, alloc)   ((size) | (alloc))  /* size와alloc(할당여부) 값을 하나의 word로 묶는다 */
53
54  /* Read and write a word at address p */
55  #define GET(p)      (*(unsigned int *)(p))  /* 포인터 p가 가리키는 주소의 값을 읽는다 */
56  #define PUT(p, val) (*(unsigned int *)(p) = (val))  /* 포인터 p가 가리키는 주소에 val 값을 쓴다 */
57
58  /* Read the size and allocated fields from address p */
59  #define GET_SIZE(p)     (GET(p) & ~0x7)     /* 포인터 p가 가리키는 주소의 값의 하위 3비트를버려서  header에서의 block size를 읽는다 */
60  #define GET_ALLOC(p)    (GET(p) & 0x1)      /* 포인터 p가 가리키는 주소의 값의 하위 1비트를 읽어서 header에서의 alloc bit를 읽는다. 0
61
62  /* Given block ptr bp, compute address of its header and footer */
63  #define HDRP(bp)    ((char*)(bp) - WSIZE)   /* 포인터 bp의 header 주소를 계산한다 */
64  #define FTRP(bp)    ((char*)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)  /* 포인터 bp의 footer 주소를 계산한다 */
65
66  /* Given block ptr bp, compute address of next and previous blocks */
67  #define NEXT_BLKP(bp)   ((char*)(bp) + GET_SIZE(((char*)(bp) - WSIZE))) /* 포인터 bp를 이용해서 다음 블록의 주소를 얻는다 */
68  #define PREV_BLKP(bp)   ((char*)(bp) - GET_SIZE(((char*)(bp) - DSIZE))) /* 포인터 bp를 이용해서 이전 블록의 주소를 얻는다 */
69
70  static char *heap_listp;    /* Heap pointer */
71  static void *coalesce(void *bp);
72  static void *extend_heap(size_t words);
```

매크로 상수 및 함수들의 설명은 다음과 같다.

- WSIZE : 1 word 만큼의 크기. 즉, 4를 나타낸다.

- DSIZE : double word 만큼의 크기. 즉, 8을 나타낸다.

- CHUNKSIZE : 초기 힙의 크기를 나타내며 그 값은 (1<<12) = 2^12 이다.

- MAX(x, y) : x, y 중 큰 값을 구한다.

- PACK(size, alloc) : size와 alloc을 묶어서 나중에 헤더나 푸터에 저장할 수 있게 한다. size와 alloc의 OR연산을 한다.

- GET(p) : p가 가리키는 곳의 word 크기만큼의 값을 얻는다.

- PUT(p, val) : p가 가리키는 곳에 val 값을 기록한다.

- GET_SIZE(p) : 포인터 p가 가리키는 주소의 값의 하위 3비트를 버려서 헤더나 푸터의 block size를 얻는다.

- GET_ALLOC(p) : 포인터 p가 가리키는 주소의 하위 1비트를 읽어서 헤더나 푸터의 할당 여부(alloc)를 읽는다. 0이면 가용 블록, 1이면 할당 블록임을 나타낸다.

- HDRP(bp), FTRP(bp) : 각각 블록 포인터 bp를 가지고 블록 헤더와 푸터의 주소를 얻는다. bp 주소에서 4 만큼 뺌으로써 헤더를, bp 주소에서 블록의 크기 - 8 만큼 더함으로써 푸터 주소를 얻을 수 있다.

- NEXT_BLKP(bp), PREV_BLKP(bp) : 포인터 bp를 가지고 각각 다음 블록과 이전 블록의 주소를 얻는다. 다음 블록은, bp 주소에 bp의 헤더에 쓰여진 크기값을 더함으로써, 이전 블록은, bp 주소에서 이전 블록의 푸터에 쓰여진 크기를 뺌으로써 주소를 구할 수 있다.

- 전역변수 heap_listp : 초기 heap을 구성할 때 쓰이기 위한 전역 변수이다.

> mm_init

```
77  int mm_init(void) {
78      /* Create the initial empty heap */
79      if ((heap_listp = mem_sbrk(4*WSIZE)) == (void*)-1)
80          return -1;
81      PUT(heap_listp, 0);                          /* Alignment padding */
82      PUT(heap_listp + (1*WSIZE), PACK(DSIZE, 1));/* Prologue header */
83      PUT(heap_listp + (2*WSIZE), PACK(DSIZE, 1));/* Prologue footer */
84      PUT(heap_listp + (3*WSIZE), PACK(0, 1));    /* Epilogue header */
85      heap_listp += (2*WSIZE);
86
87      /* Extend the empty heap with a free block of CHUNKSIZE bytes */
88      if (extend_heap(CHUNKSIZE/WSIZE) == NULL)
89          return -1;
90      return 0;
91  }
```

mm_init 함수는 초기 힙을 구성하는 함수이다. 먼저 mem_sbrk 함수에 4*WSIZE를 인자로 전달하여 16바이트만큼 힙 공간을 늘린 다음 힙의 시작부분을 heap_listp에 저장한다. 정상적으로 작업이 완료되면, 첫 4바이트 공간에는 0을, 두 번째와 세 번째 4바이트 공간에는 PACK(DSIZE, 1) 값을 기록한다. 마지막 4바이트 공간에는 PACK(0,1) 값을 기록한 다음 heap_listp를 2*WSIZE = 8만큼의 값과 더해준다. 마지막으로 extend_heap 함수를 호출하여 정상적으로 작업이 완료되면 0을 리턴한다.

> extend_heap

```
175  static void *extend_heap(size_t words){
176      char *bp;
177      size_t size;
178
179      /* Allocate an even number of words to maintain alignment */
180      size = (words % 2) ? (words+1) * WSIZE : words * WSIZE; // size를 짝수로 변환
181      if ((long)(bp = mem_sbrk(size)) == -1)  // 공간 확장 실패
182          return NULL;
183
184      // 확장 성공한 경우 블록의 헤더에 size, alloc 기록
185      /* Initialize free block header/footer and the epilogue header */
186      PUT(HDRP(bp), PACK(size, 0));       /* Free block header */
187      PUT(FTRP(bp), PACK(size, 0));       /* Free block footer */
188      PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1));   /* New epilogue header */
189
190      /* Coalesce if the previous block was free */
191      return coalesce(bp);
192  }
```

extend_heap 함수는 힙의 크기를 확장해주는 함수이며, words 파라미터에 인자를 받는다.

먼저 char형 bp 포인터 변수와, size_t 타입의 size 변수를 선언한 다음 words를 짝수 바이트 값으로 변환한다. 그다음, mem_sbrk(size)를 호출하여 힙 공간을 확장하는 작업에 성공하면, bp의 헤더와 푸터에 size와 가용블록임을 나타내는 0을 PACK하여 기록하고, 다음 블록의 헤더에 PACK(0,1)을 기록한다. 그다음, coalesce 함수를 호출하게 된다.

> coalesce

```
109  static void *coalesce(void *bp){
110      size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
111      size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
112      size_t size = GET_SIZE(HDRP(bp));
113
114      // Case 1: 이전 블럭, 다음 블럭 최하위 bit가 둘 다 1인 경우(할당) 블럭 병합 없이 bp return
115      if (prev_alloc && next_alloc){
116          return bp;
117      }
118      // Case 2: 이전 블럭 최하위 bit가 1이고(할당), 다음 블럭 최하위 bit가 0인 경우(비할당) 다음 블럭
119      else if (prev_alloc && !next_alloc){
120          size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
121          PUT(HDRP(bp), PACK(size, 0));
122          PUT(FTRP(bp), PACK(size, 0));
123      }
124
125      // Case 3: 이전 블럭 최하위 bit가 0이고(비할당), 다음 블럭 최하위 bit가 1인 경우(할당) 이전 블럭
126      else if (!prev_alloc && next_alloc){
127          size += GET_SIZE(HDRP(PREV_BLKP(bp)));
128          PUT(FTRP(bp), PACK(size, 0));
129          PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
130          bp = PREV_BLKP(bp);
131      }
132      // Case 4: 이전 블럭, 다음 블럭 최하위 bit가 둘 다 0인 경우(비할당) 이전 블럭, 현재 블럭, 다음 블
133      else {
134          size += GET_SIZE(HDRP(PREV_BLKP(bp))) + GET_SIZE(FTRP(NEXT_BLKP(bp)));
135          PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
136          PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
137          bp = PREV_BLKP(bp);
138      }
139      return bp;
140  }
```

coalesce 함수는 블록을 가용 블록으로 바꿀 때 가용 블록들끼리 합치는 작업을 하는 함수이다. 네 가지 case가 존재하며 현재 블록은 bp 포인터 변수가 가리키고 있다.

첫째로, 이전 블록과 다음 블록이 모두 할당 블록인 경우 블록을 합치는 과정 없이 bp를 그대로 리턴한다.

둘째로, 이전 블록은 할당 상태이고 다음 블록은 가용 블록이면 다음 블록과 병합한 뒤 bp를 리턴한다. 병합은, 현재 블록의 헤더에 현재 블록 크기 + 다음 블록 크기(다음 블록의 헤더로부터 구함) 값을 기록한 다음, 헤더와 푸터에 크기를 업데이트함으로써 구현한다.

셋째로, 이전 블록이 가용 블록이고 다음 블록이 할당 블록이면 이전 블록과 병합한 뒤 이전 블록의 주소를 bp에 저장하여 리턴한다. 병합은, 현재 블록의 헤더에 현재 블록 크기 + 이전 블록 크기(이전 블록의 헤더로부터 구함) 값을 기록한 다음, 이전 블록의 헤더에 size를 업데이트하고 현재 블록의 푸터에 사이즈를 업데이트 함으로써 구현한다.

마지막으로, 이전 블록과 다음 블록이 모두 가용 블록인 경우, 이전 블록과 다음 블록, 현재 블록을 하나의 블록으로 병합한 뒤 이전 블록의 주소를 bp에 저장하여 리턴한다. 병합은, 이전 블록과 다음 블록의 크기 + 현재 블록의 크기를 size에 저장하고 이전 블록의 헤더부분에 size를, 다음 블록의 푸터 부분에 size를 업데이트함으로써 구현한다.

> find_fit

```
142  static void *find_fit(size_t asize){
143      /* First-fit search */
144      void *bp;
145      // 힙의 마지막 부분에 도달할때까지 반복
146      for (bp = heap_listp; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKP(bp)){
147          // 가용블록이거나 사이즈가 같거나 큰 블록이면 찾은 위치는 bp이며 이를 리턴한다.
148          if(!GET_ALLOC(HDRP(bp)) && (asize <= GET_SIZE(HDRP(bp)))){
149              return bp;
150          }
151      }
152      return NULL; /* No fit */
153  }
```

find_fit 함수는 할당할 블록을 최초 할당 방식으로 찾는 함수이다. 힙의 처음부터 마지막 부분에 도달할 때까지, 가용 블록이거나 사이즈가 asize와 같거나 큰 블록을 찾다가, 조건에 부합하는 블록을 찾으면 해당 블록의 주소를 리턴한다. 알맞은 블록이 없다면 NULL을 리턴하고 추가 공간을 요청해야 한다.

> place

```
155  static void place(void *bp, size_t asize){
156      size_t csize = GET_SIZE(HDRP(bp));
157
158      // 현재 free block이 요청된 할당 크기보다 2*DSIZE보다 큰 경우
159      // free block에 사용할 만큼의 블록에만 alloc 1로 세팅
160      // 그리고 나머지 블록을 잘라서 free block으로 만든다
161      if((csize - asize) >= (2*DSIZE)){
162          PUT(HDRP(bp), PACK(asize, 1));
163          PUT(FTRP(bp), PACK(asize, 1));
164          bp = NEXT_BLKP(bp);
165          PUT(HDRP(bp), PACK(csize-asize, 0));
166          PUT(FTRP(bp), PACK(csize-asize, 0));
167      }else{
168          PUT(HDRP(bp), PACK(csize, 1));
169          PUT(FTRP(bp), PACK(csize, 1));
170      }
171  }
```

place 함수는 할당할 블록을 찾았을 경우 실제로 할당을 진행하는 함수이다. 할당할 블록을

가리키는 포인터 bp와, 할당할 크기를 갖는 asize를 파라미터로 갖는다. 먼저 csize 변수에 할당할 블록의 크기를 헤더로부터 얻어서 저장한다. 할당할 블록의 크기가 할당할 크기인 asize보다 DSIZE * 2(=16바이트) 이상 크면, 외부 단편화를 방지하기 위해 bp의 헤더와 푸터에 사이즈를 asize만 기록하고 alloc bit = 1로 세팅하여 할당했음을 기록한 다음, 다음 블록의 헤더와 푸터에 csize-asize 값을 업데이트하고 alloc 상태를 0으로 만든다. 만약 가용블록을 자를 필요가 없는 조건이면 할당할 블록의 헤더와 푸터에 csize를 그대로 기록하고 alloc 상태를 1로 만드는 것으로 충분하다.

> mm_free

```
96   void mm_free(void *bp){
97       if(bp == 0) return; // 잘못된 free 요청인 경우 함수 종료. 이전 프로시저로 return
98       size_t size = GET_SIZE(HDRP(bp));   // bp의 header에서 block size를 읽어온다.
99
100      // 실제로 데이터를 지우는 것이 아니라
101      // header, footer의 최하위 1bit(1, 할당된 상태)만을 0으로 수정
102
103      PUT(HDRP(bp), PACK(size, 0));   // bp의 header에 block size와 alloc = 0을 저장
104      PUT(FTRP(bp), PACK(size, 0));   // bp의 footer에 block size와 alloc = 0을 저장
105
106      coalesce(bp);   // 주위에 빈 블록이 있을 시 병합한다
107  }
```

mm_free 함수는 할당 블록을 다시 가용 블록으로 만드는 함수이다. 파라미터 *bp를 통해 free 시킬 블록 포인터를 전달받는다. 만약 잘못된 블록을 free 요청할 경우 함수를 종료한다. 그렇지 않으면, 반환할 블록의 헤더로부터 블록 크기를 얻어서 size 변수에 저장한다. 그 다음, 블록의 헤더와 푸터에 alloc bit를 0으로 세팅하여 가용 블록임을 나타내 준다. free 과정은 실제로 블록에 저장된 데이터를 지울 필요 없이 헤더, 푸터의 alloc bit를 업데이트하는 것으로 충분한데, 그 이유는 가용 블록임만 확인이 되면 나중에 블록에 할당할 때 새로운 값을 쓰기만 하면 되기 때문이다. 이 작업을 완료하면 앞뒤로 병합할 수 있는 가용 블록을 병합해 주는 coalesce 함수를 호출한다.

> mm_malloc

```
195  void *mm_malloc(size_t size){
196      size_t asize;       /* Adjusted block size */
197      size_t extendsize;  /* Amount to extend heap if no fit */
198      char *bp;
199
200      // 할당 사이즈가 0이면 할당하지 않는다.
201      if (size == 0)
202          return NULL;
203
204      // 할당하고자 하는 크기가 8보다 작으면 asize = 16 byte로 함
205      if (size <= DSIZE)
206          asize = 2*DSIZE;
207      else
208          // 8보다 크면 아래 만큼의 사이즈를 asize로 함
209          asize = DSIZE * ((size + (DSIZE) + (DSIZE-1)) / DSIZE);
210
211      // 힙에서 할당할만한 블록을 찾은 다음 찾았으면 place 함수로 할당
212      if ((bp = find_fit(asize)) != NULL){
213          place(bp, asize);
214          return bp;
215      }
216
217      // 할당할만한 블록을 찾지 못했으면 asize와 CHUNKSIZE 중 큰 값을 찾는다.
218      extendsize = MAX(asize, CHUNKSIZE);
219      if ((bp = extend_heap(extendsize/WSIZE)) == NULL)
220          return NULL;
221      // 확장하여 커진 힙 위치에 할당한다.
222      place(bp, asize);
223      return bp;
224  }
```

지금까지 구현한 함수들을 사용하여 최종적으로 동적 메모리를 할당하는 작업을 총괄하는 malloc 함수를 구현하였다. asize는 할당하고자 하는 size를 가지고 실제 할당할 크기를 나

타내며, extendsize는 할당할만한 블록이 없을 경우 힙을 확장할 크기를 나타낸다.

만약 size가 0이면 할당할 것이 없으므로 그대로 NULL을 리턴한다. 할당하고자 하는 크기가 8보다 작거나 같다면 실제로 16바이트만 할당 사이즈로 한다(헤더와 푸터에 해당하는 DSIZE만큼 더함). 8보다 크면 DSIZE * ((size + (DSIZE) + (DSIZE-1)) / DSIZE) 만큼을 할당 사이즈로 한다. 그리고 나서 find_fit 함수를 사용하여 최초할당으로 할당할 블록의 위치를 찾고, 찾았다면 place 함수를 사용하여 asize만큼을 할당한 다음 bp를 리턴한다. 만약 find_fit이 할당할 블록을 찾지 못하여 NULL을 리턴한 경우 asize와 CHUNKSIZE 중 큰 값을 찾아서 extend_heap 함수를 사용하여 공간을 확장한다. 그리고 나서 PLACE 함수를 사용하여 커진 힙의 위치에 할당을 진행한다. 마지막에는 bp를 리턴시킨다.

> realloc

```
246  void *realloc(void *oldptr, size_t size) {
247      size_t oldsize;
248      void *newptr;
249
250      /* If size == 0 then this is just free, and we return NULL. */
251      if(size == 0){
252          free(oldptr);
253          return 0;
254      }
255
256      /* If oldptr is NULL, then this is just malloc. */
257      if(oldptr == NULL) {
258          return malloc(size);
259      }
260
261      newptr = malloc(size);
262
263      /* IF realloc() fails the original block is left untouched */
264      if(!newptr){
265          return 0;
266      }
267
268      /* Copy the old data. */
269      oldsize = GET_SIZE(HDRP(oldptr));   // 헤더로부터 블록 사이즈를 읽는다.
270      if(size < oldsize)  oldsize = size;
271      memcpy(newptr, oldptr, oldsize);
272
273      /* Free the old block. */
274      free(oldptr);
275      return newptr;
276  }
```

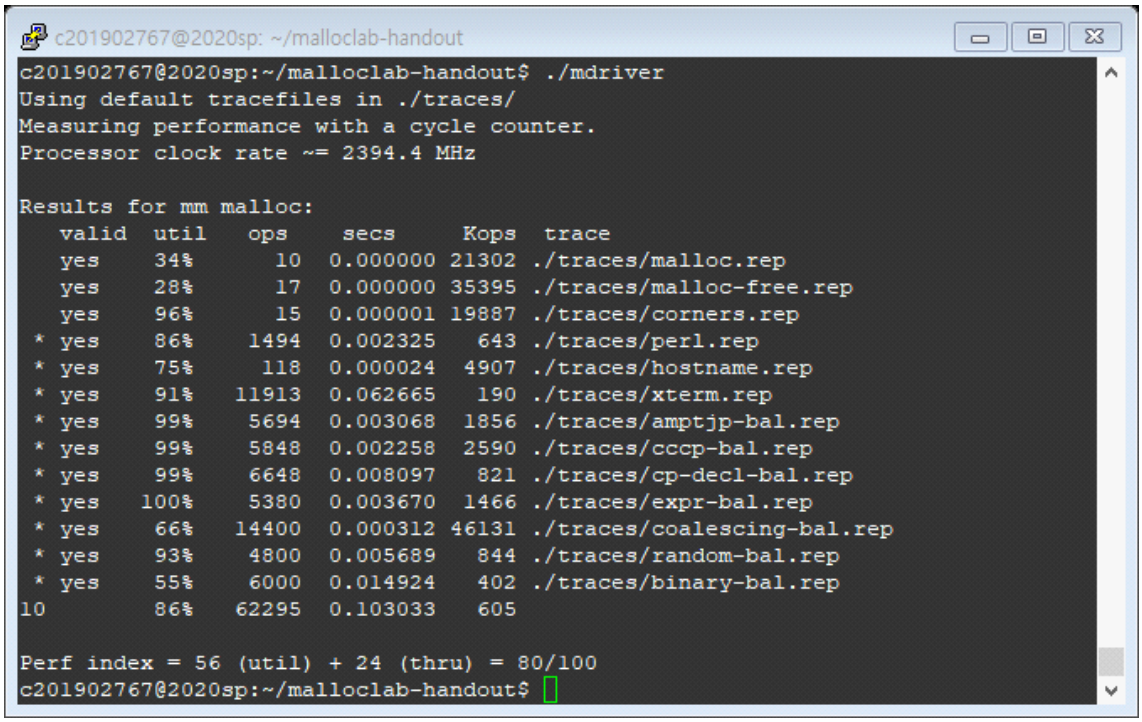
realloc 함수는 naive에서와 거의 동일한데 realloc할 블록의 크기를 읽을 때 블록의 헤더로부터 읽기 위해 GET_SIZE(HDRP(oldptr))를 계산하였다는 점이 SIZE_PTR 매크로 함수를 사용하여 사이즈를 계산한 naive 방식과의 차이점이다.

> calloc

```
283  void *calloc (size_t nmemb, size_t size) {
284      size_t bytes = nmemb * size;
285      void *newptr;
286
287      newptr = malloc(bytes);
288      memset(newptr, 0, bytes);
289
290      return newptr;
291  }
```

calloc 함수는 ./mdriver로 테스트할 때 테스트 항목에서 제외되지만 trace들을 실행하는 데에 필요하여서 naive에서 구현한 것과 동일한 구현을 사용하였다.

## Explicit




explicit (직접 리스트) 방식으로 동적 할당기를 구현하면 implicit (간접 리스트) 방식과 다르게 가용 블록들만을 리스트로 관리하기 때문에 처리율(thru)이 implicit의 처리율보다 좋아진 것을 확인할 수 있었다.

## 구현 방법

```
38  /* single word (4) or double word (8) alignment */
39  #define ALIGNMENT 8
40
41  /* rounds up to the nearest multiple of ALIGNMENT */
42  #define ALIGN(p) (((size_t)(p) + (ALIGNMENT-1)) & ~0x7)
43
44  /* Basic constants and macros */
45  #define HDRSIZE     4       /* header size (bytes) */
46  #define FTRSIZE     4       /* footer size (bytes) */
47  #define WSIZE       4       /* word와header/footer의사이즈*/
48  #define DSIZE       8       /* double word 사이즈*/
49  #define CHUNKSIZE   (1<<12) /* 힙의 확장 사이즈 */
50  #define OVERHEAD    8       /* overhead size */
51  #define MINIMUM     2*DSIZE + 4*WSIZE   /* 최소할당 사이즈 */
52
53  #define MAX(x, y)   ((x) > (y)? (x) : (y))  /* x, y 중 큰 값을 구하는 매크로함수 */
54  #define MIN(x, y)   ((x) < (y)? (x) : (y))  /* x, y 중 작은 값을 구하는 매크로함수 */
55
56  /* Pack a size and allocated bit into a word */
57  #define PACK(size, alloc)   ((unsigned) ((size) | (alloc))) /* size와alloc(할당여부) 값을 하나의 word로 묶는다 */
```

mm-explicit.c 에서의 매크로 상수 및 함수들은 mm-implicit.c 에서의 것에서 조금 추가되거나 수정되었다. 우선 헤더와 푸터 사이즈를 묶어서 OVERHEAD라는 매크로를 정의하였으며, 가용 블록 리스트로 블록이 관리되기 때문에 이전 가용 블록과 다음 가용 블록을 가리키는 NEXT, PREV 필드가 추가되어, 최소 할당 블록 사이즈가 24가 되는데 이는 MINIMUM 매크로로 정의하였다.

```c
/* Read and write a word at address p */
#define GET(p)      (*(unsigned int*)(p))   /* 포인터 p가 가리키는 주소의 값을 읽는다 */
#define PUT(p, val) (*(unsigned int*)(p) = (val))   /* 포인터 p가 가리키는 주소에 val 값을 쓴다 */
#define GET8(p)     (*(unsigned long*)(p)) /* 포인터 p가 가리키는 주소의 8바이트 값을 읽는다 */
#define PUT8(p, val)     (*(unsigned long*)(p) = (unsigned long)(val))  /* 포인터 p가 가리키는 주소에 8바이트 값을 쓴다 */

/* Read the size and allocated fields from address p */
#define GET_SIZE(p)     (GET(p) & ~0x7)     /* 포인터 p가 가리키는 주소의 값의 하위 3비트를버려서  header에서의 block size를 읽는
#define GET_ALLOC(p)    (GET(p) & 0x1)      /* 포인터 p가 가리키는 주소의 값의 하위 1비트를 읽어서 header에서의 alloc bit를 읽는다

/* Given block ptr bp, compute address of its header and footer */
#define HDRP(bp)    ((char*)(bp) - WSIZE)   /* 포인터 bp의 header 주소를 계산한다 */
#define FTRP(bp)    ((char*)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)  /* 포인터 bp의 footer 주소를 계산한다 */

/* Given block ptr bp, compute address of next and previous blocks */
#define NEXT_BLKP(bp)   ((char*)(bp) + GET_SIZE(HDRP(bp)))  /* 포인터 bp를 이용해서 다음 블록의 주소를 얻는다 */
#define PREV_BLKP(bp)   ((char*)(bp) - GET_SIZE(((char*)(bp) - DSIZE))) /* 포인터 bp를 이용해서 이전 블록의 주소를 얻는다 */

#define NEXT_FREEP(bp)  ((char*)(bp))  /* 현재 free block의 NEXT 필드를 가리키는 포인터를 얻는다 */
#define PREV_FREEP(bp)  ((char*)(bp) + DSIZE)          /* 현재 free block의 PREV 필드를 가리키는 포인터를 얻는다 */

#define NEXT_FREE_BLKP(bp)  ((char*)GET8((char*)(bp)))    /* 다음 Free block의 block pointer를 얻는다 */
#define PREV_FREE_BLKP(bp)  ((char*)GET8((char*)(bp) + DSIZE))    /* 이전 Free block의 block pointer를 얻는다 */

#define FREE_LISTP      ((char*)heap_start)        /* Free block list를 가리키는 포인터를 얻는다 */
#define GET_FREE_LISTP  ((void*)GET8(FREE_LISTP))  /* Free block의 첫 번째 블록 */

void *heap_listp;   /* Heap pointer */
void *heap_start;   /* free list head */
static void *coalesce(void *bp);
static void *extend_heap(size_t words);
```

또한 포인터 P가 가리키는 주소의 8바이트 값을 읽거나 해당 주소에 값을 쓰는 GET8, PUT8 함수가 추가되었으며, 현재 가용 블록의 NEXT와 PREV 필드를 가리키는 포인터를 얻는 매크로 함수 NEXT_FREEP, PREV_FREEP가 추가되었다. 또한, 현재 블록의 NEXT 블록의 포인터와 PREV 블록의 포인터를 얻는 함수 NEXT_FREE_BLKP, PREV_FREE_BLKP 도 추가되었다. FREE_LISTP는 가용 리스트의 시작을 가리키는 포인터를 얻는 함수이며, GET_FREE_LISTP는 가용 리스트의 시작 주소의 8바이트 값을 읽어서 가용 리스트의 첫 번째 블록 포인터를 얻는다. 전역 변수로는 힙을 가리키는 포인터 변수 heap_listp와, 가용 리스트 시작부분을 가리키는 heap_start가 있다.

> mm_init

```c
int mm_init(void) {
    /* Request memory for the initial empty heap */
    /* 메모리에 초기 힙 요청: 최소 블록 크기는 24 */
    if ((heap_listp = mem_sbrk(MINIMUM)) == NULL){
        return -1;
    }
    heap_start = heap_listp;    // 힙의 가장 앞을 heap_start 전역변수에 저장

    PUT8(heap_listp, NULL);         // root next
    PUT8(heap_listp + DSIZE, NULL); // root prev(필요하지는 않음)

    heap_listp = heap_listp + 2*DSIZE;

    PUT(heap_listp, 0);                             /* alignment padding */
    PUT(heap_listp + WSIZE, PACK(OVERHEAD, 1));     /* prologue header */
    PUT(heap_listp + 2*WSIZE, PACK(OVERHEAD, 1));   /* prologue footer */
    PUT(heap_listp + 3*WSIZE, PACK(0, 1));          /* epilogue header */

    /* Move heap pointer over to footer */
    heap_listp += DSIZE;

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

mm_init 함수는 초기 힙을 구성하는 함수이다. 먼저 mem_sbrk 함수에 MINIMUM를 인자로 전달하여 24바이트만큼 힙 공간을 늘린 다음 힙의 시작부분을 heap_listp에 저장한다. 정상

적으로 작업이 완료되면, 첫 8바이트 공간에는 NULL을, 두 번째 8바이트 공간에도 NULL을 저장한다. heap_listp 포인터를 16바이트만큼 뒤로 이동시킨 다음, 첫 4바이트 공간에는 0을, 두 번째와 세 번째 4바이트 공간에는 PACK(OVERHEAD, 1) 값을, 마지막 4바이트 공간에는 PACK(0,1) 값을 기록한 다음 heap_listp를 DSIZE 값과 더해준다. 마지막으로 extend_heap 함수를 호출하여 정상적으로 작업이 완료되면 0을 리턴한다.

> extend_heap

```
124  static void *extend_heap(size_t words){
125      char *bp;
126      size_t size;
127
128      /* Allocate an even number of words to maintain alignment */
129      size = (words % 2) ? (words+1) * WSIZE : words * WSIZE; // size를 짝수로 변환
130      if ((long)(bp = mem_sbrk(size)) == -1)  // 공간 확장 실패
131          return NULL;
132
133      // 확장 성공한 경우 블록의 헤더에 size, alloc 기록
134      /* Initialize free block header/footer and the epilogue header */
135      PUT(HDRP(bp), PACK(size, 0));         /* Free block header */
136      PUT(FTRP(bp), PACK(size, 0));         /* Free block footer */
137      PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1));   /* New epilogue header */
138
139      /* Coalesce if the previous block was free */
140      return coalesce(bp);
141  }
```

extend_heap 함수는 힙의 크기를 확장해주는 함수이며, words 파라미터에 인자를 받는다.

먼저 char형 bp 포인터 변수와, size_t 타입의 size 변수를 선언한 다음 words를 짝수 바이트 값으로 변환한다. 그다음, mem_sbrk(size)를 호출하여 힙 공간을 확장하는 작업에 성공하면, bp의 헤더와 푸터에 size와 가용블록임을 나타내는 0을 PACK하여 기록하고, 다음 블록의 헤더에 PACK(0,1)을 기록한다. 그다음, coalesce 함수를 호출하게 된다.

> coalesce

```
143  static void *coalesce(void *bp){
144      // 이전 블록이 할당되어 있는 블록인지 알아내기 위해 alloc bit을 구한다.
145      size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
146      // 다음 블록이 할당되어 있는 블록인지 알아내기 위해 alloc bit를 구한다.
147      size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
148      size_t size = GET_SIZE(HDRP(bp));   // 현재 블록의 크기를 얻는다.
149
150      // Case 1: 이전 블럭, 다음 블럭 최하위 bit가 둘 다 1인 경우(할당) bp 블록을 가용리스트의 맨 앞에 삽입한다.
151
152      // Case 2: 이전 블럭 최하위 bit가 1이고(할당), 다음 블럭 최하위 bit가 0인 경우(비할당) 다음 블럭과 병합
153      if (prev_alloc && !next_alloc){
154          size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
155          removeBlock(NEXT_BLKP(bp));
156          PUT(HDRP(bp), PACK(size, 0));
157          PUT(FTRP(bp), PACK(size, 0));
158      }
```

coalesce 함수는 블록을 가용 블록으로 바꿀 때 가용 블록들끼리 합치는 작업을 하는 함수이다. 네 가지 case가 존재하며 현재 블록은 bp 포인터 변수가 가리키고 있다.

첫째로, 이전 블록과 다음 블록이 모두 할당 블록인 경우 병합 작업을 하지 않고 현재 블록을 가용리스트의 맨 앞에 삽입하게 된다.

둘째로, 이전 블록은 할당 상태이고 다음 블록은 가용 블록이면 다음 블록과 병합한다. 현재 블록의 다음 블록의 크기를 size에 더하고, 다음 블록을 가용 리스트에서 제거하기 위해 removeBlock 함수를 사용한다. 그다음, 현재 블록의 헤더와 푸터에 size를 업데이트한다.

```
160     // Case 3: 이전 블럭 최하위 bit가 0이고(비할당), 다음 블럭 최하위 bit가 1인 경우(할당) 이전 블럭과 병합
161     else if (!prev_alloc && next_alloc){
162         size += GET_SIZE(HDRP(PREV_BLKP(bp)));
163         bp = PREV_BLKP(bp);
164         removeBlock(bp);
165         PUT(HDRP(bp), PACK(size, 0));
166         PUT(FTRP(bp), PACK(size, 0));
167     }
168     // Case 4: 이전 블럭, 다음 블럭 최하위 bit가 둘 다 0인 경우(비할당) 이전 블럭, 현재 블럭, 다음 블럭을 모두 병합
169     else {
170         size += GET_SIZE(HDRP(PREV_BLKP(bp))) + GET_SIZE(FTRP(NEXT_BLKP(bp)));
171         removeBlock(PREV_BLKP(bp));
172         removeBlock(NEXT_BLKP(bp));
173         bp = PREV_BLKP(bp);
174         PUT(HDRP(bp), PACK(size, 0));
175         PUT(FTRP(bp), PACK(size, 0));
176     }
177     // bp 블록을 가용리스트 맨 앞에 삽입
178     *NEXT_FREEP(bp) = *FREE_LISTP;    // 현재 bp의 NEXT를 가용리스트의 처음을 가리키도록 한다.
179     *PREV_FREEP(*FREE_LISTP) = bp;    // 가용리스트의 시작 부분의 PREV가 현재 bp를 가리키도록 한다.
180     *PREV_FREEP(bp) = NULL;           // 현재 bp의 PREV를 NULL로 세팅
181     *FREE_LISTP = bp;                 // 현재 bp가 가용리스트의 시작이 되도록 한다.
182 
183     return bp;
184 }
```

셋째로, 이전 블록이 가용 블록이고 다음 블록이 할당 블록이면 이전 블록과 병합한다. 현재 블록의 이전 블록의 크기를 size에 더하고, bp 포인터를 이전 블록으로 이동시킨 이전 블록을 가용 리스트에서 제거하기 위해 removeBlock 함수를 사용한다. 그다음, bp가 가리키는 블록(이전 블록)의 헤더와 푸터에 size를 업데이트한다.

마지막으로, 이전 블록과 다음 블록이 모두 가용 블록인 경우, 이전 블록과 다음 블록, 현재 블록을 하나의 블록으로 병합한다. 현재 블록의 이전, 다음 블록의 크기를 size에 더하고, 이전과 다음 블록을 가용 리스트에서 제거하기 위해 removeBlock 함수를 사용한다. 그다음, bp 포인터를 이전 블록으로 이동시켜서 bp가 가리키는 블록(이전 블록)의 헤더와 푸터에 size를 업데이트한다.

각 조건별로 작업을 완료하면, bp가 가리키는 블록을 가용리스트의 맨 앞에 삽입한다. 삽입 과정은 line 178~181의 주석에 설명하였다.

> removeBlock

```
320 // 가용리스트에서 블록을 제거하는 함수
321 // 이전 블록과 다음 블록을 연결시키고 현재 블록은 양 옆 블록과의 연결을 해제하여 구현한다.
322 static void removeBlock(void *bp){
323     // 이전 블록이 존재하면 이전 블록의 NEXT가 현재 블록의 NEXT를 가리키도록 한다.
324     if(PREV_FREEP(bp))
325         *NEXT_FREEP(*PREV_FREEP(bp)) = *NEXT_FREEP(bp);
326     else    // 이전 블록이 없으면 가용 리스트의 시작이 현재 블록의 NEXT가 되도록 한다.
327         *FREE_LISTP = *NEXT_FREEP(bp);
328     // 현재 블록의 NEXT 블록의 PREV 블록이 bp의 PREV 블록이 되도록 한다.
329     *PREV_FREEP(*NEXT_FREEP(bp)) = *PREV_FREEP(bp);
330 
331 }
```

가용리스트에서 블록을 제거하는 함수이다. 현재 bp가 가리키는 블록의 이전 블록이 존재하면 이전 블록이 다음 블록으로 현재 블록의 다음 블록을 가리키게 한다. 만약 bp가 가리키는 블록의 이전 블록이 없으면, 가용 리스트의 시작이 현재 블록의 다음 블록이 되도록 한다. 그다음, 현재 블록의 다음 블록이 이전 블록으로 bp가 가리키는 블록의 이전 블록이 되도록 한다. 이 과정을 모두 거치면, 이전 블록과 다음 블록이 연결되고, 현재 블록은 양 옆 블록과 연결이 해제되므로 가용리스트에서 현재 블록이 제거된 셈이 된다.

> find_fit

```
186  static void *find_fit(size_t asize){
187      /* First-fit search */
188      void *bp;
189      // 힙의 마지막 부분에 도달할때까지 반복
190      for (bp = FREE_LISTP; GET_ALLOC(HDRP(bp)) == 0; bp = NEXT_FREEP(bp)){
191          // 가용블록이고 사이즈가 같거나 큰 블록이면 찾은 위치는 bp이며 이를 리턴한다.
192          if(asize <= (size_t)GET_SIZE(HDRP(bp))){
193              return bp;
194          }
195      }
196      return NULL; /* No fit */
197  }
```

find_fit 함수는 할당할 블록을 최초 할당 방식으로 찾는 함수이다. 힙의 처음부터 마지막 부분에 도달할 때까지, 가용 리스트를 탐색하면서 사이즈가 asize와 같거나 큰 블록을 찾다가, 조건에 부합하는 블록을 찾으면 해당 블록의 주소를 리턴한다. 알맞은 블록이 없다면 NULL을 리턴하고 추가 공간을 요청해야 한다.

> place

```
199  static void place(void *bp, size_t asize){
200      // 가용 블록의 헤더로부터 블록 크기를 얻는다.
201      size_t csize = GET_SIZE(HDRP(bp));
202
203      // 현재 free block이 요청된 할당 크기보다 MINIMUM 이상 큰 경우
204      // free block에 사용할 만큼의 블록에만 alloc 1로 세팅
205      // 그리고 나머지 블록을 잘라서 free block으로 만든다
206      if((csize - asize) >= MINIMUM){
207          // 블록 할당
208          PUT(HDRP(bp), PACK(asize, 1));
209          PUT(FTRP(bp), PACK(asize, 1));
210          // 가용 블록 리스트에서 해당 블록을 제거
211          removeBlock(bp);
212          bp = NEXT_BLKP(bp); // 다음 블록으로 bp포인터 이동
213          PUT(HDRP(bp), PACK(csize-asize, 0));    // 남은 공간의 크기를 헤더에 기록하고 alloc = 0 세팅
214          PUT(FTRP(bp), PACK(csize-asize, 0));    // 남은 공간의 크기를 헤더에 기록하고 alloc = 0 세팅
215          coalesce(bp);   // 인접한 가용 블록들과 병합
216      }else{
217          PUT(HDRP(bp), PACK(csize, 1));
218          PUT(FTRP(bp), PACK(csize, 1));
219          removeBlock(bp);
220      }
221  }
```

place 함수는 할당할 블록을 찾았을 경우 실제로 할당을 진행하는 함수이다. 할당할 블록을 가리키는 포인터 bp와, 할당할 크기를 갖는 asize를 파라미터로 갖는다. 먼저 csize 변수에 할당할 블록의 크기를 헤더로부터 얻어서 저장한다. 할당할 블록의 크기가 할당할 크기인 asize보다 MINIMUM(=24바이트) 이상 크면, 외부 단편화를 방지하기 위해 bp의 헤더와 푸터에 사이즈를 asize만 기록하고 alloc bit = 1로 세팅하여 할당했음을 기록한 다음, 가용 리스트에서 해당 블록을 제거한다. 그다음, bp 포인터를 현재 블록의 다음 블록으로 이동시킨 후 다음 블록의 헤더와 푸터에 csize-asize 값을 업데이트하고 alloc 상태를 0으로 만든다. 그다음, coalesce를 호출하여 인접한 가용 블록들과 병합한다. 만약 가용블록을 자를 필요가 없는 조건이면 할당할 블록의 헤더와 푸터에 csize를 그대로 기록하고 alloc 상태를 1로 만든 후 현재 블록을 가용 리스트에서 제거하는 것으로 충분하다.

> _free

```
256  void free (void *bp) {
257      if(bp == 0) return; // 잘못된 free 요청인 경우 함수 종료. 이전 프로시저로 return
258      size_t size = GET_SIZE(HDRP(bp));   // bp의 header에서 block size를 읽어온다.
259
260      // 실제로 데이터를 지우는 것이 아니라
261      // header, footer의 최하위 1bit(1, 할당된 상태)만을 0으로 수정
262
263      PUT(HDRP(bp), PACK(size, 0));   // bp의 header에 block size와 alloc = 0을 저장
264      PUT(FTRP(bp), PACK(size, 0));   // bp의 footer에 block size와 alloc = 0을 저장
265
266      coalesce(bp);   // 주위에 빈 블록이 있을 시 병합한다
267  }
```

free 함수는 할당 블록을 다시 가용 블록으로 만드는 함수이다. 파라미터 *bp를 통해 free 시킬 블록 포인터를 전달받는다. 만약 잘못된 블록을 free 요청할 경우 함수를 종료한다. 그렇지 않으면, 반환할 블록의 헤더로부터 블록 크기를 얻어서 size 변수에 저장한다. 그다음, 블록의 헤더와 푸터에 alloc bit를 0으로 세팅하여 가용 블록임을 나타내 준다. free 과정은 실제로 블록에 저장된 데이터를 지울 필요 없이 헤더, 푸터의 alloc bit를 업데이트하는 것으로 충분한데, 그 이유는 가용 블록임만 확인이 되면 나중에 블록에 할당할 때 새로운 값을 쓰기만 하면 되기 때문이다. 이 작업을 완료하면 앞뒤로 병합할 수 있는 가용 블록을 병합해 주는 coalesce 함수를 호출한다.

> malloc

```
226  void *malloc (size_t size) {
227      size_t asize;       /* Adjusted block size */
228      size_t extendsize;  /* Amount to extend heap if no fit */
229      char *bp;
230
231      // 할당 사이즈가 0이면 할당하지 않는다.
232      if (size == 0)
233          return NULL;
234
235      // 할당하고자 하는 크기에 따라 asize 결정
236      asize = MAX(ALIGN(size) + OVERHEAD, MINIMUM);
237
238      // 힙에서 할당할만한 블록을 찾은 다음 찾았으면 place 함수로 할당
239      if ((bp = find_fit(asize)) != NULL){
240          place(bp, asize);
241          return bp;
242      }
243
244      // 할당할만한 블록을 찾지 못했으면 asize와 CHUNKSIZE 중 큰 값을 찾는다.
245      extendsize = MAX(asize, CHUNKSIZE);
246      if ((bp = extend_heap(extendsize/WSIZE)) == NULL)
247          return NULL;
248      // 확장하여 커진 힙 위치에 할당한다.
249      place(bp, asize);
250      return bp;
251  }
```

지금까지 구현한 함수들을 사용하여 최종적으로 동적 메모리를 할당하는 작업을 총괄하는 malloc 함수를 구현하였다. asize는 할당하고자 하는 size를 가지고 실제 할당할 크기를 나타내며, extendsize는 할당할만한 블록이 없을 경우 힙을 확장할 크기를 나타낸다.

만약 size가 0이면 할당할 것이 없으므로 그대로 NULL을 리턴한다. 할당하고자 하는 크기에 따라 asize를 결정하고 나서 find_fit 함수를 사용하여 최초할당으로 할당할 블록의 위치를 찾고, 찾았다면 place 함수를 사용하여 asize만큼을 할당한 다음 bp를 리턴한다. 만약 find_fit이 할당할 블록을 찾지 못하여 NULL을 리턴한 경우 asize와 CHUNKSIZE 중 큰 값을 찾아서 extend_heap 함수를 사용하여 공간을 확장한다. 그리고 나서 PLACE 함수를 사용하여 커진 힙의 위치에 할당을 진행한다. 마지막에는 bp를 리턴시킨다.

> realloc

```
272  void *realloc(void *oldptr, size_t size) {
273      size_t oldsize;
274      void *newptr;
275
276      /* If size == 0 then this is just free, and we return NULL. */
277      if(size == 0){
278          free(oldptr);
279          return 0;
280      }
281
282      /* If oldptr is NULL, then this is just malloc. */
283      if(oldptr == NULL) {
284          return malloc(size);
285      }
286
287      newptr = malloc(size);
288
289      /* IF realloc() fails the original block is left untouched */
290      if(!newptr){
291          return 0;
292      }
293
294      /* Copy the old data. */
295      oldsize = GET_SIZE(HDRP(oldptr));   // 헤더로부터 블록 사이즈를 읽는다.
296      if(size < oldsize)  oldsize = size;
297      memcpy(newptr, oldptr, oldsize);
298
299      /* Free the old block. */
300      free(oldptr);
301      return newptr;
302  }
```

realloc 함수는 implicit과 동일하게 구현하였다.

> calloc

```
283  void *calloc (size_t nmemb, size_t size) {
284      size_t bytes = nmemb * size;
285      void *newptr;
286
287      newptr = malloc(bytes);
288      memset(newptr, 0, bytes);
289
290      return newptr;
291  }
```

calloc 함수 역시 implicit과 동일하게 구현하였다.