2019시스템 프로그래밍

- Lab 09 -

제출일자	
분 반	
이 름	
학 번	

Naïve

```
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    94%      10  0.000000 68804 ./traces/malloc.rep
   yes    77%      17  0.000000 80127 ./traces/malloc-free.rep
   yes   100%      15  0.000000 59072 ./traces/corners.rep
 * yes    71%    1494  0.000018 83163 ./traces/perl.rep
 * yes    68%     118  0.000002 76073 ./traces/hostname.rep
 * yes    65%   11913  0.000128 92775 ./traces/xterm.rep
 * yes    23%    5694  0.000063 89992 ./traces/amptjp-bal.rep
 * yes    19%    5848  0.000065 90322 ./traces/cccp-bal.rep
 * yes    30%    6648  0.000074 90407 ./traces/cp-decl-bal.rep
 * yes    40%    5380  0.000059 91314 ./traces/expr-bal.rep
 * yes     0%   14400  0.000150 95736 ./traces/coalescing-bal.rep
 * yes    38%    4800  0.000067 71539 ./traces/random-bal.rep
 * yes    55%    6000  0.000053113797 ./traces/binary-bal.rep
10         41%   62295  0.000679 91795

Perf index = 26 (util) + 40 (thru) = 66/100
b201802055@2019sp:~/malloc$
```

mdriver를 실행시킨 결과이다. mm-naive.c에 대해서는 코드를 따로 작성하지 않았음으로, 이 mm-naive에서 사용된 함수를 분석해 볼 예정이다.

구현 방법

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

우선 정의부분부터 살펴보면, **ALIGNMENT**가 8이라고 정의되어 있는 모습을 볼 수 있고, 그 아래에 **ALIGN(size)**라는 것이 정의되어 있는 모습을 볼 수 있다. 이것은 8의 배수 단위로 할당을 하기 위해서 들어온 size에 맞게 8의 배수로 사이즈를 다시 정의해주는 부분이라고 해석했다. 예를 들어 할당하고 싶은 size가 4라고 가정해보았다. 그렇다면 ALIGNMENT가 8이므로, 4+7이 되고, 값은11이 되는데, 이것을 0x7의 보수와 비트 연산 and를 해주게 된다. 여기서 0x7을 이진수로 나타내면 0000 0111이 되는데, 이것을 보수로 만들어준다면 1111 1000이 되고, 11인 0000 1011과 AND를 한다면 결과적으로 0000 1000이 되는 모습을 볼 수 있는데, 결과값은 십진수로 8임을 알

수 있다. 이와 같이 size 4를 요청했지만, 8의 배수를 만들어 반환하는 매크로 함수라는 것을 알 수 있었다.

밑의 **SIZE_T_SIZE**부분은 사이즈를 정의해주는 부분인데, 여기서 size_t는 long타입을 재정의해준 부분으로 보였다. 이 부분도 역시 8를 위해 정의된 부분이다.

마지막 **SIZE_PTR(p)**부분은 부분은 말 그대로 사이즈를 반환해주는 부분인데, 블록을 할당하고 나서, 맨 앞 블록(이 함수에서는 아니지만, Header의 개념과 비슷한 것. 이곳에서는 그저 SIZE를 담고 있는 앞부분의 블록 정도로 해석된다.)을 참고하여 사이즈를 얻어내는 부분으로 이해하였다. 맨 앞부분에는 size의 정보를 알 수 있지만, 정보를 얻어내기 위해서는 이곳에 접근해야 하는데, 그 방법을 코드를 참고하여 이해해보았다. 만약, p가 int형이나 long형의 숫자라면 숫자 1이 들어왔다는 가정 하에, 4바이트 혹은 8바이트씩 움직여 원래의 한 칸씩 움직이려고 했던 의도와 달라지게 된다. 그렇기 때문에 (char*)형으로 형 변환을 해주고, 8만큼을 빼 주게 된다. 그러나 반환하려는 값은 8의 배수 값이여야 하므로 다시 (size_t*)형으로 형 변환을 진행하게 된다. 이렇게 해서 할당된 사이즈를 알 수 있다.

다음은 함수에 대해서 이해를 위해 하나씩 해석을 진행해 보았다.

[mm_init]

```
54 int mm_init(void)
55 {
56   return 0;
57 }
```

특별한 특이사항은 없다. 본래는 heap을 초기화해주는 함수로 알고 있는데, 특별한 구현이 없어 현재는 0의 값만을 리턴하고 있다.

[malloc]

```
63 void *malloc(size_t size)
64 {
65   int newsize = ALIGN(size + SIZE_T_SIZE);
66   unsigned char *p = mem_sbrk(newsize);
67   //dbg_printf("malloc %u => %p\n", size, p);
68
69   if ((long)p < 0)
70     return NULL;
71   else {
72     p += SIZE_T_SIZE;
73     *SIZE_PTR(p) = size;
74     return p;
75   }
76 }
77
```

할당을 원하는 만큼의 값을 size인자로 받게 된다. 인자를 받고 난 후에 할당이 진행되게 되는 모습은 아래와 같다. 우선 newsize 변수에 위에서 정의되었던 매크로 함수 ALLGN을 사용하여 할당

받을 만큼의 size를 정해주게 된다. 여기서 주의할 점은, 앞의 사이즈를 기록하는 헤더 역할을 하는 부분까지 고려를 해야한다는 점이다. 본래 할당 받을 사이즈 + 사이즈를 기록하는 헤더 역할 부분의 사이즈의 size 값을 매크로 함수에 입력해야 한다는 것이다. 여기서 그 사이즈 기록하는 부분은 8byte로 간주하여 계산을 하게 된다. 이렇게 해서 얼마나 할당 받을지 결정이 되었다면, mem_sbrk 함수를 이용하여 *p에 newsize만큼을 heap에 할당해주게 된다.

그리고 나서 이전의 p의 사이즈 (long으로 형 변환하여 체크)가 0보다 작다면 당연히 할당되지 않았다는 뜻이 되므로 NULL을 반환하고, 그렇지 않은 경우에는 포인터 p를 SIZE만큼, 8만큼 옮겨주고 이것의 size를 헤더 역할을 하는 사이즈 기록 부분에 써준다. 그리고 나서 할당된 포인터의 시작 부분을 리턴한다.

[free]

```
82 void free(void *ptr)
83 {
84 }
85
```

원래는 할당해주는 동작이 있다면, 반대로 free를 시켜주어 할당되었던 블록을 가용 블록으로 변환해주는 함수도 있어야 하지만, 이번 naive에서는 free를 따로 구현하지 않았다. 그래서 naïve는 할당 기능만을 하므로 다소 공간 활용에 있어 비효율적인 부분이 존재한다고 생각했다.

[realloc]

```
91 void *realloc(void *oldptr, size_t size)
92 {
93   size_t oldsize;
94   void *newptr;
95
96   /* If size == 0 then this is just free, and we return NULL. */
97   if(size == 0) {
98     free(oldptr);
99     return 0;
100  }
101
102  /* If oldptr is NULL, then this is just malloc. */
103  if(oldptr == NULL) {
104    return malloc(size);
105  }
106
107  newptr = malloc(size);
108
109  /* If realloc() fails the original block is left untouched  */
110  if(!newptr) {
111    return 0;
112  }
113
114  /* Copy the old data. */
115  oldsize = *SIZE_PTR(oldptr);
116  if(size < oldsize) oldsize = size;
117  memcpy(newptr, oldptr, oldsize);
118
119  /* Free the old block. */
120  free(oldptr);
121
122  return newptr;
123 }
124
```

Realloc 함수는 할당된 공간을 다시 리사이징하여 할당해주는 함수이다. 그렇기 때문에 원래 할당되었던 oldptr과 다시 할당할 size 이렇게 두 개의 인자를 받는다. 그리고 size_t 형의 oldsize 변수를 선언하고, 아무것도 리턴하지 않는 void형 newptr 포인터 변수를 선언해준다.

그리고 나서 각각의 if문을 거쳐가며 동작을 실행하는데, 우선 인자로 들어온 size가 0이라면 아무것도 할당을 원하지 않으므로, oldptr에 대한 모든 할당 공간을 free 시켜주고, 0을 리턴한다.

다음 조건은 만약 oldptr이 null이라면, 애초에 할당 받은 것이 없었으므로 리사이즈를

해주는 것이 아니라 malloc 함수로, 처음의 할당을 먼저 해 주어야 하기 때문에 malloc함수를 호출하여 size만큼 우선 할당을 해준다. 반대로, 이미 할당되어있음을 확인했다면, newptr에 malloc 함수로 size만큼 할당을 해준 다음, newptr이 0이라면 0을 리턴 하도록 하고 이 조건에 걸린다면 제대로 할당이 된 것이므로 oldsize에 원래 가지고 있었던 사이즈를 리턴 받아서 변수에 담고, 할당하려고 했던 사이즈와 비교하여, 만약 지금 realloc으로 할당 받은 사이즈보다 크다면, oldsize를 size 값으로 바꾸고 memcpy() 함수를 사용하여, 할당한 것을 복사하게 된다. 그리고 예전에 할당 받았던 함수를 모두 다시 가용 블록으로 만들어준 다음, 새롭게 할당 받은 공간의 시작 포인터를 반환하게 된다.

[calloc]

```
128 void *calloc (size_t nmemb, size_t size)
129 {
130   size_t bytes = nmemb * size;
131   void *newptr;
132
133   newptr = malloc(bytes);
134   memset(newptr, 0, bytes);
135
136   return newptr;
137 }
```

Calloc 함수의 코드를 분석해보면, bytes에 nememb와 size를 곱한 값을 넣는다. Size의 이름을 보아 nmemb개가 가지고 있는 공간이 bytes임을 유추할 수 있었다. 그래서 bytes의 크기만큼 malloc으로 bytes만큼 공간을 할당한 뒤에, memset 함수를 이용하여 0으로 바꿔준 뒤에 newptr을 리턴하게 된다.

[+mem_brk]

```
18 static unsigned char heap[MAX_HEAP];
19 static char *mem_brk = heap; /* points to last byte of heap */
20 static char *mem_max_addr = heap + MAX_HEAP;  /* largest legal heap address */
```

추가로 mem_sbrk에 대해서 해석을 해 보았다. 실습 시간에 들은 내용을 토대로 간단히 적어볼

예정이다. 우선 위와 같이 선언 되어있는 변수들을 보면 mem_brk에는 마지막 주소를 알 수 있는

변수이고, Mem_max_add 가장 큰 힙의 주소를 담는 변수이다.

```
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 }
```

Mem_sbrk 함수는 위와 같은데, 이 함수의 동작은 아래와 같다. 우선 인자 incr 를 받는다. 이 incr 는 얼마나 증가할 것인지를 알려주는 변수가 된다. 제일 처음으로 Oldbrk 에 membrk 를 넣어주는데, 이것은 메모리에서 oldbrk 와 membrk 둘다 같은 곳을 가리키고 있는 것을 표현한 것이다. 그리고 나서 Mem_brk 에 incr 값을 넣어줘서, 할당을 해주는 것이고, 리턴은 oldbrk 를 해줘서, 시작점을 리턴해주게 되는 동작을 하는 함수이다.

implicit

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

Processor clock rate ~= 2394.4 MHz

Results for mm malloc:
   valid  util   ops    secs     Kops  trace
   yes    34%      10  0.000001 19403 ./traces/malloc.rep
   yes    28%      17  0.000001 33093 ./traces/malloc-free.rep
   yes    96%      15  0.000001 26026 ./traces/corners.rep
 * yes    81%    1494  0.000148 10107 ./traces/perl.rep
 * yes    50%     118  0.000004 28459 ./traces/hostname.rep
 * yes    88%   11913  0.001546  7707 ./traces/xterm.rep
 * yes    86%    5694  0.000426 13377 ./traces/amptjp-bal.rep
 * yes    91%    5848  0.000535 10937 ./traces/cccp-bal.rep
 * yes    89%    6648  0.000585 11366 ./traces/cp-decl-bal.rep
 * yes    91%    5380  0.000414 12993 ./traces/expr-bal.rep
 * yes    66%   14400  0.000364 39601 ./traces/coalescing-bal.rep
 * yes    85%    4800  0.001286  3733 ./traces/random-bal.rep
 * yes    55%    6000  0.000228 26352 ./traces/binary-bal.rep
10        78%   62295  0.005534 11256

Perf index = 50 (util) + 40 (thru) = 90/100
b201802055@2019sp:~/malloc$
```

mm-implicit.c 파일에 implicit 기능들을 구현한 결과이다. 이것을 어떻게 구현했는지는 아래의 구현한 함수들을 토대로 자세히 설명해 볼 예정이다.

구현 방법

[Macro]

```
41  /* rounds up to the nearest multiple of ALIGNMENT */
42  #define ALIGN(p) (((size_t)(p) + (ALIGNMENT-1)) & ~0x7)
43  /*매크로 정의*/
44  #define WSIZE 4 //word크기 정렬
45  #define DSIZE 8  // double word //크기결정
46  #define CHUNKSIZE (1<<12) //초기 heap의 크기를 설정한다
47  #define OVERHEAD 8 //헤더와 푸터의 크기를 합친 크기
48  #define MAX(x,y) ((x)>(y) ? (x) : (y)) //둘 중에 더 큰값 찾기
49  #define PACK(size, alloc) ((size) | (alloc)) //size와 alloc 값 묶기
50  #define GET(p) (*(unsigned int*)(p)) //p의 word 크기 값
51  #define PUT(p,val) (*(unsigned int*)(p)=(val)) //p가 가리키는 word크기의val값
52  #define GET_SIZE(p) (GET(p) & ~0x7) //p가 가리키는 곳에서 하위 3bit를 버린다= blck size
53  #define GET_ALLOC(p) (GET(p)&0x1) //p가 가리키는 곳에서 하위 1비트 읽기(할당 여부)
54  #define HDRP(bp) ((char*)(bp)-WSIZE) //bp의 헤더 주소
55  #define FTRP(bp) ((char*)(bp)+GET_SIZE(HDRP(bp))-DSIZE) //bp의 푸터 주소
56  #define NEXT_BLKP(bp) ((char*)(bp) + GET_SIZE(((char*)(bp)-WSIZE)))//BP를 이용하여 다음 블럭 주소를 계산한다
57  #define PREV_BLKP(bp) ((char*)(bp)-GET_SIZE(((char*)(bp)-DSIZE)))
58  //bp를 이용하여 다음 block 주소를 계산한다
59  /*
60   * Initialize: return -1 on error, 0 on success.
61   */
62  void* heap_listp;
63  void* last;
```

#define된 여러 함수들에 대한 설명은 아래와 같다. 하나씩 설명해보겠다.

- WSIZE 4: word의 크기를 결정하기 위해서 정의하였다.
- DSIZE 8: double word의 크기를 결정하기 위해서 정의하였다.
- CHUNKSIZE(1<<12): 1을 12만큼 left shift 함으로써 초기의 heap 크기를 설정해준다.
- OVERHEAD 8: header와 footer 두 개를 합친 크기이다.
- MAX(x, y): 두 개의 인자 중에 더 큰 값을 찾는다.
- PACK(size, alloc): size와 alloc의 값을 묶기 위해 사용하는 것이다.
- GET(p): p의 word 크기의 값을 반환한다.
- PUT(p, val): p가 가리키는 word 크기의 val 값을 반환한다. 여담으로 이 부분을 size_t로 형 변환을 하는 실수를 했더니, segement fault 오류가 나서 당황했던 경험이 있었다. 왜냐하면 size_t는 8 단위인데, unsigned int는 4 단위였음으로, 참조를 원하지 않는 곳으로 하고 있었기 때문에 나는 오류임을 알고, 제대로 정의를 해 주게 되었더니 오류가 해결이 되었던 문제 해결 에피소드가 있었다.
- GET_SIZE(p): p가 가리키는 곳에서 3bit를 버리는 것으로, 이것은 block 크기를 나타내게 된다. 원래 naive에서 썼던 realloc과는 달리, 헤더 역할을 하는 사이즈를 담는 블록의 ptr의 주소를 리턴하는 부분을 수정하기 위해서는 implicit는 헤더에 접근하는 HDRP 매크로와 그 곳의 사이즈를 가져오는 GET_SIZE함수를 사용해서 같은 동작을 하게 할 수 있다.

- GET_ALLOC(p): p가 가리키는 곳에서 하위 1비트를 읽는 것으로, 할당된 블록인지 아닌지의 여부를 판단하는 용으로 쓰인다. 1이면 할당, 0이면 비할당이다.
- HERP(bp): 헤더의 주소를 리턴한다. 헤더를 알 수 있으므로 이것을 이용해 사이즈를 확인하거나 할당 여부를 확인할 수 있다.
- FTRP(bp): bp 푸터(footer)의 주소이다. 사이즈와 할당 등의 정보를 얻을 수 있다.
- NEXT_BLKP(bp): bp를 이용해서 다음 블록의 주소를 계산한다. 여기서 (char*)형으로 형 변환을 진행하여 사이즈를 얻는 이유는 이전에 말한 size를 반환하는 매크로 함수의 동작 원리와 같은 의미이다. (주소를 바로 다음 블록으로 한 칸씩 옮기는 방식으로 의도대로 움직이게 하기 위해서 필요한 형 변환이다.)
- PREV_BLKP(bp): bp를 이용하여 다음 block의 주소를 계산한다. 이전의 NEXT_BLKP와 같이 동작 원리는 비슷하다. NEXT와 다른 점은 NEXT는 다음의 주소를 계산하므로 더하는 연산을 하고, 현재 PREV는 이전의 주소를 계산해야 하므로 빼는 연산을 진행한다.
- 전역 변수 heap_listp, last: last는 heap을 초기화 할 때 쓰이는 값으로 사용하기 위해서 선언된 전역 변수이며, heap_listp도 마찬가지이다. Void* 형으로 선언하였다.

[mm_init]

```
189  int mm_init(void) {
190
191      //초기에 empty heap을 생성한다
192      if((heap_listp = mem_sbrk(4*WSIZE)) == NULL)
193          return -1; //heap_listp= 새로 생성되는 heap 영역의 시작이다
194
195      PUT(heap_listp, 0); //의미없는 값 삽입
196      PUT(heap_listp + WSIZE, PACK(OVERHEAD, 1)); //prologue header
197      PUT(heap_listp + DSIZE, PACK(OVERHEAD, 1));
198      PUT(heap_listp + WSIZE + DSIZE, PACK(0,1)); //에필로그 헤더
199      heap_listp += DSIZE;
200
201      last = heap_listp;
```

Mm_init 함수는 처음에 아무런 할당을 하지 않았을 때 어느정도 값을 세팅해 놓기 위해서 처음 동작을 시작하게 되는 함수이다. 우선 먼저, heap_listp에 mem_sbrk() 함수를 사용해서 할당을 해준다. 만약 NULL이라면 생성을 실패한 것이므로 -1을 리턴해주고, 그렇지 않으면 heap에 값들을 각각 삽입해주게 된다. 처음 부분에는 의미 없는 값을 삽입해주고, 그 다음에는 WSIZE 만큼을 더해 다음 주소에 overhead와 1을 pack함수로 묶은 값들을 세팅해준다. 그 다음 주소에도 값을 세팅을 해 준 다음, 제일 마지막에 에필로그 헤더에 0과 1을 세팅해준다. 그리고 나서 heap_listp에 DSIZE 8의 값만큼을 더해주고, last의 전역 변수에 이것을 저장해주게 된다.

```
214         if((extend_heap(CHUNKSIZE/WSIZE)) == NULL)
215             return -1;
216 
217         return 0;
218     }
```

그리고 이 할당들이 끝나게 되면, 할당이 잘 되었는지 확인을 하는데, extend_heap이라는 함수로 플로우가 흘러가게 된다. 그 다음은 extend_heap 함수를 보겠다.

[extend_heap]

```
166 //힙의 크기를 확장해준다
167 static void *extend_heap(size_t words) {
168 
169     char *bp;
170     size_t size;
171     //words를 짝수로 변환
172     size = (words % 2) ? (words+1) * WSIZE : words *WSIZE;
173 
174     //mem_sbrk()함수의 size가 -1이면 확장을 실패했다는 뜻이다
175     //long으로 형변환을 해서 실패 성공 여부를 알 수 있다
176     if((long) (bp = mem_sbrk(size)) == -1)
177         return NULL;
178     //확장에 성공한 경우
179     PUT(HDRP(bp), PACK(size,0)); //확장 블록 헤더에 사이즈와 할당 여부 저장
180     PUT(FTRP(bp), PACK(size,0));
181 
182     PUT(HDRP(NEXT_BLKP(bp)), PACK(0,1)); //다음 블록 해더를 0과 1로 초기화 시킴
183     //이 코드가 지금 안됨 위에거
184 
185     return coalesce((void*)bp);
186 }
```

다음은 힙의 크기를 확장해주는 extend_heap 함수이다. 이 함수는 words를 변수로 받는다.

Char형 bp 포인터 변수를 선언해주고, size_t 형의 size 변수를 선언해 준 뒤에 size를 짝수로 변환하는 과정을 거친다. 그리고 조건 문으로 어떠한 조건에 대해 검사를 하게 되는데, mm_sbrk() 함수에 size 인자 값을 넣고 그 반환 값이 -1일 경우에는 확장에 실패한 경우이므로 return null을 해주지만, 아닐 경우에는 확장에 성공했음으로, HERP와 PUT 매크로 함수를 사용하여 확장된 블록의 헤더 부분에 사이즈와 할당 여부를 저장하게 된다. 그리고 나서 다음 블록 헤더를 0과 1로 각각 초기화를 시켜주는데, 이때 다음 블록을 알 수 있는 NEXT_BLKP 매크로를 사용하여 처리를 하였다. 그 다음에, coalesce()라는 함수로 흐름이 넘어가게 되는데, 다음은 이 함수에 대해서 살펴볼 예정이다.

[coalesce]

```
107  //빈 공간을 합쳐주는 역할을 하는 함수
108  static void *coalesce(void *bp){
109
110      size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
111
112      //이전 블럭이 할당 되었는지 안 되었는지 0, NO 1 = Yes
113      size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
114      //다음 블럭 할당 여부
115      size_t size = GET_SIZE(HDRP(bp));
116      //현재 블럭의 크기
117      /*case 1 :
118        이전 블럭, 다음 블럭 최하위 비트 둘다 1인경우 -> 할당됨
119        블럭 병합 없이 bp를 return한다.
120       */
121      if(prev_alloc && next_alloc) {
122          return bp;
123      }
```

이 함수는 빈 공간, 즉 free 상태인 블록들을 합쳐주는 역할을 하는 함수인데, 여러 if문을 거쳐 각각의 동작을 진행하게 된다. 우선 첫번째 if문에 대한 처리는 위와 같다.

Free 블록을 합치기 위해서는 어느 부분이 free 상태인지를 판단해야 한다. 그렇기 때문에 이전 블록이 할당되었는지, 다음 블록이 할당 되었는지를 확인해보기위해, 헤더의 다음과 푸터의 이전에 있는 할당 여부를 살펴보아야 한다. 각각의 매크로 함수를 이용하여 할당 여부를 변수에 저장하게 된다. 그 다음 size도 구하기 위해서 헤더로 접근하여 GET_SIZE 함수를 이용하여 블록의 크기를 알아낸다.

첫번째 if문은 이전 블록과 다음 블록의 최 하위 비트가 둘다 1인 경우, 즉 이미 둘 다 할당 되어 있을 경우 합칠 것들이 존재하지 않으므로 bp를 리턴하는 동작으로만 끝이 난다.

```
125      /*case 2 :
126       이전 블럭 최하위 비트가 1이고(할당) 다음 블럭 최하위 비트가 0인 경우(비할당)
127       다음 블럭과 병합한 뒤 bp return
128       */
129      else if(prev_alloc && !next_alloc) {
130          size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
131          PUT(HDRP(bp), PACK(size, 0));
132          PUT(FTRP(bp), PACK(size, 0));
133          //PACK은 alloc에 결합하는 함수이다 size만큼 OR연산을 하는 것
134          //PUT은 이 포인터너 위치에다가 이 값을 넣겠다, 하고 쓰는 함수
135          // HDRP(헤더 포인터) , FTRP(footer 포인터)
136
137      /*case 3 : 이전 블럭 최하위 bit가 0이고,(비할당)
138       다음 블럭의 최하위 비트가 1인 경우(할당) 이전 블럭과 병합한 뒤 새로운 bp retrun*/
139      } else if(!prev_alloc && next_alloc) {
140          size += GET_SIZE(HDRP(PREV_BLKP(bp)));
141          //bp가 가리키는 블록의 사이즈 + 이전 블록의 사이즈
142          PUT(FTRP(bp), PACK(size, 0));
143          //bp의 푸터에 합친 사이즈와 할당이 안됐으므로 0을 넣음
144          PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
145          bp = PREV_BLKP(bp);
146          //bp는 bp의 이전 블록을 가리키게 된다.
147      }
```

두 번째는 이전 블록이 할당 되어있고, 다음 블록이 할당 되어있지 않을 경우에 대한 처리이다. 이 경우네는 다음 블록과 병합한 뒤에 bp를 리턴해야 하므로, size에 헤더의 이전 블록에 있던 사이즈를 size 변수에 더해준 다음, 각각의 헤더와 푸터에 size와 할당 여부를 PUT 함수를 이용하여 바꿔준다. 세 번째 경우는 이전 블록이 할당 되어있고, 다음 블록이 비할당인 경우에 대한 처리이다. 이 경우에는 이전 블록과 병합한 뒤에 bp를 리턴하게 된다. Size에 헤더의 이전 블록에 대해 size를 얻고 더한다. 다음 bp에 footer에 합친 사이즈를 세팅해주고, 할당이 되지 않았으므로 0을 넣어준다. 그리고 헤더에도 같은 size와 할당되지 않았다는 의미의 0을 넣어준다. 마지막으로 bp를 bp의 이전 블록을 가리키게 하도록 PREV_BLKP 매크로를 사용하여 반환된 값을 받아준다.

```
151     else {
152         //이전, 현재, 다음 블록을 다 합친다
153         size += GET_SIZE(HDRP(PREV_BLKP(bp))) + GET_SIZE(FTRP(NEXT_BLKP(bp)));
154         //이전 블록의 헤더와 푸터에 size할당이 안되었으므로 0 삽입하기
155         PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
156         PUT(FTRP(PREV_BLKP(bp)), PACK(size, 0));
157 
158         bp = PREV_BLKP(bp);
159         //bp는 이전 블로을 가리키게 한다.
160     }
161     last = bp; //마지막으로 탐색한 곳 저장하기
162     return bp;//리턴하rl
163     //병합된 블럭의 주소 bp return
164 }
```

네 번째 경우는 두 개의 블록이 모두 할당되지 않았을 경우이다. 이 경우에는 이전, 현재, 다음 블록을 합친 사이즈를 구하기 위해 이전 블록과 다음 블록의 사이즈와 현재 블록의 사이즈를 더해 size 변수에 세팅한다. 그런 다음, 이전에 했던 방법들과 동일하게 PUT 매크로 함수를 사용해서 헤더와 푸터에 size와 할당 여부 비트를 세팅해준 다음 bp를 이전 블록을 가리키게 한다. Next_fit 시에 다음부터 탐색해야 하므로 전역 변수 last에 마지막으로 탐색한 곳을 저장하고 bp을 리턴하며 함수를 종료하게 된다.

[find_fit]

```
65  //어느곳에 할당할지 찾는 함수
66  static void * find_fit(size_t asize) {
67 
68      //초기값을 last로 설정해준다
69      void *first = last;
70      //힙의 끝부분
71      void *end = mem_heap_hi();
72      //힙의 끝부분에 도달할때까지 계속 반복한다
73      while(first <end) {
74          //조건 : 비할당이거나, 사이즈가 같거나 큰 블록이라면
75          if(!GET_ALLOC(HDRP(first)) && (GET_SIZE(HDRP(first))>=asize)) {
76              //위치를 찾았다며 리턴을 해준다
77              return first;
78          }
79          //아니라면 블록을 이동한다
80          first = NEXT_BLKP(first);
81      }
82      return NULL;
83  }
```

Find_fit() 함수는 어느 곳에 할당할 지 찾는 함수이다. 할당하기 위해서는 free 블록을 찾고, 그 블록이 할당할 수 있는 블록인지, 사이즈가 되는지 여부를 판단하여 적절한 위치를 반환한다. 우선 first 포인터 변수에 이전에 위치를 담고 있는 last 전역 변수를 넣어주어서 탐색할 위치를 갱신한다. 이 방법은 next_fit으로, 탐색하고 난 다음의 위치부터 탐색하기 위해서 last에 마지막 탐색 위치를 넣어주고, 다음에 그 위치를 사용하는 형식으로 구현한 것이다. 그리고 end 포인터 변수에는 mem_heap_hi() 함수를 사용하여 힙의 끝부분을 구한다. 이렇게 구해진 구역으로, while 문을 돌면서 first부터 시작해서 end까지 동작을 하게 되는데, place를 하여 블록을 할당할 수 있는 위치를 찾는 조건은, 할당이 되어있지 않은 가용 블록이여야 하고, 사이즈가 같거나 큰 블록이여야 하므로, 이 조건에 부합한다면 이 위치를 리턴하고, 그렇지 않다면 NEXT_BLKP 매크로 함수를 이용하여 다음 주소로 first를 옮겨가며 합당한 위치를 찾는다. 만약에 합당한 위치가 없다면 NULL을 리턴하고 공간을 확장해야 한다.

[place]

```
84  //조건에 맞는 곳을 찾았다면 할당해주기
85  static void place(void *bp, size_t asize) {
86
87      size_t size = GET_SIZE(HDRP(bp));
88      //bp가 가리키는 블록의 사이즈 받기
89      //외부 단편화를 막이 위해서
90      if((size-asize)>=(2*DSIZE)){
91          //size-asize한 것이 >=16이면 블럭을 나누고 할당하기 시작
92          PUT(HDRP(bp), PACK(asize,1)); //asize만큼 할당, 할당됨을 알리는 1세팅하기
93          PUT(FTRP(bp), PACK(asize,1));
94
95          bp = NEXT_BLKP(bp); //bp는 다음 bp를 가리킨다. 헤더와 푸터 세팅이 완료되어서
96
97          PUT(HDRP(bp), PACK(size-asize,0));//size-asize만큼의 크기, 할당함 세팅
98          PUT(FTRP(bp), PACK(size-asize,0));
99
100     } else {
101         PUT(HDRP(bp), PACK(size,1)); //bp 헤더에 size 크기와 할당함 1 세팅
102         PUT(FTRP(bp), PACK(size,1));
103     }
104
105 }
```

공간을 할당할 곳을 find_fit() 함수로 찾았다면, 이제는 공간을 할당해 주어야 할 차례이다. 이 함수는 조건에 맞는 곳에 공간을 asize만큼 할당하는 함수이다. 우선 size 변수에 bp가 가리키는 블록의 사이즈를 받는다. 외부 단편화를 막기 위해서 size에서 asize를 뺀 만큼이 DISZE* 2 즉 최대 메모리 이용율을 넘게 되면 처리를 하게 되는데, asize만큼 헤더와 footer에 할당을 해주고, 할당을 해주게 되므로 할당 여부를 가리키는 비트는 1로 세팅을 해준다. 그리고 bp는 다음 bp를 가리키게 해 준 다음, size-asize 만큼의 사이즈를 세팅해주고, 할당 여부를 0으로 바꾸어 세팅을 해준다. 그 이외의 경우에는 단순히 bp 헤더에 size 크기와 할당함의 여부를 1을 세팅해준다.

[free]

```
274  void free (void *ptr) {
275      //할당된 블록을 free시키기
276      if(ptr == 0) return; //할당이 되어있지 않을 경우
277
278      size_t size = GET_SIZE(HDRP(ptr));//헤더의 사이즈를 저장한다
279      PUT(HDRP(ptr), PACK(size,0)); //할당 -> 가용
280      PUT(FTRP(ptr), PACK(size,0)); //할당 -> 가용
281
282      coalesce(ptr);
283      //할당안된 블록과 지금 free시킨 블록을 합치려고 함수 실행
284  }
```

다음 free 함수는 할당된 블록을 다시 가용 블록으로 돌려주는 함수이다. 만약에 들어온 ptr이 할당이 되어있지 않은 블록이라면 그냥 리턴을 하고 함수를 끝내주면 된다. 그런 경우가 아니라면 size에 헤더에서 사이즈를 구하여 세팅하기 위해서 HERP 매크로로 헤더에 접근한 뒤에 GET_SIZE 함수를 이용하여 블록의 사이즈를 얻는다. 그리고 나서 할당 블록의 할당 여부를 가용 블록 임을 알려주는 0으로 세팅을 하고, 나머지 사이즈를 세팅한 다음 free 후에 남은 블록들을 합치기 위해서 coalesce() 함수를 호출한다.

[malloc]

```
223  void *malloc (size_t size) {
224      size_t asize;
225      size_t extendsize;
226      char *bp;
227
228
229      if(size == 0) //할당 사이즈가 0이면 할당할 것이 없음
230          return NULL;
231
232      if(size <= DSIZE) {
233          asize = 2*DSIZE; //할당희망 사이즈가 8보다 작거나 같으면 16만큼 넣기
234      } else {
235      //8보다 클 경우 메모리 사이즈를 정해서 저장
236          asize = DSIZE * ((size+(DSIZE) + (DSIZE-1))/DSIZE);
237      }
238
239      if((bp = find_fit(asize)) != NULL) {
240          place(bp, asize);
241          return bp;
242      }
243      extendsize = MAX(asize, CHUNKSIZE);
244      //만약 할당 공간을 찾지 못하면 둘중에 큰 값을 찾는다
245      if((bp = extend_heap(extendsize/WSIZE)) == NULL) {
246          return NULL;
247      }
248      place(bp, asize);
249      //새롭게 커진 위치에 할당해준다.
250      last = NEXT_BLKP(bp);
251      //bp의 다음 블록을 가리키게 한다
252      return bp;
253
254
255  }
```

Malloc 함수는 이제껏 구현했던 함수들을 사용해서 메모리 사용을 원활하게 도와주는 동작을 하는 총체적인 함수라고 생각했다. Asize, extendsize, bp 등 각각의 변수들을 형에 맞춰 선언해 준 다음 각각의 조건 문을 이용하여 동작을 수행하였다.

만약 size가 0이라면 할당된 것이 없기 때문이므로 그대로 NULL을 리턴한다. 그리고 이 조건을 통과한 다음 조건은 할당을 희망하는 사이즈가 DSIZE 즉 8보다 작거나 같다면 16만큼을 asize에 넣어 세팅해준다. DSIZE를 더해주는 이유는, 헤더와 푸터의 사이즈를 포함하여 세팅을 하기 때문이라고 생각했다. 그리고 나서, 이전에 구현했던 find_fit() 함수를 사용하여 할당 위치를 찾고, 블록을 할당할 곳을 찾는다면 반환 값은 NULL이 아니므로 place 함수를 이용해서 asize 만큼을 할당하고, bp를 리턴하게 된다. 하지만 만약 find_fit()에서 반환되는 값이 NULL이라면 적절한 할당 공간을 찾지 못한 것이므로 asize와 CHUNKSIZE 둘 중에 큰 값을 찾은 뒤에 extend_heap() 함수로 공간을 확장 시켜준다. 그리고 나서 place 함수를 사용하여 새롭게 커진 위치에 할당을 시켜준 뒤에, 구현 방법에서 free 블록을 탐색 시에 next_fit()을 사용하므로 last 전역 변수에 다음 탐색할 곳의 위치를 세팅해준다. 그리고 마지막에는 bp를 리턴해주는 것으로 함수의 동작을 마무리 시켜준다.

[realloc]

```
289  void *realloc(void *oldptr, size_t size) {
290      size_t oldsize;
291      void *newptr;
292
293      if(size == 0) {
294          free(oldptr);
295          return 0;
296      }
297
298      if(oldptr == NULL) {
299          return malloc(size);
300      }
301      newptr = malloc(size);
302      if(!newptr) {
303          return 0;
304      }
305
306      oldsize = GET_SIZE(HDRP(oldptr));
307      //헤더가 가리키는 곳에서 사이즈를 읽는 것
308      if(size < oldsize) oldsize = size;
309      memcpy(newptr, oldptr, oldsize);
310
311      free(oldptr);
312      return newptr;
313  }
```

Realloc함수는 naive함수에서 사용했던 것과 거의 동일하다. 단지 oldsize = GET_SIZE(HERP(oldptr)) 이 부분이 조금 바뀌었을 뿐이다. 본래 naïve에서는 SIZE_PTR(p) 매크로 함수가 존재하였기 때문에 헤더에 있는 사이즈를 바로 읽어올 수 있었지만, implicit에는 이것에 대한 매크로 함수가 선언되어있지 않았으므로, 두 개의 정의된 매크로 함수를 사용해 HDRP 매크로 함수로 헤더에 접근하고 GET_SIZE 함수로 사이즈를 읽어와서 위의 매크로와 같은 역할을 하도록 구현하였다.

[calloc]

```
320   void *calloc (size_t nmemb, size_t size) {
321       size_t bytes = nmemb * size;
322       void *newptr;
323
324       newptr = malloc(bytes);
325       memset(newptr, 0, bytes);
326
327       return newptr;
328   }
```

Calloc도 마찬가지로 naive에서 구현했던 것과 다른 부분이 없다. 같은 동작을 하며, 테스트 과정인. /mdriver에서는 문제가 생기지 않지만 trace에서는 필요하다고 하여 그대로 구현을 해 놓았다.

구현 후 느낀 점

Malloc 이라는 것을 프로그래밍 시간에 잠시 들어보긴 했지만, 이렇게 많은 원리로 복잡하게 구현되는 줄은 몰랐다. 직접 메모리를 내가 구현한 소프트웨어적 측면으로 조절을 해 보면서 신기하기도 했고, 생각하는 부분이 너무 까다로웠지만 효율적인 프로그램을 관리하기 위해서는 어떻게 프로그래밍을 해야 하는지 조금이라도 감이 온 것 같다. 어려운 개념이고, 시간이 충분치 못해 테스트에서 완벽한 점수는 얻을 수 없었지만 시간이 된다면 100점이 나올 만큼 효율적인 메모리 관리를 하는 malloc을 구현해 보고 싶다는 생각이 들었다.