

2021시스템 프로그래밍

- Lab 04 -

제출일자	2021.12.06
분 반	01
이 름	조해창
학 번	2020XXXXX

Naïve

```
b201702897@eslab-server:~/malloclab-handout$ ./mdriver
Using default tracefiles in ./traces/
Measuring performance with a cycle counter.
Processor clock rate ~= 800.0 MHz

Results for mm malloc:
  valid  util    ops    secs   Kops  trace
  yes    94%      10  0.000001 18018 ./traces/malloc.rep
  yes    77%      17  0.000001 25612 ./traces/malloc-free.rep
  yes   100%      15  0.000001 17021 ./traces/corners.rep
* yes    71%    1494  0.000069 21604 ./traces/perl.rep
* yes    68%     118  0.000006 21276 ./traces/hostname.rep
* yes    65%   11913  0.000503 23700 ./traces/xterm.rep
* yes    23%    5694  0.000247 23048 ./traces/amptjp-bal.rep
* yes    19%    5848  0.000252 23232 ./traces/ccc-p-bal.rep
* yes    30%    6648  0.000290 22912 ./traces/cp-decl-bal.rep
* yes    40%    5380  0.000226 23781 ./traces/expr-bal.rep
* yes     0%   14400  0.000552 26070 ./traces/coalescing-bal.rep
* yes    38%    4800  0.000249 19301 ./traces/random-bal.rep
* yes    55%    6000  0.000232 25875 ./traces/binary-bal.rep
10     41%   62295  0.002625 23727

Perf index = 26 (util) + 40 (thru) = 66/100
b201702897@eslab-server:~/malloclab-handout$
```

구현 방법

[매크로 함수 및 상수]

```

39
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))
50
51 /*
~/malloclab-handout/mm.c [utf-8,unix][c] 0,45/145 28%

```

- ALIGNMENT : 메모리 접근 효율성을 높이기 위해 할당할 메모리의 크기를 4 또는 8의 배수로 설정하기 위한 매크로이다. 현재는 8로 정의 되어있다.
- ALIGN(size) : 입력받은 size를 기준으로 ALIGNMENT의 배수에 가장 근접한 값을 할당 받기 위한 매크로이다. 만약 9를 size로 입력했다면 $9 + (8 - 1) = 16 = 0001\ 0000$ 이고 이를 $\sim 0x7 = 1111\ 1000$ 과 AND 연산 하면 입력 받은 size와 가장 근접한 8의 배수인 16을 얻을 수 있다.
- SIZE_T_SIZE : size_t는 64bit machine에서 8byte 크기를 가지고 32bit machine에서 4byte 크기를 가지기 때문에 여러 machine 간의 portability 높이기 위한 매크로이다. 현재 실습환경은 64bit machine 이기 때문에 8을 얻을 수 있다.
- SIZE_PTR : 블럭의 사이즈 정보가 있는 header의 포인터를 반환하는 매크로이다.

[함수]

- mm_init :

```

51 /*
52 * mm_init - Called when a new trace starts.
53 */
54 int mm_init(void)
55 {
56     return 0;
57 }

```

naive에서 구현하지 않는다.

- malloc :

```

59 /*
60 * malloc - Allocate a block by incrementing the brk pointer.
61 *           Always allocate a block whose size is a multiple of the alignment.
62 */
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 }
```

[memlib.c]

```

17 /* private variables */
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 */
21
22 void *mem_sbrk(int incr)
23 {
24     char *old_brk = mem_brk;
25
26     if ((incr < 0) || ((mem_brk + incr) > mem_max_addr)) {
27         errno = ENOMEM;
28         fprintf(stderr, "ERROR: mem_sbrk failed. Ran out of memory...\n");
29         return (void *)-1;
30     }
31     mem_brk += incr;
32     return (void *)old_brk;
33 }
```

인자로 받은 size 크기의 메모리를 힙에 할당하는 함수이다. line 65에서 newsize를 size + SIZE_T_SIZE를 ALIGN한 크기로 초기화 한다. 이때 SIZE_T_SIZE는 헤더를 확보하기 위한 크기이다. line 66에서 newsize를 mem_sbrk의 인자로 넣어 반환 값을 p에 초기화한다. mem_sbrk는 인자로 받은 size 만큼 힙을 늘리고 성공하면 할당되기 이전 힙영역의 마지막 바이트를 가르키는 포인터(old_brk) 즉, 할당된 블럭의 시작 위치를 리턴한다. 실패하면 -1를 리턴한다. line 69 ~ 74에서 할당에 실패하면 NULL값을 리턴하고 성공하면 p를 SIZE_T_SIZE 만큼 이동하고 header에 입력받은 size 정보를 써준 뒤 할당된 블럭의 주소값을 반환한다.

- free :

```

78 /*
79 * free - We don't know how to free a block. So we ignore this call.
80 *        Computers have big memories; surely it won't be a problem.
81 */
82 void free(void *ptr)
83 {
84 }
```

naive에서 구현하지 않았다.

- realloc :

```
86  /*
87  * realloc - Change the size of the block by mallocing a new block,
88  *           copying its data, and freeing the old block. I'm too lazy
89  *           to do better.
90  */
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 }
```

realloc 의 본래 목적은 이미 할당 되어있는 메모리 블럭의 data를 복사하여 새로운 크기의 블럭으로 재할당하고 free를 호출하여 이전 블럭을 가용블럭으로 만드는 함수 이지만, naive에서는 free가 구현되어 있지 않기 때문에 메모리 이용율이 낮다.

line 97 ~ 100 에서 인자로 받은 size가 0이면 이전 블록을 가용블럭으로 만들어야 하지만 naive에서는 지원하지 않는다.

line 103 ~ 105 에서 인자로 받은 oldptr이 NULL 이면 인자로 받은 size 만큼 malloc하여 새로 할당 된 블럭의 주소값을 반환한다.

위 두 조건에 만족하지 않는다면 line 107 에서 newptr 를 size 만큼 malloc 한 주소로 초기화한다. line 110 ~ 112 에서 만약 할당에 실패 했으면 0을 리턴한다.

line 115 ~ 117에서 만약 oldsize(이전 블럭의 크기)가 재할당할 크기보다 클 경우 재할당 될 사이즈로 초기화한뒤 memcpy를 통해 newptr에 oldptr을 oldsize 만큼 복사한다.

line 120은 이전 블럭을 가용블럭으로 만들어야하지만 naive에서는 지원하지 않는다.

- calloc :

```

125 /*
126  * calloc - Allocate the block and set it to zero.
127 */
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 }

```

`nmemb * size` 크기의 블럭을 할당하고 0으로 초기화하는 함수이다.

line 130 ~ 133에서 `bytes` 를 `nmemb * size` 으로 초기화한뒤, `bytes` 크기만큼 `malloc` 한 블럭을 0으로 초기화한다. line 136에서 초기화된 블럭의 포인터를 반환한다.

implicit

```

b201702897@eslab-server:~/malloclab-handout$ ./mdriver
Using default tracefiles in ./traces/
Measuring performance with a cycle counter.
Processor clock rate ~= 4000.0 MHz

Results for mm malloc:
    valid  util   ops    secs      Kops  trace
    yes    34%     10  0.000000  46620 ./traces/malloc.rep
    yes    28%     17  0.000000  48227 ./traces/malloc-free.rep
    yes    96%     15  0.000000  41322 ./traces/corners.rep
* yes    81%    1494  0.000086  17471 ./traces/perl.rep
* yes    75%    118  0.000002  52550 ./traces/hostname.rep
* yes    91%   11913  0.000971  12271 ./traces/xterm.rep
* yes    91%    5694  0.002236   2546 ./traces/amptjp-bal.rep
* yes    91%    5848  0.001479   3954 ./traces/cccp-bal.rep
* yes    95%    6648  0.004393   1513 ./traces/cp-decl-bal.rep
* yes    97%    5380  0.004433   1214 ./traces/expr-bal.rep
* yes    66%   14400  0.000283  50844 ./traces/coalescing-bal.rep
* yes    90%    4800  0.004614   1040 ./traces/random-bal.rep
* yes    55%    6000  0.003284   1827 ./traces/binary-bal.rep
10     83%   62295  0.021780   2860

Perf index = 54 (util) + 40 (thru) = 94/100
b201702897@eslab-server:~/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 #define WSIZE 4
45 #define DSIZE 8
46 #define CHUNKSIZE (1 << 12)
47 #define OVERHEAD 8
48 #define MAX(x, y) ((x) > (y) ? (x) : (y))
49 #define PACK(size, alloc) ((size) | (alloc))
50 #define GET(p) (*(unsigned int *)(p))
51 #define PUT(p, val) (*(unsigned int *)(p) = (val))
52 #define GET_SIZE(p) (GET(p) & ~0x7)
53 #define GET_ALLOC(p) (GET(p) & 0x1)
54 #define HDRP(bp) ((char*)(bp) - WSIZE)
55 #define FTRP(bp) ((char*)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)
56 #define NEXT_BLKP(bp) ((void*)(bp) + GET_SIZE(HDRP(bp)))
57 #define PREV_BLKP(bp) ((void*)(bp) - GET_SIZE(HDRP(bp)) - WSIZE))
58
59 static void *extend_heap(size_t words);
60 static void *find_fit(size_t size);
61 static void *coalesce(void *bp);
62 static void place(void *bp, size_t asize);
63
64 static char *heap_listp;
65 static char *last_bp;

```

[상수]

- ALIGNMENT, ALIGN : naive와 동일하다.
- WSIZE : 1word 크기를 4로 정의한다.
- DSIZE : double word 크기를 8로 정의한다.
- CHUNKSIZE : 2^{12} (4KB) 으로 초기 heap의 크기를 정의한다.
- OVERHEAD : header + footer의 크기를 정의한다. 실제 데이터가 저장되는 공간이 아니므로 overhead가 된다.

[매크로 함수]

- MAX(x, y) : x, y 중 더 큰 값을 반환한다.
- PACK(size, alloc) : header 와 footer에 쉽게 저장하기 위해 size와 alloc 을 하나의 word(4byte)로 묶는다. size는 상위 28bit, alloc은 하위 3bit에 해당한다.
- GET(p) : 포인터 p 가 가르키는 위치에서 한 word 크기의 값을 읽는다.
- PUT(p, val) : 포인터 p 가 가르키는 한 word크기의 위치에 val을 저장한다.
- GET_SIZE(p) : header 와 footer 에서 block size를 읽기 위해서 포인터 p가 가르키는 위치에서 한 word를 읽은 다음 하위 3bit를 버린다. ~0x7은 11...1111000 이므로 AND연산하여 상위 28bit에 해당하는 size 정보만 가져올 수 있다.
- GET_ALLOC(p) : 포인터 p가 가르키는 위치의 한 word를 읽고 하위 1bit를 읽어서 block 할당 여부를 확인한다. 할당된 블록이면 0 아니면 1이다. 0x1 즉 00000001와 AND연산하여 최하위 1bit에 해당하는 가용여부 정보만 가져올 수 있다.

- HDRP(bp) : 주어진 포인터 bp의 header 주소를 계산한다. header는 포인터 bp의 한 word 앞에 위치한다.
- FTRP(bp) : 주어진 포인터 bp의 footer 주소를 계산한다. footer는 주어진 포인터 bp의 다음 블록((char*)(bp) + GET_SIZE(HDRP(bp)))에서 두 word 전에 위치한다.
- NEXT_BLKP(bp) : 주어진 포인터 bp를 이용하여 다음 block의 주소를 계산한다. 주어진 포인터 bp의 header에서 블록 사이즈(GET_SIZE(HDRP(bp)))를 읽어서 더해준다. 다음 블록의 bp를 반환한다.
- PREV_BLKP(bp) : 주어진 포인터 bp를 이용하여 이전 block의 주소를 계산한다. 주어진 포인터 bp의 footer에서 블록 사이즈(GET_SIZE(HDRP(bp)) - WSIZE)를 읽어서 빼준다. 이전 블록의 bp를 반환한다.

[전역 변수]

- heap_list : first block의 포인터이다.

[함수]

- mm_init :

```

69 int mm_init(void) {
70     // 초기 16bytes 크기의 empty heap 생성
71     if ( (heap_listp = mem_sbrk(4 * WSIZE)) == (void*)-1 ) return -1;
72     PUT(heap_listp, 0); // 정렬을 위해서 의미없는 값 삽입
73     PUT(heap_listp + (1*WSIZE), PACK(OVERHEAD, 1)); // prologue header
74     PUT(heap_listp + (2*WSIZE), PACK(OVERHEAD, 1)); // prologue footer
75     PUT(heap_listp + (3*WSIZE), PACK(0, 1)); // epilogue header
76
77     heap_listp += DSIZE;
78     last_bp = heap_listp;
79
80     if ( extend_heap(CHUNKSIZE/WSIZE) == NULL ) return -1;
81     return 0;
82 }

```

line 71에서 16bytes ($4 * \text{WSIZE}$) 만큼 힙 공간을 초기화한다.

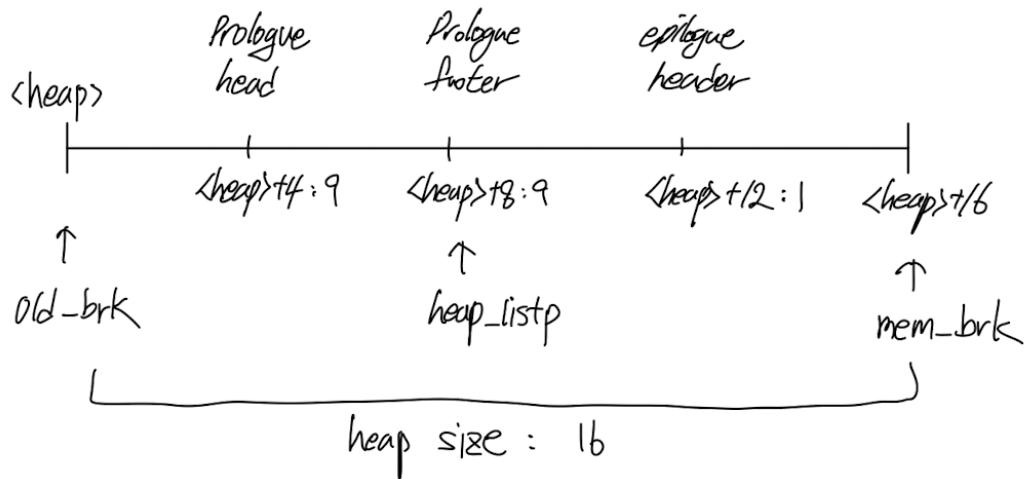
line 72 ~ line 75에서 한 word 간격마다 prologue header, prologue footer, epilogue header를 초기화한다.

line 77 ~ 78에서 heap_list는 8bytes 만큼 이동시켜서 prologue footer 시작 주소를 가르키도록 한다. 마찬가지로 last_bp도 prologue footer를 가르키도록 한다. last_bp는 next fit에 사용할 마지막으로 탐색한 블록의 주소를 저장한다.

line 80에서 extend_heap을 CHUNKSIZE의 워드단위로 환산한 값($\text{CHUNKSIZE} / \text{WSIZE}$)을 인자로 주어서 호출한다. 따라서 empty heap은 CHUNKSIZE 바이트만큼 확장되고 초기 가용 블록이 생성된다.

```
(gdb) n
75         heap_listp += (DSIZE);
(gdb) n
77         if ( extend_heap(CHUNKSIZE/WSIZE) == NULL ) return -1;
(gdb) x heap_listp
0x5555556e548 <heap+8>: 0x00000009
(gdb) █
```

다음은 mm_init 호출 후, extend_heap 호출 전 heap 상태이다.



```
B+ 68     int mm_init(void) {
69         // ^, ^ *^ 16bytes ^-^ ^ *^ , ^ empty heap ^, ^, ^
70         if ( (heap_listp = mem_sbrk(4 * WSIZE)) == NULL ) return -1;
71         PUT(heap_listp, 0); // ^, ^ +^ , ^ ^, ^ -^ , ^ ^, ^ +^ , ^
72         PUT(heap_listp + (1*WSIZE), PACK(OVERHEAD, 1)); // prologue h
73         PUT(heap_listp + (2*WSIZE), PACK(OVERHEAD, 1)); // prologue f
74         PUT(heap_listp + (3*WSIZE), PACK(0, 1)); // epilogue header
75         heap_listp += (DSIZE);
76
77         if ( extend_heap(CHUNKSIZE/WSIZE) == NULL ) return -1;
78         return 0;
79     }
80
81     /*
82      * malloc
```

native process 59248 In: mm_init L75 PC: 0x5555555594b7
(gdb) x/4x heap_listp
0x55555556e540 <heap>: 0x00000000 0x00000009 0x00000009 0x00000001

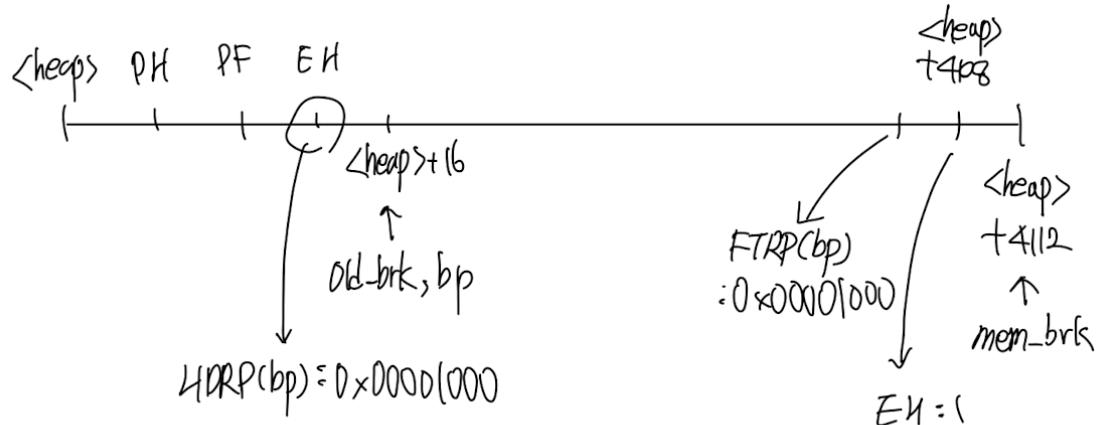
- extend_heap :

```

189 static void *extend_heap(size_t words) {
190     char *bp;
191     size_t size;
192     // words를 짹수로 변환한다.
193     size = (words%2) ? (words+1) * WSIZE : words * WSIZE;
194
195     // 힙 사이즈 늘리기를 성공하면 bp에 이전 mem_brk 저장
196     if ((long)(bp = mem_sbrk(size)) == -1) return NULL;
197
198     // header, footer에 size 정보 초기화, epilogue header 초기화
199     PUT(HDRP(bp), PACK(size, 0));
200     PUT(FTRP(bp), PACK(size, 0));
201     PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1));
202
203     return coalesce(bp);
204 }

```

다음은 extend_heap 호출 후 heap의 상태이다.



line 193에서 `words` 를 짹수 값으로 변환한다.

line 196에서 힙 사이즈를 늘리고 성고하면 `bp`를 이전 `mem_brk` 으로 초기화 하고, 아니면 `NULL`을 반환한다.

line 199 ~ 201에서 header, footer의 사이즈 정보와 가용 여부를 초기화 하고, 다음 블럭의 헤더가 될 epilogue header의 정보를 초기화한다.

- `coalesce` :

```

236 static void *coalesce(void *bp){
237     // 이전 블록의 가용여부
238     size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
239     // 다음 블록의 가용여부
240     size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
241     // 현재 블록의 크기
242     size_t size = GET_SIZE(HDRP(bp));
243
244     /*
245      * case 1: 이전, 다음 블록 모두 비가용인 경우
246      * 블록을 병합하지 않고 bp 반환
247      */
248     if (prev_alloc && next_alloc){
249         return bp;
250     }
251     /*
252      * case 2: 이전 블록은 비가용, 다음 블록은 가용인 경우
253      * 다음 블록과 병합한 뒤 bp 반환
254      */
255     else if (prev_alloc && !next_alloc){
256         size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
257         PUT(HDRP(bp),PACK(size,0));
258         PUT(FTRP(bp), PACK(size,0));
259     }
260     /*
261      * case 3: 이전 블록은 가용, 다음 블록은 비가용인 경우
262      * 이전 블록과 병합한 뒤 bp 반환
263      */
264     else if(!prev_alloc && next_alloc){
265         size+= GET_SIZE(HDRP(PREV_BLKP(bp)));
266         PUT(FTRP(bp), PACK(size,0));
267         PUT(HDRP(PREV_BLKP(bp)), PACK(size,0));
268         bp = PREV_BLKP(bp);
269     }
270     /*
271      * case 4: 이전, 다음 블록 모두 가용인 경우
272      * 이전, 다음 블록과 모두 병합한 뒤 bp 반환
273      */
274     else {
275         size+= GET_SIZE(HDRP(PREV_BLKP(bp))) + GET_SIZE(FTRP(NEXT_BLKP(bp)));
276         PUT(HDRP(PREV_BLKP(bp)), PACK(size,0));
277         PUT(FTRP(NEXT_BLKP(bp)), PACK(size,0));
278         bp = PREV_BLKP(bp);
279     }
280     // next fit 을 위해 last_bp를 초기화된 bp로 초기화한다.
281     last_bp = bp;
282     return bp;
283 }

```

coalesce의 목표는 블록을 free했을 때 앞, 뒤 가용 블록과 결합시켜 미래에 생길 외부단편화를 줄이는 것(Utilization을 높이는 것)이다.

line 237 ~ 242에서 주어진 bp의 이전 블럭의 가용여부, 다음 블럭의 가용여부, 현재 블록의 크기를 각 변수에 초기화한다.

line 248 ~ 279에서 이전 블럭과 다음 블럭이 가용여부의 가능한 경우의 수 4가지에 대해 처리해준다. case 3을 예로 들면 이전 블럭이 가용(prev_alloc == 0), 다음 블럭이 비가용(next_alloc == 1) 일때 성립한다. 주어진 bp의 크기인 size에 이전 블럭의 크기를 더하여 초기화한다. 현재 bp의 footer, 이전 블럭의 header의 size 와 alloc 값을 초기화하고 bp는 이전 블럭을 가르키도록 한다.

- find_fit :

[first fit]

```
234 static void *find_fit(size_t asize){  
235     // first fit 검색을 수행한다.  
236     void *bp;  
237  
238     /* bp는 heap_listp (prologue footer) 에서 부터 epilogue header 까지  
239     * 블럭 단위로 이동한다.  
240     */  
241     for(bp = heap_listp; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKP(bp)){  
242         if ( !GET_ALLOC(HDRP(bp)) && (asize <= GET_SIZE(HDRP(bp))) ) {  
243             /* 해당 블럭이 가용블럭이고, 블럭의 크기가 asize보다 작다면  
244                 * 해당 블럭의 bp 를 반환한다.  
245                 */  
246             return bp;  
247         }  
248     }  
249     // 알맞은 블록을 찾지 못했다면 NULL을 반환한다.  
250     return NULL;  
251 }
```

[next fit]

```
279 static void *find_fit(size_t asize){  
280     // next fit  
281     void *bp;  
282     /*  
283     * bp는 last_bp부터 epilogue header 까지  
284     * 블록 단위로 이동한다.  
285     */  
286     for(bp = last_bp; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKP(bp)){  
287         if ( !GET_ALLOC(HDRP(bp)) && (asize <= GET_SIZE(HDRP(bp))) ) {  
288             /* 할당 가능한 블록이 fit 되었다면  
289                 * last_bp를 초기화하고 해당 블록의 bp 를 반환한다.  
290                 */  
291             last_bp = bp;  
292             return bp;  
293         }  
294     }  
295     /*  
296     * 만약 last_bp부터 탐색해서 알맞은 블록을 찾지 못했다면  
297     * first fit 방법과 동일하게 last_bp 전까지 탐색한다.  
298     */  
299     for(bp = heap_listp; bp < last_bp; bp = NEXT_BLKP(bp)){  
300         if ( !GET_ALLOC(HDRP(bp)) && (asize <= GET_SIZE(HDRP(bp))) ) {  
301             last_bp = bp;  
302             return bp;  
303         }  
304     }  
305     return NULL;  
306 }
```

first fit 방법은 첫번째 블럭부터 탐색하여 맨 처음 fit된 블록을 찾는 방식이다. Best case가 아닌 이상 모든 블럭의 수의 비례해서 탐색시간이 소요되기 때문에 Throughput(처리량)이 좋지 않다. 그래서 Throughput 이 더 높은 next fit을 이용하여 find fit을 구현 하였다.

line 286 ~ 294 에서 초기값을 마지막으로 탐색한 블록의 주소(last_bp)으로 하여 epilogue footer 까지 블럭단위로 탐색한다. line 287에서 만약 탐색한 블록이 가용블록이고, 할당할 사이즈보다 같거나 크다면 last_bp 를 초기화 하고 fit된 블록의 bp를 반환한다.

line 300 ~ 305에서 만약 next fit을 이용한 방법(last_bp 부터 탐색하는 방법)이 실패하였다 면 first fit을 이용하여 last_bp 전 까지 탐색한다. 만약 알맞은 블록을 찾았다면 last_bp를 초기화하고 해당 bp값을 반환한다.

만약 알맞은 블록을 찾지 못했다면 NULL을 반환한다.

- place :

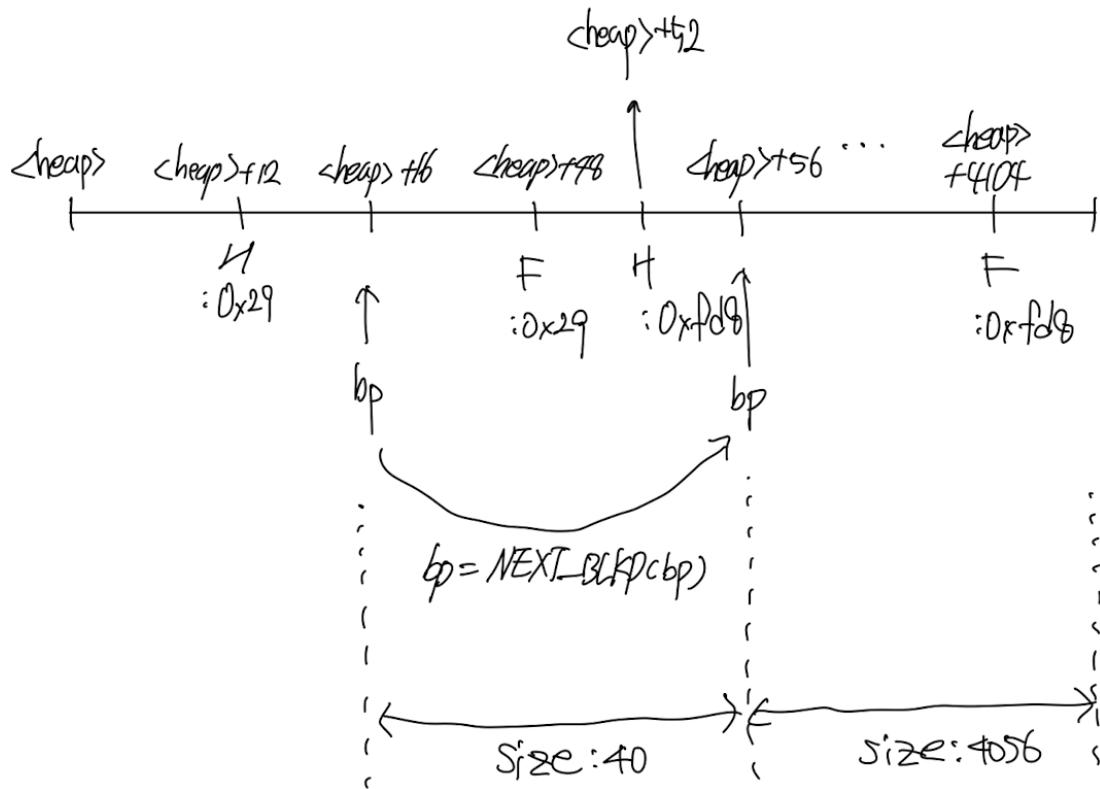
```
342 static void place(void *bp, size_t asize){
343     // bp는 find fit으로 찾은 bp, asize는 할당할 크기이다.
344
345     // csize를 주어진 bp의 크기로 초기화한다.
346     size_t csize = GET_SIZE(HDRP(bp));
347
348     // 블럭 쪼개기의 최소 크기를 8bytes로 한다. header(4), footer(4), payload(0)
349     if ((csize-asize) >= (DSIZE)){
350         PUT(HDRP(bp), PACK(asize,1)); // header의 size, alloc 정보 초기화
351         PUT(FTRP(bp), PACK(asize,1)); // footer의 위치가 바뀜과 동시에 size, alloc
352                                         // 정보 초기화
353         bp = NEXT_BLKP(bp);          // bp를 다음 블록의 위치로 초기화
354         PUT(HDRP(bp), PACK(csize-asize,0)); // header의 size, alloc 정보 초기화
355         PUT(FTRP(bp), PACK(csize-asize,0)); // footer의 size, alloc 정보 초기화
356     }
357     else{
358         /*
359         * 남은 공간이 가용 공간으로써 충분한 공간이 아니라면
360         * 해당 bp를 csize 크기의 비가용 블록으로 초기화한다.
361         */
362         PUT(HDRP(bp), PACK(csize,1));
363         PUT(FTRP(bp), PACK(csize,1));
364     }
365 }
```

place의 목표는 가용블록보다 작은 크기의 구조체를 할당할 때, 할당 후 내부에 남은 공간을 또 다른 가용 블록으로 쪼개어서 내부 단편화를 줄이는 것(Utilization을 높이는 것)이다.

line 349 ~ 356에서 할당하고도 남는 내부공간의 크기가 header 와 footer의 크기(8bytes), 보다 크다면 블록을 쪼개는 작업을 한다. 따라서 가용블록의 최소 payload의 크기는 0이된다.

line 357 ~ 364에서 남는 공간이 가용공간으로 충분하지 않으면 쪼개는 과정없이 현재 블록 사이즈만큼만 할당한다.

다음은 place에 인자로 bp 가 <heap>+16, assize 가 0x28 주어져 호출 되었을 때 heap의 상황이다.



- free :

```

120 void free (void *bp) {
121     if ( bp == 0 ) return;
122
123     // free할 블록의 사이즈
124     size_t size = GET_SIZE(HDRP(bp));
125
126     // free할 블록의 header, footer의 size, alloc을 초기화한다.
127     PUT(HDRP(bp),PACK(size,0));
128     PUT(FTRP(bp), PACK(size,0));
129
130     // coalesce를 호출하여 앞 뒤 가용블록과 결합한다.
131     coalesce(bp);
132 }
```

free할 블럭의 bp값을 인자로 받아서 header와 footer의 정보를 초기화해주고, coalesce를 호출하여 앞 뒤 가용블록과 결합할 수 있도록 한다.

- malloc :

```

87 void *malloc (size_t size) {
88     size_t asize; // Adjusted block size
89     size_t extendsize; // Amount to extend heap if no fit
90     char *bp;
91
92     if (size <= 0) return NULL;
93
94     // 할당할 사이즈를 인접한 8의 배수로 조정한다.
95     asize = DSIZE* ( (size + (DSIZE)+(DSIZE-1)) / DSIZE );
96
97     if ((bp = find_fit(asize)) != NULL){
98         /*
99          * find_fit을 호출하여 적절한 가용블록을 찾는다.
100         * 찾은 가용블록의 bp와 asize를 인자로 하여 place를 호출한다.
101         */
102         place(bp,asize);
103         return bp;
104     }
105
106    /* 적절한 가용블럭을 찾지 못한경우
107     * asize와 CHUNKSIZE 중 더 큰값만큼 heap공간을 늘려서 블럭을 위치시킨다.
108     */
109    extendsize = MAX(asize,CHUNKSIZE);
110    if ( (bp=extend_heap(extendsize/WSIZE)) == NULL){
111        return NULL;
112    }
113    place(bp,asize);
114    return bp;
115 }
```

malloc 의 목표는 위에서 구현한 함수(extend_heap, find_fit, place)를 이용하여 할당하려는 사이즈를 함수내에서 조정하여 힙 내부에 할당하는 것이다.

line 95 에서 asize를 할당하려는 크기보다 큰 인접한 8의 배수로 조정하여 초기화한다.

line 97 ~ 104 에서 find_fit을 이용하여 적절한 가용블록을 찾아서 place 함수를 호출한다. 만약 find_fit이 실패하면 line 109 ~ 114 에서 extend_heap을 호출하여 heap의 공간을 늘려서 블록을 할당한다.

- realloc :

```

151 void *realloc(void *bp, size_t size) {
152     if ( size <= 0 ) {
153         // 입력 크기가 0 보다 작거나 같으면 free해준다.
154         free(bp);
155         return 0;
156     }
157     if ( bp == NULL ) {
158         // bp 가 NULL 이면 입력 사이즈만큼 malloc해준다.
159         return malloc(size);
160     }
161     // size만큼 malloc 한 주소로 newptr을 초기화한다.
162     void *newptr = malloc(size);
163     if ( newptr == NULL ) {
164         return 0;
165     }
166     // 이전 블록의 정보를 복사한다.
167     size_t oldsize = GET_SIZE(HDRP(bp));
168     if ( size < oldsize ) oldsize = size;
169     memcpy(newptr, bp, oldsize);
170
171     // 이전 블록을 free한다.
172     free(bp);
173
174     return newptr;
175 }
176 }
```

naïve에서 구현한 방식과 동일하지만 이전 블록이 free가 된다는 점에서 메모리 이용률을 높일 수 있게되었다.

- calloc :

```

183 void *calloc (size_t nmemb, size_t size) {
184     size_t bytes = nmemb * size;
185     void *newptr;
186
187     newptr = malloc(bytes);
188     memset(newptr, 0, bytes);
189
190     return newptr;
191 }
```

naïve에서 구현한 방식과 동일하다.