2018 시스템 프로그래밍

- Lab 09 -

제출일자	2018.12.08
분 반	01
이 름	이규봉
학 번	201502085

우선 mm-naive.c에 정의되어 있던 각종 매크로와 전처리 지시사들에 대한 설명하겠습니다.

1 – 디버깅 기능

#define DEBUG

#ifdef DEBUG

define dbg_printf(...) printf(__VA_ARGS__)

#else

define dbg_printf(...)

#endif

DEBUG를 정의한 후 만약 정의되어 있다면 dbg_printf()가 __VA__ARGS__를 출력하도록 만듭니다.

만약 throughput 향상을 위해 디버깅 기능을 빼고 싶다면, //#define DEBUG로 주석처리 해 주면 dbg_printf()가 아무 일도 하지 않게 됩니다.

2 – ALIGN(size)에 대해

#define ALIGN(size) (((size) + (ALIGNMENT-1)) & ~0x7)

ALIGN(size)는 size를 받아 (((size) + (ALIGNMENT-1)) & ~0x7)로 만드는 일종의 함수 역할을 해 주는 매크로 입니다. 하지만, 실제로 함수로 구현해 사용하면 스택에서 size를 input으로 받아오고 값을 리턴해야 하므로, #define문으로 매크로 처리해 사용하면 throughput을 향상시키는데 도움이 됩니다.

이 매크로가 하는 일은 size를 받아 8의 배수가 되도록 내림하는 것입니다. 예를 들어 1110(2)를 size에 넣으면 뒤의 세 자리를 버려 1000(2) 이 되므로 십진수로 8, 즉 8의 배수가 된 것을 확인할 수 있습니다.

또한 여기서 ALIGNMENT는 64비트 운영체제 이므로, 8로 정의해 사용합니다.

3. SIZE_PTR(p)에 대해

#define SIZE_T_SIZE (ALIGN(sizeof(size_t)))

#define SIZE_PTR(p) ((size_t*)(((char*)(p)) - SIZE_T_SIZE))

sizeof(size_t)는 size_t가 64비트 운영체제에서 64비트 (즉, 8바이트)이므로 언제나 8이 됩니다.

(만약 32비트 운영체제에서 진행한다면 4가 됩니다.)

SIZE_PTR(p)는 포인터 p를 인자로 받아 char* 포인터로 캐스팅 합니다. 캐스팅 하는 이유는, 포인터 산술연산을 위한 것으로, SIZE_T_SIZE만큼 값을 빼기했을 때, char의 크기가 1바이트이므로 언제나 8바이트 뒤의 주소를 얻을 수 있게 합니다. 즉, SIZE_PTR(p)는 블록의 헤더에 들어 있는 값(블록의 크기)을 size_t로 캐스팅한 값으로 replace하는 매크로입니다.

1. Naive

naive의 소스 코드는 기존에 주어진 것을 그대로 사용하였습니다.

mdriver의 실행 결과는 아래와 같습니다.

```
Results for mm malloc:
   valid  util   ops    secs     Kops  trace
   yes     94%      10  0.000000 82258 ./traces/malloc.rep
   yes     77%      17  0.000000106127 ./traces/malloc-free.rep
   yes    100%      15  0.000000 90412 ./traces/corners.rep
 * yes     71%    1494  0.000010153842 ./traces/perl.rep
 * yes     68%     118  0.000001164679 ./traces/hostname.rep
 * yes     65%   11913  0.000078151920 ./traces/xterm.rep
 * yes     23%    5694  0.000068 84208 ./traces/amptjp-bal.rep
 * yes     19%    5848  0.000071 82769 ./traces/cccp-bal.rep
 * yes     30%    6648  0.000082 81423 ./traces/cp-decl-bal.rep
 * yes     40%    5380  0.000062 87051 ./traces/expr-bal.rep
 * yes      0%   14400  0.000177 81140 ./traces/coalescing-bal.rep
 * yes     38%    4800  0.000052 92014 ./traces/random-bal.rep
 * yes     55%    6000  0.000072 83812 ./traces/binary-bal.rep
10         41%   62295  0.000672 92729

Perf index = 26 (util) + 40 (thru) = 66/100
```

void *malloc(size_t size) {

```
  int newsize = ALIGN(size + SIZE_T_SIZE);
  /*
        newsize에 size에서 8바이트 더한 값을 8의 배수로 내림합니다.
  */
  unsigned char *p = mem_sbrk(newsize);
  // sbrk를 호출하여 brk포인터에 newsize를 더해 힙을 늘리고, 새로운 힙을 가리키는 포인터 변
  //수 p를 정의합니다.
  if ((long)p < 0)
    return NULL;
  // p가 음수일 때 NULL을 리턴합니다.
  else {
    p += SIZE_T_SIZE;
    *SIZE_PTR(p) = size;
    return p;
 /*
위 에러가 생기는 경우가 아니라면,
  이 위치에 (즉 SIZE_PTR(p+SIZE_T_SIZE) 주소의) size를 대입합니다. 할당할 크기가 size와 같아야 하므로, 8바이트를 빼는 SIZE_PTR 매크로를 사용하기 위해 8 바이트를 미리 더해준 것입니다. 이렇게 하면, 1워드 (여기선 8바이트) 만큼의 size값을 갖는 헤더를 사용하고, p는 헤더 앞의 포인터를 반환하게 되므로, 정의한 매크로 SIZE_T_SIZE()와 부합하게 됩니다.
*/
  }
}
```

void *realloc(void *oldptr, size_t size) {

```
  size_t oldsize;
  void *newptr;

  /*
  size 인자가 0이라면, free와 완전히 같은 역할을 합니다. 따라서, free에 oldptr을 넣어 호출해주기만 하
  면 됩니다.
  */
  if(size == 0) {
    free(oldptr);
    return 0;
  }

  /*
  oldptr이 NULL이라면, malloc과 완전히 같은 역할을 합니다. 따라서 malloc을 호출하여 size를 인자로 넘겨
  주면 됩니다.
  */
  if(oldptr == NULL) {
    return malloc(size);
  }
/*
위 특수한 두 가지 경우가 아니라면, realloc은 malloc으로 'size'를 크기로 갖는 메모리 블럭을 할당합니다.
할당한 블럭을 가리키는 포인터 변수 newptr을 정의합니다.
*/
  newptr = malloc(size);

if(!newptr) {
    return 0;
// 만약 malloc이 0을 NULL을 리턴한 경우 (할당에 실패한 경우) realloc은 0을 리턴합니다.
  }
```

```
  /* SIZE_PTR 매크로를 사용해 원래 포인터가 가리키던 블록의 사이즈 값을 얻습니다. */
  oldsize = *SIZE_PTR(oldptr);

  if(size < oldsize) oldsize = size;
  // 옮기려는 블록의 크기가 기존의 블록 크기보다 작다면 size만큼만 옮기도록 합니다.
  memcpy(newptr, oldptr, oldsize);
  // memcpy로 블록의 내용을 복사합니다.

  free(oldptr);
 // 기존의 블록을 가리키던 oldptr은 제거합니다.
  return newptr;
}
```

void *calloc (size_t nmemb, size_t size) {

```
  size_t bytes = nmemb * size;
  // nmemb * size 값을 bytes에 대입합니다.
  void *newptr;

  newptr = malloc(bytes);
  // newptr에 bytes 만큼 malloc 합니다.
  memset(newptr, 0, bytes);
  // newptr이 가리키는 값을 bytes만큼 0으로 채웁니다.

  return newptr;
  // newtptr 리턴
}
```

2. implicit

```
Measuring performance with a cycle counter.
Processor clock rate ~= 2097.6 MHz

Results for mm malloc:
   valid  util   ops    secs     Kops  trace
   yes    34%      10  0.000000 51411 ./traces/malloc.rep
   yes    28%      17  0.000000 68312 ./traces/malloc-free.rep
   yes    96%      15  0.000000 52178 ./traces/corners.rep
 * yes    81%    1494  0.000060 25002 ./traces/perl.rep
 * yes    75%     118  0.000002 75762 ./traces/hostname.rep
 * yes    91%   11913  0.000887 13437 ./traces/xterm.rep
 * yes    91%    5694  0.002240  2542 ./traces/amptjp-bal.rep
 * yes    91%    5848  0.001447  4041 ./traces/cccp-bal.rep
 * yes    95%    6648  0.007184   925 ./traces/cp-decl-bal.rep
 * yes    97%    5380  0.012958   415 ./traces/expr-bal.rep
 * yes    66%   14400  0.000136106211 ./traces/coalescing-bal.rep
 * yes    90%    4800  0.012824   374 ./traces/random-bal.rep
 * yes    55%    6000  0.014424   416 ./traces/binary-bal.rep
10        83%   62295  0.052161  1194

Perf index = 54 (util) + 40 (thru) = 94/100
b201502085@2018-sp:~/Malloclab/malloclab-handout$
```

next_fit 사용 시

```
Results for mm malloc:
   valid  util   ops    secs     Kops  trace
   yes    34%      10  0.000000 39502 ./traces/malloc.rep
   yes    28%      17  0.000000 72038 ./traces/malloc-free.rep
   yes    96%      15  0.000000 36543 ./traces/corners.rep
 * yes    86%    1494  0.001617   924 ./traces/perl.rep
 * yes    75%     118  0.000016  7479 ./traces/hostname.rep
 * yes    91%   11913  0.379151    31 ./traces/xterm.rep
 * yes    99%    5694  0.029365   194 ./traces/amptjp-bal.rep
 * yes    99%    5848  0.029791   196 ./traces/cccp-bal.rep
 * yes    99%    6648  0.057665   115 ./traces/cp-decl-bal.rep
 * yes   100%    5380  0.035042   154 ./traces/expr-bal.rep
 * yes    66%   14400  0.000130110860 ./traces/coalescing-bal.rep
 * yes    93%    4800  0.022482   214 ./traces/random-bal.rep
 * yes    55%    6000  0.222170    27 ./traces/binary-bal.rep
10        86%   62295  0.777431    80

Perf index = 56 (util) + 3 (thru) = 59/100
b201502085@2018-sp:~/Malloclab/malloclab-handout$
```

first_fit 사용 시

```
Results for mm malloc:
   valid  util   ops    secs     Kops  trace
   yes    34%      10  0.000000 36994 ./traces/malloc.rep
   yes    28%      17  0.000000 57982 ./traces/malloc-free.rep
   yes    96%      15  0.000000 34843 ./traces/corners.rep
 * yes    86%    1494  0.001591   939 ./traces/perl.rep
 * yes    75%     118  0.000014  8271 ./traces/hostname.rep
 * yes    91%   11913  0.380201    31 ./traces/xterm.rep
 * yes    99%    5694  0.038100   149 ./traces/amptjp-bal.rep
 * yes    99%    5848  0.033701   174 ./traces/cccp-bal.rep
 * yes    99%    6648  0.053715   124 ./traces/cp-decl-bal.rep
 * yes   100%    5380  0.037869   142 ./traces/expr-bal.rep
 * yes    66%   14400  0.000133108391 ./traces/coalescing-bal.rep
 * yes    97%    4800  0.065386    73 ./traces/random-bal.rep
 * yes    55%    6000  0.228742    26 ./traces/binary-bal.rep
10         87%   62295  0.839452    74

Perf index = 56 (util) + 3 (thru) = 59/100
b201502085@2018-sp:~/Malloclab/malloclab-handout$
```

best_fit 사용 시

따라서, 저는 가장 높은 점수를 내는 next_fit을 선택했습니다.

소스 코드와 작동 방식

소스코드에 대한 설명은 주석으로 넣어놓았으나, 추가 설명이 필요하다고 판단되는 부분은 사진을 넣거나 글로 부연설명 했습니다.

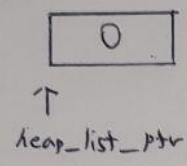
(find_fit은 #define문으로 next_fit으로 정의했습니다.)

```
// 201502085
int mm_init(void) {

    mem_init();

    if((heap_list_ptr = mem_sbrk(4*WSIZE)) == (void*)-1)
        return -1;

    PUT(heap_list_ptr, 0); // 처음 (그냥 의미 없는 값. 뭐가 들어가도 상관 없음)
    PUT(heap_list_ptr + (1*WSIZE), PACK(DSIZE,1)); // 프롤로그
    PUT(heap_list_ptr + (2*WSIZE), PACK(DSIZE,1)); // 프롤로그
    PUT(heap_list_ptr + (3*WSIZE), PACK(0,1)); // 에필로그 헤더
    heap_list_ptr += DSIZE;

    // 초기화 되면 next_fit_ptr이 프롤로그를 가리키게 함.
    next_fit_ptr = heap_list_ptr;

    // 1024만큼 extend.
    // 실제로 쓸 땐 4를 곱해서 사용.
    if(extend_heap(CHUNK_SIZE/WSIZE) == NULL)
        return -1;

    return 0;
}
```

- mm_init()

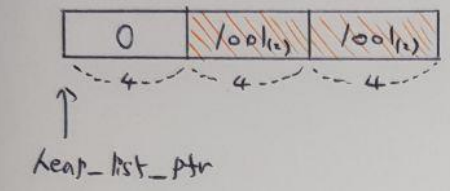
※ mm_init() 호출 시

①



(PUT(heap_list_ptr, 0) 실행 후)

②



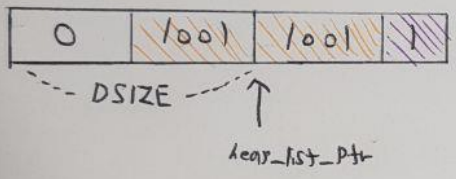
WSIZE = 4이므로

4씩 이동 후 PUT함

⇓

프롤로그 헤더, 푸터 생성

③



에필로그 블록 생성 후

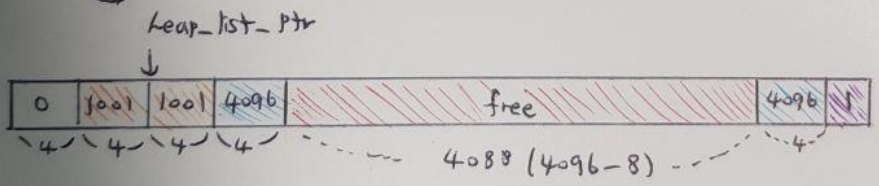
heap_list_ptr에 8을 더해

언제나 프롤로그 헤더를

가리키게 함.

④ extend_heap 호출

⑤



크기 4096을 가지는 가용블록을 프롤로그 블록과 에필로그 블록 사이에 생성. 가용이므로 헤더, 푸터 모두 사이즈만 저장

Coalesce를 호출하지만 앞 뒤가 모두 할당되어 있으므로 아무 일도 하지 않고 리턴

```
// 201502085
static void *extend_heap(size_t words){

    char *bp;
    size_t size;

    // 4096 (4를 다시 곱해줌)
    size = (words % 2) ? (words+1) * WSIZE : words * WSIZE;
    if ((long)(bp = mem_sbrk(size)) == -1) return NULL;

    PUT(HDRP(bp), PACK(size, 0));  // 헤더에도 풋터에도 size를 저장.
    PUT(FTRP(bp), PACK(size, 0));
    PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1)); // 에필로그

    return coalesce(bp);

}
```

- extend_heap()

```
// 201502085
static void* coalesce(void* bp){

  size_t size = GET_SIZE(HDRP(bp));

  // 앞이 할당되었는지, 뒤가 할당되었는지를 봐서 case를 4개로 쪼갠다.

  switch(GET_ALLOCED(FTRP(PREV_BLKP(bp))) | (GET_ALLOCED(HDRP(NEXT_BLKP(bp))) << 1)){

    case 1:
    // 앞 블록만 할당되어 있던 경우 (뒤가 가용)
    size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size, 0));
    break;

    case 2:
    // 뒷 블록만 할당되어 있던 경우 (앞은 가용)
    size += GET_SIZE(HDRP(PREV_BLKP(bp)));
        PUT(FTRP(bp), PACK(size, 0));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);
    break;

    case 3:
    // 둘 다 할당되어 있는 경우 (아무 작업도 하지 않고 bp를 반환)
    return bp;

    case 0:
    // 둘 다 가용블록인 경우
    size += GET_SIZE(HDRP(PREV_BLKP(bp))) + GET_SIZE(FTRP(NEXT_BLKP(bp)));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);
    break;

  }

  /*
     coalesce 작업 후, 가용블록이 변했다면 (case 3이 아닌 경우)
     next_fit_ptr을 bp로 변경시켜야 한다.

  */

  next_fit_ptr = bp;
  return bp;
}
```

- coalesce()

init에서 호출한 extend_heap가 호출한 coalesce의 경우 앞 뒤 블럭 (프롤로그 블럭과 에필로그 블럭) 이 모두 할당되어 있으므로 아무 일도 하지 않고 리턴합니다.

```
// 201502085
static void* first_fit(size_t asize){
  /*
   언제나 프롤로그를 가리키는 heap_list_ptr에서 시작해, 전체 블록에서 가장
   먼저 할당 가능한 블록에 할당한다.
   */
  void *bp;

  for (bp = heap_list_ptr; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKP(bp)) {
    if (!GET_ALLOCED(HDRP(bp)) && (asize <= GET_SIZE(HDRP(bp)))) {
      return bp;
    }
  }
  return NULL;
```

- first_fit()

```
// 201502085
static void* next_fit(size_t asize){

  char *bp = next_fit_ptr;

  // 우선 다음 블록부터 검색해본다.

  while(GET_SIZE(HDRP(next_fit_ptr)) > 0){
      if(!GET_ALLOCED(HDRP(next_fit_ptr)) && (asize <= GET_SIZE(HDRP(next_fit_ptr)))) return next_fit_ptr;
      next_fit_ptr = NEXT_BLKP(next_fit_ptr);
  }

  // 다음 블록부터 검색해봤는데 없다. -> 앞쪽에 가용이 있을 수 있으니 바로 NULL을 리턴하고 extend_heap 하면 안 됨
  // 처음 블록부터 다시 bp까지 검색한다. 항상 프롤로그를 가리키는 heap_list_ptr을 이용.

  for(next_fit_ptr = heap_list_ptr; next_fit_ptr < bp; next_fit_ptr = NEXT_BLKP(next_fit_ptr))
     if(!GET_ALLOCED(HDRP(next_fit_ptr)) && (asize <= GET_SIZE(HDRP(next_fit_ptr))))
              return next_fit_ptr;

  return NULL;
}
```

- next_fit()

#define find_fit next_fit으로 next_fit을 사용했습니다. next_fit의 경우 함수 정의만으로 구현할 수 없어 전역변수 void*형 포인터 next_fit_ptr를 사용했으며 이 포인터는 mm_init()에서 heap_list_ptr로 초기화 되고 coalesce()에서 bp로 변경됩니다.

```
// 201502085
static void* best_fit(size_t asize){
/*
 가장 사이즈가 잘 맞는 블록을 찾아 bp를 리턴할 것.
*/
    char* index_ptr = heap_list_ptr;
    char* minbp = NULL;
    int isFirst = 1;

    while (GET_SIZE(HDRP(index_ptr)) > 0) {
        // 가용블록의 크기가 asize보다 크거나 같고, 할당이 안 되 있고 minbp가 가리키는 공간의 크기보다 작다면 minbp를 교체
        if (!GET_ALLOCED(HDRP(index_ptr)) && (asize <= GET_SIZE(HDRP(index_ptr)))){

            if (isFirst) {
            // 처음 찾은 index_ptr을 minbp에 저장
            minbp = index_ptr;
            isFirst = 0;
        }

        else {
            // 두 번째 부턴, 찾은 블록과 지금까지의 가장 작은 블록의 크기를 비교
            if(GET_SIZE(HDRP(index_ptr)) < GET_SIZE(HDRP(minbp))) minbp = index_ptr;
        }

        }
        // 다음 블록으로
        index_ptr = NEXT_BLKP(index_ptr);
    }
    // 찾은 블록이 없다면 Null을 return
    return minbp;
```

- best_fit()

```
static void place(void* bp, size_t asize){

    size_t csize = GET_SIZE(HDRP(bp));

    /*
       free블록이 16바이트보다 큰 경우, 남은 가용블록에
       새로운 블록이 들어올 수 있기 때문에, 새로운 free블록을 만들어둔다.
    */
    if ((csize - asize) >= (2*DSIZE)) {
       PUT(HDRP(bp), PACK(asize, 1));
       PUT(FTRP(bp), PACK(asize, 1));
       bp = NEXT_BLKP(bp);
       PUT(HDRP(bp), PACK(csize-asize, 0));
       PUT(FTRP(bp), PACK(csize-asize, 0));
    }
    // 그렇지 않은 경우, 어차피 새로운 블록이 들어올 순 없기 때문에 그냥 둔다.
    else {
       PUT(HDRP(bp), PACK(csize, 1));
       PUT(FTRP(bp), PACK(csize, 1));
    }

}
```

- place()

malloc()에서 사용하는 함수입니다.

```
// 201502085
void* malloc(size_t size) {
    size_t asize;
    size_t extendsize;
    char *bp;

  if (heap_list_ptr == 0){
    mm_init();
  }
  if (size == 0)
    return NULL;

  if (size <= DSIZE)
    asize = 2*DSIZE;
  else
    // size는 내가 할당하려는 블록의 크기, asize는 프롤로그, 헤더, 정렬을 위한 블록까지 포함한 (16바이트)
    // 전체 블록의 크기
    asize = DSIZE * (((size + DSIZE + (DSIZE -1))) / DSIZE);

  if ((bp = find_fit(asize)) != NULL) {
    place(bp, asize);
    return bp;
  }

  extendsize = MAX(asize,CHUNK_SIZE);
  if ((bp = extend_heap(extendsize/WSIZE)) == NULL)
    return NULL;

  place(bp, asize);
  return bp;
```

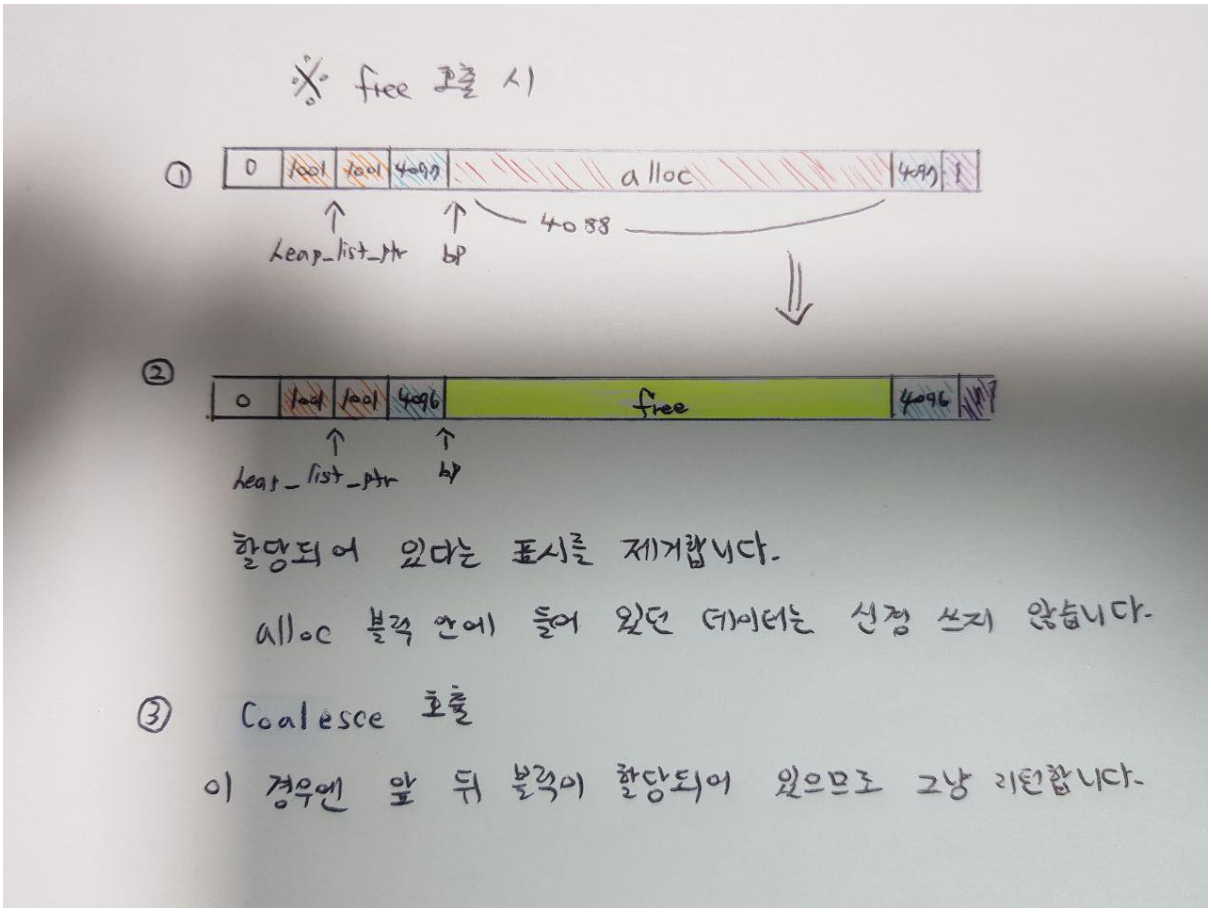
- malloc()

heap_list_ptr이 NULL인 경우 mm_init()을 호출하여, 새 블럭을 만듭니다. size가 0인 경우엔 NULL을 리턴합니다. size가 DSIZE보다 작은 경우엔 2*DSIZE (한 블럭의 최소크기) 로 사이즈를 정렬합니다. 그 후 find_fit (next_fit으로 정의) 을 사용해 가용 블럭을 찾고, place()로 데이터를 넣습니다. (free 블록이 16바이트 보다 큰 경우와 작은 경우를 나누기 위해 PUT이 아니라 place함수를 따로 정의해 사용합니다)

```
// 201502085
void mm_free(void *bp)
{

  if (bp == NULL) return;

  if (heap_list_ptr == NULL) mm_init();

  size_t size = GET_SIZE(HDRP(bp));

  PUT(HDRP(bp), PACK(size, 0));
  PUT(FTRP(bp), PACK(size, 0));

  // free한 블럭의 앞 뒤를 살펴 가용블럭을 연결
  coalesce(bp);
}
```

- free()

bp가 NULL인 경우 아무일도 하지 않습니다. heap_list_ptr이 NULL인 경우 mm_init()을 호출해, 기본 블럭을 만듭니다.



```
void *realloc(void *oldptr, size_t size) {
  size_t oldsize;
  void *newptr;

  if(size == 0) {
    mm_free(oldptr);
    return 0;
  }

  if(oldptr == NULL) {
    return mm_malloc(size);
  }

  newptr = mm_malloc(size);

  if(!newptr) {
    return 0;
  }

  oldsize = GET_SIZE(HDRP(oldptr));
  if(size < oldsize) oldsize = size;
  memcpy(newptr, oldptr, oldsize);

  mm_free(oldptr);

  return newptr;
}
```

- realloc()

realloc의 경우 size에 0이 들어온 경우, free 함수와 같은 일을 하게 되며, oldptr에 NULL이 들어온 경우엔 malloc()과 같은 일을 하게 됩니다. 그렇지 않은 경우엔 (일반적인 경우엔) memcpy()를 이용해 기존의 블럭에 있던 내용들을 size만큼 복사한 후, 새 블럭의 포인터를 리턴합니다.

```
void *calloc (size_t nmemb, size_t size) {

  // mm-naive.c와 동일
  size_t bytes = nmemb * size;
  void *newptr;

  newptr = malloc(bytes);
  memset(newptr, 0, bytes);

  return newptr;

}
```

- calloc

malloc을 호출해 메모리를 할당한 후 memset()를 사용해 값을 모두 0으로 설정합니다.