2020 시스템 프로그래밍

- Lab 04 -

제출일자	2020.12.05
분 반	01
이 름	박재우
학 번	201902694

Naïve

```
b201902694@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    94%      10  0.000000 62680 ./traces/malloc.rep
   yes    77%      17  0.000000 90858 ./traces/malloc-free.rep
   yes   100%      15  0.000000 51752 ./traces/corners.rep
 * yes    71%    1494  0.000018 84428 ./traces/perl.rep
 * yes    68%     118  0.000002 69147 ./traces/hostname.rep
 * yes    65%   11913  0.000121 98135 ./traces/xterm.rep
 * yes    23%    5694  0.000068 83589 ./traces/amptjp-bal.rep
 * yes    19%    5848  0.000065 89738 ./traces/cccp-bal.rep
 * yes    30%    6648  0.000074 89367 ./traces/cp-decl-bal.rep
 * yes    40%    5380  0.000064 84255 ./traces/expr-bal.rep
 * yes     0%   14400  0.000192 74851 ./traces/coalescing-bal.rep
 * yes    38%    4800  0.000069 69335 ./traces/random-bal.rep
 * yes    55%    6000  0.000052114789 ./traces/binary-bal.rep
10         41%   62295  0.000726 85781

Perf index = 26 (util) + 40 (thru) = 66/100
b201902694@2020sp:~/malloclab-handout$
```

구현 방법

- 사용한 매크로

  1. ALIGNMENT 8 : 저장 공간을 double word 사이즈 단위로 8 바이트로 선언
  2. ALIGN(size) (((size) + (ALIGNMENT-1)) & ~0x7) : size 의 단위를 위에 선언한 단위로 선언
  3. SIZE_T_SIZE (ALIGN(sizeof(size_t))) : size_t 형 사이즈를 ALIGN 을 이용하여 위에 선언한 단위로 선언
  4. SIZE_PTR(p) ((size_t*)(((char*)(p)) - SIZE_T_SIZE)) :

     p 에서 SIZE_T_SIZE(= 8.)를 빼서 Header 로 이동. 다음 해당 형을 size_t 로 바꿔 앞에 4byte 만 표현하여 p 의 사이즈를 받아 옴

- 사용한 함수

  1. void *malloc(size_t size) : size 바이트의 저장 공간 할당을 해주는 함수. ALIGN 으로 입력받은 size 를 SIZE_T_SIZE(= 8)을 더한 후 이 값보다 크거나 같은 8 의 배수를 구하여 newsize 에 삽입. 그후 해당 newsize 만큼의 저장 공간을 p 에 삽입. 만약 p< 0 이면 저장 공간을 받지 못했다는 뜻 이므로 NULL 을 리턴, 아니면 p += SIZE_T_SIZE, p 의 사이즈 값에 입력 받은 size 를 넣어 줌. 그리고 p 를 리턴

```
void *malloc(size_t size)
{
  int newsize = ALIGN(size + SIZE_T_SIZE);
  unsigned char *p = mem_sbrk(newsize);
  //dbg_printf("malloc %u => %p\n", size, p);

  if ((long)p < 0)
    return NULL;
  else {
    p += SIZE_T_SIZE;
    *SIZE_PTR(p) = size;
    return p;
  }
}
```

2. realloc (void *oldptr, size_t size) : 할당 받은 oldptr 를 입력받은 size 로 재 할당해주는 함수. size 가 0 이면 free 와 같으므로 free 시키고 0 리턴, oldptr 이 할당받지 않은 메모리라면 malloc 으로 입력받은 size 만큼 할당해준다. 앞의 두가지 조건을 충족하지 않을 경우 newptr 에 malloc 으로 입력받은 size 만큼 할당, 그후 할당에 오류가 없는지 확인한 후 oldsize 에 oldptr 의 사이즈를 넣어주고, 원래의 사이즈와 다시할당받기를 원하는 사이즈의 크기를 비교하여 작은 값을 oldsize 에 넣어주고 memcpy 함수를 이용하여 원래 oldptr 에 있었던 정보를 newptr 에 oldsize 만큼 복사. 입력 받은 size 가 작으면 입력 받은 size 만큼만 복사해주고 입력 받은 size 가 더 크면 oldptr 의 모든 정보를 복사해줌. 그리고 oldptr 을 free. 다시 할당 받은 newptr 을 리턴.

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

  /* If size == 0 then this is just free, and we return NULL. */
  if(size == 0) {
    free(oldptr);
    return 0;
  }

  /* If oldptr is NULL, then this is just malloc. */
  if(oldptr == NULL) {
    return malloc(size);
  }

  newptr = malloc(size);

  /* If realloc() fails the original block is left untouched  */
  if(!newptr) {
    return 0;
  }
```

3. *calloc (size_t nmemb, size_t size) : malloc 과 같이 저장 공간을 할당해 주고 0 으로 초기화 해주는 함수. malloc 과 비슷 하지만 memset 을 이용하여 0 으로 모두 초기화

```
{
  size_t bytes = nmemb * size;
  void *newptr;

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

  return newptr;
}
```

- 미 선언 함수

1. int mm_init(void) : heap 을 초기화 그후 0 을 리턴
2. free(void *ptr) : 할당 받은 ptr 메모리를 가용 메모리로 바꾸어 주는 함수
3. mm_checkheap(int verbose) : heap 을 확인하는 함수

implicit

```
b201902694@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 20395 ./traces/malloc.rep
   yes    28%      17  0.000001 32932 ./traces/malloc-free.rep
   yes    96%      15  0.000001 19909 ./traces/corners.rep
 * yes    86%    1494  0.002274   657 ./traces/perl.rep
 * yes    75%     118  0.000026  4473 ./traces/hostname.rep
 * yes    91%   11913  0.118177   101 ./traces/xterm.rep
 * yes    99%    5694  0.012385   460 ./traces/amptjp-bal.rep
 * yes    99%    5848  0.011334   516 ./traces/cccp-bal.rep
 * yes    99%    6648  0.018376   362 ./traces/cp-decl-bal.rep
 * yes   100%    5380  0.013912   387 ./traces/expr-bal.rep
 * yes    66%   14400  0.000338 42594 ./traces/coalescing-bal.rep
 * yes    93%    4800  0.009960   482 ./traces/random-bal.rep
 * yes    55%    6000  0.029264   205 ./traces/binary-bal.rep
10        86%   62295  0.216047   288

Perf index = 56 (util) + 12 (thru) = 67/100
b201902694@2020sp:~/malloclab-handout$
```

구현 방법

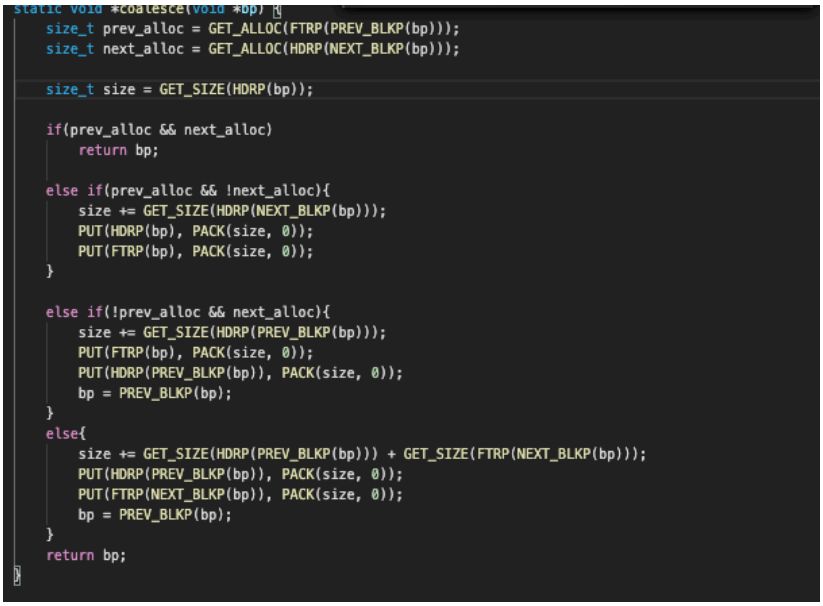
- 사용한 매크로

1. WSIZE 4 : word의 사이즈
2. DSIZE 8 : double의 사이즈
3. CHUNKSIZE (1 << 12) : 1을 12자리 Left Shift 해준 값 == 4096
4. OVERHEAD 8 : header와 footer의 합
5. MAX(x, y) ((x) > (y) ? (x) : (y)) : x와 y중 큰값
6. PACK(size, alloc) ((size) | (alloc)) : 사이즈와 할당유무를 합쳐서 표현
7. GET(p) (*(unsigned int *)(p)) : p에 있는 value를 가져옴
8. PUT(p, val) (*(unsigned int *)(p) = (val)) : p에 있는 value에 val을 넣음
9. GET_SIZE(p) (GET(p) & ~0x7) : p의 사이즈를 받아옴
10. GET_ALLOC(p) (GET(p) & 0x1) : p의 할당여부를 받아옴
11. HDRP(bp) ((char *)(bp) - WSIZE) : bp의 header의 포인터를 받아옴
12. FTRP(bp) ((char*)(bp) + GET_SIZE(HDRP(bp)) - DSIZE) : bp의 footer의 포인터를 받아옴
13. NEXT_BLKP(bp) ((char*)(bp) + GET_SIZE((char *)(bp) -WSIZE)) :
    Bp의 다음 블럭의 포인터를 받아옴
14. PREV_BLKP(bp) ((char*)(bp) - GET_SIZE((char *)(bp) -DSIZE)) :

    Bg 의 앞 블럭의 포인터를 받아옴

- 사용한 함수

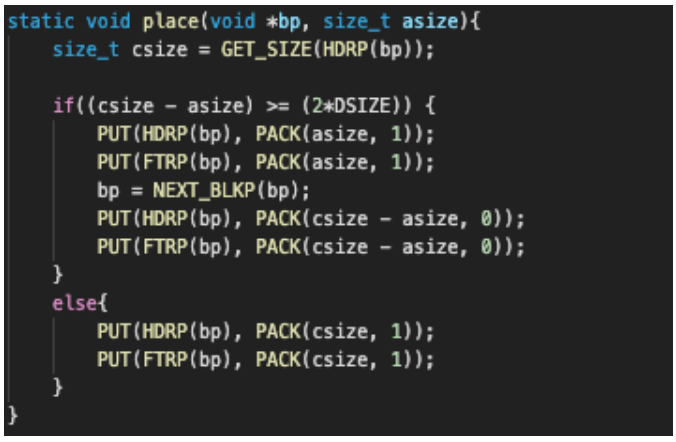
1. static void coalesce(void *bp) : free 되어진 bp 블럭 주변에 free 블럭이 있으면 정렬해주는 함수.



```
static void *coalesce(void *bp) {
    size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));

    size_t size = GET_SIZE(HDRP(bp));

    if(prev_alloc && next_alloc)
        return bp;

    else if(prev_alloc && !next_alloc){
        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size, 0));
    }

    else if(!prev_alloc && next_alloc){
        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);
    }
    else{
        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);
    }
    return bp;
}
```

입력받은 블럭의 앞블럭, 뒤블럭이 모두 할당되었다면 정렬이 불필요 함으로 bp 리턴. 뒤만 free 블럭이면 입력받은 블럭의 사이즈에 뒤에 블럭사이즈를 더해준뒤 입력받은 블럭의 헤더에 두 사이즈에 합을 넣어주고 푸터에 두 사이즈의 합을 삽입. 앞의 블럭만 free 블럭이라면 입력받은 블럭의 사이즈에 앞에 블럭의 사이즈를 더해줌. 입력받은 블럭의 풋터에 두 사이즈의 합을 삽입. 입력받은 블럭의 앞에 블럭의 헤더에 두 사이즈의 합을 삽입. Bp 를 앞에 블럭으로 옮김. 앞 뒤 블럭이 모두 free 이면 입력받은 블럭의 사이즈에 앞과 뒤의 블럭의 사이즈를 더하고 입력블럭의 앞 블럭의 헤더에 세 사이즈 합을 삽입. 입력 블럭의 뒤에 블럭의 헤더에 세 사이즈의 합을 삽입. Bp 를 앞 블럭으로 이동. Bp 를 리턴

2. static void place(void *bp, size_t asize) : 해당 위치에 사이즈 만큼 메모리를 위치 시키는 함수.



```
static void place(void *bp, size_t asize){
    size_t csize = GET_SIZE(HDRP(bp));

    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));
    }
    else{
        PUT(HDRP(bp), PACK(csize, 1));
        PUT(FTRP(bp), PACK(csize, 1));
    }
}
```

입력받은 블럭의 사이즈를 받아옴. 입력블럭사이즈 – 사이즈가 16 이상이면 블럭을 나누어준뒤 할당. 입력받은 블럭의 헤더에 사이즈만큼 할당됨을 저장. 이볅받은 블럭의 풋터에 사이즈만큼 할당됨을 저장. 블럭의 헤더와 풋터를 설정해주었으므로 나머지 블럭이 다음 블럭이 됨. 다음블럭을 bp 에 삽입. 입력받은 블럭에서 할당 받고 남은 블럭의 헤더에 남은 사이즈를 저장. 입력받은 블럭의 사이즈와 입력받은 블럭을 모두 할당. 헤더에 할당여부 저장. 풋터에 할당여부 저장.

3. static void *extend_heap(size_t words) : 요청받은 사이즈 만큼 heap 을 늘리는 함수.

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

    size = (words % 2) ? (words + 1) * WSIZE : words * WSIZE;
    if((long)(bp = mem_sbrk(size)) == -1) {
        return NULL;
    }
    PUT(HDRP(bp), PACK(size, 0));
    PUT(FTRP(bp), PACK(size, 0));
    PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1));

    return coalesce(bp);
}
```

입력된 사이즈가 홀수,짝수인지 판별후 늘려줄 사이즈 지정. Mem_sbrk 를 이용하여 추가메모리를 받아옴. 받지 못하면 null 리턴. 받아온 경우 헤더와 풋터에 사이즈 저장. 다음 블럭의 헤더 초기화. 추가된 free 블럭을 coalesce 를 이용해 정렬 후 리턴.

4. static void *find_fit(size_t assize) : 입력된 사이즈에 적당한 블럭을 찾아주는 함수.

```
static void *find_fit(size_t asize){
    void *bp;

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

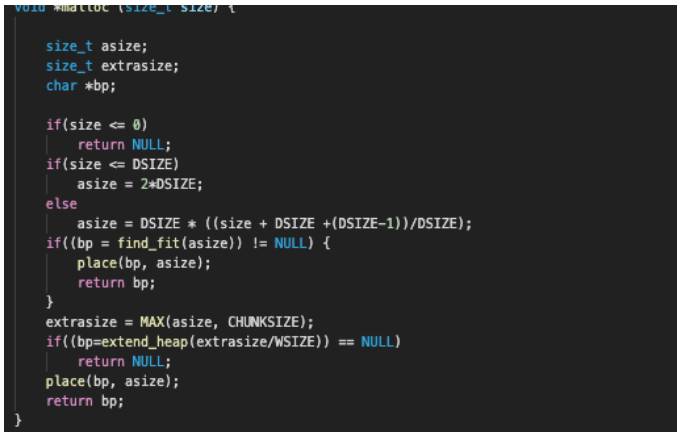
마지막으로 사용했던 블럭부터 블럭 사이즈가 0 일 될때까지의 블럭을 반복 검사. 만약 해당 블럭이 free 이고 요청 사이즈이상이면 해당블럭을 리턴. 블럭을 검사할 동안 찾지 못하면 null 리턴.

5. int mm_init(void) : 초기 empty heap 을 생성하는 함수.

```
int mm_init(void) {
    if((heap_listp = mem_sbrk(4*WSIZE))==NULL)
        return -1;
    PUT(heap_listp, 0);
    PUT(heap_listp+WSIZE,PACK(OVERHEAD,1));
    PUT(heap_listp+DSIZE,PACK(OVERHEAD,1));
    PUT(heap_listp+WSIZE+DSIZE,PACK(0,1));
    heap_listp += DSIZE;

    if((extend_heap(CHUNKSIZE/WSIZE)) == NULL)
        return -1;
    return 0;
}
```

Mem_sbrk 로 4+wsize 하여 16 만큼 메모리를 할당. 할당이 되지 않으면 리턴 -1 을 시킴. 할당되면 heap_listp 를 0 으로 맞추어줌. 헤더에 size 는 dsize, 할당은 받음을 pack 하여 넣어준다. 그후 풋터에 헤더와 같은 값을 넣어주고 다음 블럭의 헤더를 초기화 해준다. 완료되면 heap_listp 를 dsize 만큼 옮겨준다. 즉 원래 데이터가 들어가야 할 곳에 놓는다. 초기화 할때 가장 처음으로 사용한 블럭임으로 free_block 에 heap 을 넣는다.

6. void *malloc(size_t size) : 입력받은 사이즈 만큼 저장공간 할당하는 함수.



```
void *malloc (size_t size) {

    size_t asize;
    size_t extrasize;
    char *bp;

    if(size <= 0)
        return NULL;
    if(size <= DSIZE)
        asize = 2*DSIZE;
    else
        asize = DSIZE * ((size + DSIZE +(DSIZE-1))/DSIZE);
    if((bp = find_fit(asize)) != NULL) {
        place(bp, asize);
        return bp;
    }
    extrasize = MAX(asize, CHUNKSIZE);
    if((bp=extend_heap(extrasize/WSIZE)) == NULL)
        return NULL;
    place(bp, asize);
    return bp;
}
```

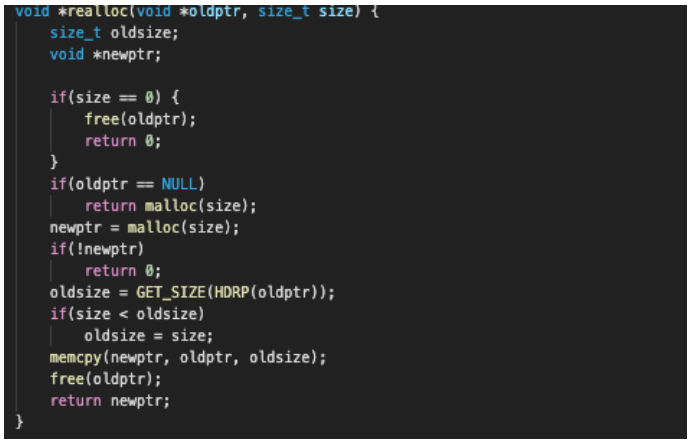
입력받은 사이즈가 0 보다 작거나 같으면 NULL 리턴. 보다 작으면 브럭에 헤더와 풋터를 포함해 16 을 할당해줌. 입력받은 사이즈가 8 보다 크면 블럭에 헤더와 풋터, 사이즈를 사이즈보다 크거나 같은 8 의 배수로 올려 할당 사이즈로 지정. Find_fit 으로 할당해줄 사이즈에 따른 적당한 블럭을 찾고 그후, place 를 이용하여 bp 에 asize 만큼 할당. 그 bp 를 리턴. 적당한 블럭을 찾지 못할 경우 해당사이즈와 chunksize 의 크기를 비교, 큰값 찾기. 그 값을 wsize 로 나눈뒤 extend_heap 을 사용하여 추가. 불가능할경우 null 리턴. 추가한 bp 에 asize 만큼 할당. Bp 리턴

7. void free(void *ptr) : bp 를 가용메모리로 변경해주는 함수.

```
void free (void *ptr) {
    size_t size;

    if(!ptr) return;
    size = GET_SIZE(HDRP(ptr));
    PUT(HDRP(ptr), PACK(size, 0));
    PUT(FTRP(ptr), PACK(size, 0));
    coalesce(ptr);
}
```

할당되어있지 않으면 종료. Heap_listp 가 없으면 mm_init 을 이용하여 초기화. Bp 블록의 사이즈를 받아옴. Bp 의 헤더에 사이즈와 가용임을 저장. Bp 의 풋터에 사이즈와 가용임을 저장. Coalesce 를 이용하여 bp 를 가용리스트 넣어 정렬해줌. 최근 사용 블럭 포인터에 정렬된 bp 를 넣어준다.

8. void *realloc(void *oldptr, size_t size) : 원래의 블럭을 새로운 사이즈로 재할당해주는 함수.



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

    if(size == 0) {
        free(oldptr);
        return 0;
    }
    if(oldptr == NULL)
        return malloc(size);
    newptr = malloc(size);
    if(!newptr)
        return 0;
    oldsize = GET_SIZE(HDRP(oldptr));
    if(size < oldsize)
        oldsize = size;
    memcpy(newptr, oldptr, oldsize);
    free(oldptr);
    return newptr;
}
```

재할당 받을 사이즈가 0 이면 free 및 0 리턴. 원래 블럭이 있지 않으면 처음 할당 받은 것과 같으므로 malloc 으로 할당 해준뒤 리턴. 재할당 방을 사이즈가 0 보다 크고 원래 블럭이 존재하면 malloc 으로 재할당 해야될 사이즈 만큼 할당한 후 newptr 에 저장. Malloc 할당이 안될경우 0 리턴. 할당 되었다면 원ㄹ ㅐ블럭의 사이즈를 받아옴. 새로운 사이즈가 원래 사이즈보다 작으면 원래 블럭의 데이터를 원래블록에 새로운 사이즈만큼만 새로운 블럭에 복사. 새로운 사이즈가 원래 사이즈보다 크거나 같으면 전부를 새로운 블럭에 복사. 원래 블럭은 free 그후 새로운 블럭 리턴.

9. void *calloc(size_t nmembm size_t size) : void *calloc (size_t nmemb, size_t size) : malloc 과 같이 저장 공간을 할당해 주고 0 으로 초기화 해주는 함수. malloc 과 비슷 하지만 memset 을 이용하여 0 으로 모두 초기화

```
void *calloc (size_t nmemb, size_t size)
{
  size_t bytes = nmemb * size;
  void *newptr;

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

  return newptr;
}
```

- 미사용 함수

10. void *calloc (size_t nmemb, size_t size) : malloc 과 같이 저장 공간을 할당해 주고 0 으로 초기화 해주는 함수. malloc 과 비슷 하지만 memset 을 이용하여 0 으로 모두 초기화
11. static int in_heap(const void *p) :
12. static int aligned(const void *p) :
13. void mm)checkheap(int verbose) :