컴퓨터 SW시스템개론 Lab7 보고서

20210207 이지현

Ⅰ. 소개

speed와 space 측면에서 모두 효율적인 malloc, free, realloc 코드를 구현하는 것이 이번 lab의 목적이다. mm.c 코드를 수정하고 평가를 계속하면서 최적의 알고리즘을 찾을 수 있었다. 나는 **implicit list와 best-fit을 이용해서 fragmentation을 줄일 수 있는 코드를 구현하였다.**

Ⅱ. Lab 11 내용 요약

-malloc은 간단하게 메모리를 할당시키는데, C에서는 malloc, 리눅스에서는 b가, sbrk로 동적할당이 가능하다

- malloc을 구현하는 방법은 여러가지가 있는데, 더 좋은 디자인을 위해 평가할 요소는 두가지가 있다. (1) execution speed : malloc과 free가 얼마나 빠르게 동작하느냐 (2) memory space consumption : data 구조의 overhead + internal fragmentation + external fragmentation을 고려한다. 이 때, malloc 및 free의 시간 복잡도를 계산해서 효율적인 구현 정도를 평가한다.

- Implicit free list는 double linked list로 관리하고 size – payload – size라는 메모리 블락이 존재할 것이다. block이 20바이트짜리 블록이면 다음 블럭을 탐색할 때에 20만큼을 넘기면 도달할 수 있고 반대편 탐색을 위해서는 20을 빼면 도달할 수 있는 것이다.

- free-free가 붙어있는 block이면 합쳐서 새로운 공간을 만들 수 있다.

- 이 때, 새로 할당할 공간을 찾는 과정에서 first-fit, best-fit 등등 여러가지 방법이 있으며, **best-fit을 이용했을 때 external fragmentation이 줄어들 것이다.**

- 평가 기준은 speed와 space 두가지 기준이 있다고 했는데, implicit free list는 시간복잡도가 O(n)이어서 별로 좋지 않다. 그러나 free하는 것은 메모리의 주소를 받아서 인식만 하면 되기에 O(1)으로 쉽다. Free를 한 다음 다른 옆의 free chunk들과 coalesce하는 과정도 O(1)이라 쉽다.

- implicit free list는 space 측면에서는 double linked list를 이용해 다음, 이전 block에 대한 pointer를 가지고 있어야 하기에 2 word 만큼의 overhead가 필수적이다.

- 또, 사용되지 않는 공간이 생겨 internal fragmentation과 external fragmentation이 발생할 것이다.

Ⅲ. 코드 설명

< 주석 내용 = 전반적인 코드 설명>

malloc이란 c언어의 동적 메모리 할당이다
동적 메모리 할당은 heap이라는 가상 메모리 영역에 이루어진다.

동적할당은 explicit or implicit free list가 가능한데,
여기서는 LAB에서 설명한 implicit free list을 이용해서 구현하였다.

implicit 리스트는 header - payload - padding - footer로 이루어져있다.
-header는 block size + 블록의 할당 여부
-payload는 할당된 블록에 값이 들어있는 부분
-padding은 double word alignment을 위한 optional한 부분
-footer는 boundary tag로, header의 값이 복사되어 있다.

(1) mm_init
: 프롤로그 블록과 에필로그 블록을 초기화하고 할당
: double word alignment를 위해 패딩 블록을 맨 앞에 붙인다.
: heap이 확장될 때 에필로그 블록은 확장된 힙의 마지막에 위치하도록 한다.

(2) extend_heap
: 힙이 초기화될때, 혹은 충분한 공간을 찾지 못했을 때 호출된다

(3) mm_free
: 블록을 할당 해제하고 반환한다

(4) coalesce
: header, footer라는 boundary를 이용해 인접한 free block들을 병합한다.

(5) mm_malloc
: 요청한 size만큼의 공간이 있는 block을 할당한다.
: 보통 header와 footer의 오버헤드 공간을 위해 16바이트의 크기를 유지해야한다
: double word alignement를 위해서는 8의 배수로 반올림한 메모리 크기를 할당해야한다.
: 메모리 할당 크기를 조절했으면, search를 통해 적합한 블록을 찾아야 하는데 이는 find_fit함수이다
: 찾은 블록을 배치하고 남는 공간을 분할하는 것이 place 함수이다

(6) find_fit
: best-fit 방식을 이용해서 가장 적합한 메모리 블럭을 찾는다.

(7) place

: 찾은 block을 배치한 후 여유공간이 있다면 분할한다.
: 즉, 현재 블록의 size를 할당한 후에 남은 size가 최소 블록(header와 footer를 포함한 4워드)보다 크거나 같으면 분할한다
: 아니라면 size 정보만 변경하면 된다.

(8) mm_realloc
: 기존에 할당되어 있던 블록을 새로운 사이즈로 재할당한다
: 새로 할당하려는 size가 기존 size보다 작을 때에는 기존 정보의 일부만 복사하도록 설정한다.

(1) #define

- WSIZE (single word size), DSIZE(double word size), CHUNKSIZE(heap을 한번 늘릴 때 필요한 사이즈), MAX함수, pack(block header)를 define으로 정의하였다
- block의 값을 return하는 get, block에 값을 넣는 put, block size를 얻는 get_size, 마지막 하위 1비트를 통해 할당여부를 얻는 get_alloc을 define으로 정의하였다
- 블록의 header 위치, footer위치, 현재 블록의 다음 위치, 이전 위치를 define으로 정의하였다.
- 이후 heap list 포인터 및 function prototype을 정의하였다.

```
#define WSIZE 4 //single-word size 4
#define DSIZE 8 //double-word size 8
#define CHUNKSIZE (1<<12) //heap을 한번 늘릴 때 필요한 사이즈

#define MAX(x,y) ((x)>(y)? (x):(y)) // max 구현 -> 큰 값을 return
#define PACK(size, alloc) ((size)|(alloc)) // block header = size + 할당여부

#define GET(p) (*(unsigned int*)(p)) //block의 값 return
#define PUT(p, val) (*(unsigned int*)(p)=(val)) //block에 값 넣기

#define GET_SIZE(p) (GET(p) & ~0x7) //block size (하위 3비트 제외 ~00000111 -> 11111000)
#define GET_ALLOC(p) (GET(p) & 0x1) //할당 여부 (마지막 하위 1비트)

#define HDRP(bp) ((char*)(bp)-WSIZE) //블록 header 위치
#define FTRP(bp) ((char*)(bp) + GET_SIZE(HDRP(bp)) -DSIZE) //블록 footer 위치
#define NEXT_BLKP(bp) ((char*)(bp) + GET_SIZE(((char*)(bp) - WSIZE))) //현재 블록의 다음 위치로 이동
#define PREV_BLKP(bp) ((char*)(bp) - GET_SIZE(((char*)(bp) - DSIZE))) //현재 블록의 이전 위치로 이동

static char* heap_listp; //heap list

//function prototype
static void* coalesce(void* bp);
static void* extend_heap(size_t words);
static void* find_fit(size_t asize);
static void place(void* bp, size_t asize);
```

(2) mm_init

- mm malloc 또는 mm realloc 또는 mm free를 call하기 전에 호출하여 초기화를 시켜주는 함수이다.
- mem_sbrk 함수를 통해 할당된 heap 영역의 바이트르 가져오고 에러가 나면 -1을 리

턴시켜 준다.

- 첫번째 word에는 double word boundary로 align된 padding을 put하고, 그다음에 프롤로그 블록이 오는데, header와 footer로만 구성된 8바이트 블록이다. 이 heap은 에필로그 블록에서 끝나는데, header로만 구성된 블록이다. 이후 heap을 확장하고 malloc이 적당한 fit을 찾지 못했을 때에도 -1이 리턴된다.

```
/*
 * mm_init - initialize the malloc package.
 */
// mm_malloc 또는 mm_realloc 또는 mm_free를 call하기 전에 호출하여 초기 힙 영역 할당, 초기화
// => 비어있는 heap을 만드는 것이 목적
//초기화 오류 시 -1, 아니라면 0
int mm_init(void)
{
    //mem_sbrk 함수를 통해 할당된 힙 영역의 바이트를 가져옴 -> 에러라면 -1이 리턴될 것
    //mem_sbrk가 -1이라면 초기화 오류 -> -1
    if ((heap_listp = mem_sbrk(4 * WSIZE)) == (void*)-1)
        return -1;

    PUT(heap_listp, 0); //alignment padding
    PUT(heap_listp + (1 * WSIZE), PACK(DSIZE, 1)); //header
    PUT(heap_listp + (2 * WSIZE), PACK(DSIZE, 1)); //footer
    PUT(heap_listp + (3 * WSIZE), PACK(0, 1)); //header
    heap_listp += (2 * WSIZE);

    //mm_malloc이 적당한 fit을 찾지 못했을 때도 호출된다
    if (extend_heap(CHUNKSIZE / WSIZE) == NULL)
        return -1;
    return 0;
}
```

(3) extend_heap

- 힙의 크기를 확장시켜주는 함수이다.

- 요청한 size (words)를 인접한 8바이트의 배수로 반올림 시키고, mem_sbrk 를 통해 추가 힙 공간을 요청한다. 이후 free block의 header/footer와 epilogue header를 초기화 시켜준다.

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

    size_t size;
    char* bp;

    //요청한 크기를 8바이트의 배수로 반올림
    if (words % 2) {
        size = (words + 1) * WSIZE;
    }
    else {
        size = (words)*WSIZE;
    }
    //추가 힙 공간 요청
    if ((long)(bp = mem_sbrk(size)) == -1)
        return NULL;

    PUT(HDRP(bp), PACK(size, 0)); //free의 header
    PUT(FTRP(bp), PACK(size, 0)); //free의 footer
    PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1)); //new header

    return coalesce(bp);

}
```

(4) mm_malloc

- 할당된 block payload에 대한 pointer를 반환한다. 할당된 전체 블록은 heap안에서 겹

치지 않게 malloc되어야 한다.

- alignment를 위한 8바이트, header와 footer각각 4바이트의 오버헤드를 확보하고, 그 뒤 사용 가능한 block을 find_fit을 통해 찾은 뒤, 있으면 place를 통해 배치한다.
- 만약 fit한 block이 없다면 extend_heap을 통해 힙을 확장시키고 배치해야한다.

```
/*
 * mm_malloc - Allocate a block by incrementing the brk pointer.
 *     Always allocate a block whose size is a multiple of the alignment.
 */
//할당된 block payload에 대한 pointer를 반환한다.(최소한 size bytes)
//할당된 전체 블록은 heap 안에 있어야 하며, 겹치지 않아야 한다.
void* mm_malloc(size_t size)
{
    char* bp;
    size_t asize;
    size_t extendsize;

    if (size == 0) return NULL;

    //alignment 8바이트, header 4바이트, footer 4바이트를 확보
    if (size <= DSIZE) {
        asize = 2 * DSIZE;
    }
    else {
        asize = DSIZE * ((size + DSIZE + (DSIZE - 1)) / DSIZE);
    }

    //사용 가능한 block을 find -> 있으면 배치
    if ((bp = find_fit(asize)) != NULL) {
        place(bp, asize);
        return bp;
    }

    //fit한 block이 없으면 확장 및 배치
    extendsize = MAX(asize, CHUNKSIZE);
    if ((bp = extend_heap(extendsize / WSIZE)) == NULL)
        return NULL;
    place(bp, asize);
    return bp;
```

(5) find_fit

- 사용 가능한 block의 주소를 best-fit으로 찾아서 return 해보았을 때, 요구 조건에 맞으면 best의 size가 bp의 size보다 크다면 bp = best하는 식으로 가장 적합한 best fit의 block을 찾는 방식을 이용하였다.

```
//사용 가능한 block의 주소를 best-fit으로 찾고 return하는 함수
static void* find_fit(size_t asize) {

    char* best = NULL;
    char* bp;

    for (bp = heap_listp; GET_SIZE(HDRP(bp))> 0; bp = NEXT_BLKP(bp)) {
        //요구 조건에 맞으면 돌리기
        if(!GET_ALLOC(HDRP(bp)) && (asize <= GET_SIZE(HDRP(bp)))) {
            //best에 아무것도 안들어갔으면 바로 bp 넣기
            if (best == NULL)
                best = bp;
            //bp의 size보다 best의 사이즈가 더 크면 bp가 best가 되도록 한다.
            if (GET_SIZE(HDRP(bp)) < GET_SIZE(HDRP(best)))
                best = bp;
        }

    }
    return best;
}
```

(6) place

- 사용 가능한 block을 찾으면 배치하고 남는 부분을 split하는 함수이다.
- 요청한 블록을 가용 블록의 시작 부분에 배치하되, 남는 부분의 크기가 최소 블록의 크기와 같거나 큰 경우에만 분할한다.
- 작으면 데이터를 담을 수 없어서 분할하지 않고 그냥 배치하는 함수이다.

```
//사용 가능한 블럭을 찾으면, block을 배치하고 남는 부분을 나눈다.
static void place(void* bp, size_t asize) {


    size_t csize = GET_SIZE(HDRP(bp));

    //double word * 2보다 (사용 가능한 block 사이즈 - 요청 block size)가 크면 분할
    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 {
        //작으면 데이터를 담을 수 없기에 분할하지 않고 그냥 배치
        PUT(HDRP(bp), PACK(csize, 1));
        PUT(FTRP(bp), PACK(csize, 1));
    }

};
```

(7) mm_free

- 이전에 할당한 블록을 free함수를 호출해서 해제한다.
- 요청한 블록을 반환한 뒤에, coalesce를 통해서 free한 부분이 인접해있으면 합친다.

```
//ptr이 가르키고 있는 할당 블록을 해제한다. 반환값은 없다.
//이전에 mm_malloc 혹은 mm_realloc으로 할당한 값이 해제되지 않았을 때 사용
/*
 * mm_free - Freeing a block does nothing.
 */
void mm_free(void* ptr)
{
    size_t size = GET_SIZE(HDRP(ptr));

    PUT(HDRP(ptr), PACK(size, 0));
    PUT(FTRP(ptr), PACK(size, 0));
    coalesce(ptr);
}

//병합이 가능하면 병합하기
static void* coalesce(void* bp) {

    size_t size = GET_SIZE(HDRP(bp));
    size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));

    //앞도 할당 뒤도 할당이면 합칠 필요 없다.
    if (prev_alloc && next_alloc) {
```

(8) coalesce

- prev_alloc과 next_alloc을 통해서 이전 블록과 다음 블록의 할당 여부를 파악한 뒤에,

4가지 경우의 수로 나눠서 병합을 진행한다.

```
//병합이 가능하면 병합하기
static void* coalesce(void* bp) {

    size_t size = GET_SIZE(HDRP(bp));
    size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));

    //앞도 할당 뒤도 할당이면 합칠 필요 없다.
    if (prev_alloc && next_alloc) {
        return bp;
    }
    //다음 블록이 free라면 합친다.
    if (prev_alloc && !next_alloc) {
        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size, 0));
        return bp;
    }
    //이전 블록이 free라면 합친다.
    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);
        return bp;
    }
    //둘다 free라면 둘다 합친다.
    if (!prev_alloc && !next_alloc){
        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;
    }

}
```

(9) mm_realloc

- realloc 함수는 여러 제약들을 가지는 least byte로 할당된 공간의 포인터를 반환한다.
- 만약 ptr이 NULL이면 기본 malloc과 같고, size가 0이면 free와 같은 동작을 한다.
- 만약 ptr이 null이 아니면 새로운 size와 비교해서 변환시켜야 하며, copy 후 기존 포인터를 free시켜야한다.

```
}
// 최소 size의 할당된 region으로의 pointer를 반환한다.
//만약 ptr이 NULL이면 mm_malloc(size)과 같은 동작
//만약 size가 0이면 mm_free(ptr)과 같은 동작
//만약 ptr이 not NULL이면, ptr은 앞서 불렀던 malloc과 realloc에 의해 리턴된 것이어야 한다.
//이후 ptr이 가르키는 메모리 블럭의 크기를 변경한다. -> 새로운 size와 새로운 block의 주소를 반환한다.
//주소의 새 주소는 기존 주소와 같을 수도 있고 다를 수도 있다.
//새 블록의 내용은 이전 ptr block의 내용과 동일하고, 초기화되지 않는다.
//
/*
 * mm_realloc - implemented simply in terms of mm_malloc and mm_free
 */
void* mm_realloc(void* ptr, size_t size)
{
    void* oldptr = ptr;
    void* newptr = mm_malloc(size);
    size_t copySize;

    //이미 NULL 포인터였으면 그냥 malloc
    if (ptr == NULL) {
        return newptr;
    }
    //size가 0이면 free하고 NULL 넘겨주기
    else if (size == 0) {
        mm_free(oldptr);
        return NULL;
    }
    else {
        //oldsize가 더 크면 줄이기
        copySize = GET_SIZE(HDRP(oldptr));
        if (size <= copySize) {
            copySize = size;
        }
        //copy
        memcpy(newptr, oldptr, copySize);
        //old free
        mm_free(oldptr);
        return newptr;

    }
```

VI. 실행 결과 (test)

```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.309695    18
 1       yes   99%    5848  0.279690    21
 2       yes   99%    6648  0.445352    15
 3       yes  100%    5380  0.361342    15
 4       yes   66%   14400  0.000090160356
 5       yes   96%    4800  0.482506    10
 6       yes   95%    4800  0.504048    10
 7       yes   55%   12000  3.177890     4
 8       yes   51%   24000 10.281531     2
 9       yes   31%   14401  2.900399     5
10       yes   30%   14401  0.053832   268
Total          75%  112372 18.796374     6

Perf index = 45 (util) + 0 (thru) = 45/100
[io0818@programming2 ~]$
```

V. 결론 및 **구현 방법에 대한 고찰**

- implicit은 모든 블록을 연결하고, explicit은 free block만을 연결시킨다는 차이점이 있다.
- segregated list는 size별로 free list를 만들어서 각 메모리 사이즈 별로 관리를 하는 list이고, 가장 빠른 탐색 속도를 가진다.
- **그렇기에 explicit list를 이용하면 할당하는 memory가 적어서 빨리 수행이 되겠고, segregated list를 이용하면 요청에 맞는 size를 훨씬 빠르게 search할 수 있을 것이라는 장점이 있을 것이다. 다음번에 기회가 된다면 이 둘을 구현하고 싶다.**
- **best fit을 통해 search하고 malloc을 시키면 first fit보다 탐색은 느릴지라도 external fragmentation을 줄일 수 있다는 장점이 있다.**
- **implicit free list 방법을 이용했기에 mm_malloc의 시간복잡도는 O(n)이라는 linear search 를 갖지만, mm_free는 주소를 받기만 하면 되니까 O(1)일 것이다.**
- **Implicit free list 방법을 이용했기에 double linked list + 앞뒤 pointer -> 2 word만큼의 overhead를 가질 것이다.**
- **best-fit 방법을 이용했기에 external fragmentation도 나쁘지 않은 space evalution을 갖는다고 생각한다.**