System Programming Project 4

담당 교수 : 박성용 교수님

이름 : 윤성민

학번 : 20191264

1. 개발 목표

-이번 프로젝트는 우리가 C언어 프로그래밍을 할 때 자주 사용하는 dynamic memory allocator를 나만의 버전으로 구현하는 것이다. 강의 시간에 배운 implicit list 코드에 explicit list의 method를 적용해 구현하였다.

2. 개발 범위 및 내용

1) Method – Explicit List

-강의 시간에 배운 implicit list방식의 allocator는 free block을 찾기 위해 allocated block도 거쳐야 하기 때문에 explicit list 방식을 사용하여 free block만을 탐색하고 관리할 수 있도록 구현하였다. Explicit list를 나타내는 포인터인 free_listp를 전역 변수로 선언하여 이 변수를 통해 free block을 찾고 관리하며 탐색의 방법은 first fit으로 설정하였다. 강의 시간에 배운 implicit list의 코드를 기반으로 코드를 작성하였다.

2) 전역 변수 선언 및 메크로 정의

-위에서 언급했듯이 free_listp를 전역 변수로 선언하여 가장 첫 번째의 free block을 가리키도록 설정하였다.

```
static char *free_listp;
```

또한 해당 block에서 다음 free block이나 이전 free block을 찾기 위해선 해당 정보를 갖는 위치로 포인터의 위치를 잡아야 하기 때문에 다음과 이전 free block에 대한 정보를 담고 있는 위치로 가기 위한 메크로를 추가였다.

```
//free block list for explicit list method
#define NEXT_FREEP(bp) (*(char **)((char*)(bp) + WSIZE))
#define PREV_FREEP(bp) (*(char **)(bp))
```

메크로 선언에 대한 설명은 아래 mm_init()함수 설명에서 진행하겠다.

3) 함수

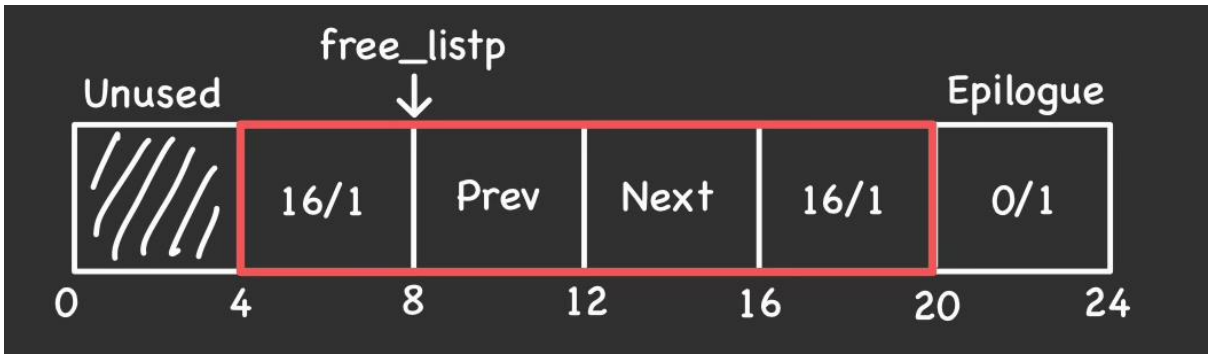
① mm_init()

-먼저 mm_init()함수는 메모리를 할당할 heap을 초기화해주는 함수이다.

```c
/*
 * mm_init - initialize the malloc package.
 */
int mm_init(void)
{
    //create the initial empty heap
    if ((free_listp = mem_sbrk(6*WSIZE)) == (void *)-1)
        return -1;
    PUT(free_listp, PACK(0,0)); //unused padding
    PUT(free_listp + WSIZE, PACK(4*WSIZE, 1));
    PUT(free_listp + 2*WSIZE, 0);
    PUT(free_listp + 3*WSIZE, 0);
    PUT(free_listp + 4*WSIZE, PACK(4*WSIZE, 1));
    //epilouge header
    PUT(free_listp + 5*WSIZE, PACK(0,1));
    free_listp += 2*WSIZE;

    //CHUNKSIZE만큼 free block을 갖게 heap extend
    if (extend_heap(CHUNKSIZE/WSIZE) == NULL)
        return -1;
    return 0;
}
```

위와 같이 먼저 mem_sbrk 함수를 통해 heap 공간에 6 word만큼을 할당하고 free_listp를 해당 할당된 영역의 시작주소로 설정한다. 그 이후 순서대로 unused padding, header, previous pointer, next pointer, footer, 그리고 epilogue를 공간에 할당한다. 이때 previous pointer와 next pointer는 이전 free block과 다음 free block에 대한 정보를 담고 있으므로 하나의 word만큼 할당한다. 이렇게 초기화한 초기 heap의 모습은 아래와 같다.



위와 같이 초기화하게 되면 이전 free block에 대한 정보는 Prev 위치에, 다음 free block에 대한 정보는 Next 위치에 있다. 따라서 위에서 언급한 메크로를 정의할 때, PREV_FREEP는 해당 bp의 주소값을, NEXT_FREEP는 해당 bp에 WSIZE(4)만큼을 더한 위치로 가게끔 정의하였다. 초기화 이후 extend_heap을 이용해 CHUNKSIZE만큼 heap을 늘려준다.

② extend_heap()

-extend_heap함수는 위의 초기화 상황에서도 사용되지만 mm_malloc시 free block 탐색 중 할당할 공간이 충분하지 않다면 할당이 가능하게끔 heap을 늘려줄 때도 사용된다.

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

    //allocate an even number of word for maintaing alignment
    size = (words % 2) ? (words+1) * WSIZE : words*WSIZE;
    if ((long)(bp = mem_sbrk(size)) == -1)
        return NULL;
    //initialize free block header&footer and epilogue header
    PUT(HDRP(bp), PACK(size,0));
    PUT(FTRP(bp), PACK(size,0));
    PUT(HDRP(NEXT_BLKP(bp)), PACK(0,1));
    //coalesce if the prev block is free
    return coalesce(bp);
}
```

③ mm_malloc()

-mm_malloc()함수는 block을 allocate하는 함수로 free block list에서 할당 가능한 free block을 찾아 free block list에서 제거해주고 block을 allocate하는 함수이다.

```
void *mm_malloc(size_t size)
{
    size_t asize; //adjusted block size
    size_t extendsize; //amount to extend heap if no fit
    char* bp;

    //ignore spurious request
    if (size == 0) return NULL;
    //adjust block size to include overhead and alignment regs
    if (size <= DSIZE) asize = 2*DSIZE;
    else asize = DSIZE * ((size + (DSIZE) + (DSIZE-1)) / DSIZE);
    //search the free list for a fit
    if ((bp = find_fit(asize)) != NULL) {
        place(bp, asize);
        return bp;
    }
    //No fit -> extend heap
    extendsize = MAX(asize, CHUNKSIZE);
    if ((bp = extend_heap(extendsize/WSIZE)) == NULL)
        return NULL;
    place(bp, asize);
    return bp;
}
```

할당할 size가 0이라면 return NULL을 해주고 size가 DSIZE, 즉 8 word보다 작다면 minimum block size인 2*DSIZE(16)만큼 할당해주며 8 word보다 크다면 alignment 8을 지키기 위해 8의 배수만큼 적절하게 할당해준다. 할당 시에는 find_fit()함수를 통해 할당 가능한 적당한 block을 탐색하고 찾았다면 place함수를 통해 해당 block을 할당한다. 만약

할당 가능한 block이 없다면 extend_heap함수를 통해 heap의 크기를 늘려준 후 할당해 준다.

④ find_fit()

-find_fit()함수는 적절한 free block을 탐색하는 함수로 first-fit 방식으로 구현하였다.

```
static void *find_fit(size_t size) {
    void *bp;
    for (bp = free_listp; GET_ALLOC(HDRP(bp)) != 1; bp = NEXT_FREEP(bp)) {
        if (size <= GET_SIZE(HDRP(bp)))
            return bp;
    }
    return NULL;
}
```

초기 bp를 첫 free block을 가리키는 free_listp로 설정한 후 for문을 통해 free block list를 순회하면서 할당에 적합한 block이 나오면 바로 return한다.

⑤ coalesce()

-coalesce() 함수는 block을 free할 때 해당 block 앞 혹은 뒤 혹은 앞뒤 모두 free된 block이 있다면 하나의 block으로 합쳐주는 함수이다. 따라서 총 4가지 경우가 존재하므로 if문을 통해 각 4가지 경우를 immediate coalescing으로 구현하였다.

```
static void *coalesce(void *bp) {
    if (bp == NULL) return NULL;
    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));

    //Case 1 - prev & alloc allocated
    if (prev_alloc && next_alloc) {
        insert_block(bp);
        return bp;
    }
    //Case 2 - prev allocated & next freed
    else if (prev_alloc && !next_alloc) {
        remove_block(NEXT_BLKP(bp));
        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(bp), PACK(size,0));
        PUT(FTRP(bp), PACK(size,0));
    }
```

위와 같이 앞뒤 모두 allocated block이라면 free한 block만 free list에 추가해주고 만약 next block이 freed block이라면 next block을 list에서 제거하고 현재 block과 합친다.

```
    //Case 3 - prev freed & next allocated
    else if (!prev_alloc && next_alloc) {
        remove_block(PREV_BLKP(bp));
        size += GET_SIZE(HDRP(PREV_BLKP(bp)));
        bp = PREV_BLKP(bp);
        PUT(FTRP(NEXT_BLKP(bp)), PACK(size,0));
        PUT(HDRP(bp), PACK(size,0));

    }
    //Case 4 - prev & next freed
    else {
        remove_block(PREV_BLKP(bp));
        remove_block(NEXT_BLKP(bp));
        size += GET_SIZE(HDRP(PREV_BLKP(bp))) + GET_SIZE(FTRP(NEXT_BLKP(bp)));
        bp = PREV_BLKP(bp);
        PUT(HDRP(bp), PACK(size,0));
        PUT(FTRP(bp), PACK(size,0));
    }
    insert_block(bp);
    return bp;
}
```

prev block만 freed block이라면 prev block을 free list에서 제거한 뒤 현재 block과 합치고 prev block과 next block 모두 freed block이라면 두 block 모두 free list에서 제거한 뒤 현재 block과 합쳐준다. 그 후 각 경우마다 합쳐진 하나의 block을 다시 free list에 추가한다.

⑥ place()

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

    //remainder space is left enough -> split
    if ((csize - asize) >= (2*DSIZE)) {
        remove_block(bp);
        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));
        insert_block(bp);
    }
    else {
        remove_block(bp);
        PUT(HDRP(bp), PACK(csize,1));
        PUT(FTRP(bp), PACK(csize,1));
    }

}
```

-mm_malloc을 통해 메모리를 할당할 때 find_fit함수를 통해 적절한 freed block을 찾고

찾은 공간에 block을 할당하게 되는데 이때 block을 알맞게 할당해주는 함수가 place함수이다. 할당하려는 block의 size가 할당할 공간의 size보다 작을 때 그 차이가 block의 최소 크기인 2*DSIZE(16)보다 크게 되면 해당 공간은 external fragmentation을 방지하기 위해 split이 수행되어야 한다. 따라서 이 경우 할당하려는 size만큼만 할당하고 남은 공간은 freed block으로 처리해 free list에 추가해준다. 반대로 차이가 16 word보다 작게 되는 경우에는 그대로 할당한다.

⑦ mm_free()

-block을 free해주는 함수로 free함수를 통해 free된 block을 free list에 추가된다. 이때 그냥 추가되는 것이 아니라 coalesce()함수를 통해 free할 block의 prev와 next block의 free 여부를 확인해 coalescing이 필요하면 수행해주는 작업이 필요하다.

```
void mm_free(void *bp)
{
    if (bp == 0) return;
    size_t size = GET_SIZE(HDRP(bp));

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

⑧ mm_realloc()

-mm_realloc()함수는 할당된 block을 parameter로 받은 size만큼 재할당해주는 함수이다.

```
void *mm_realloc(void *ptr, size_t size)
{
    if (!ptr) return mm_malloc(size);
    if (size ==0) {
        mm_free(ptr);
        return 0;
    }
    void *newptr;
    size_t copySize;

    newptr = mm_malloc(size);
    if (newptr == NULL)
      return NULL;
    copySize = GET_SIZE(HDRP(ptr)) - DSIZE;
    if (size < copySize)
      copySize = size;
    memcpy(newptr, ptr, copySize);
    mm_free(ptr);
    return newptr;
}
```

parameter로 받은 ptr이 NULL이라면 새롭게 size만큼 mm_malloc을 통해 할당만 해준고 size가 0이라면 메모리 반환, 즉 mm_free만 수행해준다. 두 경우가 모두 아니라면 mm_malloc을 통해 parameter로 받은 size만큼 할당할 수 있는 block을 찾아 할당한 뒤 기존 ptr이 가리키던 block의 크기가 새로 할당한 size보다 크다면 size를 기존 size로 조정한다. 조정한 size만큼 새로 할당한 block으로 데이터를 memcpy를 통해 복사하고 기존 ptr은 mm_free를 통해 메모리를 반환한다.

⑨ insert_block() & remove_block()

```
/*
*insert_block - insert a new free block into the free list using LIFO
*/
static void insert_block(void *bp) {
    if (bp == NULL) return;
    NEXT_FREEP(bp) = free_listp;
    PREV_FREEP(bp) = NULL;
    if (free_listp != NULL)
        PREV_FREEP(free_listp) = bp;
    free_listp = bp;
}

/*
*remove_block - remove a block from the free list
*/
static void remove_block(void *bp) {
    if (bp == NULL) return;
    if (PREV_FREEP(bp))
        NEXT_FREEP(PREV_FREEP(bp)) = NEXT_FREEP(bp);
    else
        free_listp = NEXT_FREEP(bp);
    if (NEXT_FREEP(bp))
        PREV_FREEP(NEXT_FREEP(bp)) = PREV_FREEP(bp);
}
```

-free block이 발생하게 되면 free list에 insert_block을 통해 추가한다. LIFO 방식을 사용하여 새로운 free block이 생기면 항상 free list의 맨 앞에 추가한다. 반대로 freed block에 block을 할당하게 되면 remove_block을 통해 free list에서 제거해준다. 제거 시에는 list의 연결이 유지되도록 제거되는 block의 앞뒤 block을 잘 연결해준다.

3. 구현 결과 및 성능 평가

-explicit list로 구현한 dynamic memory allocator의 performance를 측정한 결과는 다음과 같다.

```
cse20191264@cspro:~/sp/prj4/20191264$ ./mdriver -V
[20191264]::NAME: Sungmin Yoon, Email Address: zserty12@sogang.ac.kr
Using default tracefiles in ./tracefiles/
Measuring performance with gettimeofday().

Testing mm malloc
Reading tracefile: amptjp-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: cccp-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: cp-decl-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: expr-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: coalescing-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: random-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: random2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: binary-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: binary2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: realloc-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: realloc2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   89%    5694  0.001945  2927
 1       yes   92%    5848  0.001222  4784
 2       yes   94%    6648  0.000484 13750
 3       yes   96%    5380  0.001065  5052
 4       yes   66%   14400  0.000471 30547
 5       yes   88%    4800  0.003816  1258
 6       yes   85%    4800  0.000869  5522
 7       yes   55%   12000  0.010384  1156
 8       yes   51%   24000  0.005540  4332
 9       yes   26%   14401  0.226025    64
10       yes   34%   14401  0.017394   828
Total          71%  112372  0.269215   417

Perf index = 42 (util) + 28 (thru) = 70/100
```