Malloc Lab: Writing a Dynamic Storage Allocator

20190084 권민재, CSED211

개요

이 과제는 주어진 코드를 바탕으로 malloc, free, realloc을 naive하게 구현해보는 과제이다. 이 프로그램에서는 implicit free list를 이용하여 heap을 구현하였으며, 메모리 할당 정책으로는 next-fit과 best-fit을 조합한 정책을 선택하였다. 그리고, realloc 과정에서 다음 및 이전 블록으로의 확장을 malloc을 호출하지 않고 수행하도록 구현하여 단편화를 개선하였다.

mm_check를 이용한 디버깅 모드 활성화가 가능하며, 이는`#define DEBUG` 를 설정하여 이용할 수 있다.

구현

mm_init

이 함수는 본격적으로 동적 할당을 다루기 전에 메모리 등을 우리가 원하는 구조로 초기화 하는 함수이다.

```
int mm_init(void) {
#ifdef DEBUG
    // 예측되는 블럭의 정보를 초기화한다.
    num_expected_free_blocks = 0; num_expected_blocks = 0;
#endif

    // 우선 힙을 정의하는데 필요한 최소 크기를 mem_sbrk를 통해 얻어온다.
    if ((heap_listp = mem_sbrk(4 * WSIZE)) == (void *) -1) {
        return -1;
    }

    PUT(heap_listp, 0);
    PUT(heap_listp + (1 * WSIZE), PACK(DSIZE, 1)); // 프롤로그 헤더
    PUT(heap_listp + (2 * WSIZE), PACK(DSIZE, 1)); // 프롤로그 푸터
    PUT(heap_listp + (3 * WSIZE), PACK(0, 1)); // 에필로그 헤더
    heap_listp += (2 * WSIZE); // heap_listp가 블록의 주소를 가리키도록 설정한다.

    current_block = heap_listp; // init된 상태이므로, 현재 블럭을 가장 첫 블럭을 가리키도록 한다.

    return 0;
}
```

우선 블럭을 정의하는데 필요한 최소 크기를 mem_sbrk를 통해 확보한 이후에, PUT을 이용하여 프롤로그와 에필로그를 작성한다. 이후, heap_listp가 첫 블럭 주소를 가리키도록 설정하고, 현재 블럭의 주소를 첫 블럭 주소로 설정한 이후 종료시키게 해서 초기화를 진행한다.

mm_malloc

이 함수는 동적 할당 과정 중에서 malloc을 naive 하게 구현한 것이다.

```
// heap_listp가 설정되지 않은 상태라면, 초기화 되지 않았다는 뜻이므로 초기화를 진행한다.
if (heap_listp == 0) {
    mm_init();
}

// size가 0이라면, malloc을 진행하지 않아도 되므로, 미리 종료한다.
if (size == 0) {
    return NULL;
}
```

heap_listp가 설정되지 않았을 경우에는 mm_init을 통해 초기화를 진행하고, size가 0일 때에는 함수를 바로 종료하는 전처리 과정을 거친다.

```
if (size <= DSIZE) {
    // 할당해야 하는 크기가 DSIZE보다 작다면, 2 * DSIZE로 최소 크기를 맞춰준다.
    asize = 2 * DSIZE;
} else {
    // 아니라면, 아래와 같은 수식을 통해 alignment를 맞춘다.
    asize = DSIZE * ((size + (DSIZE) + (DSIZE - 1)) / DSIZE);
}
```

전처리 과정을 거친 이후에, 주어진 사이즈를 align 하는 과정을 가진다. 만약, 할당해야 하는 크기가 DSIZE 보다 작다면 블럭의 최소 크기를 맞춰주고, 아니라면 위와 같은 수식을 통해 alignment를 맞춘다.

```
// 우선 힙에 해당 사이즈가 들어갈 공간이 있는지 find_fit을 통해 탐색한다.
if ((bp = find_fit(asize)) != NULL) {
    place(bp, asize); // find_fit으로 결과를 찾았다면, place를 통해 해당 위치에
allocate를 진행한다.
#ifdef DEBUG
    num_expected_free_blocks--; // fit 된 위치에 allocate 되었을 때, free block의
개수는 1 줄어드는 것이 예측되므로, 1을 줄여준다.
    mm_check(); // 종료되기 전, 디버깅 함수를 호출한다.
#endif
    return bp;
}
```

사이즈 align을 맞춘 이후에는 우선 find_fit을 이용하여 free 된 블럭들 중에 들어갈 수 있는 블럭이 있는지 탐색한다. 만약, fit 한 블럭이 있다면, place를 이용하여 블럭을 할당하고, 해당 블럭의 주소를 반환한다. 디버깅 모드에서는, free 블럭의 감소가 예측되므로 mm_check에서 블럭 개수 예측에 이용하는 전역 변수인 num_expected_free_blocks를 1 줄이고 mm_check를 호출하여 디버깅 함수를 부른다.

```
// find_fit으로 적절한 공간을 찾지 못하였으므로, extend_heap을 이용해 힙을 늘려준다.
extendsize = MAX(asize, CHUNKSIZE);
if ((bp = extend_heap(extendsize)) == NULL) {
    return NULL;
}
place(bp, asize); // 공간이 늘어났으므로, place를 통해 해당 위치에 allocate를 진행한다.

#ifdef DEBUG
mm_check(); // 종료되기 전, 디버깅 함수를 호출한다.
#endif

return bp;
```

find_fit으로 적절한 블럭을 찾지 못했을 경우에는, extend_heap을 이용해서 힙을 늘려주고, place를 불러서 할당을 진행한다. 이후, 블럭의 주소를 반환하여 함수를 종료한다. 만약, 디버깅 보드라면 mm_check를 부른다.

mm_free

이 함수는 동적 할당 과정 중에서 free를 naive 하게 구현한 것이다.

```
// 주어진 주소가 적절하지 않은 경우이므로, 바로 종료한다.
if (ptr == 0) {
    return;
}

#ifdef DEBUG
// 블럭이 free 될때, free block의 개수가 1 증가하는 것이 예측되므로, 1 더해준다.
if(IS_BLOCK_ALLOCATED(ptr)){
    ++num_expected_free_blocks;
}
#endif

size_t size = GET_SIZE(HDRP(ptr)); // 할당 해제하고자 하는 블럭의 크기를 구한다.

// heap_listp가 적절하지 않은 값인 경우, 초기화를 진행한다.
if (heap_listp == 0) {
    mm_init();
}
```

주어진 주소가 적절하지 않은 경우에 함수를 바로 종료하고, heap_listp가 적절하지 않은 경우에 mm_init을 호출하여 초기화를 진행하고, 주어진 블럭의 size를 구하는 전처리 과정을 거친다. 디버깅 모드라면, free 블럭이 1 증가하는 것이 예측되므로, 해당 전역 변수 값을 1 증가시킨다.

```
DELETE_TAG(HDRP(NEXT_BLKP(ptr))); // 태그 삭제

// allocation 정보 삭제
PUT_WITH_TAG(HDRP(ptr), PACK(size, 0));
PUT_WITH_TAG(FTRP(ptr), PACK(size, 0));

// coalesce 수행
coalesce(ptr);

#ifdef DEBUG
// 함수가 종료되기 전 디버깅 함수 호출
mm_check();
#endif
```

free 과정이므로, 태그를 우선 삭제한 이후에, allocation 식별자도 0으로 설정해서 allocated 되지 않았음을 표시한다. 이후 coalesce를 수행해서 연속하는 free 블럭들을 합쳐준다. 만약 디버깅 모드라면, 함수가 종료되기 이전에 mm_check를 호출하여 디버깅 정보를 출력한다.

mm_realloc

이 함수는 동적할당 과정 중에서 realloc 과정을 구현한 것이다.

```
// 크기가 0이라면 바로 반환한다.
if (size == 0) {
    return NULL;
}

// malloc을 수행할 때와 같이, 사이즈를 align한다.
if (body_size <= DSIZE) {
    body_size = 2 * DSIZE;
} else {
    body_size = DSIZE * ((body_size + (DSIZE) + (DSIZE - 1)) / DSIZE);
}

body_size += PADDING;

// 남은 크기를 구한다.
remaining_size = GET_SIZE(HDRP(ptr)) - body_size;
```

본격적으로 루틴을 수행하기 이전에, 크기가 0이라면 바로 함수를 종료하고 아니라면 주어진 크기를 align 하는 전처리 과정을 거친다. 이후, 블럭에 남은 크기를 구한다.

확장을 수행해야 하는 경우

만약, reamining_size가 0보다 작다면, 확장을 수행해야 하기 때문에, 앞 뒤 블럭을 탐색하는 과정을 가질 것이다.

다음 블럭이 할당되어 있지 않은 경우

```
            // 다음 블럭이 할당되지 않았다면, 다음 블럭으로의 확장 가능성을 검토한다.

            expandable_size = remaining_size; // 우선 남은 크기를 확장 가능한 크기로 지정한다.
```

```
            if (remaining_size < 0) {
                // 남은 크기가 0보다 작다면, extend_heap_recycle 을 통해 더 확장할 수
있는 크기를 가져온다.
                expandable_size += extend_heap_recycle(ptr, -remaining_size);
            }

            if (expandable_size < 0) {
                // 확장 가능한 크기가 0보다 작다면, extend를 수행한다.
                // 최소 청크 크기를 만족하는, 확장할 크기를 결정한다.
                extendsize = MAX(-expandable_size, CHUNKSIZE);

                // extend_heap을 통해 확장을 수행한다.
                if (extend_heap(extendsize) == NULL) {
                    return NULL;
                }

                // 확장 가능한 크기에 방금 구한 크기를 더한다.
                expandable_size += extendsize;
            }

            // 현재 몸체 크기에 확장 가능한 크기를 더해서 블럭을 확장한다.
            PUT(HDRP(ptr), PACK(body_size + expandable_size, 1));
            PUT(FTRP(ptr), PACK(body_size + expandable_size, 1));
            is_reallocated = TRUE; // 재할당이 완료되었음을 마킹한다.
```

확장을 수행해야 하는데, 다음 블럭이 할당되어 있지 않으므로 다음 블럭까지의 확장을 검토한다. 만약, 다음 블럭까지의 크기 합이 필요로 하는 크기만큼 충분히 크다면, 해당 블럭까지 확장을 수행한다. 아니라면, 즉 다음 블럭까지의 크기 합이 필요로 하는 크기만큼 충분히 크지 않다면 extend_heap을 이용하여 heap을 확장해주고, 해당 크기 만큼 더 확장해서 할당한다.

이전 블럭이 할당되어 있지 않은 경우

```
            // 재할당이 진행되지 않았고, 이전 블럭이 할당되지 않은 상태라면, 이전 블럭으로
의 확장을 시도한다.
            new_ptr = PREV_BLKP(ptr); // 이전 블럭의 주소를 가져온다.

            // 확장 가능한 크기를 검토한다.
            expandable_size = remaining_size + (int)GET_SIZE(HDRP(new_ptr));
            if (expandable_size >= 0) {
                // 확장 가능한 크기가 0 이상이라면, 확장을 진행한다.
#ifdef DEBUG
                // 이때, 예측되는 블럭의 개수를 업데이트 한다.
                --num_expected_blocks;
                --num_expected_free_blocks;
#endif
                // 이전 블럭으로부터 크기를 업데이트하여 재할당을 수행한다.
                PUT(HDRP(new_ptr), PACK(body_size + expandable_size, 1));
                PUT(FTRP(new_ptr), PACK(body_size + expandable_size, 1));
                // 데이터를 이전 포인터로 옮긴다.
                memmove(new_ptr, ptr, MIN(size, body_size));
                is_reallocated = TRUE; // 재할당 되었음을 표시한다.
            }
```

다음 블럭이 비어있지 않았었는데, 이전 블럭이 할당되어 있지 않은 경우라면, 이전 블럭으로의 확장을 검토한다. 만약 이전 블럭까지의 크기 합이 필요로 하는 크기만큼 충분히 크다면, 해당 블럭까지 확장을 수행한다. 이전 블럭의 사이즈를 재설정하고, 블럭 정보를 memmove를 통해 이동함으로써 이를 실현한다.

다음 블럭과 이전 블럭 모두 할당되어 있는 경우

```
// 최종적으로 재할당이 진행되지 않았다면, 새로 malloc을 진행해서 메모리를 할당받는다.
new_ptr = mm_malloc(body_size - DSIZE);
memmove(new_ptr, ptr, MIN(size, body_size)); // 정보 이전
mm_free(ptr); // 기존 할당 받은 공간은 해제한다.
```

최종적으로 앞 뒤 블럭을 이용하여 공간을 확장하는데 실패하였으므로, malloc을 이용해 새로운 메모리를 할당받고 기존 공간은 free 하여 realloc을 수행한다.

확장이 필요한 경우, 각 확장 경우에 따라서 확장을 진행하고 해당 블럭 포인터를 반환하여 함수를 종료하고, 확장이 필요하지 않은 경우, 주소를 그대로 반환한다.

mm_check

이 함수는 힙의 consistency를 체크하는 함수로, 디버깅 모드에서 활용된다.

블럭 예측 데이터와 매칭

이 코드의 곳곳에서는 전역 변수 `num_expected_free_blocks`, `num_expected_blocks` 를 이용하여 free_block의 개수와 block의 개수를 예측하고 있다. 이를 바탕으로 mm_check 에서는 현재 힙에 있는 할당된 블럭의 개수와 free 된 블럭의 개수를 세고 이를 예측값과 비교하여 consistency를 체크한다. 이 때, 예측값은 **전체 블럭 개수에서 FREE 블럭 개수를 뺀 값을 이용하여 비교**한다.

Coalescing 체크

이 파트에서는 coalesce 되지 않은 블럭이 있는지 확인한다. 모든 블럭을 탐색하면서, free 된 블럭이 연속해서 나타난다면, 해당 횟수를 기록하여 저장한다. 만약이 횟수가 1 이상이라면, 에러를 출력한다.

Overlapping 체크

이 파트에서는 overlap하는 블럭이 있는지 체크한다. 모든 블럭을 탐색하면서, 현재 블럭에서 사이즈를 더한 주소 값이 다음 블럭의 시작 주소 값을 침범한다면, overlap으로 판단하고 그 횟수를 기록한다. 이 횟수가 1 이상이라면, 에러를 출력한다.

Alignment 체크

이 파트에서는 힙 블럭의 모든 주소가 align 되어 있는지 확인한다. 모든 블럭을 탐색하면서, 다음 블럭의 주소값과 현재 블럭의 주소 값 차이가 ALIGNMENT로 나눴을 때 그 나머지가 0인지 확인하여 alignment를 확인한다. 만약, alignment가 맞지 않는 것이 발견된다면, 그 횟수를 기록한다. 해당 횟수가 1 이상이라면, 에러를 출력한다.

Helper Functions

extend_heap

이 함수는 인자로 주어진 크기 만큼을 mem_sbrk를 이용하여 힙 공간을 확보하는 함수이다. 주어진 크기를 우선 align 한 이후, mem_sbrk로 공간을 확보하고, 확보한 공간의 블럭에 대해 헤더와 푸터를 설정하는 작업을 수행한다.

place

이 함수는 인자로 주어진 주소와 크기를 이용하여 주어진 주소의 블럭에 주어진 크기를 할당하는 작업을 수행한다. 기본적으로 주어진 주소의 블럭의 헤더와 푸터에 정렬된 사이즈를 지정하는데, 크기를 지정해준 이후에 남은 공간이 블럭이 필요로 하는 최소 공간보다 크다면 split 해서 새로운 free block을 만들어준다.

find_fit

find_fit은 주어진 크기의 블럭이 들어갈 수 있는 알맞은 free 블럭을 찾아주는 역할을 수행하는 함수이다. 기본적으로는 next-fit 정책을 채택했으나, 부분적으로 best-fit을 사용하고 있다. 마지막으로 할당된 블럭으로부터 탐색을 진행하여 충분히 큰 블럭을 찾았을 때에는 해당 블럭의 주소를 바로 반환한다. 하지만, 이 과정에서 찾지 못했을 때에는 힙의 처음부터 탐색할 때 best-fit 정책을 이용하여 블럭을 찾도록 함으로써 throughput에 지장을 주지 않으면서 단편화를 유의미하게 최적화했다.

coalesce

해당 함수는 이전 블럭과 다음 블럭의 할당 여부에 따라서 4가지 경우에 대해 연속하는 free 블럭들을 하나로 합쳐주는 역할을 수행한다.

extend_heap_recycle

이 함수는 realloc에서 사용되는, 특정 블럭 다음에 연속되는 할당되지 않은 블럭들을 더해서 충분한 공간의 크기를 반환하는 함수이다. coalesce가 정상적으로 진행되지 않았을 때를 대비해서, realloc 과정에서 coalesce 되지 않은 블럭들을 이용할 수 있도록 하는 일종의 보험과도 같은 역할을 수행한다.

결과

without mm_check

```
Using default tracefiles in ./traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.004597  1239
 1       yes   99%    5848  0.005228  1119
 2       yes   99%    6648  0.006910   962
 3       yes   99%    5380  0.004936  1090
 4       yes  100%   14400  0.000108132841
 5       yes   95%    4800  0.008306   578
 6       yes   94%    4800  0.007966   603
 7       yes   55%   12000  0.012147   988
 8       yes   51%   24000  0.006036  3976
 9       yes   99%   14401  0.000111130326
10       yes   85%   14401  0.000106136373
Total          89%  112372  0.056450  1991
```

```
Perf index = 53 (util) + 40 (thru) = 93/100

Process finished with exit code 0
```

with mm_check

```
[+] DEBUG STATUS
  - Check the expected heap status and actual heap status is matching...
(expecting based on counting function call)
    * Actual: All (0), Allocated (0), Freed (0)
    * Expect: All (0), Allocated (0), Freed (0)
    [ PASS ] It seems good :)

  - Check any contiguous free blocks that somehow escaped coalescing...
    [ PASS ] It seems good :)

  - Check any blocks overlap...
    [ PASS ] It seems good :)

  - Check the pointers in a heap block point to valid (aligned) heap
addresses...
    [ PASS ] It seems good :)


Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.175778    32
 1       yes   99%    5848  0.177080    33
 2       yes   99%    6648  0.224680    30
 3       yes   99%    5380  0.179331    30
 4       yes  100%   14400  0.299323    48
 5       yes   95%    4800  0.174391    28
 6       yes   94%    4800  0.171758    28
 7       yes   55%   12000  1.292141     9
 8       yes   51%   24000  2.375007    10
 9       yes   99%   14401  0.305249    47
10       yes   85%   14401  0.303729    47
Total          89%  112372  5.678469    20

Perf index = 53 (util) + 1 (thru) = 55/100

Process finished with exit code 0
```

토론 및 개선

- Segregated fit 을 이용하여 더 개선할 필요가 있다.
- Realloc 과정에서 extend_heap_recycle을 사용하지 않아도 되도록 coalesce를 검증할 필요가 있다.