Malloc Lab: Writing a Dynamic Storage Allocator

Report

20220127 임유진

1. 개요

이번 Lab 에서는 malloc, free, realloc 루틴을 수행하고, Space utilization 과 throughput 의 측면에서 정확하고, 효율적이며, 빠른 동적 할당기를 위한 C 프로그램을 작성해보도록 한다.

2. 이론적 배경

- Dynamic Memory Allocation

프로그래머는 런타임에 추가적인 가상 메모리를 얻기 위해 Dynamic memory allocator 를 사용하며, 런타임에만 사이즈를 아는 것이 가능한 자료 구조를 위해 사용된다. Dynamic memory allocator 는 프로세스의 가상 메모리에서 heap 을 관리하며, heap 을 allocated 또는 free 의 상태를 가지는 다양한 사이즈의 블록들의 집합으로 유지한다.

- Dynamic Memory Allocator Performance

Dynamic Memory Allocator 를 위한 performance goal 로는 throughput 과 peak memory utilization 이 존재하며, 이 두 가지 performance goal 은 서로 conflict 하다. Throughput 은 단위 시간당 완료되는 요청의 수이며, peak memory utilization 은 (k + 1)번째 요청을 수행한 뒤 heap 사이즈 대비 (k + 1) 번째까지의 요청을 수행할 때까지 Aggregated payload 의 최댓값으로 나타난다.

- Implicit list/ Explicit list/ Segregated list

Implicit list 는 각 블록에 size 와 allocation status 를 저장하여 모든 블록을 연결하는 방법이며, allocation 에 걸리는 시간이 전체 블록의 수에 비례한다. explicit list 는 pointer 를 통해 free block 들을 연결하는 방법으로, allocation 에 걸리는 시간이 free block 의 개수에 비례한다. Segregated list 는 사이즈 클래스 별로 별도의 free list 를 이용하여, 높은 throughput 을 가지는 방법이다.

- First fit/Next fit/Best fit

Free block 을 찾는 방법으로는 first fit, next fit, best fit 이 존재하며, First fit 은 list 의 시작부터 free block 을 탐색하기 시작하여 사이즈가 맞는 첫번째 free block 을 선택하는 방식이다. Next

fit 은 이전에 탐색이 종료된 지점으로부터 리스트를 탐색하기 시작하는 방식이다. Best fit 은 요구되는 사이즈에 fit 하면서도 가장 적은 바이트가 남게 되는 free block 을 탐색하는 방식으로 memory utilization 을 향상시킬 수 있지만, 매 allocation 마다 리스트를 전부 탐색해야 하기 때문에 느리다는 특징을 가진다.

3. 풀이 과정

Computer System: A Programmer's Perspective 의 malloc 코드로부터 시작하여 동적 할당기의 성능을 향상해 나가는 방식으로 Lab 을 진행하였다.

처음 교과서의 코드와 기존에 작성되어 있던 realloc 함수를 통해 실행했을 때 아래와 같은 결과를 얻을 수 있었고, 이로부터 memory utilization 과 throughput 성능을 향상해 나갔다.

```
[limyoojin@programming2 malloclab-handout]$ ./mdriver
Using default tracefiles in ./traces/
Perf index = 45 (util) + 14 (thru) = 59/100
```

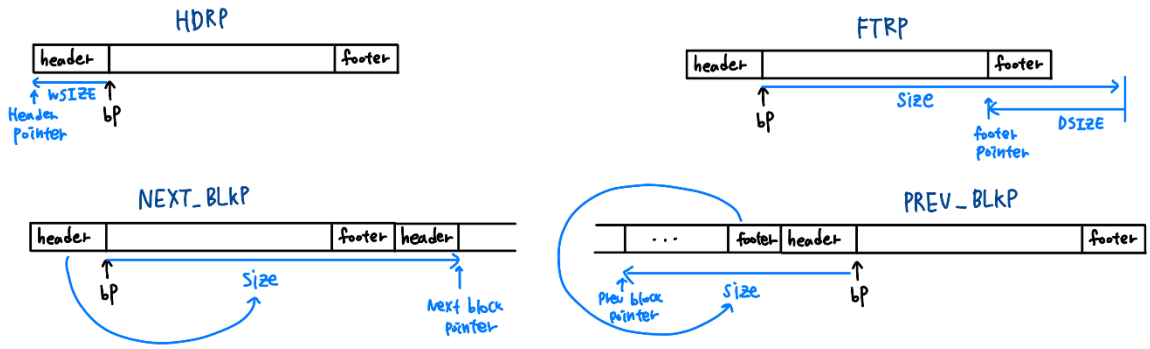
교과서 코드에 대해 분석한 내용은 아래와 같다.

#define 을 통한 매크로를 사용하며 각각의 의미는 아래와 같다.

```
#define WSIZE 4
#define DSIZE 8
#define CHUNKSIZE (1<<12)

#define MAX(x, y) ((x) > (y) ? (x) : (y))

#define PACK(size, alloc) ((size) | (alloc))

#define GET(p)      (*(unsigned int *) (p))
#define PUT(p, val) (*(unsigned int *) (p) = (val))

#define GET_SIZE(p) (GET(p) & ~0x7)
#define GET_ALLOC(p) (GET(p) & 0x1)

#define HDRP(bp)    ((char *)(bp) - WSIZE)
#define FTRP(bp)    ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)

#define NEXT_BLKP(bp) ((char *)(bp) + GET_SIZE(((char *)(bp) - WSIZE)))
#define PREV_BLKP(bp) ((char *)(bp) - GET_SIZE(((char *)(bp) - DSIZE)))
```

WSIZE	word size
DSIZE	double word size, 헤더와 푸터를 합한 사이즈와 같다.

CHUNKSIZE	heap 을 확장하는 기본 사이즈
MAX	두 숫자 중 더 큰 것을 반환
PACK	헤더와 푸터에서 사이즈는 8 의 배수로 하위 3bit 가 0 이 되어, LSB 는 해당 블록이 allocated 되어있는지 여부를 나타내기 때문에 헤더와 푸터에 쓰일 최종적인 값을 계산
GET	포인터를 받아 해당 주소에 저장된 값을 반환한다.
PUT	포인터와 값을 받아 해당 주소에 값을 쓴다.
GET_SIZE	헤더와 푸터로부터 하위 3bit 을 제외한 정보, 즉 size 에 대한 정보를 얻는다.
GET_ALLOC	헤더와 푸터로부터 LSB, 즉 allocated 에 대한 정보를 얻는다.
HDRP	블록 포인터로부터 헤더 포인터를 얻는다.
FTRP	블록 포인터로부터 푸터 포인터를 얻는다.
NEXT_BLKP	블록 포인터로부터 다음 블록의 포인터를 얻는다.
PREV_BLKP	블록 포인터로부터 이전 블록의 포인터를 얻는다.



함수들은 다음과 같다.

extend heap	memlib.c 에 정의되어 있는 mem_sbrk 함수를 호출하여 heap 을 확장하며, 새롭게 확장된 블록을 위한 헤더, 푸터와 에필로그 블록 헤더를 갱신한다.
find_fit	heap 의 시작을 가리키는 포인터인 heap_ptr 부터 시작하여, 사이즈가 0 인 에필로그 블록에 다다를 때까지 다음 블록으로 이동하며 free 이면서 요청받은 사이즈에 fit 하는 블록이 있다면 그 블록의 포인터를 반환하며, fit 하는 block 이 없다면 NULL 을 반환한다.
place	요청받은 사이즈에 맞는 free block 을 찾았을 때, 만약 해당 free 블록의 크기에서 요청받은 블록의 크기를 뺀 것이 최소 free block (헤더, 푸터, payload 를 위한 공간이 필요하며 8 byte alignment 로 인해 최소 2 * DSIZE(16 byte)가 필요)크기보다 크다면 블록을 쪼개며, 그렇지 않다면 블록을 쪼개지 않고 해당 free 블록 전체를 할당한다.

coalesce	heap 을 확장하거나 블록을 free 할 할 때, 확장되며 새롭게 생긴 블록, free 하는 블록 앞 뒤의 블록이 free 라면 false fragment 방지를 위해 coalescing 을 수행해주는 함수이며, 앞 뒤 블록이 모두 allocated 인 경우, 앞 블록은 allocated 뒤 블록은 free 인 경우, 앞 블록은 free 뒤 블록은 allocated 인 경우, 앞 뒤 블록 모두 free 인 경우로 나누어 coalesing 을 진행하고 헤더와 푸터를 갱신해준다.

교과서의 코드에서는 free block 이 size 와 allocation 정보를 포함하는 헤더와 푸터를 가지고 있어 모든 블록이 연결되어 있는 implicit list 의 방식에서, 요청 사이즈에 fit 하는 free block 을 찾을 때는 매 검색 시 처음부터 순회하는 first fit 이 적용되어 있음을 확인할 수 있다.

- realloc 최적화

우선 기존에 mm.c 에 작성되어 있던 realloc 함수에 대한 최적화를 진행하고자 하였다.

먼저 예외적인 경우에 대한 처리를 진행해주었다. 만약 realloc(void *ptr, size_t size)에 인자로 주어진 포인터가 NULL 값이라면 realloc(ptr, size) 호출이 mm_malloc(size)와 동일해야 하므로, ptr 이 NULL 이라면 mm_malloc(size)를 반환해주도록 처리하였다. 다음으로 realloc 에 인자로 주어진 size 가 0 이라면, realloc(ptr, size) 호출이 mm_free(ptr)과 동일해야 하기 때문에, size 가 0 이라면 mm_free 를 통해 ptr 을 할당해제해준 후 NULL 을 반환하도록 하였다.

다음으로는 성능 향상을 위한 최적화를 진행해주었다.

기존에 작성되어 있던 realloc 함수는 요청받은 사이즈와 관계없이 항상 새로 malloc 을 진행하고, 기존의 동적할당을 해제해주고 있음을 확인할 수 있다. 하지만 realloc 을 통해 요청받은 payload 사이즈가 기존에 요청받아 할당했던 payload 사이즈보다 작거나 같을 수 있고, 기존에 요청받았던 payload 사이즈보다 크더라도 Data alignment 를 맞추기 위해 allocation 되는 블록에 padding 이 포함되는 경우가 있어 padding 과 payload 를 합한 것보다는 작거나 같을 수 있다. 이런 경우에 대해서는 기존의 블록으로 요청받은 payload 의 사이즈를 커버할 수 있으므로, 새로운 동적할당 영역을 할당해주지 않아도 된다. 따라서 기존 블록의 사이즈보다 요청받은 사이즈에 헤더와 푸터의 사이즈를 더해 alignment 인 8 의 배수로 올림한 사이즈가 작거나 같다면 곧바로 기존의 포인터를 반환하도록 수정하였다. 해당 경우에 대한 코드는 아래와 같다.

```
size_t oldsize = GET_SIZE(HDRP(oldptr));
size_t newsize = ALIGN(size + DSIZE);
if (newsize <= oldsize) {
        return oldptr;   }
```

위에서 말한 경우와 같이 동적할당을 다시 진행하지 않아도 되는 경우가 아니라면 요청받은 사이즈에 맞게 새롭게 동적할당이 일어나고 기존 메모리의 복사가 일어나야 한다. 이때 만약 기존의 블록 뒤의 블록이 free block 이고, 기존 블록과 다음 블록을 합한 사이즈가 요청받은 사이즈를 커버할 수 있다면 기존 블록과 다음 블록을 합한 블록을 새롭게 할당하는 블록으로 했을 때 메모리의 복사가 이루어질 필요가 없이 헤더와 푸터만 갱신해주면 된다. 따라서 메모리 카피를 위해 걸리는 시간을 줄일 수 있다. 해당 경우에 대한 코드는 아래와 같다.

```
if ((GET_ALLOC(HDRP(nextptr)) == 0) && (newsize <= (oldsize + nextsize))){
    delete_from_free_list(nextptr);
    PUT(FTRP(nextptr), PACK((oldsize + nextsize), 1));
    PUT(HDRP(oldptr), PACK((oldsize + nextsize), 1));
    return oldptr;
}
```

위에서 설명했던 두 가지 경우가 아니라면, 새로운 동적할당과 메모리 카피가 발생하는 것이 필수적이다. 이때 카피되는 사이즈는 기존의 payload 사이즈와 요청받은 payload 사이즈 중 작은 것을 따르며, 기존의 payload 사이즈는 블록의 사이즈에서 헤더와 푸터의 사이즈를 뺀 것과 같다. 따라서 요청받은 사이즈인 size 와 기존 블록에서 헤더와 푸터를 제외한 사이즈인 GET_SIZE(HDRP(oldptr)) – DSIZE 중 작은 것을 copySize 로 하여 memcpy 를 통해 메모리 복사를 해준 뒤 기존의 동적할당을 해제해주었다.

기존의 코드에서 realloc 에 대한 최적화를 진행하면서 불필요한 경우에 대한 mm_malloc 과 mm_free 의 호출을 줄였기 때문에 throughput 성능 향상을 예상할 수 있었고, 실제로 아래와 같이 throughput 의 점수가 3 점 증가한 결과를 확인할 수 있었다.

```
[limyoojin@programming2 malloclab-handout]$ ./mdriver
Using default tracefiles in ./traces/
Perf index = 46 (util) + 17 (thru) = 64/100
```

- Explicit List

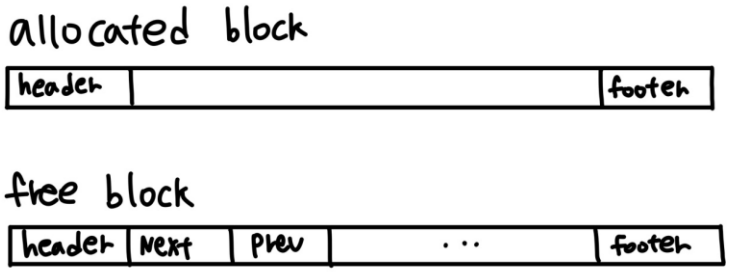
realloc 함수에 대한 최적화를 진행하며 성능이 향상되었음에도 여전히 낮은 throughput 성능을 보이고 있음을 확인할 수 있다. First fit 을 사용하는 implicit list 의 경우 allocation 시 요청되는 사이즈에 알맞은 블록을 찾기 위해 free 블록뿐만 아니라 allocated 블록까지 모두 순회하기 때문에 시간 복잡도가 전체 블록 개수에 비례하므로 throughput 성능이 낮게 측정되는 것이라고 판단하였다. 따라서 throughput 을 향상시키기 위해 allocation 시 요청되는 사이즈에 알맞은 블록을 찾기 위해 free block 만을 순회하는 explicit list 방식으로 코드를 개선하기로 하였다.

explicit list 의 방식에서는 free block 에 헤더와 푸터뿐만 아니라 다음 free block 과 이전 free block 을 가리키는 포인터를 저장하기 위한 공간이 추가적으로 필요하다. 따라서 #define 을 통한 매크로를 추가적으로 선언해주었고, 아래와 같다.

```
#define NEXT_PTR(bp) (*(char **)(bp))
#define PREV_PTR(bp) (*(char **)((char *)(bp) + WSIZE))
#define PUT_NEXT_PTR(bp, ptr) (*(char **)(bp) = (char*)(ptr))
#define PUT_PREV_PTR(bp, ptr) (*(char **)((char *)(bp) + WSIZE) =(char*)(ptr))
```

NEXT_PTR(bp)	bp 가 가리키는 블록으로부터 다음 free block 을 가리키는 포인터를 읽는다.
PREV_PTR(bp)	bp 가 가리키는 블록으로부터 이전 free block 을 가리키는 포인터를 읽는다.
PUT_NEXT_PTR(bp, ptr)	bp 가 가리키는 블록의 다음 free block 포인터를 ptr 값으로 써준다.
PUT_PREV_PTR(bp, ptr)	bp 가 가리키는 블록의 이전 free block 포인터를 ptr 값으로 써준다.

allocated block 은 size 와 allocated 정보를 갖는 헤더와 푸터를 포함하며, free block 은 size 와 allocated 정보를 갖는 헤더와 푸터, 그리고 이전 free block 과 다음 free block 으로의 포인터를 가지는 구조를 갖는다.



또한 free block list 에 대해서는 LIFO 방식을 적용하며, First Fit 을 적용하도록 구현하였다. free block list 의 관리를 위한 세 가지 함수를 추가적으로 선언하여 사용하였고 이는 다음과 같다.

```
static void insert_to_free_list(void *bp) {
    char *start_ptr = free_ptr;
    if (free_ptr == NULL) {
        free_ptr = bp;
        PUT_PREV_PTR(bp, NULL);
        PUT_NEXT_PTR(bp, NULL);
    } else {
        if (start_ptr != bp) {
            PUT_PREV_PTR(start_ptr, bp);
            PUT_NEXT_PTR(bp, start_ptr);
            PUT_PREV_PTR(bp, NULL);
        }
        free_ptr = bp;
    }
}
```

insert_to_free_list 함수는 인자로 받은 블록 포인터 bp를 free list에 삽입하는 함수이며, LIFO 방식을 사용하므로 list 의 첫번째 free block 을 가리키는 free_ptr 의 값에 따라 경우를 나누었다.

free_ptr 이 NULL 인 경우 free list 에 free block 이 존재하지 않음을 의미하므로, bp 를 free block list 의 첫번째 블록으로 설정해주면서 이전 free block 포인터와 다음 free block 포인터를 NULL 로 지정해주었다. free_ptr 이 NULL 이 아닌 경우에는 기존의 첫번째 free block 이 이전 free block 으로 bp 를 가리키고, bp 가 다음 free block 으로 기존의 첫번째 free block 을 가리키고, free_ptr 의 값을 bp 로 설정해주었다.

```
static void delete_from_free_list(void *bp) {
    char *prev_ptr = PREV_PTR(bp);
    char *next_ptr = NEXT_PTR(bp);

    if (prev_ptr == NULL && next_ptr == NULL) {
        free_ptr = NULL;
    } else if (prev_ptr == NULL && next_ptr != NULL) {
        PUT_PREV_PTR(next_ptr, NULL);
        free_ptr = next_ptr;
    } else if (prev_ptr != NULL && next_ptr == NULL) {
        PUT_NEXT_PTR(prev_ptr, NULL);
    } else {
        PUT_PREV_PTR(next_ptr, prev_ptr);
        PUT_NEXT_PTR(prev_ptr, next_ptr);
    }
}
```

delete_from_free_list 는 인자로 받은 bp 가 가리키는 free block 을 free block list 로부터

삭제해주는 함수이다. 이때 해당 free block 의 위치에 따라 이전, 다음 free block 이 존재하지 않을 수 있어 이전 free block, 다음 free block 이 모두 존재하지 않는 경우, 이전 free block 이 존재하지 않고 다음 free block 만 존재하는 경우, 이전 free block 은 존재하지만 다음 free block 은 존재하지 않는 경우, 이전, 다음 free block 이 모두 존재하는 경우로 각각 나누어 진행하였다. 이전 free block 의 다음 free block 포인터, 다음 free block 의 이전 free block 포인터를 경우에 따라 알맞게 수정해주며, 해당 bp 가 list 의 첫번째 블록이었을 경우 free block list 의 첫번째 블록을 가리키는 free_ptr 의 값 또한 수정해준다.

```
static void modify_free_list(void *prev_ptr, void *next_ptr, void *next) {

    if (prev_ptr == NULL && next_ptr == NULL) {
        PUT_PREV_PTR(next, NULL);
        PUT_NEXT_PTR(next, NULL);
        free_ptr = next;
    } else if (prev_ptr == NULL && next_ptr != NULL) {
        PUT_PREV_PTR(next, NULL);
        PUT_NEXT_PTR(next, next_ptr);
        PUT_PREV_PTR(next_ptr, next);
        free_ptr = next;
    } else if (prev_ptr != NULL && next_ptr == NULL) {
        PUT_PREV_PTR(next, prev_ptr);
        PUT_NEXT_PTR(next, NULL);
        PUT_NEXT_PTR(prev_ptr, next);
    } else {
        PUT_NEXT_PTR(prev_ptr, next);
        PUT_PREV_PTR(next, prev_ptr);
        PUT_NEXT_PTR(next, next_ptr);
        PUT_PREV_PTR(next_ptr, next);
    }
}
```

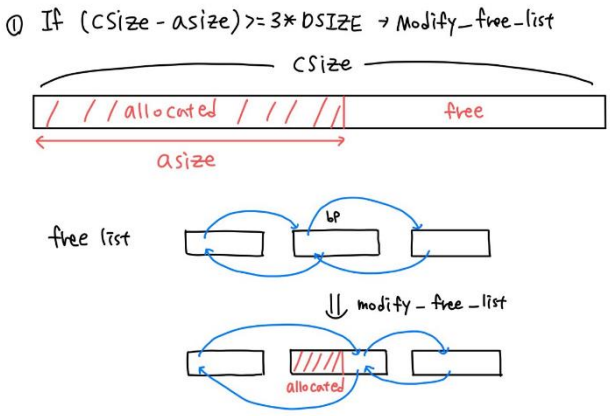
modify_free_list 함수는 place 함수의 호출 시 경우에 따라 호출되는 함수이다. 만약 free block 에서 request block 의 크기를 뺀 크기가 최소 블록 사이즈(explicit list 에서는 free block 에 대해 헤더, 푸터, next free block 포인터, previous free block 포인터, 데이터를 위한 공간이 필수적으로 필요하며 8 byte alignment 이므로 3 * DSIZE) 이상일 때 free block 을 쪼개 앞부분을 allocation 부분으로 사용하고 뒤 부분을 free 부분으로 남겨두게 되는데 이때 남겨지는 뒤 부분은 list 의 맨 앞에 새로 삽입하지 않고 free block 의 이전, 다음 free block 과 남겨진 뒤 부분이 연결될 수 있도록 하였다.

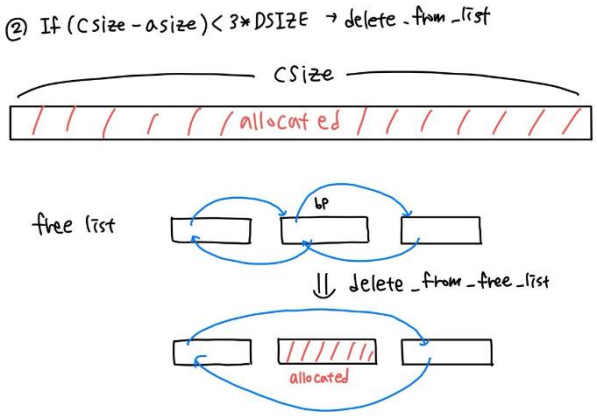
```
static void *find_fit(size_t asize) {
    void *bp;
    if (free_ptr == NULL) {
        return NULL;
    }

    for (bp = free_ptr; bp != NULL; bp = NEXT_PTR(bp)) {
        if ((GET_ALLOC(HDRP(bp)) == 0) && (asize <= GET_SIZE(HDRP(bp)))) {
            return bp;
        }
    }

    return NULL;
}
```

Explicit list 에서는 요청되는 사이즈에 맞는 free block 을 찾기 위해 모든 블록을 순회하는 implicit list 와 달리, free block list 를 순회하기 때문에 위와 같이 find_fit 함수를 수정해주었다.

```
static void place(void *bp, size_t asize) {
    size_t csize = GET_SIZE(HDRP(bp));
    char *prev_ptr = PREV_PTR(bp);
    char *next_ptr = NEXT_PTR(bp);
    char *next;

    if ((csize - asize) >= (3*DSIZE)) {
        PUT(HDRP(bp), PACK(asize, 1));
        PUT(FTRP(bp), PACK(asize, 1));
        next = NEXT_BLKP(bp);
        PUT(HDRP(next), PACK(csize-asize, 0));
        PUT(FTRP(next), PACK(csize-asize, 0));
        modify_free_list(prev_ptr, next_ptr, next);
    } else {
        PUT(HDRP(bp), PACK(csize, 1));
        PUT(FTRP(bp), PACK(csize, 1));
        delete_from_free_list(bp);
    }
}
```

수정된 버전의 place 코드는 위와 같고, place 호출 시 발생할 수 있는 경우와 각 경우에 대한 처리는 다음 그림과 같다.

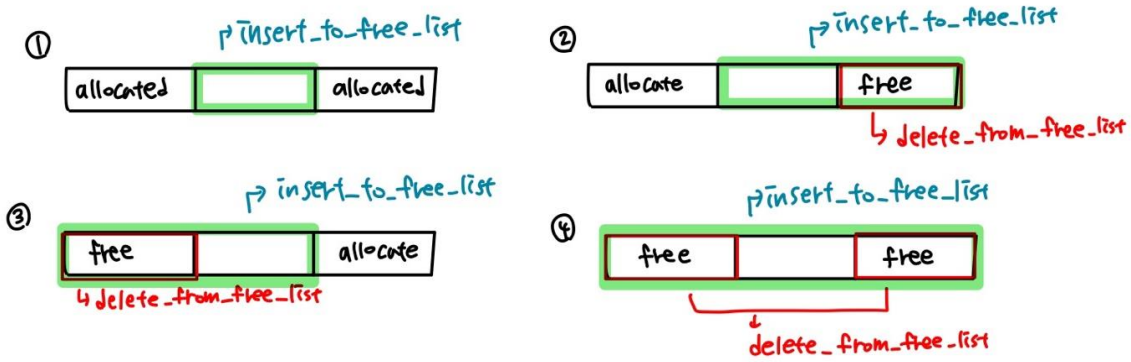




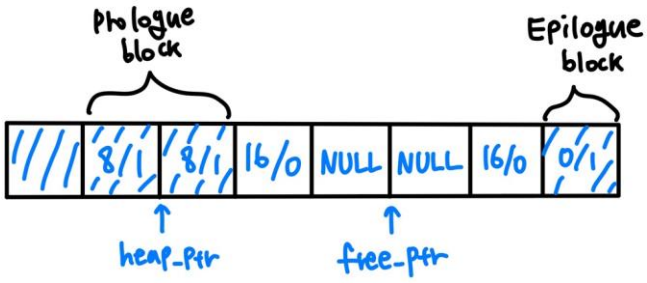

```
static void *coalesce(void *bp) {
    size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
    size_t size = GET_SIZE(HDRP(bp));

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

또한 coalescing 을 할 때도 free list 의 관리가 필요하며 이에 대한 수정된 코드는 위와 같다.

free 하는 블록의 이전 블록과 다음 블록이 모두 allocated 인 경우, 이전 블록이 allocated 다음 블록이 free, 이전 블록이 free 다음 블록이 allocated, 이전 블록과 다음 블록이 모두 free 인 경우로 나누어 생각하여 free block 을 합치고 새롭게 free block list 에 추가하는 방식으로 coalescing 을 진행하며, coalescing 시 발생할 수 있는 경우와 각각의 경우에 대한 처리는 아래 그림과 같다.



또한 mm_init 함수에서 처음 heap 을 확장하기 전 기존에 prologue block 과 Eplilogue block 을 포함해 4 개의 블록을 할당하던 것을, 8 개의 블록을 할당하여 prologue block, 첫번째 free block, epilogue block 을 가지도록 수정하였다.



이와 같이 기존의 implicit list 방식을 explicit list 방식으로 전환함에 따라, allocation 시 free block list 만을 순회하므로 throughput 측면에서의 성능 향상을 예상할 수 있었고, 실제로 아래와 같이 기존에 17 점이던 throughput 점수가 40 점까지 향상됨을 확인할 수 있었다.

```
[limyoojin@programming2 malloclab-handout]$ ./mdriver
Using default tracefiles in ./traces/
Perf index = 47 (util) + 40 (thru) = 87/100
```

```
[limyoojin@programming2 malloclab-handout]$ ./mdriver -v -g
Using default tracefiles in ./traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   93%    5694  0.000131 43300
 1       yes   93%    4805  0.000112 42864
 2       yes   55%   12000  0.002124  5649
 3       yes   55%    8000  0.002071  3863
 4       yes   51%   24000  0.002179 11014
 5       yes   51%   16000  0.002070  7731
 6       yes   94%    5848  0.000126 46561
 7       yes   94%    5032  0.000104 48571
 8       yes   99%   14400  0.000095151102
 9       yes   99%   14400  0.000095151420
10       yes   96%    6648  0.000192 34589
11       yes   96%    5683  0.000174 32623
12       yes   97%    5380  0.000176 30620
13       yes   97%    4537  0.000146 31139
14       yes   89%    4800  0.000401 11970
15       yes   89%    4800  0.000403 11905
16       yes   85%    4800  0.000431 11142
17       yes   85%    4800  0.000430 11155
18       yes   42%   14401  0.000278 51821
19       yes   42%   14401  0.000289 49796
20       yes   53%   14401  0.000087164583
21       yes   53%   14401  0.000094153202
22       yes   66%      12  0.000000 40000
23       yes   66%      12  0.000000 40000
24       yes   89%      12  0.000000 40000
25       yes   89%      12  0.000000 60000
Total          78%  209279  0.012210 17140

Perf index = 47 (util) + 40 (thru) = 87/100
correct:26
perfidx:87
```

- mm_check(void)

heap consistency 를 위해 작성한 mm_check 함수는 아래와 같다.

```
int mm_check(void) {
    char *start;
    int error = 0;
    char *heap_start = (char *)mem_heap_lo();
    char *heap_end = (char *)mem_heap_hi();
    for (start = free_ptr; start != NULL; start = NEXT_PTR(start)) {
        if (GET_ALLOC(HDRP(start)) != 0) {
            printf("[ERROR] free block(%p) is not marked as free\n", start);
            error = 1;
        }
    }
    for (start = heap_ptr; GET_SIZE(HDRP(start)) > 0; start = NEXT_BLKP(start)) {
```

```
        if (start < heap_start || start > heap_end) {
            printf("[ERROR] The block pointer(%p) is out of heap range(%p~%p)\n", start,
mem_heap_lo(), mem_heap_hi());
            error = 1;
        }

        if (GET_ALLOC(HDRP(start)) == 1 && GET_ALLOC(HDRP(NEXT_BLKP(start))) == 1) {
            if (FTRP(start) >= HDRP(NEXT_BLKP(start))) {
                printf("There is overlap between allocated blocks(%p~%p, %p~%p)",
HDRP(start), FTRP(start), HDRP(NEXT_BLKP(start)), FTRP(NEXT_BLKP(start)));
                error = 1;
            }
        }
    }
    return !error;
}
```

mm_check 에 구현한 기능은 3 가지로, (1) free list 의 블록 중 free 로 mark 되지 않은 블록이 있는지 확인, (2) heap range 를 넘어가는 block pointer 가 있는지 확인, (3) allocated 블록 간의 overlap 이 있는지 확인하는 것이다. (1)은 free block list 를 차례로 순회하며 만약 GET_ALLOC 매크로로 얻은 allocation 값이 0 이 아니라면 에러 메시지를 띄우도록 하였다. 모든 블록을 순회하면서 만약 블록 포인터의 주소가 mem_heap_lo()와 mem_heap_hi()를 통해 얻은 heap 의 첫번째 바이트보다 작거나 마지막 바이트보다 크면 heap range 밖에 있으므로 에러 메시지를 띄우도록 하여 (2) 기능을 구현하였다. 마지막으로 모든 블록을 차례대로 순회하면서 만약 이전 블록의 푸터 주소가 다음 블록의 헤더 주소보다 크다면 overlap 이 존재한다는 의미이므로 에러 메시지를 띄우도록 하여 (3) 기능을 구현하였다.

마지막으로 에러가 하나라도 발생하는 경우에 0 으로 초기화했던 변수 error 의 값을 1 로 설정하고, !error 을 반환함으로써 에러가 없는 경우에는 1 이 에러가 있는 경우에는 0 이 반환되도록 하였다.

mm_malloc(), mm_free(), mm_realloc()의 마지막 부분에서 mm_check() 함수를 호출하여 heap 의 consistency 를 확인하도록 했을 때 결과는 아래와 같이 heap consistency 체크로 인해 throughput 성능이 낮아지긴 했지만 에러 메시지가 발생하지 않음을 확인할 수 있었다.

```
[limyoojin@programming2 malloclab-handout]$ ./mdriver
Using default tracefiles in ./traces/
Perf index = 47 (util) + 14 (thru) = 60/100
```

4. 결론

이번 Lab 을 통해서 수업 시간에 다루었던 동적 할당기의 여러가지 구현 방식을 직접 구현해보고, 구현 방식에 따른 throughput 과 memory utilization 의 측면에서의 성능 차이를 확인해볼 수 있어 Dynamic Memory Allocator 에 대한 이해도를 높일 수 있었다.