System Programming Project 4

담당 교수 : 이영민

이름 : 방지혁

학번 : 20190808

1. 개발 목표

본 프로젝트4의 목표는 동적 메모리 할당자를 구현하는 것이다. 기존의 함수들과 동일한 인터페이스를 가지고 같은 동작을 하는 것뿐만 아니라 naive한 방식보다 더 향상된 성능을 보이고자 한다.

2. 개발 내용

A. 설계 방향

Implicit, explicit, segregated 방식 중에 segregated 방식을 택해 free 블록만 순회하여 시간 복잡도를 개선할 수 있도록 하였다.

또한, best-fit 방식을 사용하여 큰 블록을 불필요하게 분할하여 fragmentation이 일어나는 것을 방지하고자 했다.

B. 개발 내용

- **전역 변수 설명**

  ✓ static char *heap_listp;

  힙의 시작점을 가리키는 포인터로 mm_init으로 heap이 초기화 될 경우, prologue block의 시작 주소를 저장한다.

  ✓ static char *heap_first_free_block;

  explicit 구현 방식의 핵심 요소인 free list의 가장 첫 번째 block을 가리키는 포인터이다. NULL일 경우 사용 가능한 free 블록이 없음을 의미한다.

- **매크로**

  ✓ #define WSIZE 4 및 #define DSIZE 8

  WSIZE는 header, footer 및 word 사이즈를 의미하고, DSIZE는 align을 위한 더블 워드 크기이다.

  ✓ #define CHUNKSIZE (1<<12)

  heap 확장 시 4096 byte를 늘린다.

- ✓ #define ALIGNMENT 8
- ✓ #define ALIGN(size) (((size) + (ALIGNMENT - 1)) & ~0x7)
- ✓ #define SIZE_T_SIZE (ALIGN(sizeof(size_t)))

  8 byte 정렬을 위한 매크로이다.
- ✓ #define MAX(x, y) ((x) > (y) ? (x) : (y))

  두 값 중 큰 값을 반환한다.
- ✓ #define PACK(size, alloc) ((size) | (alloc))

  블록 크기 및 할당 여부 bit를 합쳐서 하나의 워드로 압축한다.
- ✓ #define GET(p) (*(unsigned int *)(p))
- ✓ #define PUT(p, val) (*(unsigned int *)(p) = (val))

  주소 p를 읽거나 지정한 val 값의 word를 쓰는 매크로이다.
- ✓ #define GET_SIZE(p) (GET(p) & ~0x7)
- ✓ #define GET_ALLOC(p) (GET(p) & 0x1)

  Header 및 footer에서 크기와 할당 여부를 추출하는 매크로이다.
- ✓ #define HDRP(bp) ((char *)(bp) - WSIZE)
- ✓ #define FTRP(bp) ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)

  block 포인터로 header 주소 또는 footer 주소를 계산하는 매크로이다.
- ✓ #define NEXT_BLKP(bp) ((char *)(bp) + GET_SIZE((char *)(bp) - WSIZE))
- ✓ #define PREV_BLKP(bp) ((char *)(bp) - GET_SIZE((char *)(bp) - DSIZE))

  block 포인터로 다음 또는 이전 block ptr 주소를 계산하는 매크로이다.
- ✓ #define NEXT_PTR(bp) ((char *)(bp))
- ✓ #define PREV_PTR(bp) ((char *)(bp + WSIZE))

  Free block 내의 payload에 저장되어 있는 다음 free block을 가리키는 포인터 혹은 이전 free block을 가리키는 포인터에 접근하는 매크로이다.

- **함수 별 설명**

  ✓ int mm_init(void);

  초기 heap 구조를 생성한다. 8 byte 정렬을 위한 1word 크기의 패딩, prologue header, prologue footer, epilogue header로 구성되며 exten_heap(4)를 호출하며 heap 크기를 늘린다.

  ✓ void mm_malloc(size_t size);

  size가 0일시 의미가 없기에 NULL을 반환하며 종료한다. 우선 8 byte에 맞도록 정규화를 하고 이 블록 사이즈에 맞게 find_fit(asize)를 호출한다. 후술하겠지만, 이는 best fit 알고리즘을 이용하며 만약 찾았을 경우 place()함수를 호출하여 할당하며 종료한다. 그러나 적절한 block을 찾지 못했을 경우 extend_heap 함수를 호출하여 asize와 CHUNKSIZE 둘 중 더 큰 크기만큼 확장한다.

  ✓ void mm_free(void *bp);

  인접한 블록과 병합하여 할당한 메모리 블록을 해제하는 함수이다. PUT, HDRP, FTRP, PACK 매크로를 이용하여 header와 footer를 free로 변경한다. 또한, 인접 블록과 병합하기 위해서 coalesce 함수를 호출한다.

  ✓ void *mm_realloc(void *ptr, size_t size);

  인자로 들어온 ptr이 NULL인 경우는 mm_malloc(size) 호출하여 새롭게 할당하고, 인자로 들어온 size가 0인 경우 mm_free(ptr)을 호출해 메모리를 해제해준다. 이후 크기를 정규화하고, 기존 블록과 크기를 비교한다. 만약 새 크기가 기존 크기보다 작거나 같다면 realloc이 필요 없기에 그냥 그대로 반환한다. 다음 블록이 free 상태이고, 해당 블록과 합친 크기가 요청한 크기보다 크거나 같다면 free list에서 제거하고, 헤더와 푸터를 업데이트하지만, 메모리 복사는 필요 없다. 만약, 단순 확장으로는 불가능하다면 새로운 메모리를 할당하고 기존 블록을 해제해준다.

  ✓ static void place(void *bp, size_t asize);

  find_fit을 통해 찾은 free 블록을 free list에서 제거한다. 만약 남은 크기가 임계치로 정한 최소 블록 크기 이상일 경우에만 블록을 분할한다. 이렇게 단편화를 줄이고자 한다.

- ✓ static void *find_fit(size_t asize);

  best fit 알고리즘으로 구현했다. 리스트를 순회하며 정확한 크기를 발견했을 시 즉시 반환하고, 만약 없다고 하더라도 요구되는 크기보다 큰 블록 중 가장 작은 것을 선택한다.

- ✓ static void *extend_heap(size_t words);

  크기를 정규화하고 새 블록을 초기화하는데, 이전 free 블록과 병합 가능하다면 병합한다.

- ✓ static void *coalesce(void *bp);

  4가지 케이스로 나누어서 구현한다. 이전 블록만 free라면 이전 블록을 free list에서 제거하고 병합한다. 다음 블록만 free라면 다음 블록을 free list에서 제거하고 병합한다. 양쪽 블록 모두 free라면 모두 제거해주고 병합한다. 제거된 block들은 free list에 추가해준다.

- ✓ static void insert_free_block(void *bp);

  LIFO 방식으로 구현했다. Heap_first_free_block이 NULL이라면 첫번째 free 블록이 되기 때문에 이에 맞게 구현을 했다. 기존 free블록이 있는 경우 새 블록의 next에 기존 첫 블록을 넣고, 새 블록의 prev에 NULL을 넣는다. 기존 첫 블록의 prev에는 새 블록을 넣고, head를 저장하는 전역 변수를 업데이트 한다.

- ✓ static void remove_free_block(void *bp);

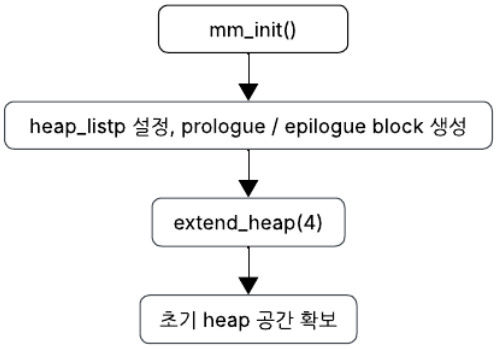
  인자로 들어온 block pointer가 NULL이면 종료하고, 다음 ptr과 이전 ptr을 얻어 변수에 저장한다. 만약 둘 다 NULL일 경우 유일한 블록이라는 뜻이다.

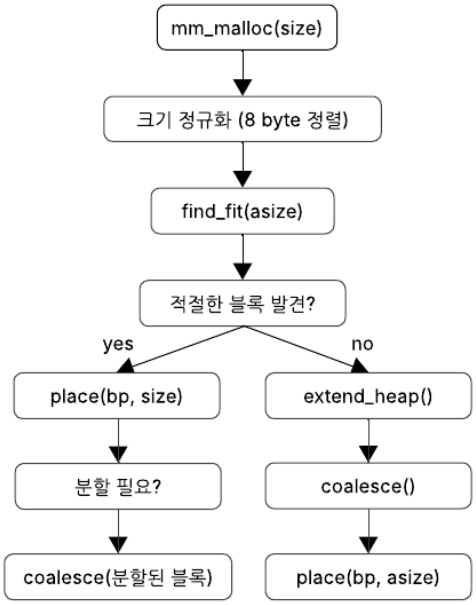
  prev_ptr만 NULL이라면 다음 블록이 새 첫 블록이기에 첫 블록을 가리키는 전역 변수에 다음 ptr을 넣는다. 그리고 prev에 NULL을 넣는다.

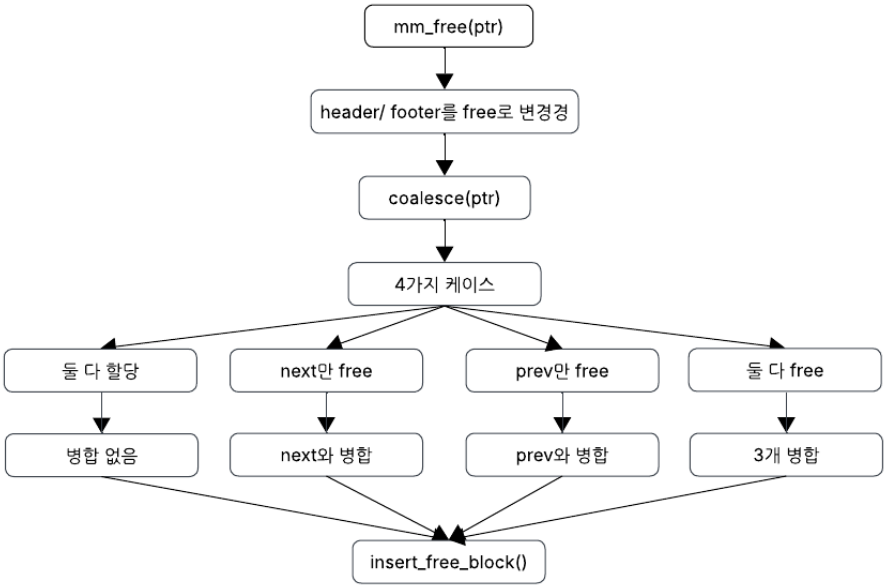
  만약 next_ptr이 NULL이라면 삭제되는 블록이 마지막 블록이라는 뜻이기에 이전 블록이 마지막 블록이 되기에 이전 블록의 next에 NULL을 넣는다.

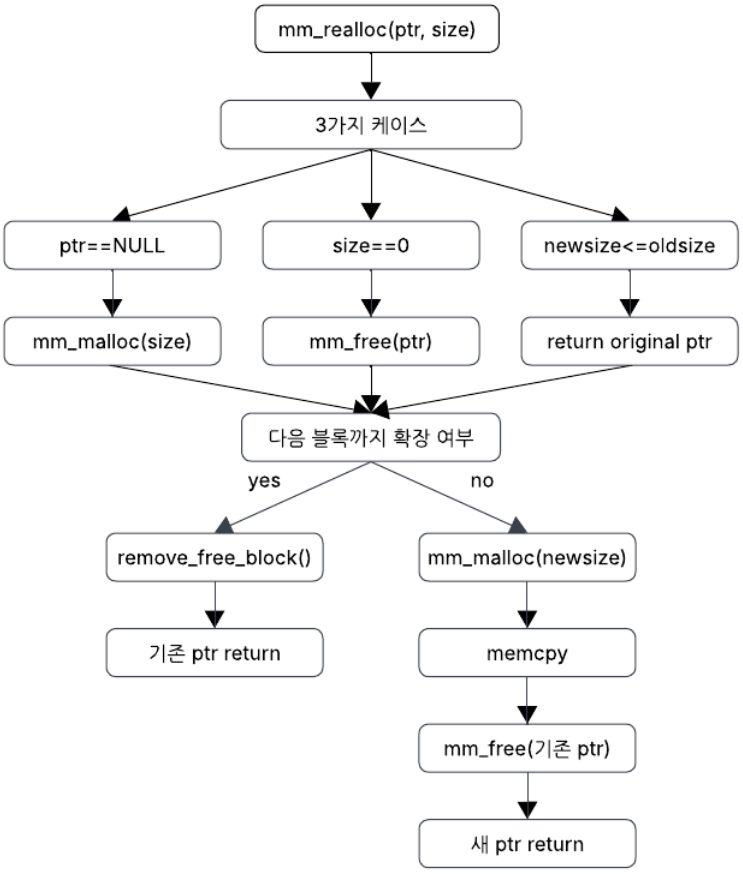
  마지막으로 둘 다 NULL이라면 중간 블록이기 때문에 이전 블록과 다음 블록을 서로 연결시켜준다.

C. 플로우 차트


mm_init()
heap_listp 설정, prologue / epilogue block 생성
extend_heap(4)
초기 heap 공간 확보


mm_malloc(size)
크기 정규화 (8 byte 정렬)
find_fit(asize)
적절한 블록 발견?
yes
no
place(bp, size)
extend_heap()
분할 필요?
coalesce()
coalesce(분할된 블록)
place(bp, asize)


mm_free(ptr)
header/ footer를 free로 변경경
coalesce(ptr)
4가지 케이스
둘 다 할당
next만 free
prev만 free
둘 다 free
병합 없음
next와 병합
prev와 병합
3개 병합
insert_free_block()


mm_realloc(ptr, size)
3가지 케이스
ptr==NULL
size==0
newsize<=oldsize
mm_malloc(size)
mm_free(ptr)
return original ptr
다음 블록까지 확장 여부
yes
no
remove_free_block()
mm_malloc(newsize)
기존 ptr return
memcpy
mm_free(기존 ptr)
새 ptr return

3. **성능 평가 결과**

```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.000342 16673
 1       yes   99%    5848  0.000337 17338
 2       yes   99%    6648  0.000349 19032
 3       yes  100%    5380  0.000233 23130
 4       yes   99%   14400  0.000214 67290
 5       yes   96%    4800  0.003697  1298
 6       yes   96%    4800  0.003569  1345
 7       yes   55%   12000  0.027310   439
 8       yes   51%   24000  0.094030   255
 9       yes   87%   14401  0.000205 70215
10       yes   67%   14401  0.000131109847
Total          86%  112372  0.130417   862

Perf index = 52 (util) + 40 (thru) = 92/100
```

Trace7 및 8에서 낮은 지표를 보여주는데 이는 복잡한 패턴으로 인해 성능이 저하된 것으로 보인다. 그러는 과정에서 throughput이 희생된 측면이 있다. 이는 크기별 별도 free 리스트를 구현하는 segregated free list를 통해 개선할 수 있다고 생각되어진다.