System Programming Project 3

담당 교수 : 김영재 교수님

이름 : 송지훈

학번 : 20201595

1. 개발 목표

강의 시간에 배운 memory allocator의 구조를 직접 실습을 통해 구현해본다. 이후 구현된 memory allocator가 어느 정도의 utility와 throughput을 가지고 있는지 주어진 mdriver을 통해 확인해 본다.

2. Allocator Design

이번 과제는 강의 시간에 배운 Implict, explicit, segregated 중 explicit free list를 구현했다. Explicit free list란 allocate 된 memory를 제외한 free된 리스트만을 관리하는 방식이다. 따라서 Implicit한 방식으로 사용될 때와는 달리 저장될 때 다른 구조가 필요하다. 그 구조는 다음과 같다.

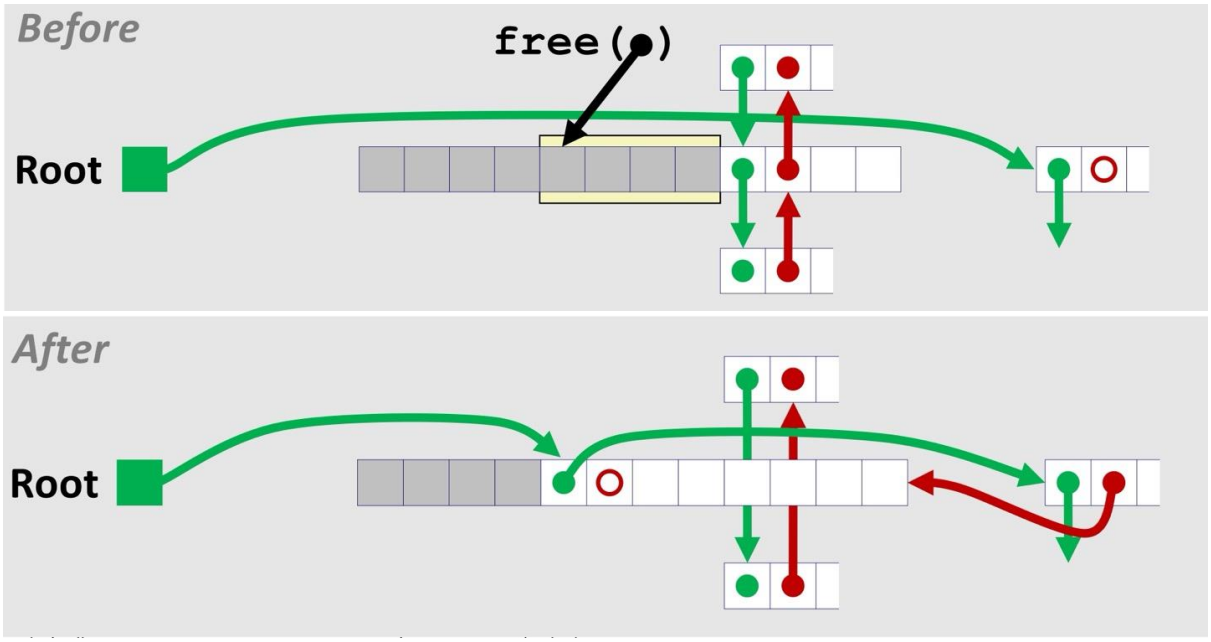
그림을 보면 알 수 있듯이 free list에는 SIZE와 할당 여부를 판단하는 a는 여전히 존재하지만 NEXT와 PREV가 새로 추가된 것을 확인할 수 있다. 여기서 NEXT는 다음 free block을 prev는 이전 free block을 가리키는 이중 연결리스트 구조로 되어있다. 이 때 HEADER를 포함한 FOOTER 또한 유지되어야 하는데 이는 만약 allocate 되어 있던 메모리가 free 될 때 바로 인접한 곳에 free 된 블록이 존재한다면 이를 합쳐 더 큰 요청이 들어왔을 때 대응할 수 있게 하기 위함이다. 구현한 allocator에서는 강의자료에서 설명된 것과 같이 Last In – First Out 방식을 채택했다. 이 방식을 구현하기 위해서는 다음 4가지 경우의 수에 대해 구현할 수 있어야 한다.

(1) 새로운 블록이 들어올 때

새로운 블록이 들어온다면 이는 맨 앞에 추가된다. 이 때 추가된 free block의 next는 기존에 root가 가리키고 있던 block을 가리키며 기존에 root가 가리키고 있던 free block의 prev는 새로 추가된 free block을 가리켜야 한다.

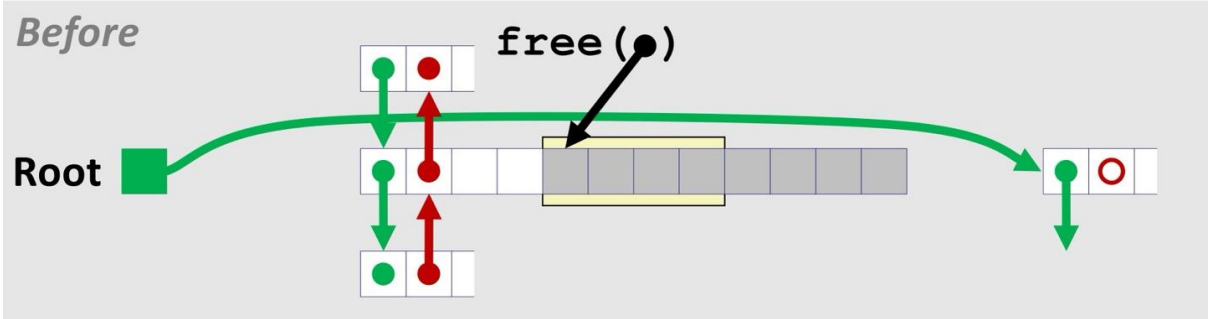
(2) 기존에 존재하던 블록에 앞에 이어서 들어올 때

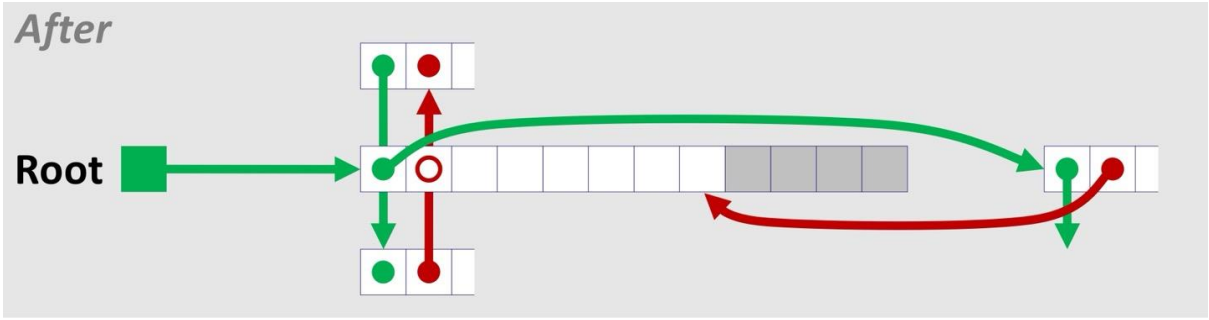
이는 위에서 설명했던 것과 마찬가지로 앞에 새로운 block을 root가 가리키며 이전 root가 가리키고 있던 block은 새로운 block을 가리킨다. 그리고 합쳐지기 전에 있던 block이 앞 뒤로 가리키던 block은 서로를 가리키게 바꾼다. 이를 그림을 나타내면 다음과 같다.



(3) 기조에 존재하던 블록 뒤에 이어서 들어올 때

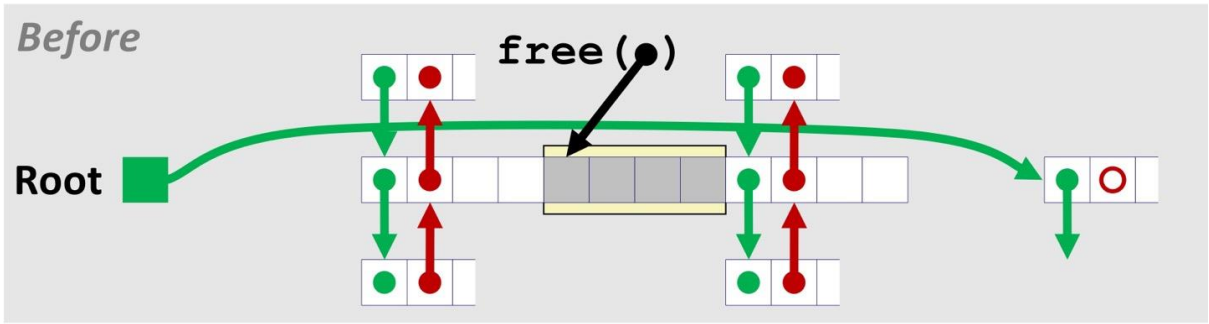
이 방식은 (2)번과 유사하다. 다만 뒤에 있는 block이 합쳐진다. 이에 대한 그림은 다음과 같다.

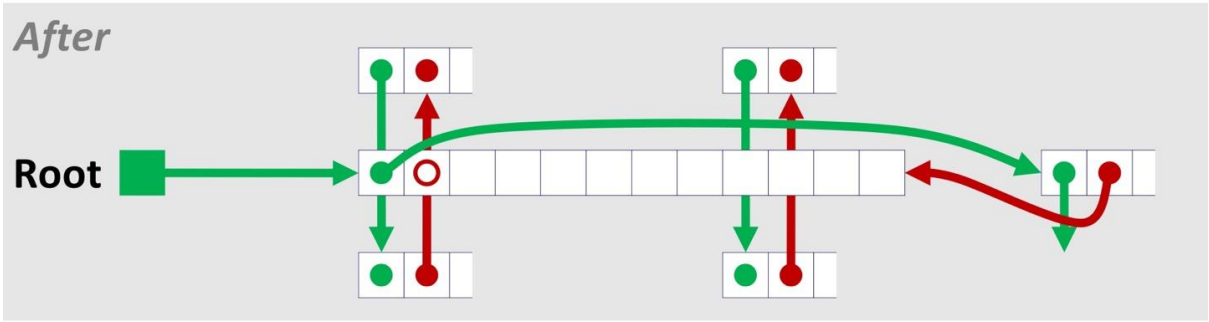




(4) 앞 뒤로 free 된 블록이 존재할 때

이는 (2)번과 (3)번에서 수행했던 것을 반복해서 수행하면 된다. 이를 그림으로 나타내면 다음과 같다.





위의 모든 경우에서 유의해야 할 것은 변경되거나 새로 생성된 free block은 항상 첫 번째로 들어와 root가 가리켜야 한다는 것이다.

3. **Newly Defined Things**

Explicit free list를 구현하기 위해 추가적으로 선언된 것은 크게 MACRO, 전역변수, 함수로 나눌 수 있다.

(1) MACRO

-MINSIZE 16

이는 할당하려는 block 사이즈의 최소 값을 나타내기 위해 선언된 MACRO이다. 입력된 size와 최소 block 크기를 비교하기 위해 선언했다.

-PREV_P(bp) (*(void **)(bp + WSIZE)

이는 free list를 관리하기 위해 선언했다. 위에서 설명했던 것과 마찬가지로 explicit free list의 각 block은 다음과 이전 free block을 가리키는 next와 prev가 존재한다. 이때 기본적으로 인자는 단일 포인터 이므로 해당하는 포인터를 더블 포인터로 변경 후 *를 통해 값을 받아 포인터의 형태로 유지시키는 역할을 한다. RPEV_P는 이전 block의 주소를 의미한다.

-NEXT_P(bp) (*(void **)(bp)

NEXT_P는 free block의 다음 free block을 나타낸다.

(2) 전역 변수

-static void root : free list의 시작 block을 가리키기 위해 선언된 전역 변수이다.

(3) 함수

-int mm_init()

초기의 heap을 생성하기 위한 함수이다. 기존에 선언되어 있던 heap_listp가 가리킨다. 초기에 생성되는 block은 각각 1 word 크기의 alignment padding, 프롤로그 header, 프롤로그 footer, 에필로그 header, next, prev 값을 가져 총 6 word size를 가

진다. 이후 heap_listp가 가리키는 곳은 root가 가리켜 next의 위치를 가리키게 한다. 초기의 next와 prev는 모두 null 값을 가진다.

-void *mm_malloc(size_t size)

Malloc의 기능을 수행하는 함수이다. Size에 해당하는 block을 할당한다. 이 때 할당되는 block의 사이즈는 최소 MINSIZE이고 doubly aligned 이므로 8 바이트 단위로 설정된다. 이 내부에서는 free block 중 현재 할당하고자 하는 block의 크기와 맞는 것이 있는지 확인하고 없다면 heap의 크기를 늘려 할당하도록 한다. 그리고 할당된 주소를 return 한다.

-void mm_free(void * bp)

Bp에 해당하는 block을 free하는 함수이다. 해당 block의 할당 여부를 free로 바꿔주고 coalesce 작업을 거치기 위해 coalesce 함수를 호출한다.

-void coalesce(void * bp)

Bp에 해당하는 block을 coalesce 하기 위한 함수이다. 위에서 설명했던 것처럼 총 4가지 경우에 대한 작업을 수행한다. Return 값을 변경된 bp를 가진다.

-void *mm_realloc(void *ptr, size_t size)

Ptr에 해당하는 block 을 size의 크기로 재할당하기 위한 함수이다. 기존의 realloc 함수와 동일하게 size가 0 이라면 free를 ptr이 NULL 이라면 malloc을 실행한다. 만약 이전 크기가 새로운 크기보다 크다면 재할당 작업을 거치지 않고 현재의 주소 그대로를 return한다. 만약 newsize가 더 크다면 두 가지 경우의 수로 나눠서 생각한다. Ptr의 바로 뒤 block이 free 상태라면 뒤 block의 크기를 확인후 그 합이 newsize보다 크다면 두 개를 합쳐 ptr을 리턴한다. 만약 할당된 블록이거나 사이즈가 작다면 새로운 메모리를 할당해준다.

-static void *extend_heap(size_t words)

Heap을 확장하기 위한 함수이다. Words 에 대항하는 값 만큼 mem_sbrk 함수를 호출해 확장하고 coalesce 함수를 호출해 free list에 추가한다.

-static void place(void *bp, size_t asize)

찾은 free block에서 할당하고자 하는 블록만큼 할당하는 함수이다. 만약 현재 free block과 할당하고자 하는 크기가 MINSIZE보다 크다면 할당하고자 하는 만큼 할당하고 나머지 뒤의 부분을 다시 free list로 돌려놓는다. 만약 아니라면 해당하는 block을 전부 할당한다.

-static void * find_fit(size_t asize)

First fit 방식으로 조건에 맞는 free block을 찾는 함수이다. Root가 가리키는 가장 처음부터 시작해 다음으로 움직이며 asize와 같거나 큰 block을 찾아 이를 return한다. 만약 찾지 못하면 NULL을 return한다.

-static void addfreelist(void * bp)

Bp를 freelist에 추가하기 위한 함수이다. 맨 앞에 삽입하는 LIFO 방식이므로 해당하는 기능을 수행하게 한다.

-static void removefreelist(void * bp)

Bp에 해당하는 block을 freelist에서 제거하기 위한 함수이다. 이는 총 3가지로 생각할 수 있다. 만약 첫 번째 블록일 경우 bp가 가리키고 있던 다음을 root가 가리키고 그 값의 전 포인터를 NULL로 변경한다. 만약 bp가 마지막 블록이면 bp의 전 블록이 가리키고 있는 주소를 NULL로 변경한다. 만약 중간 값이라면 bp의 전 블록의 다음 주소를 bp의 다음 주소로, bp의 다음 주소의 전 주소를 bp의 전 주소로 대입한다.

-size_t checksize(size_t size)

할당하고자 하는 크기인 size를 받고 이를 doubly aligned의 크기에 맞게 변환해 return하는함수이다. 만약 size가 MINSIZE 보다 작다면 MINSIZE를 return하고 아니라면 doubly aligned의 크기에 맞게 계산해 그 값을 return한다.

4. **Result**

구현된 코드를 바탕으로 mdirver를 실행했을 때의 결과는 다음과 같다.

```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   89%    5694  0.001372  4149
 1       yes   92%    5848  0.000872  6704
 2       yes   94%    6648  0.001762  3772
 3       yes   96%    5380  0.000517 10402
 4       yes   99%   14400  0.001083 13296
 5       yes   88%    4800  0.002719  1765
 6       yes   85%    4800  0.003949  1216
 7       yes   55%   12000  0.029408   408
 8       yes   51%   24000  0.015519  1547
 9       yes   42%   14401  0.002663  5407
10       yes   53%   14401  0.000196 73474
Total          77%  112372  0.060062  1871

Perf index = 46 (util) + 40 (thru) = 86/100
```

0번부터 6번 까지의 utility는 좋은 편이지만 7번부터 10번까지의 utility는 좋지 못하다. 이는 내가 작성한 memory allocator가 효율적인 할당을 제공하고 있지 못하다는 것이다. 특히 realloc에서 매우 저조한 것을 확인할 수 있다. 이는 realloc을 진행할 때 place와 동일한 기능을 할 수 있도록 코드를 작성하였지만 dubug를 하던 중 그 원인을 발견하지 못해 utilization에 실패했기 때문이다. 만약 시간이 더 주어진다면 utilization을 더 높인 버전을 작성해보고 싶다.