Multicore Programming Project 3

담당 교수 : 박성용

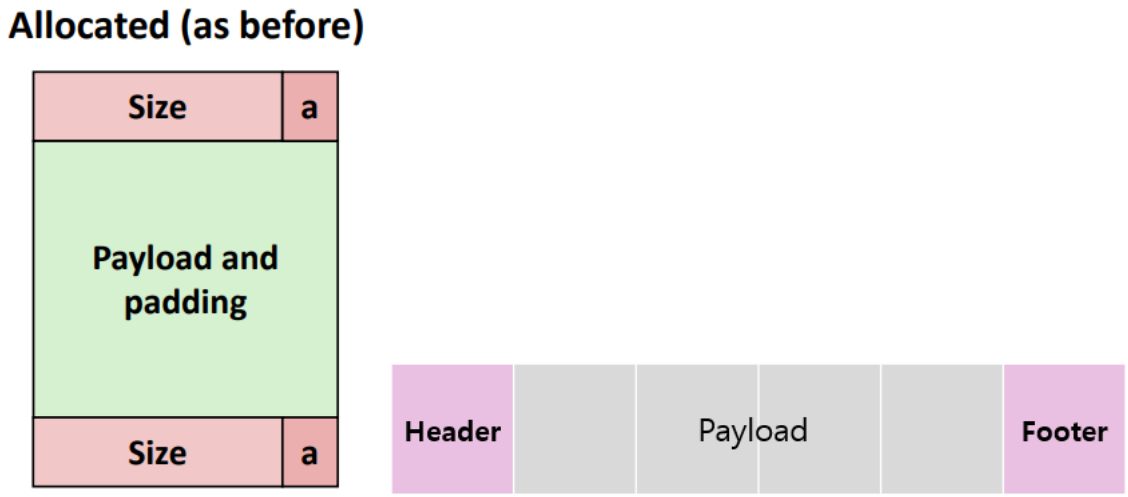
이름 : 안희원

학번 : 20211550

1. Design of my allocator

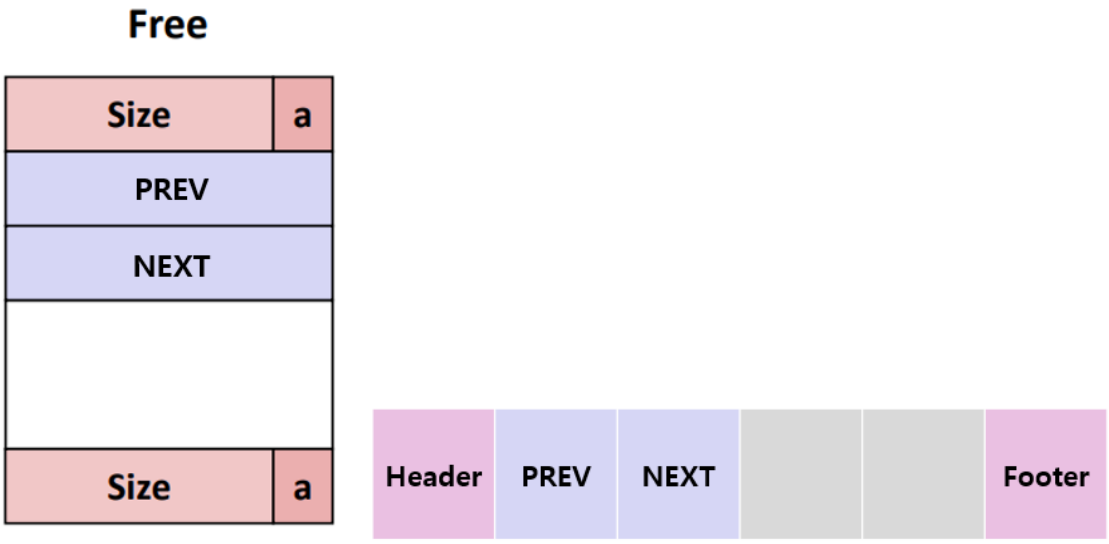
수업시간에 배운 3가지 방법(Implicit, Explicit, Segregated) 중 free block만 탐색하는 Explicit list 방법으로 Dynamic memory allocator를 구현하였다. Explicit list를 구현하기 위해 아래와 같은 block 구조를 사용하였다.

- **Allocated block**



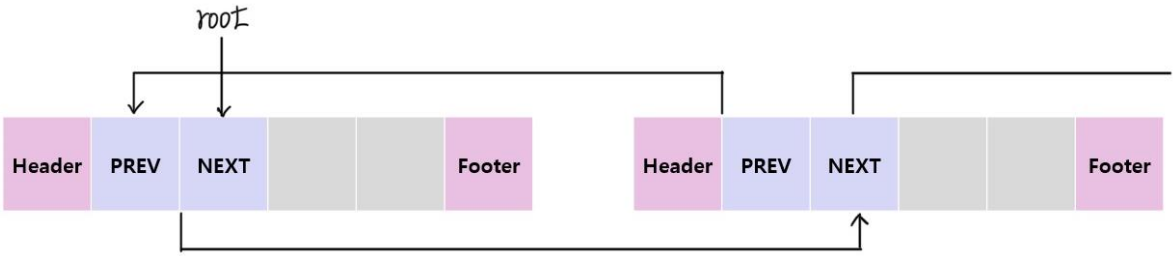
Allocated block의 구조는 위와 같다. Block의 1칸은 word(4byte) 크기로 가정한다. 오른쪽 그림처럼 Block의 첫 번째 word는 header로, 마지막 word는 footer로 사용된다. Header와 footer는 free block들을 coalescing할 때 사용된다. 왼쪽 그림처럼 header와 footer에는 header와 footer를 포함한 전체 block의 size가 저장되고, 구현하고자 하는 allocator에서는 8byte alignment를 사용할 것이기 때문에 header의 하위 3비트는 사용되지 않는다. 따라서 하위 1bit를 allocated 여부를 나타내는 state bit로 사용한다. state bit가 1이면 해당 block이 allocated 상태라는 의미이고 0이면 free 상태라는 의미이다. 이 block은 allocated block이므로 state bit가 1일 것이다.

- **Free block**



구현한 free block의 구조는 위와 같다. Block 1칸의 크기와 header, footer에 대한 구현은 위의 allocated block과 같다. allocated block과 차이점은, state bit가 0이고, 할당할 공간을 탐색할 때 free block들만 탐색하기 위해 free block list를 만들기 위해 free block에는 2개의 pointer를 추가하였다. 각 pointer는 하나 당 1word를 차지한다. PREV pointer는 해당 free block 앞에 있는 free block을 가리킨다. NEXT pointer는 해당 free block 뒤에 있는 free block을 가리킨다. 부분적인 구조를 보면 아래와 같다.

- **Free block list 구조**



위의 그림을 보면 첫 번째 block의 NEXT pointer가 두 번째 block을 가리키고, 두 번째 block의 PREV pointer는 첫 번째 block을 가리키는 것을 알 수 있다. 그리고 두 번째 block의 NEXT pointer는 또다른 세 번째 free block을 가리키는 구조가 될 것이다. 또한, 만약 첫 번째 block이 실제로 전체 heap에서 첫 번째 free block이라면 이 block은 root pointer가 가리키고 있을 것이다. root는 항상 첫 번째 free block을 가리킨다. Free block list는 항상 list 맨 앞에 새로운 block을 insert하며, 처음(root)부터

탐색하는 LIFO 구조로 구현하였다.

2. Description of subroutines, structs, any global variables

2.1 추가한 매크로 및 global 변수

- #define FREE_PREV(bp) (*(char)(bp))**

- #define FREE_NEXT(bp) (*(char)(bp + WSIZE))**

Free block들의 포인터 연산을 간단하게 구현하기 위한 매크로를 2개 추가하였다. bp는 구현한 코드에서 주로 block의 header 다음 word를 가리키기 때문에 bp가 가리키는 곳 자체를 PREV 포인터로 사용하므로 FREE_PREV() 매크로를 다음과 같이 작성하였다. 비슷한 방식으로 FREE_NEXT()는 PREV 포인터 다음 word를 NEXT 포인터로 사용하였기 때문에 PREV 포인터에 1word size를 매크로로 저장한 WSIZE를 더해주는 방식으로 구현하였다.

이외의 매크로는 전부 교재에 구현되어 있는 것을 그대로 사용하였으므로 설명을 생략하겠다.

- static char *root = 0; //free list의 root

free block list의 첫 번째 block을 가리킬 포인터이다.

- static char *heap_listp = 0;

전체 heap list에서 첫 번째 block인 prologue block을 가리키는 포인터이다.

2.2 함수

- static void deletion(void *bp)

Free block list에서 free block을 삭제하는 함수이다. 여기서 bp는 삭제하고자 하는 block을 가리키는 포인터이다. 이 함수에는 2가지 case가 존재한다. 첫째는 삭제하고자 하는 block이 list의 첫 번째 free block이었을 경우인데, 이 때는 root를 삭제할 block 다음 block의 NEXT 포인터와 연결한다. 또다른 case는 list의 첫 번째 free block이 아닌 경우다. 이 경우에는 삭제할 block의 이전 block의 NEXT 포인터에 삭제할 block의 NEXT 포인터(삭제할 block의 다음 block)를 연결시켜준다.

이후, 삭제할 block 다음 block의 PREV 포인터에 삭제할 block의 PREV 포인터를 연결해 다음 block과 이전 block을 연결시켜준다.

- static void insertion(void *bp)

Free block list의 맨 앞에 block을 삽입하는 함수다. 여기서 parameter인 bp는 삽입하고자 하는 block을 가리키는 포인터이다. 총 4가지 단계를 거쳐 연결을 하게 된다. 첫 번째로는 삽입하고자 하는 block의 NEXT 포인터가 현재 root가 가리키고 있는 block(현재 list의 첫 번째 block)을 가리키도록 한다. 두 번째로는 현재 root가 가리키고 있는 block의 PREV 포인터가 삽입하고자 하는 block을 가리키도록 한다. 세 번째로는 삽입하고자 하는 block의 PREV 포인터를 NULL값으로 하여 첫 번째 block임을 나타낸다. 마지막으로 root가 bp를 가리키도록 하면 된다.

- static void* coalesce(void *bp)

Block을 free할 때, free하고자 하는 block의 앞 뒤에 free block이 있는지 확인하고 만약 free block이 존재한다면 free block끼리 합쳐주는 함수이다. bp는 free하고자 하는 block의 포인터다. 이 함수는 총 4가지 case로 구성된다. (1)allocated block + bp + allocated block인 경우, (2) allocated block + bp + free block, (3) free block + bp + allocated block, (4) free block + bp + free block인 경우이다. 각 case에 따라 header와 footer를 바꿔주는 순서만 다르고, 전체적인 구성은 다음과 같다. 합쳐진 block의 size를 업데이트해주고, 양 옆의 free block을 deletion()을 이용해 free block list에서 삭제한다. 이후 header와 footer를 수정해 block을 합쳐주는 것이다.

- static void* extend_heap(size_t words)

해당 함수는 heap list에 할당할 size 만큼의 공간이 없을 때 sbrk()를 통해 heap list의 크기를 확장해 준다. 확장할 size를 받아 8byte-alignment에 맞게 값을 조절해준다. 이후 sbrk()를 통해 heap의 크기를 늘리고, 새롭게 확장한 size로 bp의 header와 footer 정보를 바꿔주고, epilogue block 또한 새롭게 설정해준다.

- int mm_init()

초기 heap 구조를 구성하는 함수이다. 초기의 heap 구성은 padding block 하나, header와 footer만으로 이루어진 prologue block 하나, header 하나로 이루어진 Epilogue block 하나로 구성된다. Prologue block은 heap list의 시작을 알려주며, Epilogue block은 heap list의 끝을 알려준다. 이때, heap_listp는 heap list의 처음

을 가리키는 포인터인데 이 함수에서 root를 heap_listp + (2*WSIZE)로 설정해준다. 이 뜻은, prologue block을 free block list의 root로 사용하겠다는 것이다.

- static void* find_fit(size_t asize)

Free block list를 탐색하며 할당하고자 하는 size만큼의 free block을 찾는 함수이다. root부터 free block의 NEXT 포인터를 따라가면서 검사한다. 맞는 block을 찾았다면 그 block의 포인터를 return하고, 찾지 못했다면 null값을 return한다.

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

parameter부터 설명하자면, bp 포인터가 가리키는 block에 asize만큼 할당을 하겠다는 것이다. 해당 함수는 bp 포인터가 가리키는 block의 size에서 asize만큼 할당하고 남은 size가 2*DSIZE 이상 남으면 남은 free 부분을 split하여 memory utilization을 높인다.

- void *mm_malloc(size_t size)

Parameter인 size는 user가 할당하기를 원하는 size를 말한다. size를 받으면 mm_malloc()에서는 실제로 할당할 block의 size를 alignment에 맞게 계산한다. 만약 payload size가 DSIZE보다 작으면 block을 최소 size인 2*DSIZE로 맞춰준다. 이후 find_fit()을 호출해 할당할 block의 위치를 찾고, place()를 통해 split을 해준다. 만약 find_fit()에서 할당할 size만큼의 block을 찾지 못했다면 extend_heap()을 호출해 heap list의 size를 확장해준다.

- void mm_free(void* bp)

bp가 가리키는 block의 header와 footer의 state bit를 0으로 바꿔 free 상태임을 표시하고 coalesce()를 호출해 앞, 뒤 free block과 합쳐서 free block list에 추가해준다.

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

ptr 포인터가 가리키는 block을 size만큼 realloc하는 함수이다. 가장 먼저 예외처리를 해주는데, ptr이 NULL인 경우에는 기존 malloc과 동일하게 실행하므로 mm_malloc(size)를 return한다. Size가 0인 경우는 기존의 ptr가 가리키는 block의 메모리를 삭제하라는 것이므로 mm_free(ptr)을 호출해 block을 free 시켜준다. 추가적으로 size가 음수거나 ptr의 state bit가 0인 경우(할당된 block이 아닌 경우)는 바로 null값을 return하도록 하였다. 이 함수에는 ptr의 block의 size인 psize

변수와 새롭게 할당할 block의 size인 nsize 변수가 존재한다. 만약 nsize가 psize 보다 작은 경우에는 기존 block에서 size를 늘릴 필요가 없으므로 그대로 ptr을 return한다. 이제 기존 block에서 size를 늘려야 하는 경우만 남았다. 확장을 하는 방법을 2가지로 나눠서 생각하였는데, 첫째는 기존 block 다음 block과 바로 연결하는 것이고, 나머지는 nsize만큼의 새로운 block을 다시 찾아서 새롭게 할당하는 것이다. 첫번째 방법의 경우에는 coalesce()의 (2)번 case와 거의 동일하다. Header와 footer의 정보를 업데이트해줄 때 state bit만 1로 설정해주면 두 block을 합칠 수 있다. 두 번째 방법의 경우 mm_malloc(nsize)를 호출하여 새롭게 block을 할당해주고, 만약 mm_malloc()이 실패하였을 경우에는 null값을 return한다. 이후, 기존의 data를 복사해야 하므로 memcpy()를 사용하였고 아예 새로운 block을 할당했으므로 mm_free(ptr)을 통해 ptr을 아예 free시켜준다.

3. 구현 결과

```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   89%    5694  0.000474 12018
 1       yes   92%    5848  0.000267 21894
 2       yes   94%    6648  0.000530 12536
 3       yes   96%    5380  0.000380 14150
 4       yes   99%   14400  0.000274 52555
 5       yes   88%    4800  0.000615  7805
 6       yes   85%    4800  0.000665  7217
 7       yes   55%   12000  0.004202  2856
 8       yes   51%   24000  0.004340  5531
 9       yes   97%   14401  0.000234 61438
10       yes   46%   14401  0.000226 63721
Total          81%  112372  0.012208  9205

Perf index = 49 (util) + 40 (thru) = 89/100
cse20211550@cspro:~/sp3/prj3-malloc$
```

./mdriver -V 명령어를 통해 확인한 구현 결과는 위와 같다.