Multicore Programming Project 3

담당 교수 : 박성용

이름 : 남기찬

학번 : 20211180

1. 개발 목표

이번 Dynamic memory allocator 프로젝트에서는 C언어에서 동적 할당을 할 때 썼던 malloc, free, realloc 함수들을 자신만의 함수로 직접 구현한다. 먼저 mm_malloc() 함수는 요청된 메모리의 크기를 기반으로 실제로 할당해야 하는 크기를 계산하고, explicit list에서 적절한 크기의 block을 찾아 return 값으로 반환한다. 다음, mm_free() 함수는 이전에 할당한 block을 해제하고, 인접한 free block이 있다면 병합한다. 마지막으로 mm_realloc() 함수는 앞의 mm_malloc(), mm_free()를 이용하여 이전에 할당한 block의 크기를 조절한다.

2. 개발 범위 및 내용

A. Global variables

- void* heap_list: 메모리를 할당하고 해제해주는 heap의 시작 주소이다. 처음에 빈 heap 공간을 만들고 header, footer를 넣을 때 사용된다.
- void* free_p: Explicit free list의 첫 번째 block을 가리키는 포인터이다. Explicit free list의 block들을 search하거나, 특정 block을 삽입 또는 제거할 때 사용된다.

B. Subroutines

- void FreeInsert(void* bp): 포인터 bp가 가리키는 메모리 block을 explicit free list의 가장 앞에 삽입한다. 매크로 'FBLNEXT_P'와 'FBLPREV_P'를 사용하여 기존 free_p의 앞 link가 bp에 해당되는 block을 가리키도록, bp의 뒤 link는 free_p에 해당되는 block을 가리키도록 하고, bp를 새로운 free_p로 지정한다.
- void FreeDelete(void* bp): 포인터 bp가 가리키는 메모리 block을 explicit free list에서 제거한다. 만약 이 block이 explicit free list의 첫 번째 block이라면 매크로를 사용하여 bp의 뒤 link가 가리키는 block의 포인터를 새로운 free_p로 지정한다. 그리고 이 block이 list의 중간에 있다면, 매크로를 사용하여 bp의 앞 link가 가리키는 block과 뒤 link가 가리키는 block의 뒤/앞 link가 서로를 가리키도록 하여 bp가 가리키는 block이 list에서 완전히 빠져나오도록 한다.
- static void* coalesce(void* bp): 포인터 bp가 가리키는 block이 free되었을 때 그 앞뒤의 block 중에 free 상태인 block이 있다면 서로 합친다. 할당되었는

지 여부는 매크로 'GET_ALLOC'을 통해 알 수 있다. 첫 번째로, 앞뒤의 block이 모두 할당된 경우에는 coalesce가 필요하지 않다. 두 번째, 앞 block은 할당되어있고 뒤 block이 free인 경우에는 FreeDelete() 함수를 이용하여 뒤 block을 explicit free list에서 제거하고, 합친 block의 크기는 두 free block의 크기를 더한 값으로 갱신한다. 또 header와 footer도 갱신한다. 세 번째, 앞 block이 free이고 뒤 block이 할당된 경우에는 FreeDelete()로 앞 block을 explicit free list에서 제거하고, 앞에서와 같이 합친 block의 크기와 header, footer를 갱신한다. 마지막으로, 앞뒤의 block이 모두 free인 경우에는 FreeDelete()로 앞, 뒤 block을 모두 explicit free list에서 제거하고, 합친 block의 크기와 header, footer를 갱신한다. 이때 합친 block의 크기는 free block 3개의 크기를 모두 더한 값이다. 이 과정이 끝나면, FreeInsert() 함수를 이용해서 최종적으로 합쳐진 block을 새롭게 explicit free list에 삽입한다. 이때 앞 block이 free인 경우에는 합친 block의 포인터를 bp의 앞 link 포인터로 설정해줘야 한다.

- static void* extend_heap(size_t words): block을 할당할 때 heap 공간이 부족할 경우에 호출된다. mem_sbrk() 함수를 이용하여 heap을 words x 'WSIZE'만큼 확장시키고, 이 공간에 header, footer, epilogue header를 넣는다. 마지막으로, 인접한 free block이 있다면 합쳐지도록 coalesce() 함수를 호출한다.

- static void* find_fit(size_t asize): 크기가 asize인 메모리 block을 할당하기 위해 매크로 'FBLNEXT_P'를 사용하여 explicit free list에서 크기가 asize와 같거나 더 큰 free block을 search한다. First-fit search 방법을 사용하여 가장 먼저 발견되는 적절한 크기의 block을 반환한다.

- static void place(void* bp, size_t asize): 포인터 bp가 가리키는 block에 크기가 asize인 메모리를 할당한다. 그리고 FreeDelete()를 이용하여 bp가 가리키는 block을 explicit free list에서 제거한다. 만약 할당하고 남은 공간이 2 * 'DSIZE'보다 크다면, 즉 충분하다면, 남은 공간으로 분할 block을 생성하여 header와 footer를 갱신하고 FreeInsert()를 통해 다시 explicit free list에 삽입한다. 이렇게 하면 fragmentation을 줄여서 utilization을 높일 수 있다.

C. Dynamic memory allocator functions

- int mm_init(void): mm_malloc(), mm_free(), mm_realloc()을 호출하기 전에 메모리를 할당하는데 필요한 heap 공간을 만든다. mem_sbrk()를 이용하여 초기

heap 공간을 할당하고, prologue header와 footer, epilogue header를 넣어서 초기 block을 설정한다. 그리고 extend_heap() 함수를 이용하여 heap 공간을 확장한다. 즉 mm_init()은 메모리를 할당하기 전에 heap을 초기화하는 역할을 한다.

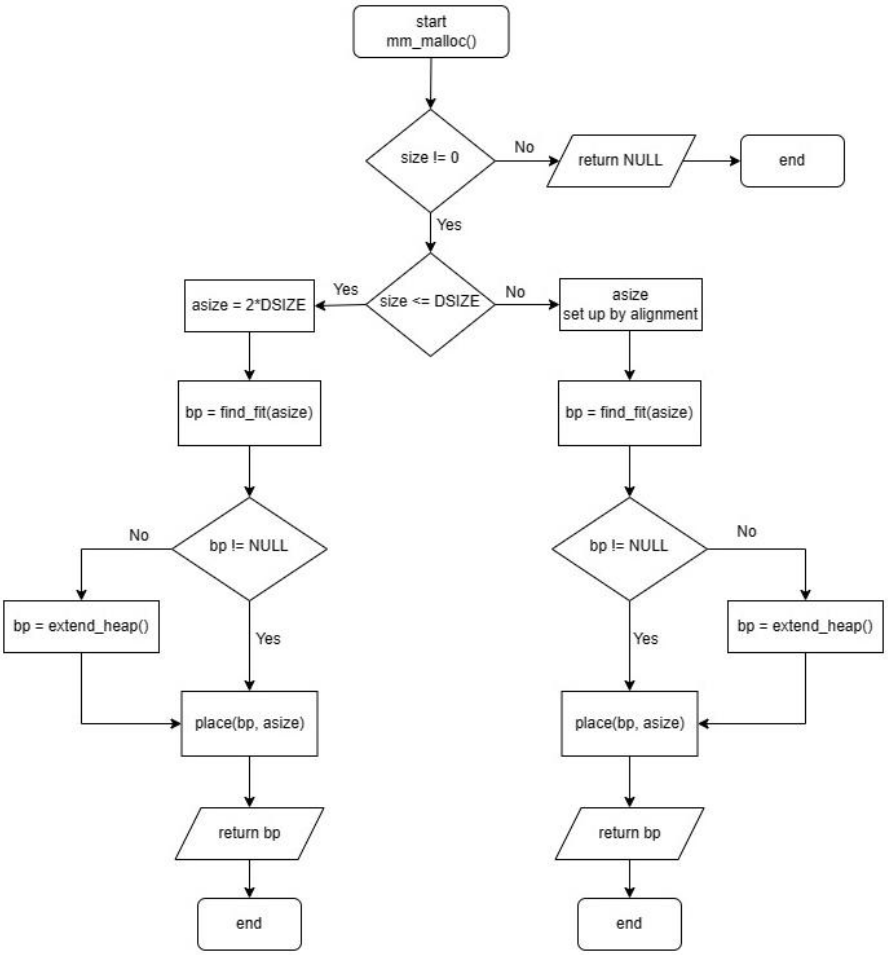
- void* mm_malloc(size_t size): 크기가 size인 메모리 block을 할당한다. 먼저 block을 할당할 때는 alignment 조건을 지켜야 하므로, size보다 더 크거나 같으면서 alignment(8)의 배수인 적절한 크기 asize를 구한다. 이때 size가 최소 block 크기인 16보다 작은 경우에는 asize를 16으로 설정한다. 다음 find_fit() 함수를 이용하여 explicit free list에서 크기가 asize보다 더 크거나 같은 free block을 찾는다. 그리고 이 free block의 포인터와 요청된 메모리의 크기를 place() 함수에 넣어 free block에 메모리를 할당하고, block의 남은 공간은 분할하여 다시 explicit free list에 넣는다. 만약 find_fit()을 했을 때 검색된 free block이 없다면, extend_heap() 함수를 통해 확장된 heap공간의 새로운 block에 메모리를 할당한다.

- void mm_free(void* ptr): 포인터 ptr이 가리키는 할당된 메모리 block을 해제한다. Header를 통해 이 block의 크기를 알아내고, allocate 부분은 0으로 바꿔 header와 footer를 갱신한다. 그리고 앞뒤로 인접한 free block이 있는 경우를 생각하여, coalesce() 함수를 통해 free block들을 하나의 block으로 병합하고, explicit free list에 삽입한다.

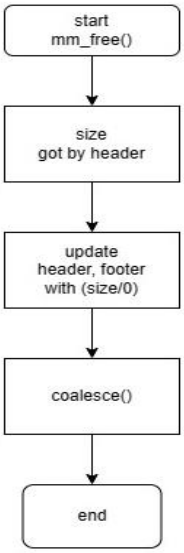
- void* mm_realloc(void* ptr, size_t size): 포인터 ptr이 가리키는 메모리 block의 크기를 size로 변경한다. 먼저 새로운 크기(size)의 메모리를 재할당하기 위해 mm_malloc()을 호출하고, memcpy() 함수를 통해 기존의 block에서 재할당된 메모리 block으로 size만큼 데이터를 복사한다. 이때 요청하는 크기 size가 기존 메모리 block의 크기보다 더 크다면 기존 block 크기만큼만 복사한다. memcpy()가 끝나면 기존의 block은 mm_free()를 호출하여 해제한다. 마지막으로 재할당된 block의 포인터를 반환하면 메모리의 크기가 size로 바뀐 결과가 나온다.

D. Flow chart

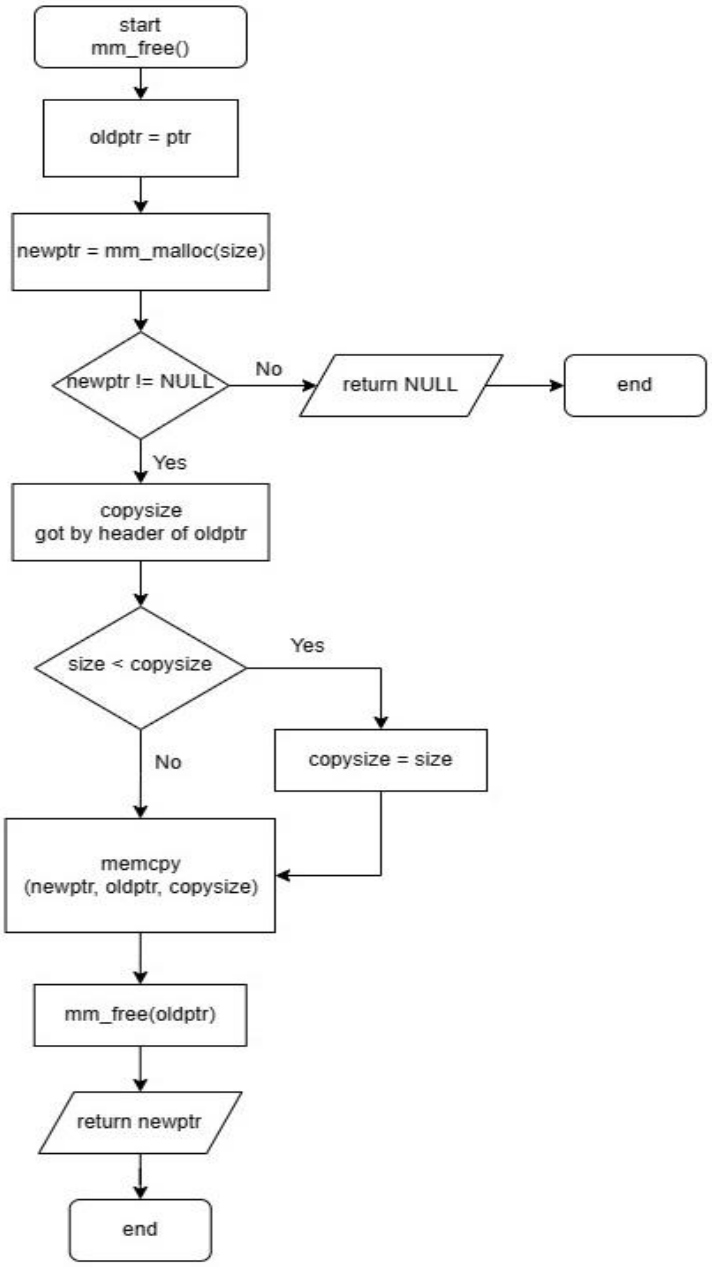
1) mm_malloc()



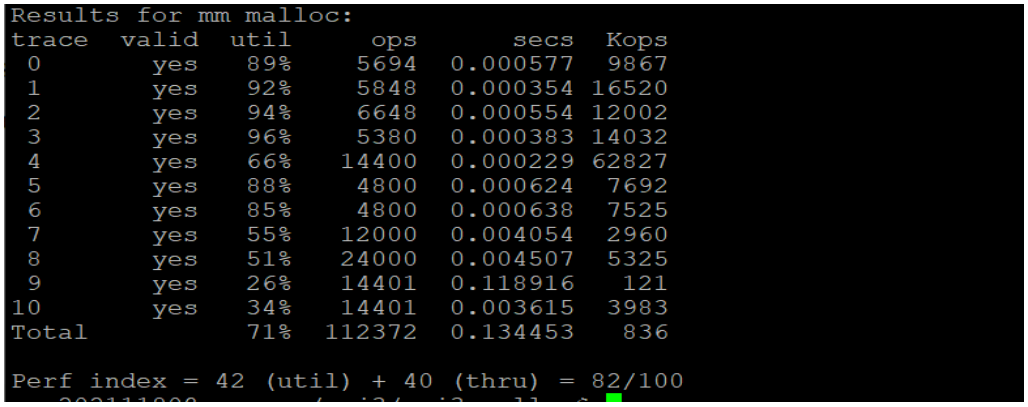
2) mm_free()



3) mm_realloc()


start
mm_free()
oldptr = ptr
newptr = mm_malloc(size)
newptr != NULL
No
return NULL
end
Yes
copysize
got by header of oldptr
size < copysize
Yes
No
copysize = size
memcpy
(newptr, oldptr, copysize)
mm_free(oldptr)
return newptr
end

3. 구현 결과



```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   89%    5694  0.000577  9867
 1       yes   92%    5848  0.000354 16520
 2       yes   94%    6648  0.000554 12002
 3       yes   96%    5380  0.000383 14032
 4       yes   66%   14400  0.000229 62827
 5       yes   88%    4800  0.000624  7692
 6       yes   85%    4800  0.000638  7525
 7       yes   55%   12000  0.004054  2960
 8       yes   51%   24000  0.004507  5325
 9       yes   26%   14401  0.118916   121
10       yes   34%   14401  0.003615  3983
Total          71%  112372  0.134453   836

Perf index = 42 (util) + 40 (thru) = 82/100
```

- 모든 test case들에 대해 성공했으나, 뒤의 case로 갈수록 utilization이 떨어지고 시간도 오래 걸리는 결과를 보였다. Performance 점수는 peak memory utilization은 60점 만점에 42점이 나왔고, throughput은 40점 만점에 40점이 나왔다. 총점은 100점 만점에 82점이다.

- Explicit free list보다 segregated free list를 이용하여 mm_malloc() 등의 함수를 구현했을 때 총점이 더 높게 나올 것으로 예상된다. Segregated free list를 이용하면 코드가 더 복잡해지므로 throughput은 조금 줄어들겠지만, segregated free list에서의 first-fit search 방법은 implicit list에서 best-fit search 방법과 비슷하므로 utilization이 매우 높을 것이기 때문이다. 그리고 performance를 측정할 때의 반영 비율은 utilization : throughput = 6 : 4로 utilization의 비율이 더 높으므로, segregated free list를 이용했을 때 더 높은 점수가 나올 것이다.

4. 피드백

처음에는 Segregated free list를 이용하여 dynamic memory allocator를 구현하려 했으나, free block이 들어갈 수 있는 적절한 크기의 free list를 찾아서 free block을 넣은 후에, 10개가 넘는 free list들의 link 관계를 정리하는 과정에서 global array 없이 구현해야 하는 것에 어려움을 느꼈다. 한 list의 앞 link와 뒤 link가 정확하지 않은 list를 가리키는지 계속 segmentation fault가 나왔다. 과제를 제출한 뒤에도 segregated free list로 구현했을 때 performance 점수 차이가 얼마나 나는지 알아보기 위해 계속해서 구현해볼 것이다.