System Programming Project 3

담당 교수 : 김영재

이름 : 송재현

학번 : 20190554

1. 개발 목표

해당 프로젝트에서는 효율적인 방식으로 가장 c언어 library의 malloc, free, realloc과 유사한 성능을 내는 mm_malloc, mm_free, mm_realloc을 구현한다. 본 프로젝트에서는 explicit free list를 이용하여 malloc, free, realloc을 구현하였다.

2. 개발 범위 및 내용

A. 개발 범위

- 아래 항목을 구현했을 때의 결과를 간략히 서술

explicit free list의 경우 free block들만으로 리스트를 유지하기 때문에 메모리가 많이 찼을 때

implicit free list 방식보다 훨씬 빠르다는 장점이 있다. Implicit의 경우 탐색 속도가 전체 블록의 수에 비례하지만 explicit의 경우 free 블록의 수에 비례하기 때문이다. 또한 allocated block은 next포인터와 prev포인터가 필요 없으므로 payload를 저장하는 곳에 이를 저장하면 되므로 별다른 문제가 생기지 않는다. 그러나 implicit에 비해 추가적인 공간을 요구하므로 internal fragmentation이 더 생긴다고 볼 수 있다.

B. 개발 내용

- 아래 항목의 내용만 서술

- **Task1 (mm_init)**

mm_init의 경우 기존의 implicit free list에서의 init과 동일하다. Initial heap을 만든 후에 prologue header, prologue footer, epilogue header에 대한 공간을 할당해준다.

- **Task2 (mm_malloc)**

mm_malloc의 경우 기존의 implicit free list와 동일하다고 볼 수 있다. 그러나 다른 점이 하나 있는데 find_fit함수 부분이 다르다. Implicit free list에서는 모든 블록을 추적하는 것이었다면 explicit free list에서는 free인 블록만 살피기 때문에 그 부분에서 다르다.

- **Task3 (mm_free)**

mm_free의 경우에도 mm_malloc과 마찬가지로 기존의 implicit free list와 동일하다. mm_free에서 coalesce함수가 있는데 이 부분만 기존의 것과 다르다. Implicit free list에서의 coalesce와 동일하게 case1~4를 앞, 뒤 블록이 free인지의 여부에 따라 나눈 후 그에 맞게 free 블록들을 join 시켜준다

- **Task4 (mm_realloc)**

mm_realloc의 경우 원하는 사이즈에서 HEADER와 FOOTER에 해당하는 추가 8바이트를 size에 더 추가해주고 만약 뒤에 블록이 free이고 현재 블록과 뒤에 블록을 합쳤을 때 길이가 위에서 구한 size+8보다 작거나 같으면 현재 위치에서 realloc해준다. 그렇지 않으면 새로운 size에 맞는 곳을 찾아서 malloc해주고 기존의 block은 free해준다.

C. 개발 방법

위의 함수들을 구현하기 위해 추가적인 static 함수들을 선언하여서 위 함수들을 구현하였다. 우선 insert_node와 delete_node를 통해서 free list를 관리해주었다. coalesce를 통해서 free하거나 heap의 크기를 늘릴 때 원래 있던 free 블록과 새롭게 free로 생긴 블록들을 join 시켜준다. incr_heap으로 mem_sbrk로 힙의 크기를 늘림과 더불어 HEADER와 FOOTER에 해당하는 정보를 삽입해준다. alloc_blk로 malloc에서 find_fit을 통해 allocate할 위치를 찾으면 그 곳에 allocate해주고 allocate 한 후에 split된 block의 크기가 16바이트보다 크면(최소 16바이트 필요) 이를 free list에 추가해준다. 마지막으로 find_fit은 free list의 앞에서부터 크기가 알맞은 block을 골라서 return 해준다.

3. 구현 결과

구현 후 mdrive를 통해 확인한 결과 코드에 문제가 없는 것으로 확인된다. heap의 크기를 늘릴 때 chunksize만큼 늘리도록 코드를 작성하였는데 chunksize를 2048, 4096, 8192로 했을 때 각각 performance가 88/100, 88/100, 89/100으로 나온 것을 확인할 수 있었다. 따라서 CHUNKSIZE는 8192로 정했다. util의 경우 49가 나왔으며 thru의 경우 40이 나와 총 89의 performance가 나왔는데 코드로 작성하진 않았으나 implicit으로 실험해보았을 때에는 50~60 사이의 performance가 나왔고 이 때 util의 값을 어느정도 높게 나왔으나 thru의 값이 낮게 나왔다. 이를 통해 explicit free list가 implicit free list보다 더 효율적인 방법임을 알 수 있다. 또한 segregated free list의 경우에도 util은 더 높게 나올 것으로 예상되나 segregated free list를 다루는데 필요한 메모리가 있을 것이므로 util의 경우는 explicit보다 낮게 나올 것으로 예상된다.

4. mm_check

mm_check를 통해 heap consistency check를 진행하였다. mm_check함수를 통해서 다음의 네 가지의 heap_consistency에 대해서 확인할 수 있도록 작성하였다. 우선 ptr을 이용해 free list를 순회하면서 ptr의 IS_ALLOC값이 0이 아닌 값이 나오면 free가 아니므로 1을 return 하도록 하였다. 또한 ptr의 prev_ptr의 footer와 ptr의 header 값의 차이가 4이고 prev_ptr이 NULL이 아니면 이는 free list에서 메모리상 인접한 두 free list가 coalescing이 일어나지 않은 것을 의미하므로 1을 return 하도록 하였다. 또한 root_free_list가 아닌 이상 모두 prev_ptr이 존재하므로 root_free_list가 아닌데 prev_ptr이 NULL이면 pointer가 valid free block을 point하지 않는 것으로 간주하여 1을 return 하였다. 마지막으로 mem_heap_lo와 mem_heap_hi를 이용해 ptr의 값이 heap의 가장 큰 값보다 크거나 가장 작은 값보다 작으면 valid한 free block이 아니라고 간주하여 1을 Return 하였다.

5. Explicit free list의 구조

본 프로젝트를 explicit free list로 구현하였는데 explicit free list를 LIFO policy를 따르도록 구현하였다. Explicit free list는 free block들을 이용해 list를 유지하는 것으로 implicit free list가 모든 블록을 사용하는 것과는 다르다. 따라서 boundary tag가 필요하므로 header와 footer가 모두 필요하다. 따라서 2 extra words가 각 block마다 필요하여 internal fragmenration이 늘어난다는 단점이 있으나 implicit free list에 비해서 memory가 거의 가득 찼을 경우 훨씬 빠르다는 장점이 있다. 대체로 implicit free list와 함수의 구조는 비슷하다고 할 수 있으나 alloc_blk함수와 coalesce함수의 부분이 특히 다르다. 그 외에는 거의 유사하지만 조금의 수정만으로도 mdriver를 이용한 결과 implicit free list보다

훨씬 높은 throughput을 기대할 수 있다.