M1522.000800 System Programming
Fall 2018

System Programming
Memory Lab Report

Daeeun Kim
2014-15395

1. Memory Lab

메모리 랩은 GNU 라이브러리의 malloc에 대응되는 메모리 할당 함수 mm_malloc을 직접 구현해 보는 과제이다. 직접 구현한 함수와 라이브러리 함수의 성능을 비교하며 지속적인 개선을 통해 효율적인 메모리 사용과 함께 짧은 응답 시간을 유지하는 것이 목적이다.

2. Implementation

mm.h에 정의된 함수 선언 인터페이스에 따라 mm_init, mm_malloc, mm_free, mm_exit을 구현하였다. 구현의 편의 및 가독성을 위해 #define을 이용한 preprocessor 상수 및 매크로를 사용하였으며, 추가적인 함수들을 이용해 코드를 모듈화했다.
전체적인 구현은, 비슷한 크기의 Free Block들을 리스트로 관리하고, 해당 size class 에 맞는 블럭이 없을 경우 다음 size 리스트를 탐색하며, coalescing으로 free block을 합하는 Segregated Fits 방법을 이용하였다. 추가 작업 없이, 요청한 크기에 맞춰서 힙을 늘려가는 경우 테스트 케이스상의 binary 와 같이 적은 크기의 malloc요청과 큰 요청이 여러 번 나눠서 들어오는 경우 메모리 효율이 현저히 낮아진다. 이러한 상황에 대비해 heap extension 부분을 개선하여 external fragmentation을 줄였다.

2.1 Preprocessor Constant Definition / Macros

연산 효율을 높이고 구현의 편의를 위해 많은 전처리기 상수와 매크로를 이용하였다. 대부분의 매크로는 CSAPP 교재에 나온 부분을 차용하였으며, Segregated Free List 구현을 위해 일부 추가하고 수정해서 사용하였다.
Segregated Free List 구현을 위해 기존에 있던 boundary tag에 추가로 내부 8바이트를 사용해 같은 size-class내의 free block list를 explicit한 doubly linked list로 구현하였다. 이 때문에 블럭의 최소 크기로 16바이트가 필요하며, 이 구현에 맞는 linked list 인터페이스를 위해 몇 가지 매크로를 추가하였다.

NEXT_FREE(bp) / PREV_FREE(bp) → 동일한 class 리스트의 다음 / 이전 free block
SET_NEXT_F(bp, val), SET_PREV_F(bp, val) → 다음 / 이전 free block을 지정하는 매크로

또한 External Fragmentation을 줄이기 위해 작은 크기의 malloc요청에 대해 size-class 상한 이상의 큰 배열을 예외적으로 리스트에 포함시키는 방법을 사용했는데, 이렇게 선언된 블럭을 bulk block이라 생각하고 인터페이스를 구성하였다. 이를 위해 allocated 비트 옆에 bulk block으로 형성된 free block을 구분하기 위한 비트를 추가했으며 다음의 매크로를 이용해 접근한다.

PACK(size, tagbit)　　((size) | (tagbit))
GET_ISBULK(p)　　(GET(p) & 0x2)

PACK의 tagbit은 bitwise 연산을 이용하여 BULKTAG | ALLOCTAG와 같이 지정한다.
마지막으로, Segregated Fits 구현을 위해 총 32개의 size class들을 이용했다.
SIZE_CLASS_NUM　32　// size class 개수
SIZE_CLASS_1 ~ SIZE_CLASS_31　// size class 구분기준

성능 최적화에 있어서 편의를 위해 #ifdef를 사용해서 SIZE_CLASS개수를 16개와 32개로 바꿔가며 사용할 수 있도록 구성하였다. 함수의 경우 get_size_class와 get_lower_bound함수가 CLASS_NUM_16의 define여부에 따라 다르게 정의된다. Default로 32개의 class를 이용한다.

2.2 Size Class

구현에 총 32개의 Size Class를 이용했으며, get_size_class함수를 이용해 어떤 size class에 속하게 될 지 결정하며, get_class_root함수를 이용해 각 class별 리스트 시작 노드를 얻도록 하였다. 이 리스트 시작 노드는 내용을 포함하지 않으며, doubly linked list의 구조만을 유지하여 경계조건에서의 예외 처리를 용이하게 하였다. 따라서 각 size class별로 8바이트의 시작 노드를 갖게 되며, mm_exit을 통한 memory leak 방지를 손쉽게 하기 위해 prologue 블럭과 epilogue블럭을 추가하였다. 이를 8byte 기준으로 align하기 위해 mm_init에서 ((SIZE_CLASS_NUM)+2) * DSIZE, 즉 144byte를 초기에 할당하여 리스트 관리에 사용하였다. 여기서 할당된 리스트 별 시작 노드는 next와 prev모두 자기 자신을 가리키는 상태로 초기화했으며, 원소가 추가될 때마다 circular doubly linked list 형태를 유지하도록 해 경계조건에서의 처리를 용이하게 하였다.

2.3 List Management

Free Block들을 Linked List로 관리하기 때문에 자료구조상 삽입과 삭제가 매우 잦다. 따라서 2.1에서 기술한 매크로와 함께 insert_block과 remove_from_list 두 함수를 만들어 모듈화하였다.

2.4 Bulk Block Allocation

작은 크기로 할당된 메모리가 흩어져 있는 경우, external fragmentation이 쉽게 생길 수 있다. 따라서 작은 크기의 메모리 할당 요청이 들어오고, 기존 리스트에 공간이 없어 heap상의 추가적인 공간 요청을 해야하는 경우, 현재 요청과 같은 작은 요청이 반복적으로 들어올 수 있다는 가능성을 고려해서 요청한 메모리의 8배에 해당하는 공간을 확보한다. 이렇게 확보된 Free Block은 연결된 size-class 범위를 벗어나지만, 단순한 free block이 아닌, bulk free block으로 분류하여 예외적으로 처리하였다. 따라서 작은 크기의 메모리 할당 요청들이 따로따로 들어오더라도 인접한 구역에 메모리를 할당받을 수 있도록 하였다. mm_malloc 함수에서 요청한 size에 따라 extend_heap함수를 호출해 일반적인 힙 확장을 할 수도 있고, 요청의 크기가 작은 경우 extend_heap_bulk함수를 호출해 특수한 입력에 대한 메모리 관리 효율을 높일 수도 있다.
이는 place 함수 구현에 있어서도 고려해야한다. find_fit 함수를 통해 size-class에 비해 훨씬 큰 bulk free block을 받게 되었을 때, 사용하고 남은 부분을 여전히 bulk형태로 남겨두어야 한다. 따라서, 요청 크기에 맞게 할당받은 블록이 bulk태그를 가진 경우 해당 블럭을 리스트에서 제거함과 동시에, 사용하고 남은 부분에 bulk태그를 유지시켜 동일한 list에 다시 삽입한다. 이 경우에 유일한 예외가 남은 부분이 해당 size-class범위보다 더 작아지는 경우인데, 이 경우를 체크하기 위해 find_fit함수와 place함수에 size-class를 나타내는 정수형 변수 하나를 추가로 이용했다. 블럭 할당 후에 남은 공간이 size-class 크기범위의 하한보다 작을 경우, 그대로 방치하면 알고리즘상 절대로 공간을 할당받을 수 없기에 bulk block이 아닌 일반 free block처럼 처리하였다.
하지만 coalescing에서는 불필요한 false fragmentation을 방지하기 위해 bulk 여부를 고려하지 않고 인접한 블럭을 모두 merge 하도록 구현했다.

2.5 Optimize

mm_init과 mm_malloc을 포함한 여러 함수들이 모두 정상적으로 작동해야 결과를 확인할 수 있는 만큼, 처음에는 CSAPP교재에 나타나있는 Implicit Free List 구현을 그대로 적용했고, 60점을 얻을 수 있었다. 메모리 입력 요청이 많아짐에 따라 작동속도가 현저히 느려졌으며 일부 테스트케이스에 대해서는 0.1초 이상의 시간이 소요되었다.
이후, size-class에 따라 free list를 관리하는 Segregated Free List 구현을 시도하였고, 불필요한 free block 탐색을 줄여 응답시간을 단축하였다. Segregated Free List를 이용한 최초 구현에서는 8개의 Size Class를 이용했고, 라이브러리 malloc보다 수행속도가 빨라

Throughput 부분에서 30점 만점의 최고 점수를 얻을 수 있었다. 하지만 제공된 binary 테스트 케이스와 같은 극단적인 입력에서 메모리 사용 효율이 50% 근처에 머무르는 등 비효율적인 작동이 나타나 메모리 사용 측면에서 70점 만점에 58점으로 총점 88점을 받았다.
부족한 메모리 사용 효율을 높이기 위해 Size Class를 8개에서 16개로 세분화했다. 이렇게 구현된 mm.c는 메모리 효율이 약간 증가해 총첨 89점을 얻을 수 있었고, 여전히 수행시간은 라이브러리보다 빨랐다. 하지만 극단적인 입력에서의 메모리 사용 효율은 크게 증가하지 않았으며, 짧게 생성되었다 사라지는 다시 말해 life cycle이 짧은 작은 크기의 요청들은 인접한 주소공간에 모여있도록 구현해야 external fragmentation을 줄일 수 있음을 알았다. 이는 단순히 Segregated Free List 아이디어 내에서는 해결할 수 없었기에 bulk block이라는 커다란 free block을 만들어 예외적으로 관리하도록 구현했다.
그 결과 16개의 size class와 작은 입력에 대한 bulk block 정책을 동시에 사용해 90점이 넘는 점수를 받을 수 있었다. 몇 번의 테스트를 거치며 '작은 입력'의 기준과 함께 bulk block 생성 기준을 변경한 결과 헤더 포함 총 블럭의 크기가 512바이트 이하인 경우에 요청의 8배에 해당하는 블럭을 할당하게 되었고, 총점 94점을 얻을 수 있었다. 이후 size class를 32개로 늘려서 테스트했고, 95점을 얻었으나, 이 이상 size class를 세분화했을 때는 유의미한 차이를 얻기 힘들었다.

2.6 Implement mm_realloc function

mm_realloc(void *ptr, size_t t) 함수는 입력과 인접한 다음 블럭의 상태에 따라 경우를 나누어 구현했다. 먼저 ptr == NULL 인 경우, 내부적으로 mm_malloc을 호출하도록 했으며, t == 0인 경우 내부적으로 mm_free(ptr)을 호출하도록 했다.
다음으로, 갱신될 크기 t가 요구하는 블럭의 크기가 기존 블럭보다 작을 때 그 크기 차이가 MIN_BLOCK_SIZE보다 작은 경우 아무 작업도 하지 않고 ptr을 리턴했다. 크기 차이가 MIN_BLOCK_SIZE보다 큰 경우, 블럭의 크기를 축소하고 남아있는 뒷부분은 free block으로 지정해 insert_block함수를 이용해 알맞는 size-class free list에 삽입했고 coalescing을 진행했다. 이 경우에는 기존 데이터가 그대로 유지되기 때문에 데이터 복사 과정이 필요없다.
갱신될 크기 t가 요구하는 새 블럭의 크기가 현재 블럭보다 클 경우, 다음 블럭이 free인지를 검사한 후, free 라면 다음 블럭을 free_list에서 제외시킨 뒤 현재 블록에 흡수시켰다. 그 후, mm_realloc(ptr, t)를 재호출 해 함수 처음부터 다시 크기를 비교하도록 하였다.
위 모든 경우에 해당하지 않는다면 realloc을 위해 새로 메모리를 할당받아야 한다는 의미이다. 따라서 newbp=mm_malloc(t)를 이용해 새로운 메모리의 시작 포인터 newbp를 얻은 뒤, copy_index를 4씩 이동시켜가며 GET과 PUT매크로를 이용해 기존 메모리 공간에서 새 메모리 공간으로 데이터를 복사했다. 그 이후, mm_free(ptr)을 통해 기존 메모리 공간을 free 시키고, 새로 할당받은 공간인 newbp를 리턴했다.

2.6 Implement mm_check function
작성한 mm_check함수는 두 단계로 동작한다. 먼저, size class별로 free list들을 순회하며 allocate된 블럭이 리스트에 포함되어있는지를 검사하며, 동시에 블럭의 사이즈가 해당 사이즈 클래스 범위를 벗어나는지 검사한다. 이때, bulk block으로 생성된 경우, 예외적으로 size범위의 하한보다 작은지만 검사한다. 이와같이 size class별 list의 root부터 시작하여 접근할 수 있는 모든 free block들을 마크한다. 이 때, FREE_MARK(p) 매크로를 사용하며, 0x4와 or연산을 진행하여 헤더의 3번째 lsb에 1로 마크한다. 각 단계에서 에러가 발견되면 즉시 루프를 종료하고 에러를 출력한다.
첫 번째 과정이 끝나면, prologue블럭부터 epilogue블럭까지 순회하며, free block들이 인접해 있는지 확인한다. 이 경우, 인접한 두 free block이 bulk block으로 생성된 경우는 예외로 한다. 또한 모든 free block이 마크되어있는지 체크하고, 마크를 지운다. 마크되지 않은 free block을 만난 경우 에러를 출력한다.

3. Conclusion

여러 함수가 모두 정상적으로 작동해야 결과를 확인할 수 있다는 점 때문에 버그 없이 구현하는게 매우 중요했고, 모든 테스트케이스에서 올바르게 작동하기까지 오랜 시간이 걸렸다. 교재에 나온 Implicit List를 이용해 60점, 기초적인 Segregated Fits 리스트를 이용해 80점을 받았고, 지속적인 개선과 최적화를 통해 95점까지 점수를 끌어올릴 수 있었다.
최종적으로 Segregated Fits를 이용했으며, 작은 크기의 입력은 인접한 메모리 공간을 할당받을 수 있도록 bulk block개념을 도입하여 함께 사용했다. mdriver를 이용한 지속적인 피드백을 통해 가장 높은 점수를 얻을 수 있도록 size-class 구분 기준과, bulk block 처리 기준을 조정하여 지금의 값을 얻어냈다.