System Programming Project 4

담당 교수 : 김영재

이름 : 차수환

학번 : 20222089

1. 개발 목표

본 프로젝트의 목표는 C 언어 환경에서 표준 라이브러리의 malloc, free, realloc 함수와 동일한 기능을 수행하는 사용자 정의 동적 메모리 할당기를 구현하는 것이다. 이 할당기는 메모리 활용도와 처리량을 모두 고려하여 높은 성능을 가지도록 하는 것을 목표로 한다. 프로젝트는 수업에서 소개된 메모리 관리 개념을 바탕으로 하며, 특히 Segregated List 기반 할당 방식을 사용하여 메모리를 효율적으로 관리하도록 하였다.

2. 개발 범위 및 내용

동적 메모리 할당기를 구현하기 위해 먼저 mm_init, mm_malloc, mm_free, mm_realloc 함수를 통해 힙 초기화, 메모리 할당 및 해제, 크기 조정을 수행하도록 했다.

메모리 관리 방식은 탐색 시간과 비용을 줄이기 위하여 Implicit, Explicit Free List가 아닌 Segregated Free List 를 선택했다. 구현한 할당기는 16바이트부터 2^19바이트까지 총 20개의 크기 클래스를 갖고 있으며, 각 클래스는 내부적으로 크기순으로 정렬된 자유 블록 리스트를 유지한다.

Best-Fit 전략을 사용해 메모리 활용도와 처리량을 높였고, 모든 페이로드 포인터는 8바이트 정렬을 만족하도록 했다.

각 크기 클래스의 free list 시작점을 저장하는 free_lists라는 포인터 배열을 사용하였으며, 크기 클래스 계산, 힙 확장, 블록 병합 및 분할, 리스트 관리 등의 역할을 하는 get_class_index, extend_heap, coalesce, insert_list, delete_list, place 함수들을 추가하여 사용하였다.

프로그램의 구현은 초기 함수들을 완성한 뒤 realloc을 추가하고, 분할 및 병합을 점진적으로 최적화하는 방식으로 진행했다.

3. 구현 결과

구현한 함수들과 그에 대한 설명은 아래과 같다.

- int mm_init(void)

이 함수는 힙과 free list를 초기화하는 역할을 한다.

먼저 힙에 20개의 크기 클래스를 관리할 포인터 배열 free_lists를 동적으로 할당하고, 모든 리스트 헤드를 NULL로 초기화한다. 다음으로 힙 시작부에 정렬을 위한 4바이트 패딩, 첫 블록의 header와 footer(각각 4바이트씩), 그리고 힙의 끝을 알리는 에필로그 헤더(4바이트)를 차례로 설정한다. 마지막으로 extend_heap을 호출해 초기 힙 공간을 확장하고 첫 free block 만들어져 할당 준비를 마친다. mem_sbrk 호출 결과를 확인하여 실패 시 오류를 반환하도록 했다.

- void *mm_malloc(size_t size)

사용자가 요청한 크기의 메모리 블록을 할당하는 함수다. 요청 크기를 8byte 단위로 정렬하고, 최소 블록 크기인 16바이트 이상으로 조정한다. 그리고, 적절한 클래스의 크기를 계산하여 그 클래스부터 더 큰 클래스까지 free list를 순서대로 탐색하고, 크기가 충분한 가장 작은 블록을 찾는다(Best-Fit 전략). 적합한 블록이 없으면, 힙을 확장해 새로운 free block을 확보한다. 찾은 블록은 place 함수를 통해 allocated 상태로 변경하고, 필요하면 블록을 분할한다. 크기가 0인 요청은 NULL을 반환하도록 했다.

- void mm_free(void *bp)

할당된 메모리 블록을 해제하는 함수다. 먼저, 블록의 헤더와 풋터로 해당 블록의 크기를 가져오고, allocated flag를 free 상태로 설정한다. 그리고 인접한 이전과 다음 블록의 할당 상태를 확인하여, free 상태이면 병합한다. 만약 NULL 포인터가 들어오면 아무 동작도 하지 않도록 했다.

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

기존 할당된 블록의 크기를 변경하는 함수다. 만약 포인터가 NULL이면 mm_malloc을 수행한다. 요청 크기가 0이면 해당 블록을 해제하고 NULL을 반환한다. 요청 크기가 현재 블록 크기보다 작거나 같으면, 남은 공간은 분할하여 처리한다. 크기가 더 큰 경우에는 인접한 다음 블록의 상태를 확인하고 free 상태이고 충분한 크기를 가지고 있다면 두 블록을 병합한다. 힙의 마지막 블록이라면 힙을 확장하고, 모든 방법이 불가능하다면 새 블록을 할당하고 기존 데이터 내용을 복사한 뒤, 기존 블록을 해제한다.

- static int get_class_index(size_t size)

주어진 블록 크기에 대해 어느 크기 클래스에 속하는지 결정한다. 16 byte부터 2^19 byte까지 20개의 크기 구간을 가지며, 크기에 따라 0부터 19까지의 인덱스를 반환한다.

- static void *extend_heap(size_t size)

extend_heap은 힙 공간을 추가로 확장하는 함수다. 요청 크기를 8byte 정렬 기준으로 조정하고, mem_sbrk를 호출해 힙을 확장한다. 새로 확보된 공간의 시작 주소를 free block 으로 초기화하며, 헤더와 풋터에 그 크기와 free 상태임을 기록한다. 또한 기존 에필로그 헤더를 새 위치로 옮기고, 새로 추가된 블록과 연결된 free block 과 병합한다.

- static void *coalesce(void *bp)

인접한 free block 들을 병합하는 함수이다. 블록의 이전과 다음 블록이 할당되었는지의 여부에 따라 네 가지 경우로 나누어 작동하는데, 이전과 다음 블록 모두 할당되어 있다면 병합 없이 현재 블록을 free list에 넣고 종료하고, 이전 또는 다음 블록 중 하나만 free 상태라면 그 블록과 병합하며, 양쪽 모두 free 상태인 경우에는 세 블록을 모두 합친다. 병합 후에 병합된 블록을 다시 적절한 크기 클래스 리스트에 넣어 관리한다.

- static void insert_list(void *bp)

free block을 해당 크기 클래스의 free list에 크기순으로 삽입하는 함수이다. 이중 연결 리스트 형태로 삽입하여 free list가 항상 크기순으로 정렬되게 한다. 이를 통해 Best-Fit 검색을 빠르게 수행할 수 있도록 한다.

- static void delete_list(void *bp)

free list에서 특정 블록을 제거하는 함수이다. 블록이 리스트의 어느 위치에 있는지를 확인하고, 이전과 다음 포인터를 적절히 연결해 리스트 연결을 유지한다. 리스트 헤드가 변경될 경우에 즉시 반영한다.

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

주어진 free block에 실제 할당을 수행하는 함수이다. 블록의 크기와 요청받은 크기를 비교하여, 남는 공간이 최소 블록의 크기 이상이면 블록을 분할한다. 요청 크기가 100 byte 미만이라면, free block을 앞에 두고 allocated block을 뒤에 배치하여 이후 큰 블록 요청이 들어올 때를 대비해 큰 연속 공간을 확보할 수 있도록 한다. 100 byte 이상이면, 그 반대로, allocated block을 앞에 두고 남은 free block을 뒤에 둔다. 분할 후 새로 생긴 free block은 insert_list로 다시 free list에 삽입된다.

4. **성능 평가 결과**

mdriver와 제공된 trace 파일들을 통해 구현의 정확성과 성능을 검증했다. mdriver은 성능을 측정하는 함수로, 메모리 활용도와 처리량을 측정하고, 구현이 정확하지 않으면 실행되지 않는다.

메모리 활용도는 프로그램이 실제로 필요로 하는 메모리 양을, 할당기가 전체 힙에서 사용하는 크기와 비교한 비율이고, 처리량은 단위 시간당 수행된 할당/해제/재할당 연산의 평균 수를 의미한다.

구현한 프로그램은 메모리 활용도 53점, 처리량 40점으로 총점 93점을 획득하였다.

5. 결론

본 프로젝트에서 segregated list 기반의 동적 메모리 할당기를 구현하여, 높은 메모리 활용도와 처리량을 동시에 달성하였다. Best-Fit 전략, 크기순 삽입, 병합 및 분할 최적화 등의 방법을 적용하여, 정확성과 성능 모두에서 우수한 결과를 얻었고, 할당 결과도 안정적이었다.