Project #4 : Mallocator

CSE4100 시스템프로그래밍 과제4 보고서

학번: 20190328

이름: 조준희

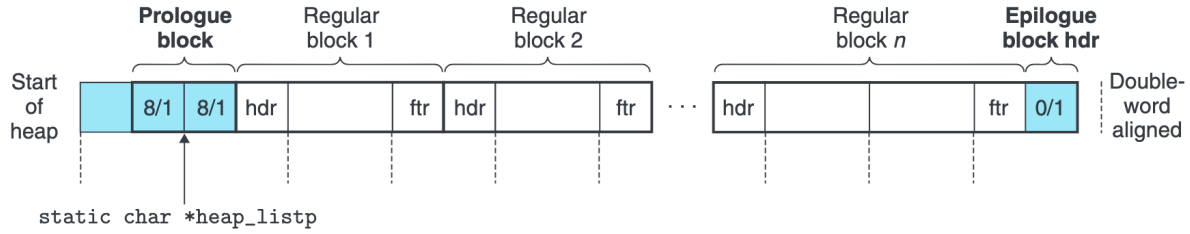
목차

머리말

이번 프로젝트의 목표는 동적 메모리 할당자의 설계 및 구현이다. `mm.c`에 작성된 동적 메모리 할당자에는 표준 C 라이브러리 함수인 `malloc`, `free`, `realloc`과 유사한 기능을 수행하는 `mm_init`, `mm_malloc`, `mm_free`, `mm_realloc` 함수가 구현되어 있으며, 정확성, 공간 효율성(utilization), 그리고 처리율(throughput) 측면에서 만족스러운 결과를 도출해야 한다. 본문에서는 구현된 동적 메모리를 구성하는 메모리 블록의 구조와 동적 메모리 할당자에서 가용 메모리를 관리하는 방법인 Segregated Free List, 그리고 `mm.c`에 구현된 각 함수와 전역 변수들의 구현에 대해 설명한다.

`mm_init`

이번 프로젝트에서 구현된 동적 메모리 할당자는 힙 메모리를 다음과 유사한 형태로 초기화한다. 변수나 블록 구조에서 다소 차이가 있으나, 8바이트 메모리 정렬을 위하여 워드 하나를 비워두는 것과 프롤로그 블록 및 에필로그 블록의 존재, 각 블록마다 존재하는 헤더와 푸터, 그리고 메모리 연산의 시작점이 프롤로그 블록의 정중앙에 위치한다는 공통점이 있다.



이번 프로젝트에서 구현한 `mm_init` 함수는 프롤로그 블록과 에필로그 블록을 포함하는 가용 블록을 생성한 뒤, `extend_heap` 함수를 호출하여 프롤로그 블록 뒤에 초기 힙 공간을 할당한다. 이 공간의 크기는 `INITCHUNKSIZE`에 $2^6 = 64$바이트로 정의되어 있다.

`extend_heap` & Segregated Free List

`extend_heap` 함수는 새로운 가용 블록을 생성하며 힙 메모리 공간을 확장한다. 새로운 가용 블록을 생성할 때는 생성할 블록의 헤더와 푸터, 그리고 새로운 에필로그 헤더를 생성한다. 그 후 `insert_node` 함수를 통하여 만들어진 블록을 가용 메모리를 관리하기 위한 Segregated Free List에 추가한 다음, `coalesce` 함수를 통하여 인접한 가용 블록을 하나로 통합한다.

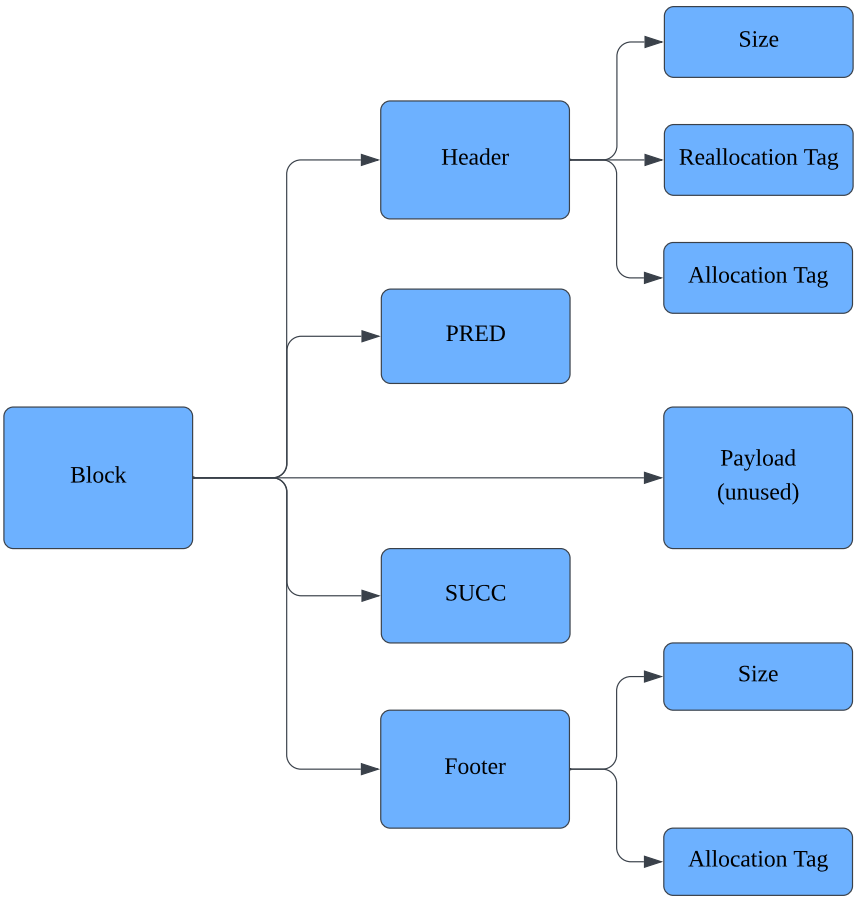
이번 프로젝트의 동적 메모리 할당자는 가용 블록을 효율적으로 찾기 위하여 2의 거듭제곱을 기준으로 크기별 클래스를 나누고 각 클래스에 대한 가용 리스트를 관리한다. 가용 리스트는 `segregated_free_lists`라는 이름의 전역 변수로 선언되어 있으며, 클래스의 개수는 `SEGLISTNUM`을 넘지 못하도록 구현되어 있다. `segregated_free_lists`는

`void*` 포인터 배열이며, 각 원소는 해당 크기 클래스에 속하는 가용 리스트의 시작점을 가리킨다.

- list[0] : size <= 1
- list[1] : 1 < size <= 2
- list[2] : 2 < size <= 4
- ...
- list[19] : 2^18 < size

`insert_node` & Free Block

`insert_node` 함수는 가용 블록을 가용 리스트에 넣기 위한 함수이며, 가용 블록을 가리키는 `void*` 포인터와 가용 블록의 크기를 통하여 가용 블록의 크기에 맞는 클래스에 가용 블록을 보관한다. 이때, 같은 클래스에 속하는 가용 블록들은 하나의 이중 연결 리스트로 보관하며, 이를 위해 현재 위치한 리스트에서 자신의 앞과 뒤에 위치한 블록을 가리키기 위한 PRED와 SUCC라는 포인터를 위한 공간이 추가로 할당된다. 따라서, 최종적으로 가용 블록의 구조는 아래의 다이어그램과 같다.



가용 블록은 블록을 파악하기 위한 여러 정보가 담겨 있는 헤더, 가용 리스트에서 앞뒤에 위치한 블록을 가리키는 포인터가 저장되는 공간인 PRED와 SUCC, 가용 메모리 블록을 빠르게 병합하기 위한 푸터, 그리고 사용되지 않는 나머지 공간으로 이루어져 있다. 여기서 Size는 현재 블록에서 사용 가능한 메모리의 크기를 나타내며, Allocation Tag는 자기 자신의 할당 여부를 나타내고, Reallocation Tag는 바로 앞에 위치한 블록의 재할당 여부를 나타낸다. 간략하게 정리하자면 아래와 같다.

- Header (4바이트): 블록의 전체 크기(size), 현재 할당 여부(LSB, 최하위 비트), 그리고 재할당 태그(reallocation tag, 끝에서 두 번째 비트) 정보를 담고 있다.
- Payload: 가용 블록에서는 사용되지 않는 남은 공간이다.
- Footer (4바이트): Header와 동일하게 블록의 크기와 할당 여부 정보를 저장한다. 덕분에 블록의 끝에서 시작 위치를 계산할 수 있게 하여 블록 병합(coalescing)을 효율적으로 수행하도록 한다.
- PRED : 가용 리스트에서 이전 가용 블록을 가리키는 포인터이다.
- SUCC : 가용 리스트에서 다음 가용 블록을 가리키는 포인터이다.

`coalesce` & `delete_node`

`coalesce` 함수는 인접한 가용 블록을 하나로 합치기 위한 함수인데, 메모리 공간에서 인접한 블록의 할당 여부를 Allocation Tag만으로 빠르게 파악한 다음 가용 블록이 인접해 있다면 가용 리스트에서 자기 자신과 인접한 가용 블록을 모두 삭제하고, 자기 자신과 합쳐 하나의 가용 블록으로 만든 다음 다시 이 블록을 가용 리스트에 추가한다. 네 가지 경우에 따라 병합을 수행한다.

- Case 1: 이전과 다음 블록 모두 할당됨 -> 병합 없음.
- Case 2: 다음 블록만 가용 -> 현재 블록과 다음 블록 병합.
- Case 3: 이전 블록만 가용 -> 현재 블록과 이전 블록 병합.
- Case 4: 이전과 다음 블록 모두 가용 -> 세 블록 모두 병합.

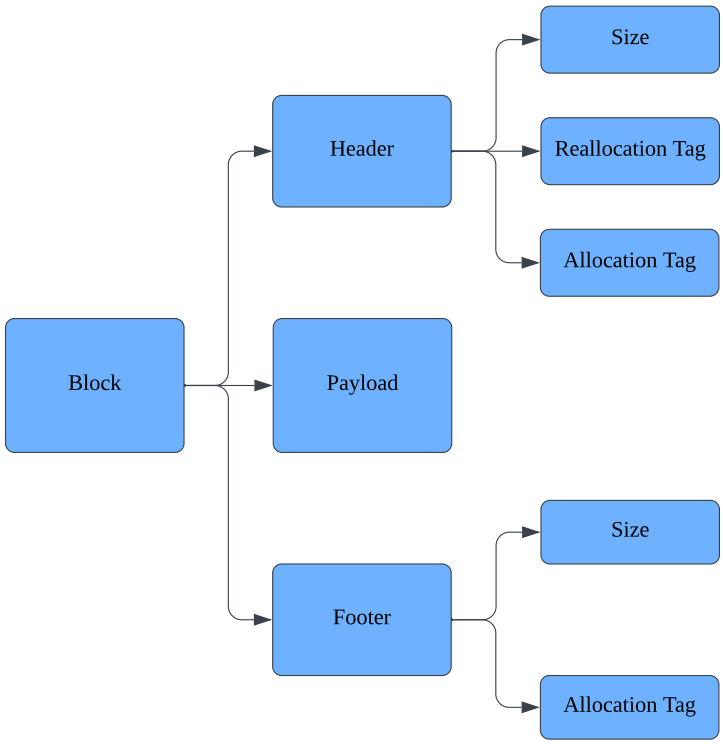
병합이 발생하면, 병합에 사용된 블록들은 가용 리스트에서 제거(`delete_node`)되고, 병합으로 생성된 더 큰 새 가용 블록이 리스트에 추가(`insert_node`)된다. `delete_node` 함수 또한 `insert_node`와 비슷하게 가용 리스트에서 삭제할 블럭에 맞는 클래스를 탐색한 뒤 해당 클래스에서 자기 자신을 삭제한다.

동적 메모리를 실제로 할당하는 함수는 `mm_malloc` 함수로, 요청받은 `size`에 헤더/푸터 크기와 정렬(`ALIGNMENT`) 요구사항을 반영하여 실제 할당할 블록 크기 `asize`를 계산한다. `segregated_free_lists`에서 `asize`에 맞는 크기에 해당하는 클래스의 가용 리스트부터 탐색을 시작하는데, 이때 단편화 현상을 최소화하기 위해 다음의 과정을 거친다.

1. 리스트를 순회하며 `asize`보다 크고, 재할당 태그가 없는 첫 번째 가용 블록을 찾는다.
2. 적절한 가용 블록을 찾지 못하면 `extend_heap` 함수를 호출하여 힙을 확장하고 새 블록을 사용한다.
3. `place` 함수를 호출하여 찾은 가용 블록에 요청된 메모리를 할당하고, 남은 공간이 충분하면 분할한다.

`place` 함수에서는 단편화 현상을 최소화하기 위하여 요청된 크기(`asize`)를 할당한 후 남는 공간(`remainder`)이 최소 블록 크기(16바이트)보다 큰지 확인한다. 남는 공간이 충분하면, 현재 블록을 `asize`만큼 줄이고 남는 부분을 새로운 가용 블록으로 만들어 가용 리스트에 추가한다. 특히 할당 크기가 100바이트 이상일 경우, 분할된 가용 블록을 앞쪽에 배치하고 할당된 블록을 뒤쪽에 배치하는데, 이는 큰 블록과 작은 블록이 섞이는 것을 줄여 단편화 현상을 최소화시키기 위함이다. 남는 공간이 충분하지 않아 분할하지 않을 경우, 전체 블록을 할당 상태로 변경한다.

`mm_malloc` 함수를 통하여 할당된 메모리 블록의 구조는 아래의 다이어그램과 같다.



헤더와 푸터는 동일하지만, Payload 영역은 `mm_malloc` 호출 시 사용자가 실제로 데이터를 저장하는 영역이다. 할당된 블록은 Allocation Tag의 값이 1로 고정된다.

반대로 할당했던 힙 메모리를 회수할 때에는 `mm_free` 함수를 사용한다. 회수하고자 하는 블록의 헤더와 푸터의 Allocation Tag를 0으로 설정하고 이 블록의 다음 물리적 블록에 설정되어 있을지 모르는 Reallocation Tag를 제거한 뒤, `insert_node` 함수를 호출하여 해제된 블록을 다시 가용 리스트에 추가한다. 마지막으로 `coalesce` 함수를 호출하여 인접한 가용 블록과의 병합을 시도한다. 따라서 이번 프로젝트에서 구현한 동적 메모리 할당기는 메모리를 회수할 때마다 병합을 진행하는 즉시 병합(Immediate Coalescing) 정책을 따르고 있다.

메모리의 재할당은 `mm_realloc` 함수를 통하여 이루어지는데, 이 함수에도 단편화 문제를 최소화하기 위한 여러가지 방법이 구현되어 있다. 할당하려는 블록의 주소가 `NULL`이면 `mm_malloc` 함수를 호출하고, `size`가 0이면 `mm_free` 함수를 호출하는 것까지는 기존의 `malloc`과 동일하다. 할당된 메모리의 크기를 바꾸려는 정상적인 상황에서는 다음과 같이 작동한다.

- 크기 감소: 만약 새로 요청된 메모리의 크기인 `new_size`가 기존에 할당되었던 메모리의 크기 `old_size`보다 작으면 블록을 분할하지 않고 그대로 사용하며 크기 변경에 대비한다. 또한 다음 블록에 Reallocation Tag를 설정하여 공간을 예약한다.

- 크기 증가: 기존에 할당된 크기를 조절하지 않고도 감당 가능하다면 그대로 사용하지만, 기존에 할당된 크기만으로 부족할 경우, 메모리에서 바로 뒤에 위치한 블록이 가용 블록이거나 힙의 끝인지 확인한다. 만약 그렇다면, 다음 블록을 흡수하여 블록을 확장하고 필요한 경우 `extend_heap` 함수를 호출한다. 이러한 방법으로 확장할 수 없다면, `mm_malloc` 함수를 호출하여 새 블록을 할당하고, `memcpy` 함수로 데이터를 복사한 뒤, 기존 블록은 `mm_free` 함수로 해제한다.

재할당 후 남은 버퍼가 작으면, 다음 블록의 Reallocation Tag를 1로 설정하여 미래의 재할당에 대비한다.