**2022-2 CSED211 Lab12&13 Report**

학번 : 20210479

이름 : 이주현

**명예서약 (Honor Code)**

나는 이 프로그래밍 과제를 다른 사람의 부적절한 도움 없이 완수하였습니다.

I completed this programming task without the improper help of others.

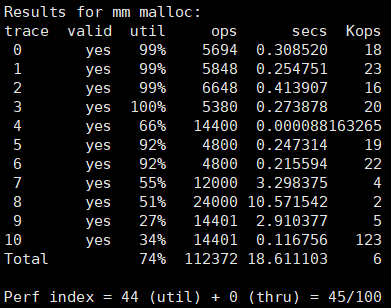
1. **코드 설명**

mm.c 내에서 구현한 함수별로 나누어 각각의 기능과 개선 과정, 최종 구현 방법 및 time complexity를 설명했다.

* + **int mm\_init(void): heap 초기화**
    - 먼저 처음에는 mem\_sbrk()를 이용해 4 \* WSIZE만큼 heap을 늘려주었다. 만약 이 과정에서 문제가 생긴다면 -1을 반환하고 종료되도록 했다.
    - 그 다음으로는 차례대로 unused alignment padding, prologue header, prologue footer, epilogue header를 세팅해 주었다. 이때는 PUT이라는 preprocess statement를 사용해주었다. PUT은 size정보와 할당 여부 bit을 묶어주는 function을 수행한다.
    - 그 후 heap의 시작 부분을 가리키는 포인터를 2 \* WSIZE만큼 당겨 두었다.
    - 그리고 heap을 (CHUNKSIZE / WSIZE) bytes만큼 늘려주었다. 이때 이 과정에서 문제가 발생하였을 경우에는 역시 -1을 반환하고 종료하도록 해주었다. 그리고 만약 이 과정까지 아무 문제 없이 수행되었다면, mm\_init()은 0을 반환하고 종료된다.
    - 대략적인 time complexity를 생각해 보면, 여기에는 일부 조건문만 있을 뿐 반복문이 사용되지 않았으며, 이 함수 내에서 호출하는 다른 함수들 역시 반복문을사용하지 않기 때문에 time complexity는 O(1)일 것임을 추측할 수 있다.
    - **static void \*extend\_heap()**: mm\_init()을 구현하는 과정에서 필요했던 함수이다. 이 함수에서는 우선 필요한 size를 alignment에 맞도록 보정해 새 사이즈를 구해준 후, 그 보정된 사이즈만큼 heap을 확장하는 동작을 수행한다. 확장 후에는 header와 footer까지 세팅해주고, 만약 주변에 free block이 있을 경우에 대비해 coalesce() 함수를 호출하고 종료된다.
    - **static void \*coalesce()**: mm\_init()을 구현하는 과정에서 필요했던 함수이다. 이 함수에서는 주어진 포인터 부근의 block들을 살펴본 후, coalescing이 필요한지를 판단하고 실제 필요한 경우라면 coalescing을 수행한다. coalescing이 필요한 경우는 총 4가지로 나뉘며, 이를 각 블록의 header 혹은 footer에 저장된 할당 여부 비트를 통해 나타내어 각각에 알맞은 과정을 처리하도록 해주었다.
      * free block이 없을 때: argument로 받았던 bp를 다시 반환함
      * 뒷 block이 free일 때: 뒷 블록과 병합 진행
      * 앞 블록이 free일 때: 앞 블록과 병합 진행
      * 앞뒤 모두 free일 때: 세 블록을 한번에 병합
  + **void \*mm\_malloc(size\_t size): 실제 malloc 수행**
    - 이 함수에서는 우선 윗부분에서 예외 처리를 먼저 해주었다. 우선 입력받은 size가 0인 경우는 어떠한 동작을 수행해줄 필요가 없는 의미없는 명령이다. 따라서 이 경우에는 NULL을 반환하고 종료하도록 한다.
    - size가 유의미한 값일 때에는 malloc 동작을 수행한다. 먼저 size의 크기에 따라 alignment를 지키도록 보정을 해준다. 크기를 나누는 기준은 8bytes로, 이보다 작으면 16byte로 보정되며 이보다 클 시 적절한 8의 배수로 보정된다.
    - 이후에는 적절한 size의 블록을 탐색한다. 만약 적절한 블록이 있을 시, size만큼의 블록을 할당하고 mm\_malloc() 함수가 종료된다.
    - 만약 적절한 블록이 없다면 heap을 확장해야 한다. 필요한 사이즈는 보정했던 size와 CHUNKSIZE 중 큰 값으로 정해주었다. 그리고 extend\_heap()를 호출해 heap을 필요한 만큼 확장해 준 뒤, 여기에 필요한 size만큼 블록을 할당해 준다. 이후 할당된 블록의 포인터를 반환하며 mm\_malloc()이 종료된다.
    - 이 함수는 자체에서 반복문을 가지지는 않지만, 아래에 설명된 것과 같이 이 함수가 호출하는 함수 중 반복문을 갖는 것이 있으므로 time complexity가 O(n)이 될 것임을 알 수 있다. (단, n = heap 내부의 block의 전체 개수)
    - **static void \*find\_fit()**: 적절한 사이즈의 블록을 탐색하기 위한 함수이다. 여기에서는 heap의 처음부터 시작하여 할당이 되지 않은 free 상태이며 적절한 사이즈를 갖는 블록이 있는지 탐색한다. 앞서 언급한 두 조건을 동시에 만족하는 블록이 존재할 시 해당 블록의 주소를 반환하며 함수가 종료된다. 그러나 블록을 찾지 못하고 heap의 끝에 도달했을 시, NULL을 반환하고 종료된다. 이 반환값을 통해 탐색의 성공 여부를 알 수 있다.
    - **static void place()**: 실제로 지정된 heap 내의 블록에 size만큼이 할당되었음을 표시해주는 함수이다. 이 함수에서는 사용하려는 블록과 실제 필요한 size를 우선 비교한다. 그리고 그 차이가 16bytes 이상이라면 나머지 공간을 분리해 새로운 free block으로 표시한다. 만약 차이가 그 이하이면 해당 블록 전체를 할당으로 표기하고 넘어간다.
  + **void mm\_free(void \*ptr): 할당 해제**
    - 이 함수에서는 우선 할당 해제하려고 하는 블록을 가리키는 포인터를 argument로 받게 된다. 그러면 mm\_free()에서는 이 포인터가 가리키는 블록의 사이즈를 구하고, 해당 블록의 header와 footer에 접근해 할당 여부 표시 비트를 수정한다. 그리고 바로 coalesce() 함수를 불러 주변 free block을 탐색해 possible merging을 수행한다.
    - 이 함수 역시 별다른 반복문 등을 가지고 있지 않고, 할당 해제를 위한 블록을 탐색하는 과정 등이 필요한 것도 아니므로 time complexity는 역시 O(1)일 것임을 추측할 수 있다.
  + **void \*mm\_realloc(void \*ptr, size\_t size): 새로운 size로 재할당**
    - 우선 이 함수의 경우 앞 부분에서 두 가지 예외를 처리한다. 먼저 size가 0으로 입력된 경우 이는 mm\_free()를 호출하라는 것과 같다. 따라서 mm\_free()를 호출한 뒤 NULL을 반환하고 종료하도록 했다.
    - 그리고 만약 입력받은 ptr이 NULL인 경우 mm\_malloc()을 호출하라는 것과 같다. 따라서 mm\_malloc()을 호출하고 종료하도록 했다.
    - 만약 이 두 경우에 해당하지 않는다면, 바로 이어서 realloc operation을 수행하도록 했다. 우선 newp라는 새로운 포인터를 선언해 mm\_malloc()을 선언해주도록 했다. 그리고 만약 이 과정에서 문제가 발생할 때에도 NULL을 반환하고 종료되도록 했다.
    - 만약 문제없이 진행되었다면 이번에는 입력받은 사이즈 또는 원래 블록의 사이즈 중 최솟값을 구하고, 그 사이즈만큼 메모리를 기존 블록에서 새 블록으로 복사하도록 했다. 이후 원래 블록을 할당 해제하기 위해 mm\_free()를 불러주었고, 여기까지 문제없이 진행되었다면 새로운 블록을 가리키는 포인터를 반환하고 종료되도록 했다.
    - 이때 mm\_realloc() 역시 명시적으로 사용한 반복문은 없으나, mm\_malloc()을 불러 간접적으로 반복문을 사용하기 때문에 time complexity가 O(n)일 것임을 예상할 수 있다.

1. **구현한 mm.c의 space evaluation**

* 우선 validity의 측면에서는 매우 높은 정확도를 보였다. 아래의 사진을 통해 그것을 확인할 수 있다.

****

* 그러나 구현에 있어 first-fit 방법을 채택하였기 때문에 어쩔 수 없는 internal fragmentation이 발생하였을 것이다. 16 bytes를 기준으로 그 이상이 되어야만 새로운 free block으로 분리될 수 있으므로 0~15 bytes의 internal fragmentation이 다수 존재할 것이다. 또한 first-fit 방법을 사용해 이러한 fragmentation이 한 곳에 집중되어 있을 것이다. 가장 기초적인 implicit list와 기본적인 탐색방법을 사용하였기 때문에 spatial 효율이 낮은 것을 알 수 있다.

1. **구현하지 못하였으나 생각해본 점들**
   * **next-fit and segment free list**
     + 이번에 구현한 mm.c에서는 find\_fit을 사용해 필요한 size를 넘는 크기의 블록을 마주치면 바로 그것을 사용하도록 반환하는 방식을 선택했다. 만약 segment free list를 구현해 next-fit 방법을 사용했다면, 탐색을 위한 시간은 조금 더 걸렸을지언정 heap 공간을 사용하는 효율성을 많이 올릴 수 있었을 것이다.
     + 이는 segement free list가 free block들만을 골라내어 관리하는 list로, 새롭게 할당을 하기 위해 탐색을 진행할 때의 시간 효율을 올려주기 때문이다. 그리고 next-fit이란 first-fit과 유사하게 모든 노드를 탐색하지 않는 방법이면서도 동시에 fragmentation이 연속적으로 누적되어 있을 확률을 줄여주는 효율적인 방법이다.
     + 그러나 place()를 구현함에 있어서 너무 낭비가 심하다고 생각되는 부분은 적절히 블록을 분할해 새로운 free block으로 만들어 주는 과정을 거치기 때문에, 적어도 블록이 최대 16byte까지만 낭비된다는 성과를 얻을 수 있었다.
2. **Lab12&13에서 배운 점**

* Lab 12&13 malloc lab을 해결하면서 동적할당을 위한 기본적인 원리와, 원리를 코드로 구현하기 위해 필요한 사항들을 학습할 수 있었다.
* 특히 header와 footer를 구현하는 방법과 그 이유를 깨달을 수 있었다. footer를 붙여 boundary를 표시해주면 용량을 차지하게 되지만 coalescing과 여타의 동작에 많은 도움이 될 수 있다는 것을 알게 되었다.