Project #3: Dynamic Memory Allocator

전공: 컴퓨터공학과 학년: 3학년 학번: 20191619 이름: 이동석

1. 개발 목표

이번 프로젝트의 주요 목표는 dynamic memory allocator를 C언어로의 구현이다. 수업 시간에 배웠던 implicit list와 explicit list를 활용하여 구현해도 된다. 하지만, 본 프로젝트에서는 performance 값이 높게 나올 수 있도록 Segregated free list와 Best-fit 방법, 그리고 Realloc 함수의 수정으로 효율적이고 정확하며, 빠른 dynamic memory allocator를 구현했다.

2. 개발 및 내용

대부분의 코드는 책의 코드를 참고하여 작성하였다. 그 외 performance의 향상을 이룬 부분이나, 중요 자료구조 및 함수 코드의 동작방식과 내용에 대해 설명한다. 메모리 블록은 힙을 8바이트로 정렬하는데 필요한 Alignment pad, Prologue Block을 위해 필요한 각 header / footer, 그리고 힙의 마지막 부분에 위치하는 epliogue header로 나타난다. 힙이 확장되면 새로운 epilogue가 생성되며 항상 크기가 0이고 할당상태는 1이다.

2-1 Segregated free list 구현

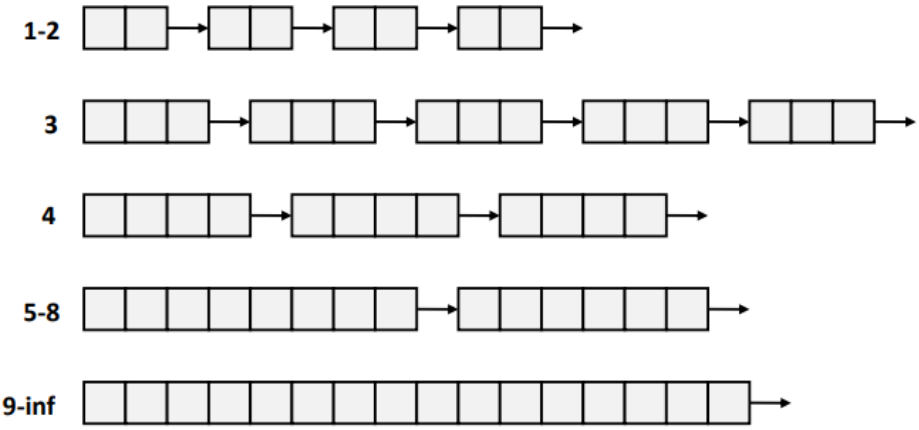
우선 새로 추가한 매크로 코드들은 다음과 같다. 각 매크로들을 간단하게 설명하면 다음과 같다.

```
#define SET_PTR(p, ptr) (*(unsigned int *)(p) = (unsigned int)(ptr))
#define NEXT_PTR(ptr) ((char *)(ptr))
#define PREV_PTR(ptr) ((char *)(ptr) + WSIZE)
#define NEXT(ptr) (*(char **)(ptr))
#define PREV(ptr) (*(char **)(PREV_PTR(ptr)))
```

SET_PTR : 주어진 위치 p에 포인터 ptr의 값을 저장한다.
NEXT_PTR : 주어진 포인터를 char* 로 변환한다. 이때 char *의 사용이유는 바이트 단위로 이동하기 위함이다.
PREV_PTR : WSIZE만큼을 더하여 char *로 변환한다. 이전 노드를 가리키는 포인터를 구하는데 사용한다.
NEXT : 주어진 ptr에 저장된 값(다음 노드)를 반환한다.
PREV : 주어진 포인터의 위치의 PREV_PTR을 구하고 반환한다. 이전 노드를 가리키는 포인터를 반환한다.

```
#define CHUNKSIZE (1<<9)
#define INITCHUNKSIZE (1<<3)
#define LISTLIMIT 10
```

Segregated free list는 다음 그림과 같이 free 블록을 저장하는 방식이다. explicit과 implicit에 비해서 Higher throughput을 가지며, 더 나은 memory utilization이 생기므로, performance의 좋은 성능을 예상해볼 수 있다.



이번 프로젝트에서는 2, 4, 8, 16, 32 .. 1024의 크기로 나누고 다음 전역변수에 각각의 크기에 해당하는 포인터의 시작주소를 저장했다.

```
void *segregated_free_lists[LISTLIMIT];
```

각 인덱스에 저장되는 포인터들은 explicit list 형식으로 저장해 관리하는 방식을 사용했다. 이후, 다음 단계를 통해 해당 리스트에 free 포인터가 추가되어 관리된다.

1. 초기 mm_init에서 해당 포인터들을 모두 NULL로 초기화 시켜준다.
2. extend_heap이나 place함수에서 insert_node가 호출된다.
3. insert_node 함수에서는 다음과 같은 단계를 가진다.
   3-1 입력받을 size에 따라 적절한 index를 찾는다.
   3-2 리스트를 검색하고, 삽입할 위치를 찾는다. 이때 삽입할 위치는 해당 리스트 내에서 삽입하려는 블록 크기보다 큰 첫 번째 블록 앞이다.
      I) 이미 있는 노드 사이에 삽입
      II) 리스트의 시작부분에 삽입
      III) 리스트의 끝부분에 삽입
      IV) 유일한 노드
   3-3 삽입한 이후, 다음 블록과 이전 블록의 포인터를 연결한다.
4. delete_node 함수에서는 insert_node 함수와 동일하게 size에 따라 적절한 index를 찾는다. 이후, 해당 블록을 리스트에서 제거하고 마찬가지로 올바르게 포인터를 연결한다.

2-2 Best-fit 구현

find_fit 함수는 주어진 크기(asize)에 맞는 빈 메모리 블록을 찾는 역할을 한다. 이 함

수를 first-fit 방식으로 구현해도 되지만, util의 성능을 조금이라도 더 높이기 위하여 best-fit방식을 사용했다. 실제로, 해당 방식의 사용으로 util이 1점 올라감을 확인할 수 있었다. find_fit 함수는 아래와 같은 단계로 진행된다.

1. 먼저, 요청된 크기(asize)와 일치하는 크기의 segregated free list를 찾는다.
2. 크기에 맞는 index를 찾은 후, 해당 리스트에서는 각 노드를 순회하며 요청된 크기 이상의 빈 free 메모리 블록을 찾는다.
3. 만약 free 메모리 블록의 크기가 정확히 요청한 크기와 일치하면, 그 블록의 포인터를 즉시 반환한다.
4. 일치하는 크기의 블록을 찾지 못한 경우, 현재까지 발견된 free 메모리 블록 중에서 요청한 크기보다는 크지만 가장 작은 크기의 블록을 선택한다.
5. 해당 블록의 포인터를 반환하며 NULL을 반환하는 경우, 요청한 크기의 free 메모리 블록을 찾을 수 없었다는 것을 의미한다.

2-3 Realloc 구현

Realloc의 수정을 통해 realloc 케이스에서 util의 성능이 높아질 수 있었다. 해당 함수는 이미 할당된 메모리 블록의 크기를 재조정하는 역할을 하며, 이 함수는 사용자로부터 원래의 메모리 블록과 변경하고자 하는 새로운 크기를 입력으로 받는다. 동작은 다음 단계를 따른다.

1. 입력된 size를 asize로 복사한다. asize는 최종적으로 요구하는 메모리 블록의 크기를 나타낸다. mm_malloc 함수와 동일하게, 만약 size가 0이라면 NULL을 반환하며, size가 DSIZE 이하인 경우에는 asize를 2*DSIZE로 설정한다. 그렇지 않은 경우에는 size에 오버헤드를 더해 DSIZE로 나눠주어 asize를 설정한다. asize는 최소 더블 워드 크기이고 더블 워드 크기의 배수가 된다.
2. 현재 메모리 블록의 크기를 확인하고, 만약 cur_size가 asize보다 크거나 같다면, 메모리 블록의 크기를 변경할 필요가 없으므로 입력으로 받은 ptr를 그대로 반환한다.
3. 현재 블록 다음에 있는 블록의 크기와 할당 상태를 확인한다. 만약 다음 블록이 빈 상태(Epilogue header)이고, 현재 블록과 다음 블록의 합이 asize보다 크거나 같은지 확인한다.
4. 두 블록을 합친 후의 남은 메모리가 0 이상인 경우, 다음 블록을 delete_node 함수로 free list에서 삭제한다.
5. 현재 블록의 헤더와 푸터를 업데이트하여 asize + remainder 크기의 할당된 블록으로 표시한다. 이후 ptr를 반환한다.
6. 만약 두 블록을 합친 후에도 asize보다 여전히 작다면, 힙을 asize - cur_size만큼 extend_heap를 통해 확장한다.
7. 현재 블록의 헤더와 푸터를 asize + remainder 크기를 가진 할당된 블록으로 표시한다. 그리고 ptr를 반환한다.
8. 위 조건이 모두 아니라면, mm_malloc 함수를 통해 asize - DSIZE 크기의 새로운

메모리 블록을 할당한다.

9. 원래 블록(ptr)의 내용을 새로운 블록(new_ptr)로 복사(memcpy)하고, 원래 블록을 해제(mm_free)한다. 이후 new_ptr를 반환한다.

이렇게 mm_realloc 함수는 메모리 블록의 크기를 기존함수에 비해 효율적으로 재조정하여 util의 performance를 향상시켰다.

2-4 Place

이 place 함수는 요청한 크기(asize)에 대한 메모리 블록을 배치하고, 남은 공간이 있다면 새로운 free 메모리 블록으로 분할한다.

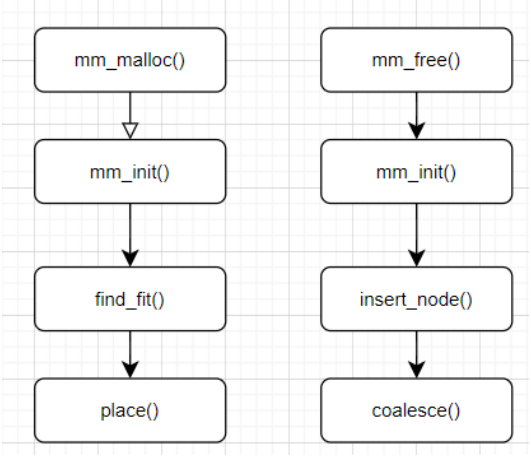
1. 이 함수는 분할할 수 있는 충분한 공간이 있는지 확인한다.
2. 새로운 프리 블록이 2*DSIZE보다 큰지 확인한다. 메모리의 최소 크기로 2*DSIZE를 생각했다. 해당 크기보다 작다면 분할하지 않고 전체 블록을 사용한다.
3. 분할이 가능한 경우, 이 함수는 요청한 크기가 64바이트(1<<6) 이상인지 확인한다. 요청된 크기가 64바이트 이상인 경우, 이 함수는 free 메모리 블록을 앞부분에 할당하고, allocate할 블록을 뒷부분에 할당한다. (64는 performance에 따라 임의로 정한 값이다.)
4. 반면 64바이트 이하라면, 앞 부분에 할당한 후 뒷 부분의 메모리 블록을 free블록으로서 insert_node 함수를 호출한다.

위 방식을 통해서 util의 성능이 좋아졌음을 확인할 수 있었다. 이런 방법은 메모리효율성과 external fragmentation 문제 해결에 도움을 준다.

이 외에도 segregated free list 관리를 위해 coalesce함수를 미세하게 수정하거나, 다른 함수에 insert / delete_node 함수의 추가부분이 있지만, 본 프로젝트에서 주요한 부분은 아니므로 생략한다.

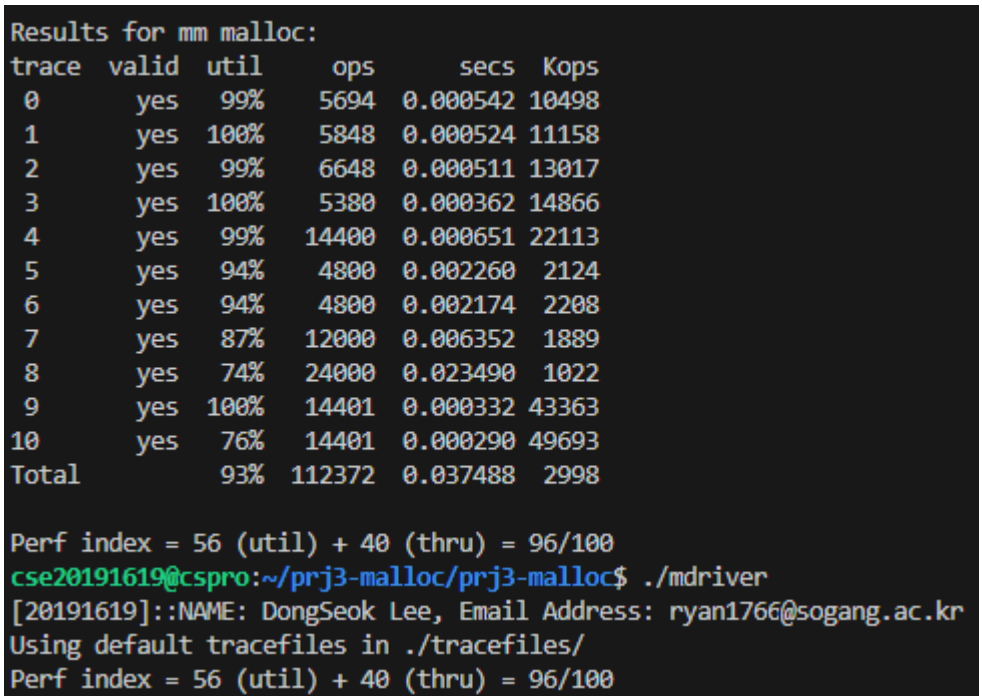
3. Flow Chart

구현한 함수의 동작 순서를 간단히 순서도로 작성하면 아래 그림과 같다.



4. 구현 결과 및 분석

작성한 코드를 mdriver -V로 확인한 performance 및 각 테스트 케이스의 정확도와 util , ops의 결과는 다음 사진과 같다.



```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.000542 10498
 1       yes  100%    5848  0.000524 11158
 2       yes   99%    6648  0.000511 13017
 3       yes  100%    5380  0.000362 14866
 4       yes   99%   14400  0.000651 22113
 5       yes   94%    4800  0.002260  2124
 6       yes   94%    4800  0.002174  2208
 7       yes   87%   12000  0.006352  1889
 8       yes   74%   24000  0.023490  1022
 9       yes  100%   14401  0.000332 43363
10       yes   76%   14401  0.000290 49693
Total          93%  112372  0.037488  2998

Perf index = 56 (util) + 40 (thru) = 96/100
cse20191619@cspro:~/prj3-malloc/prj3-malloc$ ./mdriver
[20191619]::NAME: DongSeok Lee, Email Address: ryan1766@sogang.ac.kr
Using default tracefiles in ./tracefiles/
Perf index = 56 (util) + 40 (thru) = 96/100
```

모든 테스트 케이스에서 정확도 검사를 통과하였으며, 공간 이용률은 대체로 높게 유지되었다. 특히 테스트 케이스 1, 3, 9에서는 100%의 이용률을 달성하였다

성능 지표(Perf index)는 96/100으로, util 56과 thru 40을 합한 값이다. 이는 segregated free list로 구현한 dynamic memory allocator 가 준수한 성능을 보여준다는 증거이다. 책의 베이스 코드인 implicit-list와 first fit은 54점을 보여준다. Next-fit의 경우에는 84점으로 나름 준수한 성능을 보인다.

본 코드에서 segregated free list와 first-fit을 사용할 경우 95점으로 강의자료에 나와있는 것처럼 Best-fit과의 성능차이는 거의 없다고 봐도 무방하다. 그럼에도 1점의 향상을 얻을 수 있었다.

- First-fit search of segregated free list approximates a best-fit search of entire heap
- Extreme case: If each block size is given its own size class, then it is equivalent to best-fit

Util의 경우 realloc의 개선은 realloc 테스트 케이스의 util 향상에 큰 효과를 주었으며 extend_heap에서 사용되는 값 CHUNCK_SIZE / INITCHUNKSIZE의 값의 조정은 전반적으로 util의 성능이 70%를 넘을 수 있는 효과를 주었다. 또한, 64바이트를 기준으로 메모리 할당을 나누는 방법 역시 util performance 향상에 큰 도움이 되었다.

Segregated free list의 경우 어떤 방법을 사용하던, 처리량이 40으로 나온 것을 보아 확실히 implicit list나 explicit list에 비해 빠른 속도를 보여주었다.