# 멀티코어 프로그래밍 프로젝트3 보고서

20211531 나호영

## 1. 개요

동적 할당 기능을 제공하는 malloc 패키지를 explicit list 방식으로 구현하였다.

## 2. 각 함수에 대한 설명

mm_init()
malloc 패키지를 초기화하기 위해 사용되며 비어있는 초기 힙을 생성한다. 프롤로그 헤더와 푸터, 에필로그 헤더, explicit list에서 free 상태 일 때 사용하는 prev, next를 각각의 블록에 넣었는데 처음에 prev와 next가 서로를 가르키게하여 free list를 초기화한 후 extend_heap을 사용하여 힙을 초기 확장한다.

extend_heap()
힙을 주어진 워드만큼 확장한다. 새로운 확장된 힙 영역의 블록의 헤더와 푸터를 초기화하고 coalesce 함수를 호출하여 free block들이 있으면 합친다.

coalesce()
동적 메모리가 할당되고 해제되는 과정에서 free block들이 이어지게 하는 함수이다. 앞, 뒤, 앞뒤에 free block의 존재여부를 확인하고 있으면 free block을 합치게 한다.

find_fit()
동적 메모리 할당시 알맞은 free block을 찾아준다. explicit list로 구현하였기 때문에 free block list를 순회하며 free block의 위치를 반환해준다.

place()
주어진 블록에 주어진 크기의 블록에 해당하는 메모리를 할당한다. 이 때 필요하면 분할을 진행한다.

mm_malloc()
주어진 크기의 페이로드를 가진 블록을 할당한다. malloc 패키지의 가장 핵심적인 함수로 이 함수로 메모리 동적 할당을 할 수 있다. 요청 크기를 조정하여 필요한 alignment를 고려한 뒤 빈 공간을 탐색하여 알맞은 블록을 찾고, 해당 블록에 메모리를 할당한다. 만약 공간이 없다면 extend_heap을 호출하여 힙을 확장한다.

mm_free()
할당되었던 메모리를 해제한다. 헤더와 푸터를 다시 업데이트하고 coalesce를 수행하여 빈 공간이 없도록 한다.

mm_realloc()
이미 할당되었던 블록을 재할당하는 함수이다. 요청된 크기가 0인 경우는 메모리를 해제하고, 주어진 포인터가 NULL인 경우는 메모리를 할당한다. 이 두 가지 경우가 아니면 해당하는 사이즈만큼 새로운 메모리를 할당하고 기존 데이터를 복사한 후 이전 메모리를 해제한다.

deleteBlockFromList()
free block list에서 주어진 블록을 삭제시키는 함수이다. 해당 블록의 이전 블록의 다음 블록 포인터를 해당 블록의 다음 블록 포인터로 연결하고 반대로 다음 블록의 이전 블록 포인터를 해당 블록의 이전 블록과 연결하는 방식으로 삭제한다.

addBlockToList()
free block list에 주어진 블록을 추가하는 함수이다. LIFO 방식으로 구현하였는데 이는 블록을 추가할 때 맨앞에 추가하는 방식이다. 위의 블록 삭제 함수와 비슷한 방식으로 구현되는데 free list의 처음을 가리키는 free_listp의 다음 블록을 주어진 블록으로 하게 한다.

checkheap() printblock() checkblock()
힙의 일관성을 체크하기 위한 검사를 수행하는 함수이다.

## 3. 최적화를 위해 진행했던 방법

- explicit list 방식 채택
implicit list 방식은 free block 뿐만 아니라 allocated block 까지 모두 연결하고 있기 때문에 할당할 블록을 찾을 때 할당된 블록까지 모두 순회해야 하는 단점이 존재하였다. 이를 힙 내부에서 free block들만을 연결 리스트로 구성해서 free block list로 유지하다가 할당할 때는 free block 개수 만큼만 순회하면 되기 때문에 Troughput을 개선할 수 있었다. 위의 함수 설명에서 설명했듯이 free list를 초기화할 때 prev와 next가 서로를 잇게 하는 식으로 구현하였고 블록 추가 방식은 LIFO 방식을 사용하여 블록을 추가할 때 첫 번째에 추가되도록 하였다.

- realloc 최적화
realloc의 최적화를 진행하였는데 같은 크기의 블록이 들어온다면 기존 포인터를 그대로 반환하여 메모리 복사와 malloc 호출하지 않게 하였다. 하지만 점수를 분석한 결과 큰 효과는 존재하지 않았다.

- 매크로 사용
가능하다면 함수가 아닌 매크로를 사용하여 호출 오버헤드를 줄였다. 이로써 Throughput을 개선할 수 있었으며 코드의 가독성 또한 높였다.

## 4. 결과 화면 및 분석

implicit list를 활용했을 때(50점)보다 32점이 높은 performance 점수를 얻을 수 있었다.

```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   89%    5694  0.000516 11041
 1       yes   92%    5848  0.000286 20419
 2       yes   94%    6648  0.000552 12035
 3       yes   96%    5380  0.000394 13641
 4       yes   66%   14400  0.000289 49913
 5       yes   88%    4800  0.000631  7606
 6       yes   85%    4800  0.000656  7323
 7       yes   55%   12000  0.003953  3036
 8       yes   51%   24000  0.004444  5400
 9       yes   26%   14401  0.112301   128
10       yes   29%   14401  0.003993  3606
Total          70%  112372  0.128015   878

Perf index = 42 (util) + 40 (thru) = 82/100
```