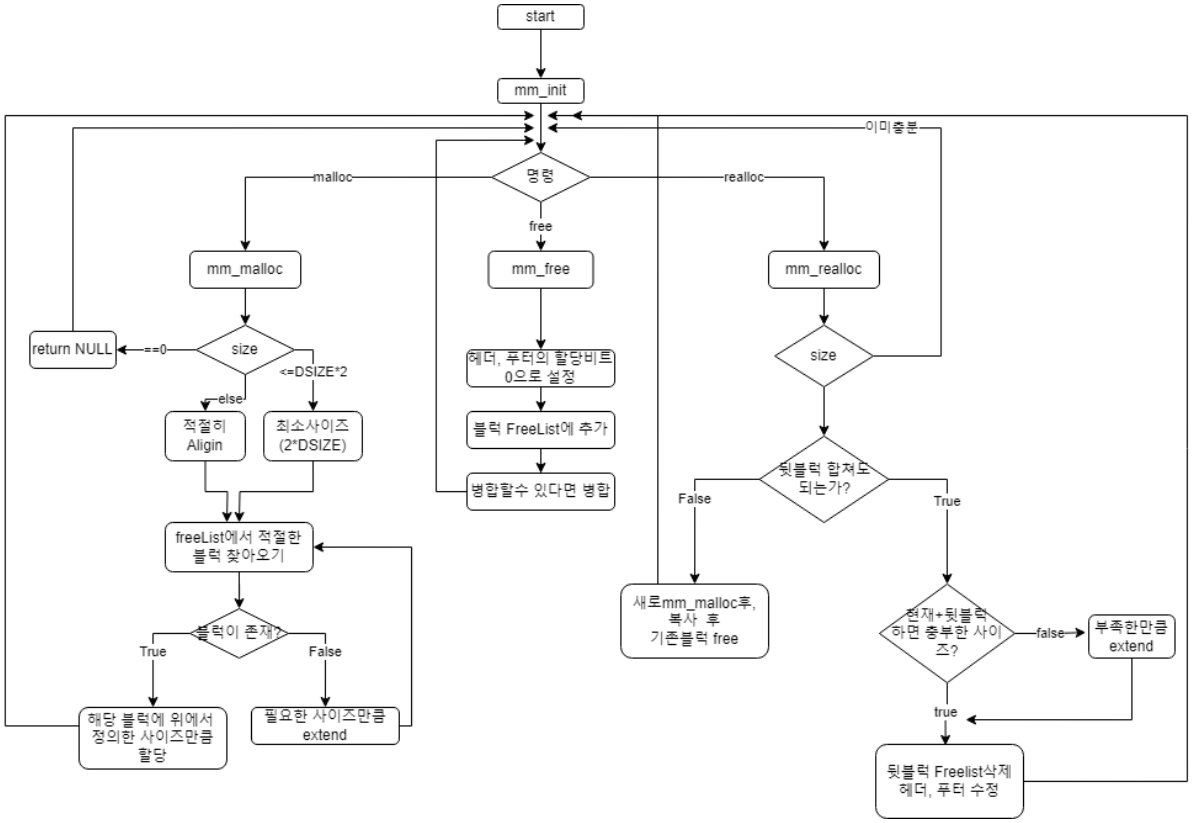
SP Project 4 Malloc

담당 교수 : 박성용

학번 : 20201558

이름 : 김명준

1. 전체적인 프로그램 설계



[그림 1] 전체 Flow chart

이번 malloc project의 전체적인 플로우 차트는 위와 같다. 우선 세가지 명령에 대한 수행이 요구된다. malloc, free, realloc이다. 우선 프로그램 시작 시, init으로 프롤로그, 에필로그 등에 대한 설정을 하고, 이 과정에서 이번 프로젝트에서 선택한 segregated list에 대한 설정도 완료해둔다. 이에 대한 설명은 2번 항목에서 자세히 설명한다. 초기설정을 한 뒤, 요구되는 명령에 맞는 동작을 실시한다.

malloc 명령에 대해서는 우선 사이즈를 과제에서 요구한 8 Alignment에 맞게 사이즈를 조작해주고, segregated list에서 가장 적절하다고 판단되는 블록을 가져온다. 이 때 만약 적절한 블록을 찾지 못했다면, 할당하고자 하는 블록만큼 heap을 늘려준 후(utilization을 위해 최소로 확장한다) 다시 적절한 블록을 찾아온다. 적절한 블록을 찾는 데 성공하면, 해당 블록에 place함수를 사용하여 정확한 사이즈만큼 할당해준 후, 해당 블록의 포인터를 반환한다.

free 명령에 대해서는 헤더와 푸터에 존재하는 할당 비트(lsb)를 '0', free상태로 설정해주고, 해당 블록을 segregated list에 다시 삽입해주며 free상태로 바꾼다. 이후 만약 블록이 다른 free 블록과 병합 가능하다면 병합해준다.

realloc 함수에 대해서는 이미 할당되어있는 블록의 사이즈를 바꾸는 기능을 수행해야한다. 만약 이미 사이즈가 충분하면, 그대로 포인터를 반환해준다

할당 사이즈가 더 커져야 한다면, 앞뒤 블록과 합쳐 메모리를 충분히 할당해줄 수 있는지 확인한다. 가능하다면, 앞뒤블록과 병합하여 해당 블록에 메모리 내용을 카피해준 후 반환한다. 만약 이게 불가능하다면, utilization은 낮아질 수 있지만 어쩔 수 없이 mm_malloc으로 새로운 블록을 할당 받은 후, 해당 블록에 기존 내용을 복사하고, 기존 블록을 mm_free로 반환시켜주어야한다.

2. 세부 구현 사항 (특정 함수, 전역변수)

전역변수 사용

init 과정에서, segregated도 함께 설정해주었는데, 이 과정에서 segregated list에 대한 접근을 용이하게 하고자 전역변수 포인터를 하나 선언했다. (void ** segFList)

이후 init과정에서 unused 블록 앞 메모리 공간을 사용하여 20개의 segFList[]의 자리를 확보해주었다. 이를 위해 sbrk 또한 기존 예제코드처럼 4개를 확보하는 것이 아닌, index 개수를 포함하여 24개를 확보해주었다.

이렇게 Free List List를 위한 메모리 공간을 힙의 맨 앞에 확보시켜 준 후, 미리 선언해둔 segFList에 해당 리스트리스트의 시작주소를 저장해주었다. 이후 각 리스트를 NULL로 초기화해주었다.

리스트 인서트 딜리트

Segregated List의 insert, delete 는 일반적인 linked list의 insert, delete와 매우 유사하게 작동한다. 다만 적절한 index를 찾는 과정이 우선적으로 수행되어져야 한다는 것이 차이가 있다. 이번 프로젝트에서 가장 작은 인덱스인 SegFList[0]에는 최대 SMALLESTSIZE (최소 블록 크기인 16으로 define) 크기의 블록까지 저장하도록 하고, 각 SegFList[i]에는 최대 SMALLESTSIZE*(2^i)사이즈까지 저장이 가능하도록, 마지막 리스트인 SegFList[19]에는 이전 리스트에서 받지 못한 모든 블록을 가능하도록 설정했다. 2의 제곱승으로 하는것

이를 위해 insert나, delete, 적절한 블록찾기 함수에서 tmpSize를 SMALLESTSIZE부터 시작하여, 반복문을 돌면서, Idx는 1씩 증가, tmpSize는 2배로 증가를 시키며 해당 사이즈보다 크거나 같은 첫번째 인덱스를 선택하게끔 설정했다. insert의 경우 반드시 그곳에 삽입

이 가능하므로 별도의 처리를 하지 않았고, delete의 경우에도 해당 리스트에 블록이 존재함을 아는 상태에서, 불러오기 때문에 별도의 처리를 하지 않았다. 적절한 블록 찾는 함수에서는 해당 리스트에 원하는 사이즈보다 큰 블록이 없다면, (예를 들어 30사이즈를 원하는데, SegFList[1]에 블록이 아예 없다던가) 그 다음 리스트에서 탐색하여 대신 찾아올 수 있게끔 처리를 해주었다. 이러한 과정으로 segFList[Idx]가 적절한 리스트임을 알 수 있다. 적절한 리스트를 찾으면, 크기순으로 정렬하기 위해 코드를 작성했다. address order와 크기 order중 고민했으나, 퍼포먼스 점수의 차이가 없었기에, first fit이 best fit이 될 수 있는 크기 순으로 삽입을 선택했다. 이후 삽입, 삭제에 대해서는 일반적 더블 링크드 리스트와 거의 동일한 작동이기 때문에 자세한 설명은 생략한다.

병합

Free상태인 리스트가 인접한 공간과 합칠 수 있다면 즉시 합치는 immediately 방식을 택했는데, 때문에 매번 free List가 insert할때마다 coalesce함수를 불러오며 병합을 할 수 있는지 체크하고, 가능하다면 수행하도록 했다.

병합은 앞뒤 블록 중 Free상태인 블록이 있는지 확인한 후, 존재한다면 해당 블록과 병합을 수행하게 했다. 병합하는 방법은 각 Free Block을 모두 FreeList에서 제거한 후, 적절한 위치에 헤더와 푸터를 설정해주고 (병합한 블록들의 가장 앞과 가장 뒤에, 사이즈는 모든 블록의 사이즈를 더해서) 다시 블록을 insert해주는 방식으로 병합을 진행했다.

place

malloc과정에서 어떤 malloc을 어떠한 free List에 존재하는 Block에 무조건 적으로 어주는 것은 utilization을 낮출 수 있다. 왜냐하면, 들어갈 수 있는 가장 작은 블록일지라도, fragmentation이 발생할 수 있다. 때문에 free block과 할당해야 할 크기의 차이가 min보다 크다면, 정확한 할당해야 할 사이즈만큼만 할당해주고, 나머지 남는 부분에 대해서는 다시 쪼개어 List에 삽입해주는 방식을 택해 fragmentation을 최소화하고자 했다

3. 퍼포먼스 높이기

위의 내용들에 대해서 수행한 후, 자잘한 디버깅을 해결한 후에는 90점의 점수를 얻을 수 있었다. Thru는 40점 만점을 받았으나, 4,7,8번에서 부족한 점수가 낮게 나와 util점수

가 낮게 잡혔다. 이를 해결하기 위해 place에서 특정 크기 이하에 대해서는 [할당] [나머지] 가 아닌, [나머지] [할당]과 같이 반대로 할당 위치를 선택하게 하여 7,8번에 대한 점수를 높였다. 또한 chunkSize나 init과정에서 extend의 사이즈를 계속해서 바꾸어 보며 점수 변화를 살펴보며 가장 높은 점수를 택할 수 있는 값으로 최종적으로 결정하며 점수를 조금 더 높였다. 하지만 이 부분에 대해서는 왜 Util이 크게 상승하는지는 정확히 알 수 없었고, 단지 실험적으로 높은 점수를 받을 수 있어서 해당 값들을 선택했다.