System Programming Project 3

담당 교수 : 김영재 교수님

이름 : 이재진

학번 : 20181671

1. 개발 목표

- 해당 프로젝트에서 구현할 내용을 간략히 서술.

해당 프로젝트에서는 libc의 malloc, free, realloc 함수의 기능을 가능한 한 효과적이고 빠르게 수행할 수 있는 나만의 동적 메모리 할당기를 구현하는 것이 목표이다. 여기서 효과적이고 빠르다는 의미는 수업 시간에 배웠듯이 memory utilization과 throughput이 모두 극대화될 수 있는 경우를 의미한다. 또한 메모리 할당기를 만드는 과정에서 포인터 연산을 많이 필요로 하기 때문에 잘못된 주소를 참조하거나 할당하는 오류를 범하기 쉬운데, 이를 방지하기 위해 디버깅을 위한 checker 함수도 구현하는 것까지를 최종적인 목표로 한다.

2. 개발 방향 / 동기 설명

수업 시간에 배운 바에 의하면 일반적으로 implicit free list에서 free block을 찾는 방법에는 first fit, next fit, best fit이 있다. 그 중 internal fragmentation을 최소화해서 궁극적으로 memory utilization을 높여줄 수 있는 방법은 best fit이지만 매번 free block을 찾을 때마다 free list를 전부 순회해야 하기 때문에 throughput이 안 좋아지는 문제가 있었다. throughput을 높이기 위해서 implicit list의 구조를 개선하여 free block들로만 이루어진 explicit free list를 사용하는 방법도 있지만, size group별로 정렬된 여러 개의 free list들을 관리하면서 internal fragmentation과 throughput 모두를 개선시킬 수 있는 segmentation list (seglist)를 구현하기로 결정하였다.

3. 개발 범위 및 내용

A. 개발 범위

- 아래 항목을 구현하였을 때의 결과를 서술

1. Segmentation List

Segmentation List는 size별 각 인덱스에 그 size에 해당하는 free list의 맨 처음 free block의 주소를 저장하고 있는 투포인터 변수로 구현한다. mm_malloc에서 메모리를 할당할 블록을 first-fit 방법으로 찾는 과정이나 mm_free에서 할당되어 있던 메모리가 해제되면서 연속적인 블록이 비할당된 (unallocated) 상태가 생길

경우 등을 생각해보자. 이런 경우들을 생각해보면 free list에 있는 블록들의 size를 업데이트하고 size group을 옮겨주거나, mm_free에서 해제된 새로운 free block을 free list에 넣어주는 과정이 필요하다. 이와 같은 기능을 위해 seg_insert_freeblock() 과 seg_remove_freeblock()을 구현하였다. 각각의 함수는 free block이 생겼을 때 그걸 가장 알맞은 size group에 넣어주거나, 알맞은 size group에서 블록을 찾을 수 있어야 한다.

2. mm_malloc & mm_free & mm_realloc 함수

mm_malloc 함수의 경우 교재에 있는 코드를 참조하여 작성하였다. 하지만 Seglist 구조로 구현할 때 주의해야 할 점은 first-fit 방식을 사용하여 가장 적합한 free block을 찾을 때 seglist에서 그 블록을 찾아야 한다는 점과, 알맞은 크기의 free block을 찾고 난 이후에 만약 할당하려는 메모리의 크기가 찾은 free block의 크기보다 작다면 splitting을 해 주어 찾은 free block을 seglist에서 제거하고 할당하고 남은 메모리 block을 seglist에 새로 insert 해주어야 한다는 점이다.

mm_free 함수의 경우에도 교재에 있는 코드를 참조하여 작성하였다. 하지만 Seglist로 free를 구현하는 경우에는 강의 때 coalescing을 해야 하는 **4가지 케이스** (1. 현재 해제되는 블록의 다음 블록이 free block / 2. 현재 해제되는 블록의 이전 블록이 free block / 3. 현재 해제되는 블록의 전후 블록이 모두 할당된 블록 / 4. 현재 해제되는 블록의 전후 블록이 모두 free blocks) 를 고려하여 합친 블록을 **seglist에 넣어 주어야 한다.**

mm_realloc 함수의 경우 단순하게 모든 경우에 요청한 size 만큼의 할당된 블록에 원래 블록의 메모리를 덮어씌우는 데서 끝나지 않도록 설계하였다. 재할당하려는 size가 원래 블록과 같은 경우, 큰 경우, 작은 경우로 경우를 나누어 기능을 구현하였다. 자세한 내용은 뒤에서 후술하겠다.

3. checker 함수

checker 함수 (mm_check()) 의 경우 말 그대로 heap에 있는 할당된 블록 및 free block 들이 valid 한지, coalescing이 미처 되지 못한 블록이 존재하는지, 그리고 seglist에 있는 free block들이 valid한지 등을 체크할 수 있도록 만든 디버깅용 함수이다.

B. 개발 내용

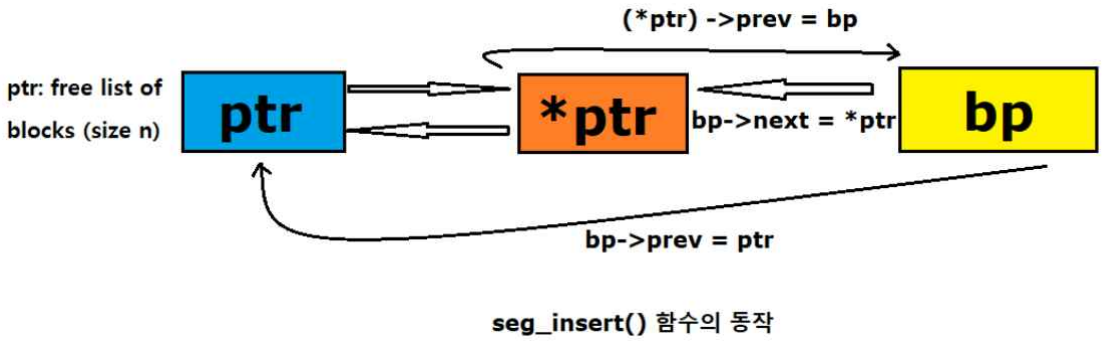
- **1. Segregated List :**

char형 투포인터 seg_listp를 선언했다. 이 때 처음에 힙을 초기화할 때 sbrk()로 힙의 프롤로그 블록 앞에 seglist를 위한 총 16워드의 메모리 공간을 확보하고, 1차원 배열처럼 각 size group에 해당하는 free list의 주소를 저장할 수 있도록 한다. 효율적인 size group들의 관리를 위해, 총 16워드의 seglist의 각 워드에는 index 0 (size $1 \sim 2^4$), index 1 (size $2^4+1 \sim 2^5$), index 2 (size $2^5+1 \sim 2^6$) .... index 15 (size $2^{18}+1 \sim 2^{19}$) 에 해당하는 free list의 시작 주소가 저장된다. 이러한 Segregated List에서 두 가지의 작업을 수행 가능하다. 첫째는 free block을 seglist에 insert하는 것이다. 그리고 둘째는 seglist에 존재하는 특정 free block을 seglist에서 remove하는 것이다.

첫번째 작업은 **seg_insert_freeblock()** 함수에서 이루어진다. 처음에는 free block이 들어갈 수 있는 가장 알맞은 size group을 찾는 과정이 필요하다. 가장 알맞은 size group을 찾기 위해서는 애초에 seglist의 free list들이 size group에 대해 정렬이 되어있기 때문에 요청된 free block의 size의 값이 최소 블록 size인 16바이트 (4 word) 보다 작아지거나 같아질 때까지 2로 계속 나눠주면 된다. 알맞은 size group을 찾았다면 이제 seglist의 주소값에 size group을 더해 줘서 해당하는 size group의 free list로 이동하고, free list가 비었는지 확인한다. 비었다면 해당 free list의 header가 새로 insert하는 free block을 가리키도록 하고, 새로 insert하는 freeblock->prev는 list의 header를 가리키게 하고, freeblock->next는 NULL을 가리키도록 한다.

만약에 free list가 비어있지 않다면 문제가 약간 더 복잡해진다. 그림을 예시로 설명하도록 하겠다. free_list에 하나의 free_block이 list header에 연결되어 있고,

그 상황에 새로운 free block의 insertion을 시도하는 상황을 가정해보자.



seg_insert() 함수의 동작

새로 들어오는 freeblock->prev가 list header를 가리키도록 한다. freeblock->next는 *ptr를 가리키도록 하는데, 여기서 *ptr은 원래 freeblock에 존재했던 free block을 가리킨다. 이렇게 하는 이유는 free list에 새로운 free block이 LIFO 방식으로 insert되기 때문이다. 그리고 *ptr->prev가 bp를 가리키도록 하면 *ptr의 이전 free block이 bp가 된다. 마지막으로 ptr이 bp를 가리키게 하면 free list에 새로운 free block의 insertion이 완료된다.

두 번째 작업은 **seg_remove_freeblock()**에서 이루어진다. 먼저 각 free block이 논리적으로는 seglist의 각 size group의 free list내에 존재하지만, **물리적으로는 그렇지 않다**는 것을 알아야 한다. 그래서 실제로 특정 size group의 free block의 이전 free block인 prv와 다음 free block인 nxt가 (실제로는 prev, next 지만 free list에서 이전 블록과 다음 블록을 나타내는 포인터인 prev, next와 혼동되므로 prv, nxt로 표기하였다.) 실제로 존재하는지 확인하려면 seglist를 제외한 힙 공간에 free block이 존재하는지 확인해야 한다. 후술하겠지만 두 가지 if 문으로 현재 제거하려는 free block에 대한 prv와 nxt의 존재성을 확인한다.

만약에 prv가 존재하지 않고 nxt 또한 존재하지 않는다면, 현재 제거하려는 free block이 free list의 유일한 블록이라는 의미가 된다. 이 때는 단순히 list header가 NULL을 가리키게 하면 된다. (free list를 empty하게 만들기 위함)

만약 prv가 존재하지 않고 nxt가 존재한다면, 현재 제거하려는 free block이 free list의 첫번째 블록이면서 다음 블록인 nxt가 존재함을 의미한다. 이 때는 list header가 현재 제거하려는 free block의 다음 블록을 가리키게 하고, 제거하려는 free block의 다음 블록인 nxt->prev가 list header를 가리키도록 하면 된다.

만약 prv가 존재하고 nxt가 존재한다면, 현재 제거하려는 free block이 free list의 중간 어딘가에 위치해 있다고 볼 수 있다. 이런 경우, prv->next가 nxt를 가리키도록 하고, nxt->prev가 prv를 가리키도록 하면 된다.

마지막으로 prv가 존재하고 nxt가 존재하지 않는다면, 현재 제거하려는 free block이 free list의 마지막 블록이 된다. 이런 경우에는 단순히 list header가 NULL을 가리키게 하면 된다.

그리고 free list에서 제거하려는 free block에 원래는 다른 free block이 연결되어 있었을 수 있으므로, 해당 연결을 제거한다.

이 두 작업 (insert, remove) 를 수행하는 함수로는 find_fit(), extend_heap(), coalesce(), place() 가 있는데, 뒤에서 더욱 자세하게 설명하도록 하겠다.

2. Heap Checker :

Heap Checker의 기능은 크게 두 가지 목적을 가지고 구현하였다. 첫 번째 목적은 힙에 정확히 어떤 블록들이 어떻게 위치해있는지를 확인하는 것이었고, 두 번째 목적은 본인이 구현한 seglist에 block들이 얼마나 효율적으로 잘 관리되고 올바르지 않은 invalid한 블록들은 없는지 확인하는 것이었다. 첫 번째 목적을 이루기 위해, 힙에 모종의 이유로 coalescing에 실패한 연속된 free block들이 있는지 체크하기 위한 기능/ 힙의 할당 블록들이 valid 한지 확인하기 위한 기능을 구현하였다.

우선 **coalescing이 제대로 됐는지 확인**하는 방법은 간단하게 구현하였다. 에필로그 블록 이후부터 힙을 순회하면서, 현재 블록이 만약에 free block이고 이전 블록과 다음 블록 중 하나라도 그 상태가 free인 블록이 있다면 올바르게 coalescing 된 블록이 아니므로 오류를 출력하도록 했다.

그 다음으로 **힙의 할당 블록들이 valid한지 확인**하는 방법은 세 가지 케이스를 고려하여 구현하였다. 우선 첫 번째로 해당 할당 블록의 header와 footer의 size bit들이 일치하지 않을 경우 오류를 출력하게 했고, 두 번째로 header와 footer의 allocation bit이 일치하지 않을 경우에도 오류를 출력하게 했다. 마지막으로 해당 할당 블록이 double-alignment (8바이트) 에 맞지 않을 경우에도 오류를 출력하도록 했다.

두 번째 목적을 이루기 위해서 두 가지 케이스를 고려하여 구현하였다. 첫 번째는 free list에 있는 free block이 valid한지 확인하는 것인데, 이는 앞에서 힙에서 확인하는 방법과 동일하다. 두 번째는 과연 모든 free block들이 free로 표기되어 있는지 확인하는 것이다. 이 경우 header의 allocation bit가 0인지 확인하고 아니면 오류를 출력하도록 하였다.

3. mm_malloc(), mm_free(), mm_realloc() :

mm_malloc의 경우 큰 틀은 교재에 있는 내용과 유사하다. 다만 교재의 경우 implicit free list에서 free block을 찾고 만약 찾는 데 실패했다면 힙의 공간을 extend_heap (더 정확히는 sbrk())을 통해 넓히고 그 공간에 place() 함수를 사용하여 블록을 할당하고, 찾는 데 성공했다면 해당 free block에 place() 함수를 통해 splitting 방식으로 블록을 할당하거나 아니면 블록을 통째로 할당하게 되는데, 본인이 구현한 함수에서는 이와 같은 과정을 seglist에서의 first-fit 방식을 사용하여 더 효율적으로 기능하도록 하였다. 우선 근본적인 방식은 비슷하지만 seglist에서의 first-fit은 implicit list에서의 best-fit의 memory utilization에 맞먹으면서 throughput은 best-fit보다 훨씬 좋다고 알려져 있다. 여기서 언급한 place(), extend_heap() 함수에서 추가한 내용은 뒤에서 후술하도록 하겠다.

mm_free의 경우에는 상당히 내용이 짧고 교재의 내용과 유사하다. 다만 다른 점은 free 된 block을 주변의 가능한 블록과 coalesce 하는 것에서 멈추지 않고 coalesce 이후 생성된 새로운 블록을 seglist에 insert하게 된다.

mm_realloc의 경우에는 기본적으로 ptr이 NULL일 때 mm_malloc(size)를 실행하고, 요청된 size가 0일 때 mm_free(ptr)를 실행하는 것까지는 동일하다. 다른 점은, 앞에서 이야기했듯이 재할당하려는 size와 원래 블록(ptr)의 size를 비교하고 그에 따른 작업을 수행한다. 우선 처음엔 재할당하려는 size를 alignment requirement에 맞게 align 시킨다. **이 align된 size가 ptr의 size보다 작다면** 요청한 size가 원래 블록의 size보다 작은 경우이므로 블록이 축소되는 경우이다. 이 경우에는 원래 블록에 요청한 size만큼의 메모리를 할당하고, 남는 블록은 unallocated state로 바꿔서 seglist에 넣는다. 단 주의해야 할 점은 seglist에 남는 블록을 넣을 수 있는 경우는 오직 남는 블록이 최소 블록 단위인 16 바이트보다 클 경우이다.

align 된 size가 ptr의 size와 동일하다면 요청한 size가 원래 블록의 size와 일치하는 경우이므로 그대로 ptr을 리턴한다.

align 된 size가 ptr의 size보다 크다면 문제가 복잡해진다. 처음 구현할 때 메모리 효율성을 높이기 위해 어떤 방법이 있을지 고민하다가 힙에서 coalesce할 수 있는 블록끼리 free block을 합칠 수 있을 것이란 생각을 했다. ptr의 이전 블록과 다음 블록을 모두 조사하는 방법은 비효율적이므로 본인이 구현한 방법에서는 다음 블록만을 조사한다. 다음 블록이 우선 존재하는지를 조사해야 한다. 이를 위해 힙에서 다음 블록이 NULL인지를 확인하면 된다. 다음 블록이 존재한다면 먼저 그 블록을 seglist에서 제거한다. 그 이전에 다음 블록과 현재 할당 블록을 합쳤을 때 전체 size가 얼마나 되는지 계산해야 하는데 이 값만큼 힙의 블록에 공간을 할당하고 ptr를 리턴하면 '확장되어' 할당된 블록이 리턴된다.

C. 개발 방법

- **B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)**

1. 포인터 연산을 간단화하기 위한 여러 가지 포인터 매크로들

이 중 대다수는 교재의 내용을 참고하여 구현하였다. 새로 구현한 매크로로는

1) #define FREE_PREV(bp) ((char *) (bp))

2) #define FREE_NEXT(bp) ((char *) (bp + WSIZE))

3) #define FREE_PREV_BLKP(bp) ((char *) GET(FREE_PREV(bp)))

4) #define FREE_NEXT_BLKP(bp) ((char *) GET(FREE_NEXT(bp)))

가 있다. 설명하면

1) seglist의 특정 size group에 해당하는 free list에서, 현재 블록의 이전 블록을 가리키는 포인터

2) seglist의 특정 size group에 해당하는 free list에서, 현재 블록의 다음 블록을

가리키는 포인터

3) seglist의 특정 size group에 해당하는 free list에서, 현재 블록의 이전 블록의 **힙에서의 위치 (참조된 위치)** 를 가리키는 포인터

4) seglist의 특정 size group에 해당하는 free list에서, 현재 블록의 다음 블록의 **힙에서의 위치 (참조된 위치)** 를 가리키는 포인터

나머지는 교재에 있는 매크로들이므로 설명을 생략하도록 하겠다.

2. char **seg_listp (seglist를 나타내는 투포인터 변수) :

앞에서 자세히 설명했으므로 설명을 생략한다.

3. static void *extend_heap(size_t words) :

메모리를 할당하면서 heap 공간이 부족할 때 확장하기 위해 구현된 함수이다. 우선 요청된 size (words) 를 double-word alignment에 맞게 align시킨다. 그리고 align 된 size만큼 sbrk()를 사용하여 힙을 확장시키고 그 size만큼의 free block을 heap에 추가한다. 그리고 그 뒤에 heap의 끝을 의미하는 에필로그 블록이 오도록 조정한다. 추가로 구현한 부분은 free block이 heap에 생성되고 coalescing이 필요할 수도 있으므로 coalescing을 하고, coalescing된 free block을 seglist에 넣는 부분이다.

4. static void *coalesce(void *bp) :

external fragmentation을 줄이기 위해 꼭 필요한 함수이다. 힙 상에서 연속된 free block들을 합쳐 하나의 free block으로 만드는 역할을 한다. 우선 강의에서 배웠던 것처럼 전후 블록의 할당 상태를 조사한다. 여기서 4가지 경우로 나뉘게 되는데, 첫번째로는 현재 블록의 전후 블록 모두가 할당된 경우이다. 이 경우 만약 mm_free()를 통해 메모리를 해제하는 경우라면 그냥 해당 블록만 unallocate 하고 끝나게 된다. 두번째 경우는 현재 블록의 이전 블록은 할당되었고, 다음 블록은 free block인 경우이다. 이 경우 다음 블록과 현재 블록을 합쳐줘야 한다. 합치기 전에 현재 블록의 다음 블록을 seglist에서 제거해주는 부분을 추가하였다.

세 번째 경우는 현재 블록의 이전 블록은 free block이고 다음 블록은 할당된 경우이다. 강의에서 배웠듯이 이 경우에는 boundary tag가 필요한데, 마침 본인의 구현은 boundary tag를 지원하고 있다. 이 경우에는 이전 블록과 현재 블록을 합쳐줘야 한다. 합치기 전에 현재 블록의 이전 블록을 seglist에서 제거해주는 부분을 추가하였다. 마지막 네 번째 경우는 이전 블록, 다음 블록 모두가 free block인 경우이다. 이 경우에는 이전 블록, 현재 블록, 다음 블록 모두를 합쳐줘야 한다. 합치기 전에 이전 블록, 현재 블록 두 개를 seglist에서 모두 제거해주는 부분을 추가하였다.

5. static void *find_fit(size_t req_size) :

이 함수는 first-fit 방식으로 seglist에서 가장 적합한 free block을 찾아주는 함수이다. 처음부터 seglist에 맞춰서 새롭게 구현한 함수이다. 먼저 seglist에서 가장 적합한 size의 free block을 찾아주는 부분은 앞에서 seg_insert_freeblock()에서의 과정과 유사하다. (size group을 찾는 부분)

이제 적합한 size group을 찾았다면 최적의 size group을 찾아야 한다. 찾은 size group에서부터 size group을 1씩 높이면서, mm_malloc 등에서 요청한 블록의 크기보다 현재 free list 안에 있는 블록의 크기가 더 크거나 같을 경우 최적의 size의 블록을 찾았으므로 그 free block을 리턴한다.

6. static void place(void *bp, size_t req_size) :

전반적인 틀을 교재에서 참고한 이 함수는 mm_malloc()에서, find_fit()에서 찾은 최적의 free block이나 extend_heap()을 통해 얻은 free block에 두 가지 경우를 고려해서 메모리를 할당한다. 첫번째 경우는 요청한 블록의 크기가 실제 할당할 전체 블록의 크기보다 작을 경우, 강의에서 들었던 바와 같이 splitting을 해 주게 된다. 정확히는 먼저 할당할 free block을 seglist에서 제거해준 다음, 요청된 size 만큼 header와 footer의 size 값을 수정해주고, free block에서 남은 부분은 seg_insert_freeblock()을 통해 seglist에 넣어주어 관리하게 된다. (이 부분을 추가하였다) 두번째 경우는 요청한 블록의 크기가 실제 할당할 전체 블록의 크기보다 큰 경우인데, 이 경우에는 할당할 free block 전체의 size만큼 통째로 할당하고, 이 free block은 이제 할당된 상태이므로 seglist에서 제거해준다. (이 부분을 추가

하였다.)

7. int mm_init(void) :

맨 처음에 힙 공간의 초기 구조를 initialize하는 함수이다. 앞에서 설명했던 seglist의 구조가 힙의 맨 앞의 16 x Word size만큼 메모리 공간을 차지하도록 설계하였고, 그 다음에 alignment requirement를 충족하기 위해 1 Word size 만큼의 padding block을 놓았다. 바로 다음에 이어지는 두 개의 워드로는 힙의 시작을 나타내는 프롤로그 header 와 footer 블록을 놓았고, 맨 처음에 힙은 비어 있으므로 바로 다음 워드에 힙의 마지막을 나타내는 에필로그 header 블록을 놓았다. 또한 프롤로그 footer 블록 뒤에 heap의 메모리 블록들 (free/allocated)이 위치해야 하므로 heap의 시작 주소인 heap_listp를 프롤로그 footer 블록과 에필로그 header 블록 사이를 가리키도록 하였다. 또한 mm_malloc과 mm_free, mm_realloc 등의 요청들이 수행되기 이전에 extend_heap() 함수를 실행하여 미리 충분한 만큼의 (정확히는 4096 바이트 = 1024 Word) free block을 reserve 해준다. 전체적으로 본인이 구현한 부분은 기본적인 seglist의 initialization이다. 나머지는 교재의 내용을 참조하였다.

8. mm_malloc(), mm_free(), mm_realloc() :

사실상 동적 할당기의 기본적인 기능을 나타내는 함수들로, 앞에서 자세히 설명했으므로 설명을 생략한다.

4. 구현 결과 및 성능 평가

- **구현 결과를 간략하게 작성하고 프로그램의 전체적인 실행 과정을 flowchart로 작성**
- **memory utilization + throughput으로 나타낸 프로그램의 전반적인 성능 평가**

사실 seglist를 이용하여 구현하기 전에 implicit list의 first-fit, next-fit 과 explicit list의 next-fit 방식으로 미리 구현을 해 보았는데, implicit list에서는 35 (memory utilization) +30 (throughput) = **65점**으로 70점도 넘기지 못했다. throughput의 향상을 꾀하고 방법을 next-fit으로 바꿔본 결과 32 + 40 = **72점**으로 memory utilization은 약간 감소한 반면 throughput은 10점이나 증가하여 70점을 넘겼다. 여기서 explicit list로 구현해본 결과 37 + 45 = **82**점으로 throughput과 memory utilization이 같이 증가하여 더 좋은 성능을 보였다. 이는 free block만을 free list에 관리하여 throughput이 개선된다는 강의 내용과 일치하는 모습을 보였다. 성능을 더욱 개선하기 위해 마지막으로 seglist를 이용하여 구현을 해 보았다.

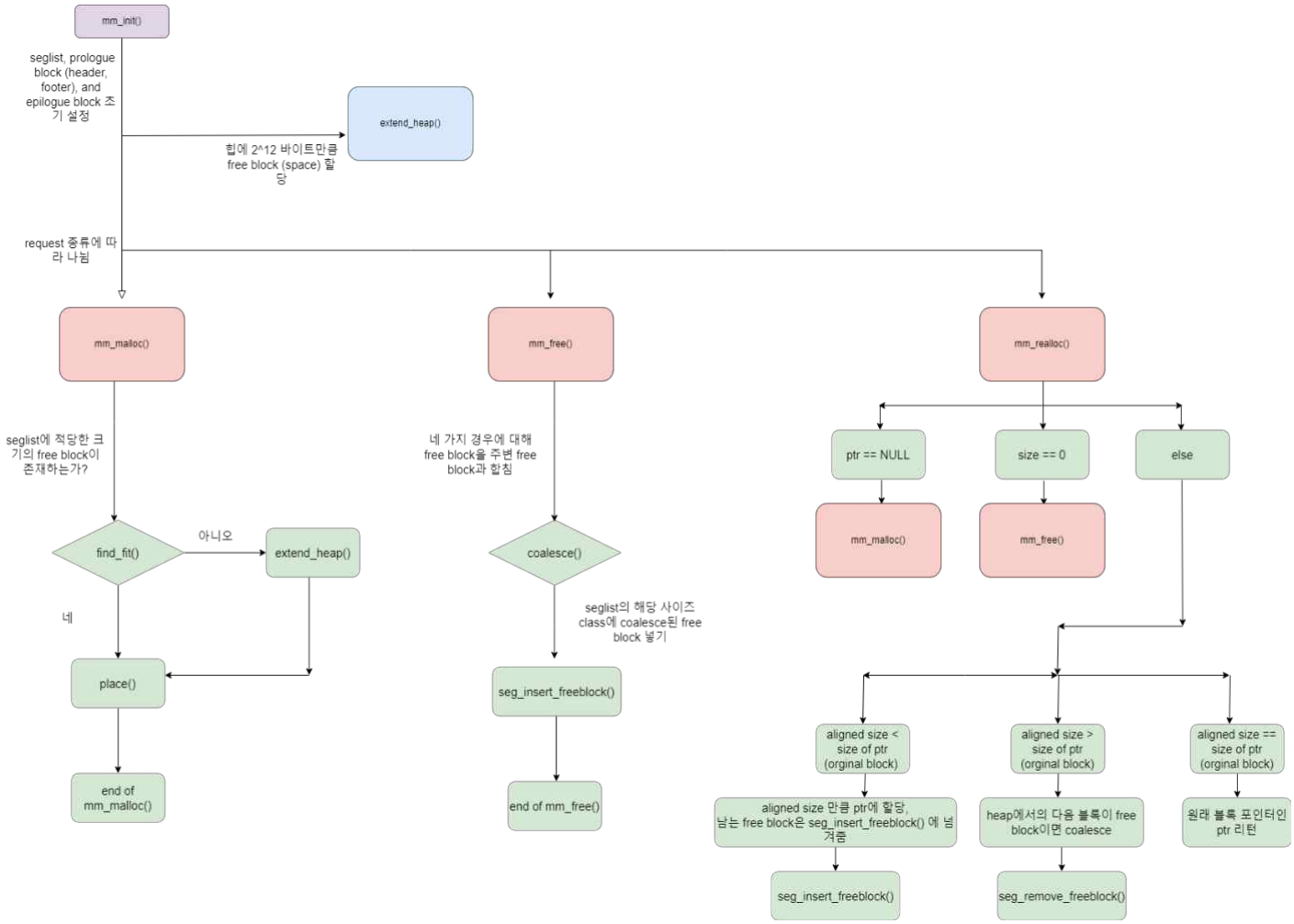
```
[20181671]::NAME: Jaejin Lee, Email Address: jlee4923@naver.com
Using default tracefiles in ./tracefiles/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   98%    5694  0.000640  8895
 1       yes   98%    5848  0.000532 10984
 2       yes   97%    6648  0.000704  9440
 3       yes   99%    5380  0.000455 11827
 4       yes   66%   14400  0.000798 18056
 5       yes   93%    4800  0.000570  8427
 6       yes   90%    4800  0.000565  8490
 7       yes   55%   12000  0.000542 22128
 8       yes   51%   24000  0.000909 26417
 9       yes   99%   14401  0.000349 41216
10       yes   62%   14401  0.000253 56921
Total          83%  112372  0.006317 17788

Perf index = 50 (util) + 40 (thru) = 90/100
cse20181671@cspro:~/test$
```

전반적으로 7, 8, 10 번의 tracefile에 대한 memory utilization은 아직 저조하지만 이전에 구현한 방법들보다 성능이 30%가량 오른 것을 알 수 있었다. 무엇보다도 전반적인 점수가 50 + 40 = **90점**으로 memory utilization이 크게 올랐고, throughput의 점수도 tradeoff가 많이 일어나지 않아 40점이라는 높은 점수를 유지함을 알 수 있었다. 이는 강의에서 segmentation list 방식이 best-fit 방식을 사용한 경우의 memory utilization에 근접하고 그와 동시에 throughput도 높은 수준을 유지한다는 설명과 일치했다.

마지막으로 전반적인 프로그램의 실행도 (flowchart) 는 아래와 같다.



5. 구현 과정에서 겪었던 어려운 점 / 애로사항 + 개선점

애로 사항

1. 복잡한 포인터 연산 :

처음에는 교재에 있는 매크로를 잘 이해하지 못하고 기본적으로 알고 있던 포인터 연산을 통해 주소를 참조했는데, 아무래도 seglist가 투포인터 형태로 선언되고 설계되었다 보니 포인터의 참조 및 수정이 까다로웠다. 다시 교재의 관련 내용을 숙지하고 seglist의 free block들을 효율적으로 관리하기 위한 매크로들을 사용하니 몇 번의 시행착오 끝에 성공적으로 seglist를 구현할 수 있었다. 특히 GET, PUT 등의 매크로를 유용하게 사용하였다. 여기에 추가로 mm_check 함수를 통해 더욱 효율적인 디버깅을 할 수 있었다.

2. 블록들을 Align하는 작업 :

가끔 메모리 할당을 하는 과정에서 memory allocator의 double-word alignment를 벗어난 블록들이 생성되어 애를 먹게 했다. 이를 고치기 위해 mm_check에서 double-word alignment가 되어 있는지 확인하는 과정을 추가하였고, 실제로 블록을 할당하거나 seglist에 새로운 free block을 추가하기 전에 그 블록들을 미리 align 하여, 메모리 할당/ 해제 과정에서 블록이 align 되어 있지 않다는 오류를 없애려고 노력하였다.

개선점

시간이 더 있었다면 아마도 seglist 말고도 강의에서 언급된 레드블랙 트리를 이용한 free list 구조를 구현해보거나 교재에서 언급된 buddy system을 구현해보았을 것이다. 이것들 또한 검증된 방법들이고 실제로도 많이 쓰이고 있는 방법들이니만큼 성능을 더 개선할 수 있었을 것이다. 또한 realloc 함수에서 요청 블록의 크기가 원래 블록의 크기보다 클 경우에 원래 블록 주변에 있는 블록들과 coalesce 함으로써 memory utilization을 높이려고 노력했는데, 본인이 구현한 함수에서는 오직 원래 블록의 다음 블록만 검사하고 있다. 이전 블록을 검사하는 부분과 / 전후 블록을 모두 검사하는 부분은 시도를 해 보았지만 (주석 처리되어 있다) 잘 되지 않아서 그만두게 되었다. throughput에 어느 정도의 영향을 미치는지는 모르겠지만 만약 throughput에 큰 지장을 주지 않는다면 memory utilization 점수를 더 높일 수 있을 것이라 생각한다.