**[LAB 12 & 13] Malloc Lab Report**

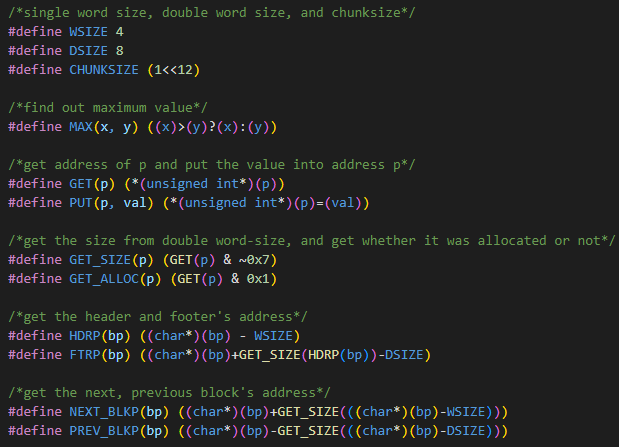
**20210643 hyunjune\_Kim**

**<Lab 수업 Summary>**

이번 Lab 시간에서는 Dynamic Memory Allocation과 Malloc에 대해 다루고, 간단한 실습을 진행하였다. Dynamic Memory Allocation은 CSED101에서 배운 것처럼, 동적으로 메모리를 필요한 만큼 할당 및 해제하여 메모리 사용을 줄이는 작동 방식이고, libc 라이브러리에 대표적인 동적 할당 함수인 malloc과 free가 들어 있다. 이번 Lab 과제에서는 malloc, realloc, free 함수를 구현해야 했는데, 신경써야 할 점은 execution speed와 memory space이다. Free memory block과 releasing memory block에 있어서 speed를 신경써야 하고, data structure overhead와 internal 및 external fragmentation을 고려해야 한다. 가장 간단한 malloc 구현 예시는 Implicit free-list algorithm인데, 이는 double linked list로 구현된다. Memory block은 header와 footer를 가지며, double word alignment이므로 LSB쪽 3비트는 항상 0인데, LSB bit을 1로 설정하여 allocated 여부를 확인한다. Malloc, free 구현은 linked list 주소를 변경해주고, LSB만 따로 다시 세팅해주면 되므로 비교적 간단하지만, Speed 및 memory Space 관점에서 다른 algorithm들에 비해 낮은 성능을 보인다. 특히, external fragmentation에 취약하다. 본 Lab 과제에서는 최대한 효율적인 malloc을 만들면 된다. 간단한 실습의 경우, libc malloc 함수의 동작을 gdb에서 간단하게 확인해보는 내용이었다.

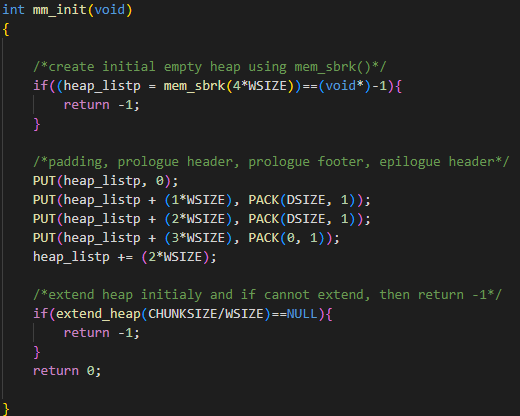
**<malloc lab>**

먼저, 최대한 효율적으로 malloc을 구현하기 위해서 가장 간단한 implicit allocator부터 구현해보는 것이 좋을 것으로 생각하였다. 교과서에 코드도 설명이 나와있고, 구현이 비교적 간단하기 때문에 이를 구현해보면 감을 잡을 수 있지 않을까 생각하였다. 하지만 implicit allocator는 효율이 떨어지고 fragmentation 발생 가능성이 있으므로 먼저 구현을 마친 후 implicit을 기반으로 explicit, segregated list 방식을 구현해보려고 계획하였다. Segregated list Allocator, best-fit에 가깝게 구현을 성공한다면 충분히 효율적인 allocator를 만들 수 있을 것으로 생각하였다. 먼저 implicit allocator 구현을 시작하였다. 교과서를 참고하여 필요한 macro와 상수들을 #define으로 선언해 주었다.

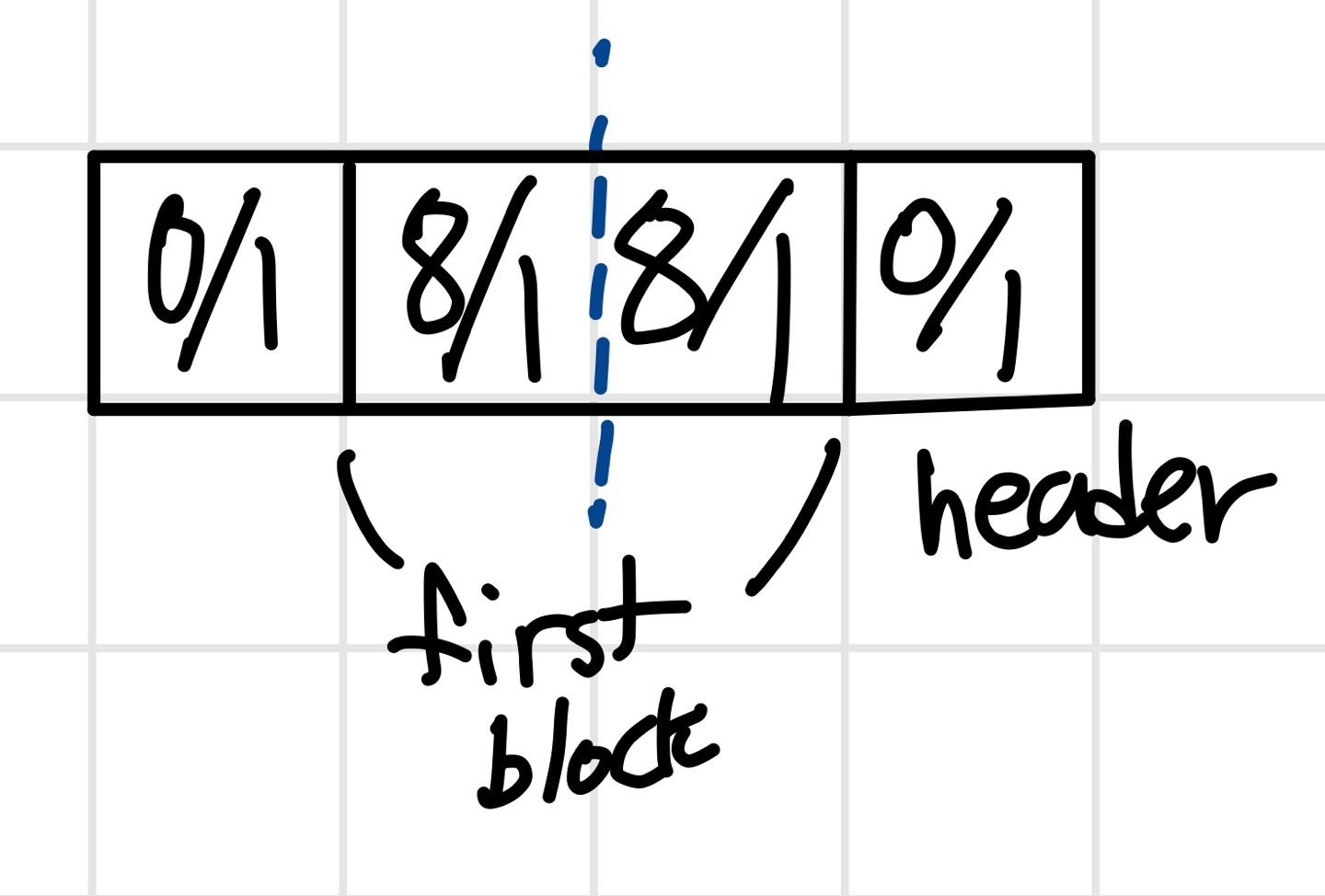
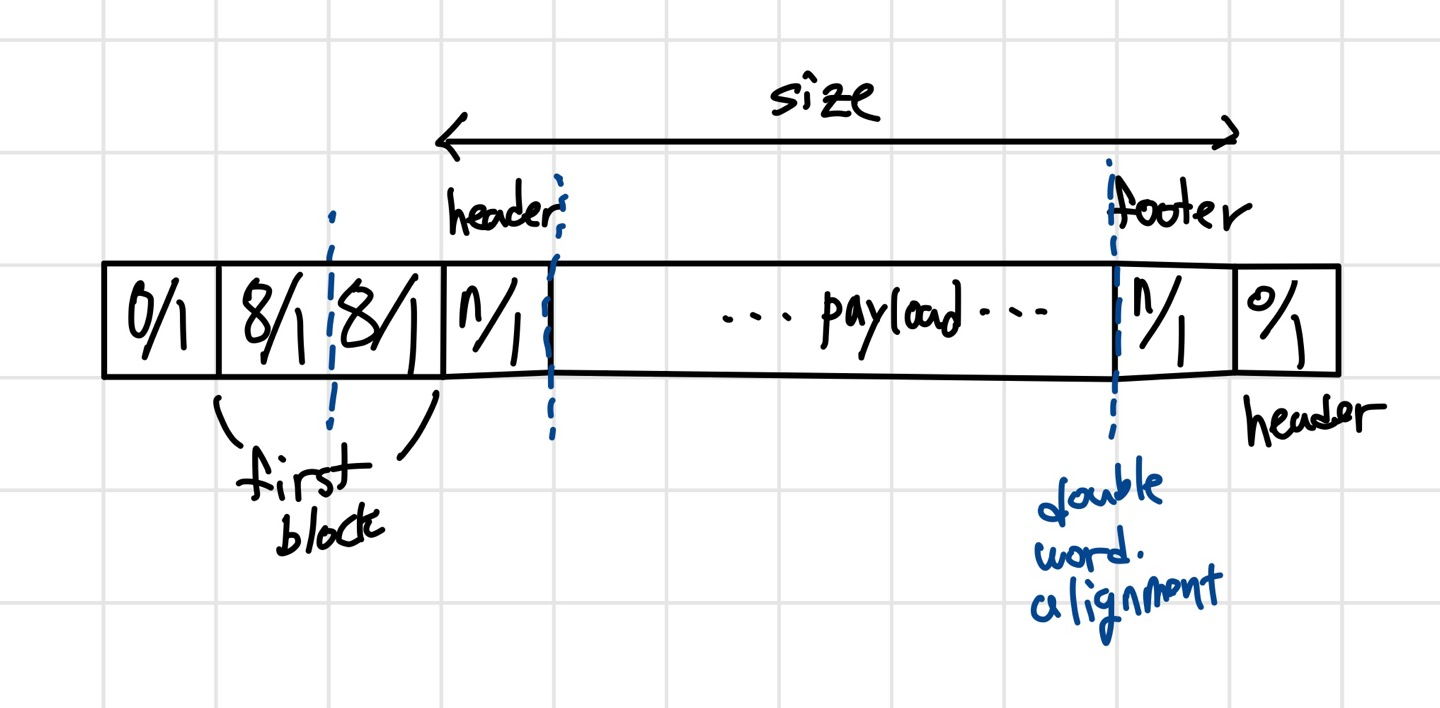


Word size들 정보와 삼항 연산자로 max값을 구하는 macro와, GET과 PUT macro를 선언해주었다. 그리고 header 또는 footer에 해당 block의 size와 allocated 여부가 저장되어 있으므로, 해당 값들을 받아오는 GET\_SIZE와 GET\_ALLOC도 선언해 주었다. HDRP는 header의 pointer를 return하도록 하여 bp가 payload의 시작점을 가리킬 것이므로 word size만큼 빼 주어 구하였고, FTRP의 경우 bp에 size만큼 더하면 block의 맨 끝부분의 주소+4이므로 double word size를 빼 주어 footer의 주소를 구하도록 하였다. 그리고 PREV\_BLKP과 NEXT\_BLKP는 각각 이전 블록의 footer와 현재 블록의 header를 체크하여 size를 구하고, 해당 size만큼 GET\_SIZE로 구해서 빼 주어 이전 block payload의 시작 주소와 다음 block payload의 시작 주소를 각각 구하였다.

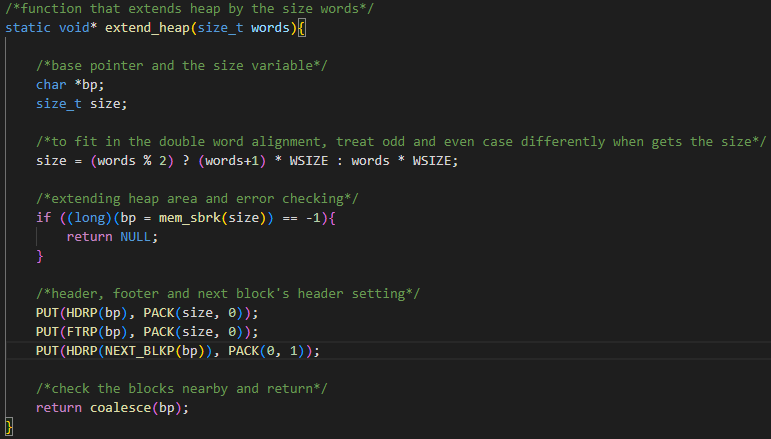
추가로, 여기까지는 교과서에도 있는 내용이지만, 일단 implicit, first-fit을 기반으로 최종 단계인 segregated list, best-fit으로 넘어갈 것이기 때문에, 이를 구현하기 위해 추가로 macro와 상수들을 어떤 식으로 선언할 지 조금씩 염두에 두기 시작했다. Segregated-list algorithm의 경우 첫 번째 block에 각 block의 size별로 linked list의 시작 pointer가 있을 것이므로, 그것들을 바로 꺼내 쓸 수 있는 Macro가 필요할 것으로 생각하였다. 몇 가지 우선 생각만 해 두고 initialize 부분으로 넘어갔다.



Mm\_init같은 경우 먼저 위와 같이 작성하였고, 먼저 맨 앞의 block에서 header와 footer를 적절히 넣어주고, 마지막 부분에 새롭게 block이 이어져 시작될 header를 넣어주었다. 그리고 extend\_heap 함수를 호출하여 초기 CHUNKSIZE/WSIZE만큼의 공간의 heap을 확장하도록 하였다.

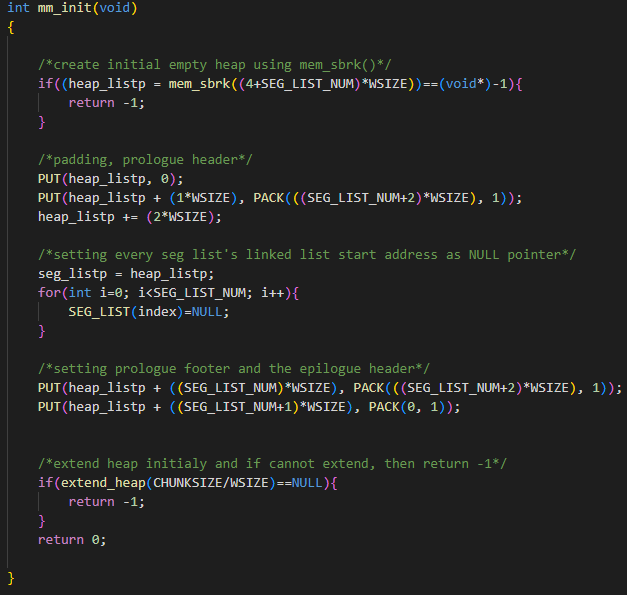
즉 extend\_heap 전까지는 왼쪽 그림 상태이고, extend 후에는 오른쪽 그림과 같은 상태가 되도록 하였다. 이를 위해 extend\_heap 함수를 아래와 같이 구현하였다.



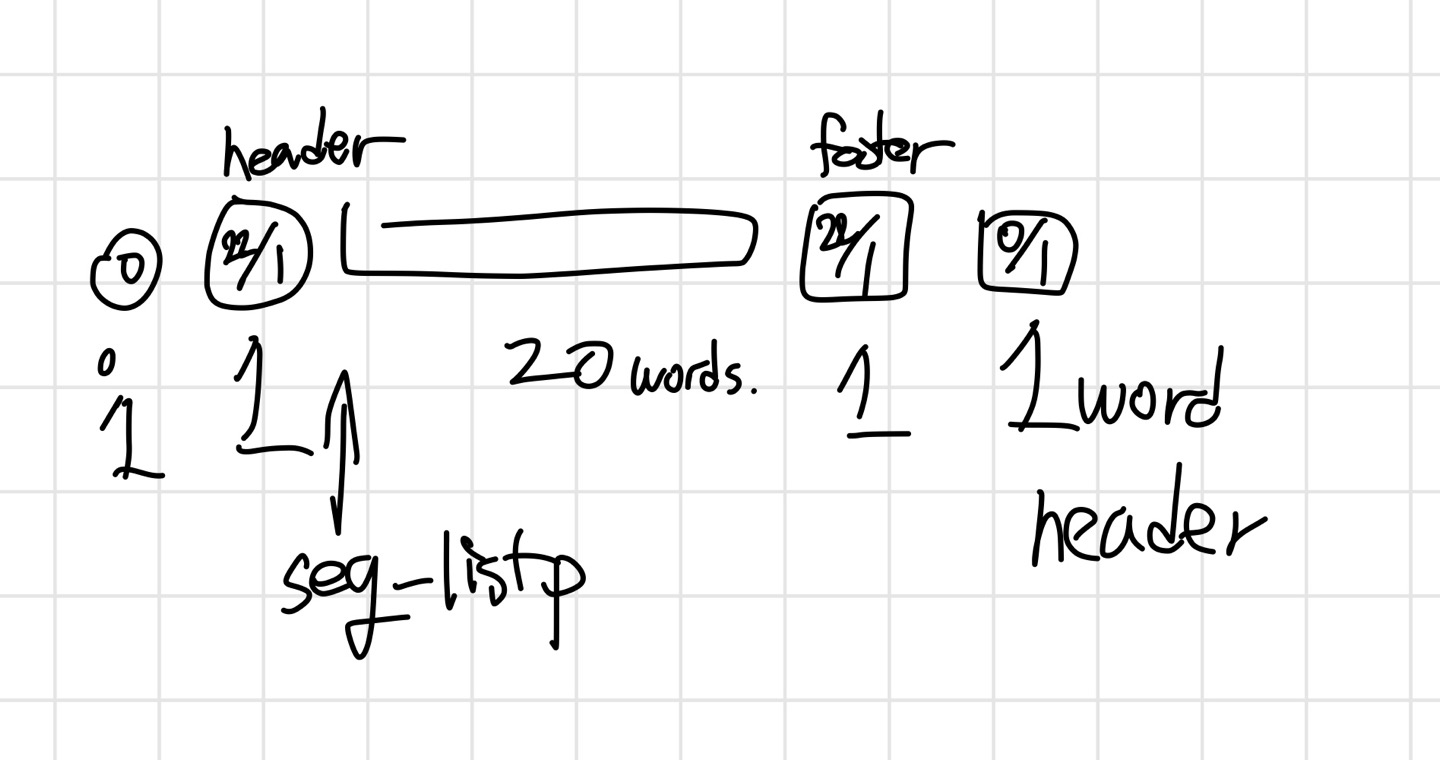
Double word alignment 만족을 위해 size를 홀수와 짝수 때 나눠서 구하였고, sbrk함수로 heap을 확장하고, header, footer, 다음 블록의 header를 각각 다시 세팅하였다. 그리고 coalesce함수를 호출하여 이전, 다음 블록 체크를 해 주고 리턴하도록 하였다.

그런데, segregated list를 만들기 위해선, 첫 block 부분에 각 범위에 맞은 seg list들의 linked list 시작 주소들이 포함되어야 하므로, 이 부분을 구현해보려고 하였다. 따라서 먼저 seg\_listp 전역 변수를 선언하여 list 시작 포인터를 가진 첫 블록의 payload 시작 지점을 가리키도록 구현하고, SEG\_LIST macro를 만들어 index에 따라 seg list의 주소를 얻어내는 것을 구현하였다. SEG\_LIST\_NUM은 20으로 설정하여 20개의 list를 만들어보려고 하였다. 그리고 init 함수를 아래와 같이 수정하였다.

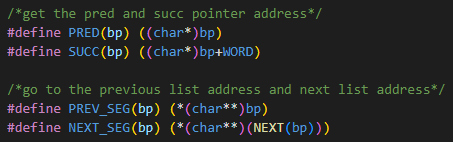




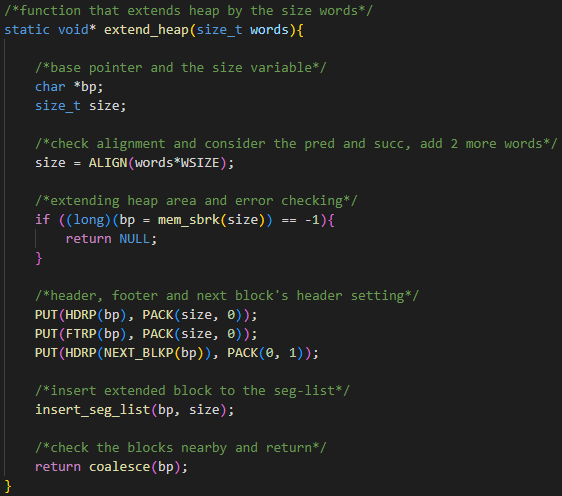
이 함수로 아래 그림과 같이 구현이 될 것으로 예상하였다.



다음으로, seg list 구현을 위해서는 pred와 succ포인터도 필요하므로, extend\_heap도 수정하여 pred, succ포인터를 넣어주었고, pred와 succ포인터를 구하는 macro도 새로 만들어주었다. 추가로, pred, succ 포인터로 탐색이 진행될 것이므로, linked list에서 이전 블록의 address와 이후 블록의 address를 찾는 macro도 만들었다.

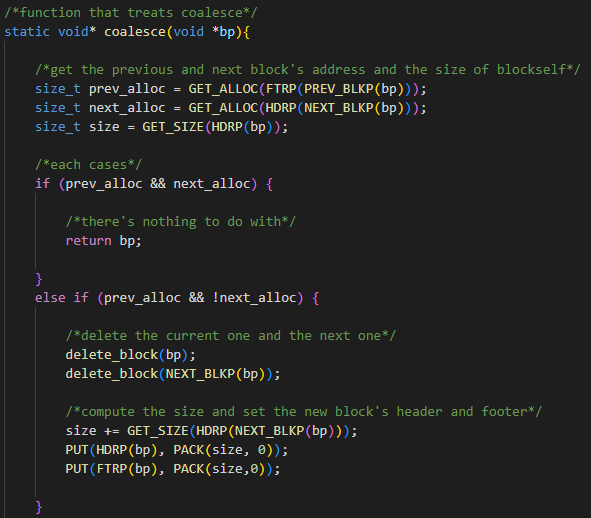


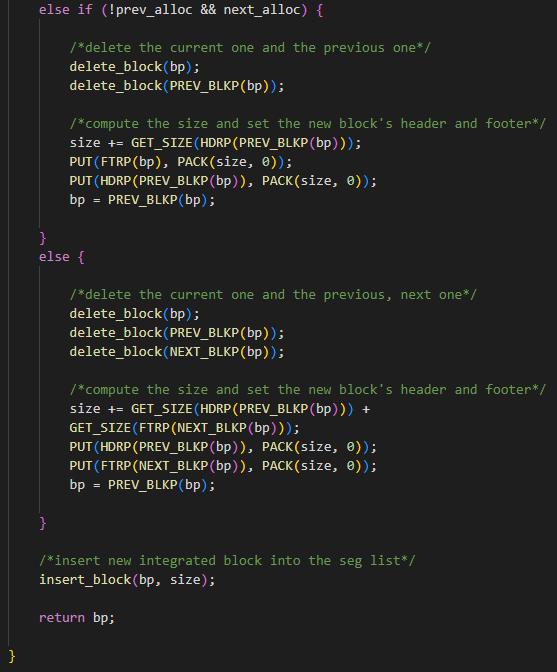
그런데 extend\_heap을 수정하다 보니, extend\_heap 내에서 pred와 succ들을 고려하는 것보다 다른 함수에서 처리하는 것이 나을 것으로 생각하였고, ALIGN의 존재를 알게 되어 아래와 같이 구현하였다.



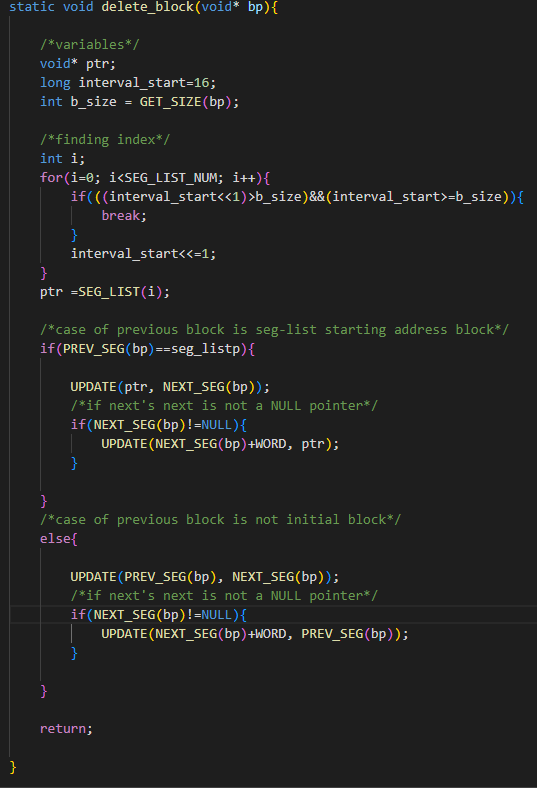
여기서 새롭게 insert\_seg\_list 함수를 적어 놓았는데, 이 함수가 새로 확장된 block을 seg list에 연결함과 동시에 pred, succ 세팅도 해주는 것으로 구현해보려고 계획하였다.

다음으로, coalesce 함수를 교과서를 참고하여 작성하였다. 교과서에서는 implicit list이므로 구현할 필요가 없었지만, 이제는 coalesce 작업을 해줄 때, 만약 free block을 합쳐 더 큰 block을 만들게 되는 경우라면 seg list에도 변화가 필요했다. 따라서 delete\_block()함수로 seg list에서 block을 삭제하고, 다시 insert\_seg\_list 함수를 이용하여 새로운 size대로 insert하는 구현을 해보았다.

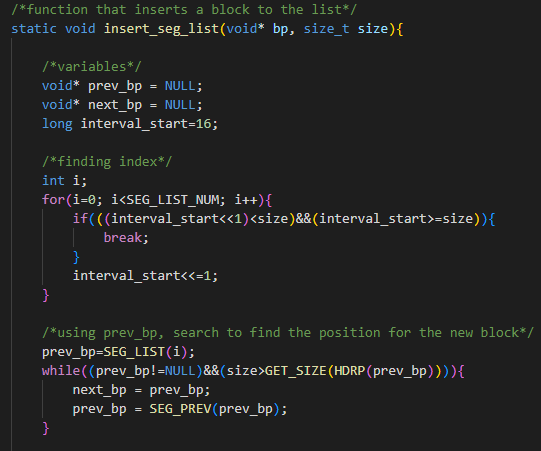


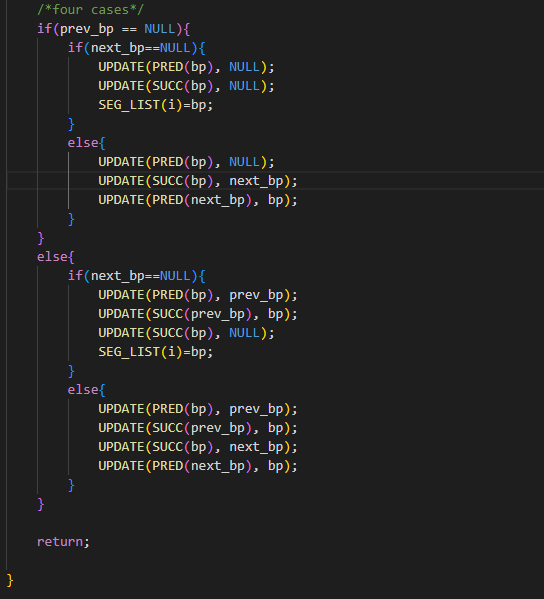


다음으로, insert\_seg\_list 함수 구현은 난관이 예상되어, 상대적으로 쉬워 보이는 delete\_block함수를 구현해보았다. Insert가 별다른 오류 없이 실행되었다는 가정 하에, delete\_block은 linked list의 이전 블록과 이후 블록을 연결해주기만 하면 될 것으로 생각하였다. 그런데 문제는, PREV\_SEG로 list의 첫 주소는 구할 수가 없다는 점이었다. 따라서 먼저 index를 구해야 했다. 한 block의 사이즈는 header, pred, succ, footer를 포함하므로 16byte 이상이므로, i=0의 index는 16~32byte 범위에 대응되고, i=1은 32~64byte 범위에 대응되고, i=19까지 총 20개가 대응되게 된다. 따라서 반복문을 돌려서 index를 찾아주었다. 그러고 나서 next block의 next가 NULL인지, previous block이 initial block인지를 따져서 포인터를 update해 주었다. 이 때 필요한 UPDATE macro도 따로 선언해주었다. 결국 delete\_block 구현은 아래와 같이 하였다. Delete\_block 함수 같은 경우에는 index를 탐색하며 위치를 찾아주어야 하는 부분이 있으므로 O(n)의 complexity가 소요될 것이라고 생각하였다.

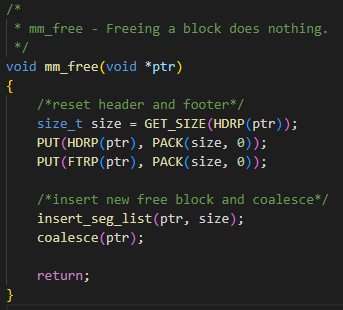


다음으로 insert\_seg\_list 함수를 구현해보았다. 이 함수는 list에서 어디 사이로 들어갈지 위치를 찾아서 들어가고, 그에 따라 pred와 succ 포인터를 적절히 설정해주면 되는 함수였다. Insert 함수도 delete 때와 마찬가지로 먼저 list에서 index를 찾아주었고, ascending order를 따라 들어갈 위치를 찾아서 pred와 succ 값들을 수정하는 형식으로 작성하였다. 이를 위해서 먼저 들어갈 자리를 찾아서 prev\_bp를 설정해준 다음에, prev\_bp와 next\_bp의 NULL 여부를 따져 총 4개 경우에 대해 각각 따로 나누어 UPDATE 처리를 해 주었다. 그 결과 아래와 같은 코드를 작성하였다.

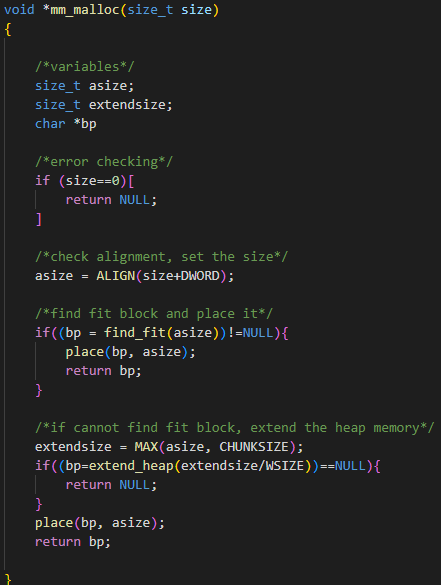




다음으로는 이제 free 함수를 작성해보았다. Free는 header와 footer의 allocated 확인 bit을 0으로 바꾸고, coalesce를 호출해주면 되기 때문에, 간단하게 구현할 수 있을 것을 생각하였다. 따라서 교과서의 code처럼 작성하였고, free를 해 주면 seg-list에 들어가야 하기 때문에 insert\_seg\_list 함수 또한 호출해주었다. 따라서 아래와 같은 코드를 작성하였다. 추가로, insert를 coalesce 이전에 해준 것은 coalesce에서 delete 후 합쳐져서 insert하는 부분이 있기 때문이었다.

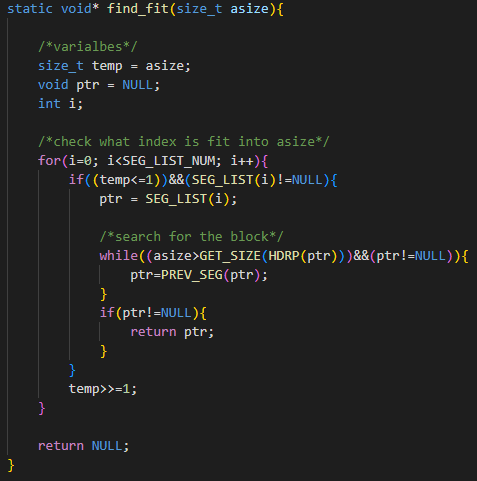


이제, 가장 메인으로 볼 수 있는 malloc을 구현해 보려고 하였다. malloc같은 경우에는, size에 따라 어느 범위의 list를 search할지 고르고, 해당 list를 search하였는데 맞는 block이 없다면, 더 큰 범위의 index로 증가를 하다가 그래도 못 찾으면 expend\_heap으로 heap을 새로 할당해 allocate하는 방식으로 작동하도록 하면 될 것으로 생각하였다. 또한, 적절한 block을 찾았을 때, 앞에서부터 할당을 하고, 뒤에 남은 block은 새로운 free block으로 list에 연결하도록 하였다. 다만 여기서 궁금했던 점은 만약 남은 block이 16보다 작은 size이면 어떻게 할 것인지였다. 또한, 교과서를 참고하니, 추가로 place함수와 free\_fit함수를 구현해야 할 것으로 보였다.



교과서를 참고하여 malloc 함수 코드를 작성하고, ALIGN을 이용하여 asize 계산 부분을 단순하게 바꾸었다. 나머지는 딱히 바꿀 만한 내용이 없고, find\_fit 함수와 place함수를 구현해주면 될 것으로 생각되었다. Malloc code는 위와 같다.

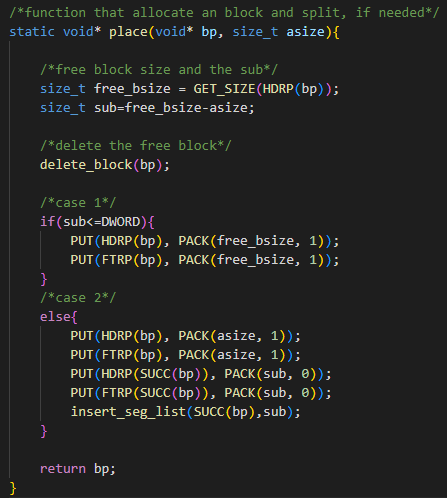
둘 중 먼저 find\_fit 함수를 구현하였다. Find\_fit은 first-fit 방식으로, asize의 크기를 보고 어느 index의 linked list를 탐색할지 정한 다음, 탐색해보고 없으면 다음 index(더 큰 index)에서 찾아보는 것을 반복하고 하나도 없다면 NULL을 리턴하도록 구현하였다. Find\_fit 함수는 list를 따라 탐색을 하며 해당 list에서 출발하는 linked list 또한 탐색하여야 한다. 따라서 O(n+N)의 complexity를 가질 것으로 생가각하였다.



이후, 지금까지 place함수를 작성하기 앞서서, 지금까지 작성한 코드가 컴파일은 잘 되는지 확인해보기 위해 terminal에 make를 입력하였다. 그 결과 상당히 많은 error, note, warning 메시지가 출력되었고, 이를 하나씩 해결하였다. 오타가 몇 개 있었고, 가장 큰 오류는 PACK을 define하지 않았던 것에서 비롯된 것이었는데, 제대로 선언해주어 바로잡았다.

다만 make를 해줄 때 mdriver에서도 warning이 뜨기도 하고, 몇 가지 코드를 수정했을 때 안 뜨기도 했는데, 이와 관련해서는 최종 완성본에서는 안 뜨게 할 수 있도록 해야겠다고 생각하였다. 최종본에서 다시 확인해보도록 하고 나중에 분석해보고자 하였다.

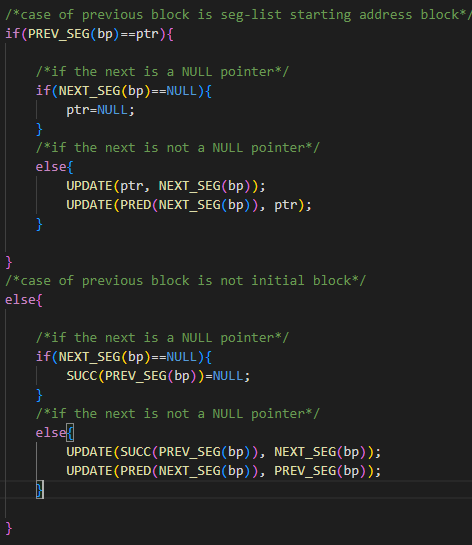
다음으로 place함수를 구현하였다. place함수는 find\_fit으로 알맞은 free block을 찾아 주었다면, 해당 block을 size만큼 allocated 상태로 바꾸고, 나머지 부분은 free block으로 남겨 다시 list에 insert하는 것이다. 이때, free block의 size와 할당할 size의 차가 8이하이면, 그냥 두 block을 대응시켰고, 차가 16이상인 경우(ALIGN하여서 8~16 사이 값은 존재하지 않으므로)는 free block을 쪼개도록 하였다. 왜냐하면 free block의 경우 header, footer, pred, succ 4개의 word가 필요하므로, 16byte가 최소 크기이기 때문이다. 여기서 어쩔 수 없이 internal fragmentation이 발생하게 된다. Place 함수는 아래와 같이 구현하였다.



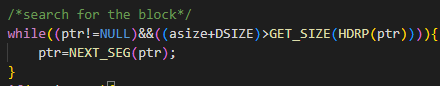
이렇게 마지막으로 realloc만 구현하면 되는 상황이 되었는데, realloc 구현에 앞서, malloc과 free가 잘 돌아가는지 확인하기 위해, test를 진행하였다. 그 결과 segmentation fault가 나와 디버깅이 필요하였다. 이에, 몇 가지 잘못된 부분을 수정하였는데, insert\_seg\_list함수와 find\_fit 함수를 일부 수정하였다.

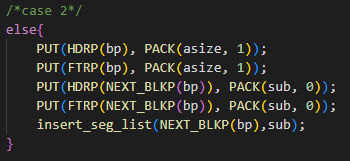


또한, mdriver로 지금까지 구현된 allocator를 실행해보고, 어디서 segmentation fault가 나오는지 확인해본 결과 delete\_block 함수가 문제인 것으로 생각되었다. 함수에서 address 사용 관련해서 잘못 작성된 부분이 있어서 고쳐주었다.

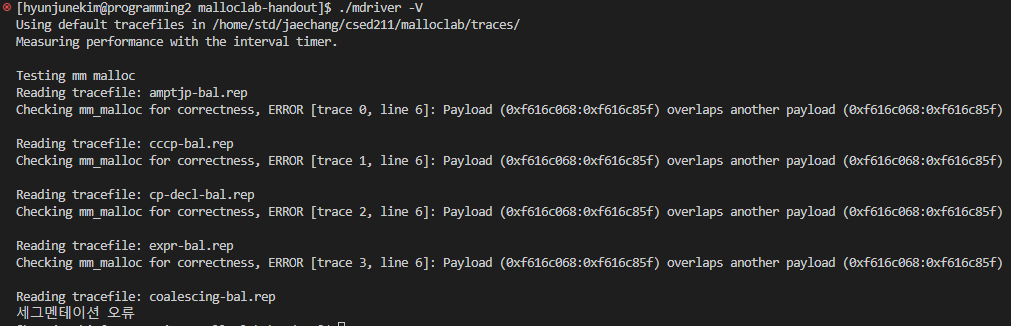


추가로, find\_fit에서 segmentation fault를 발견하였고, while문에서 ptr의 null 여부를 먼저 검사하지 않은 문제여서 둘의 순서를 바꾸어 주었다. 그리고, NEXT\_BLKP가 들어갈 자리에 SUCC가 잘못 들어간 것도 있어 변경해주었다.

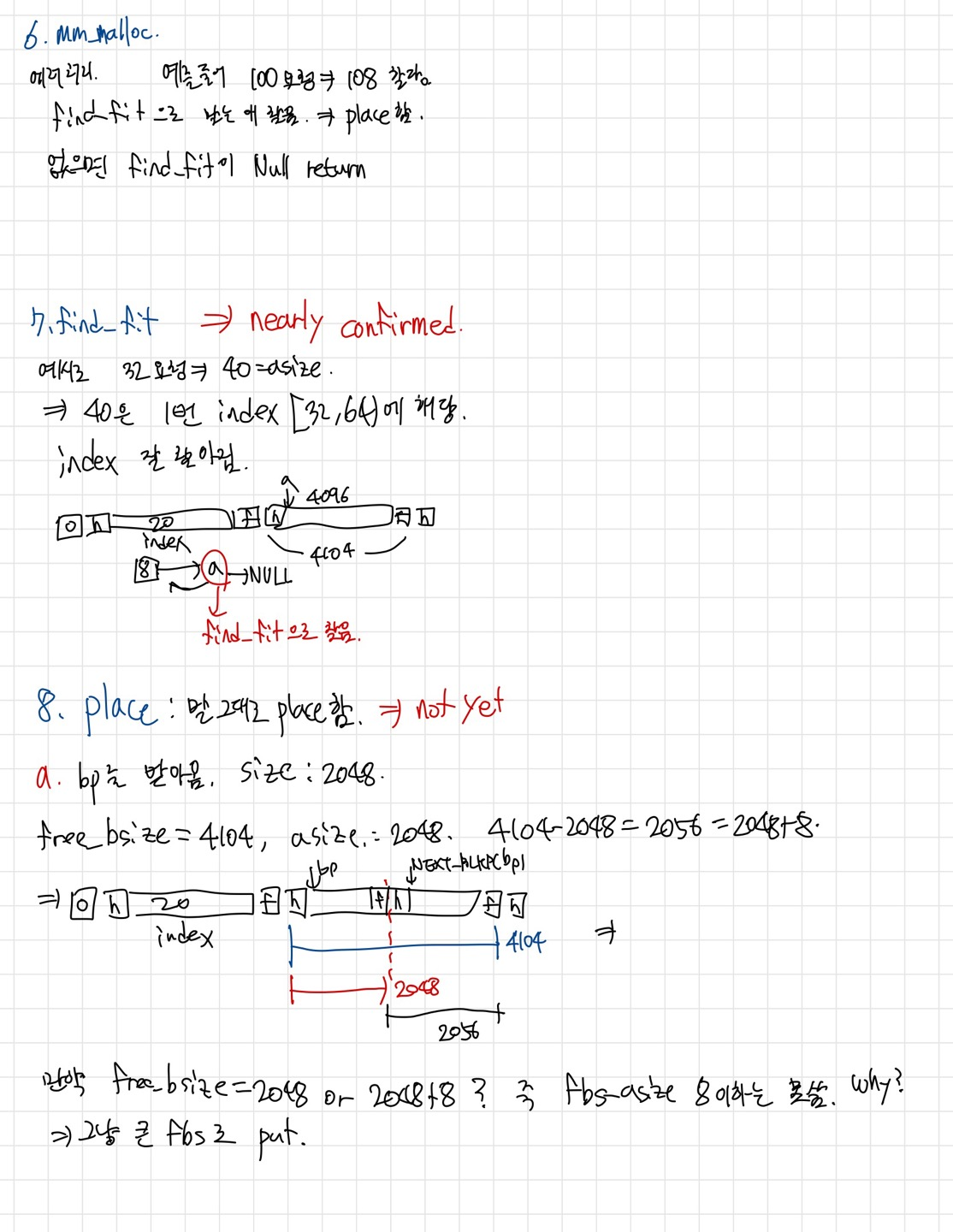
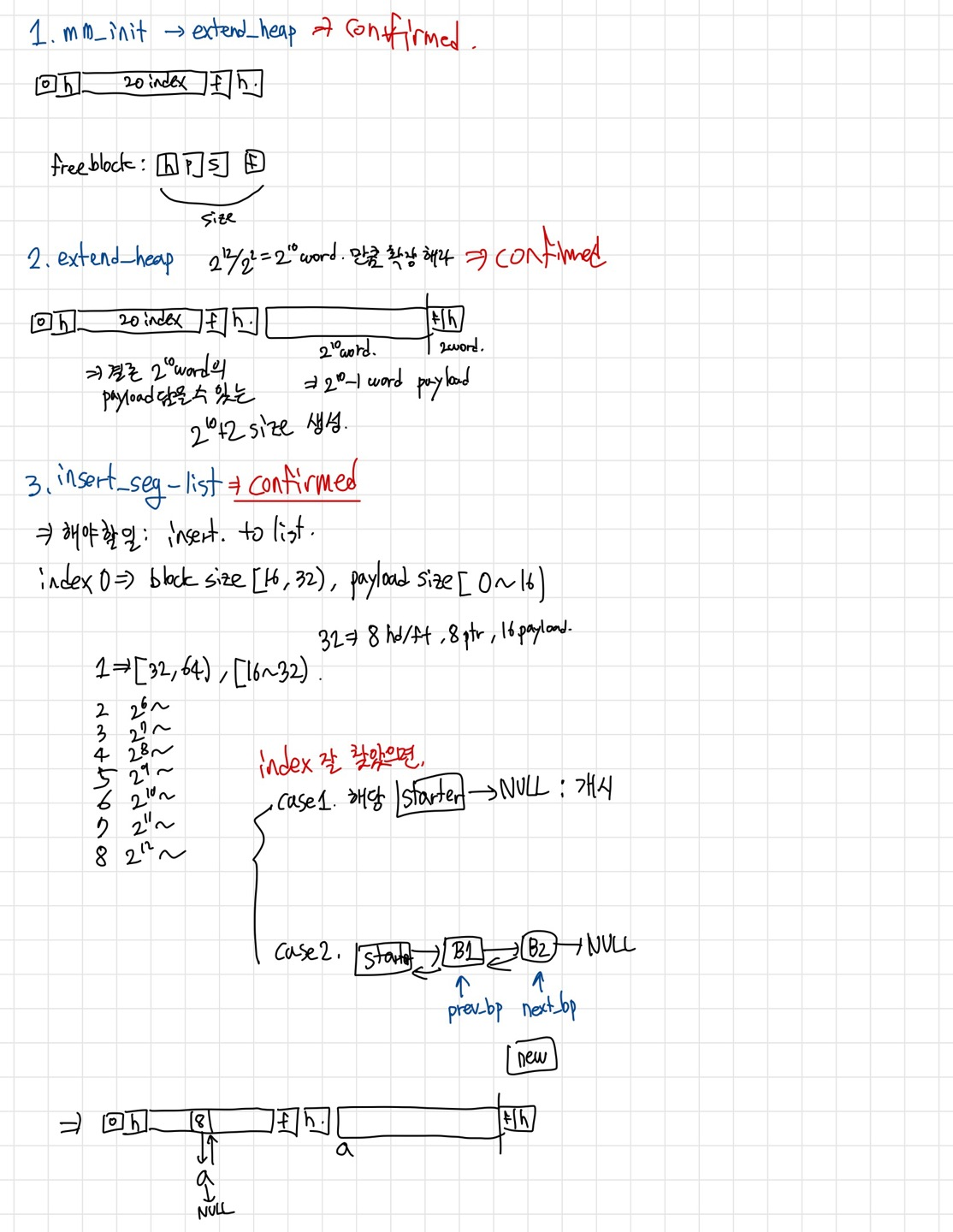


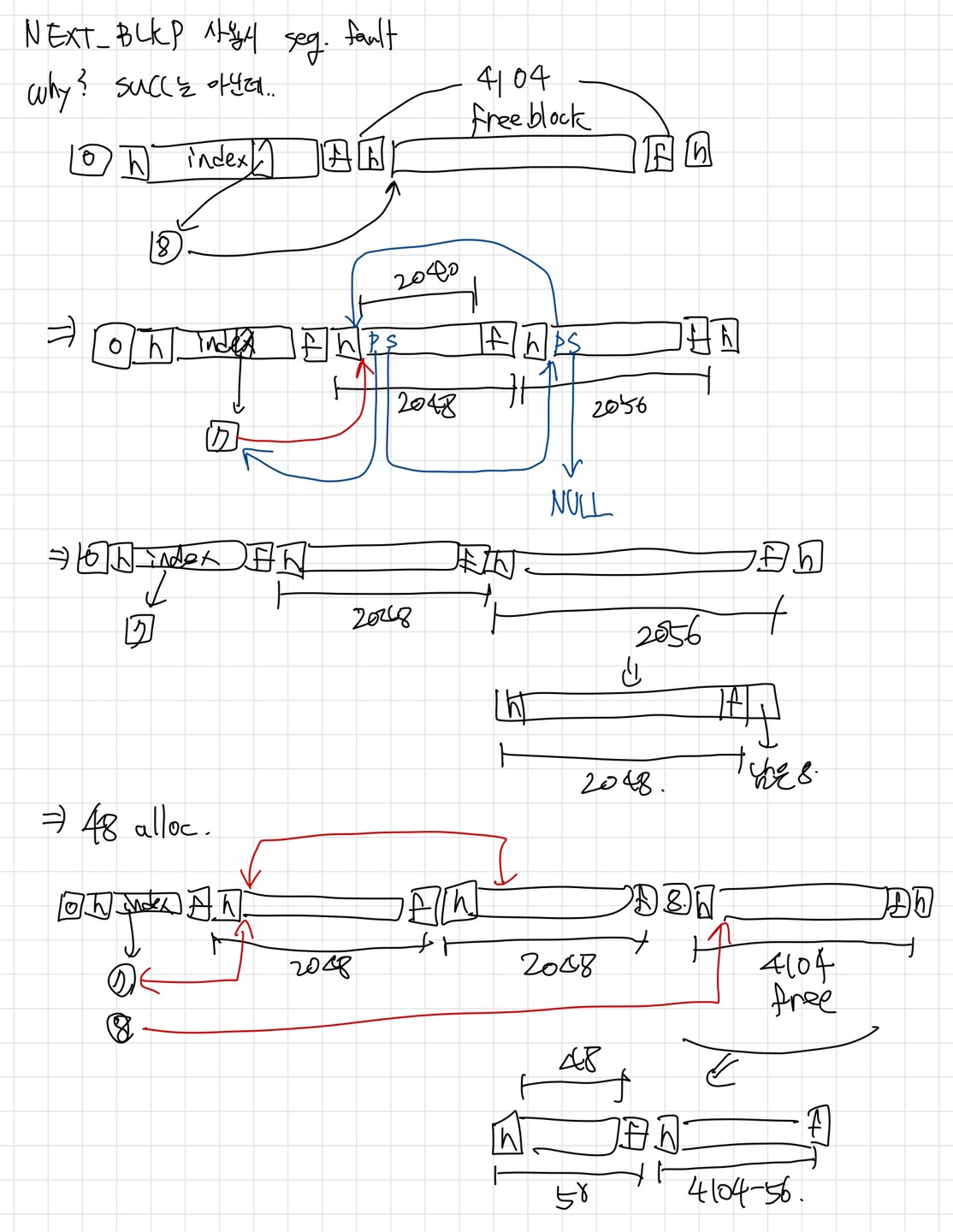
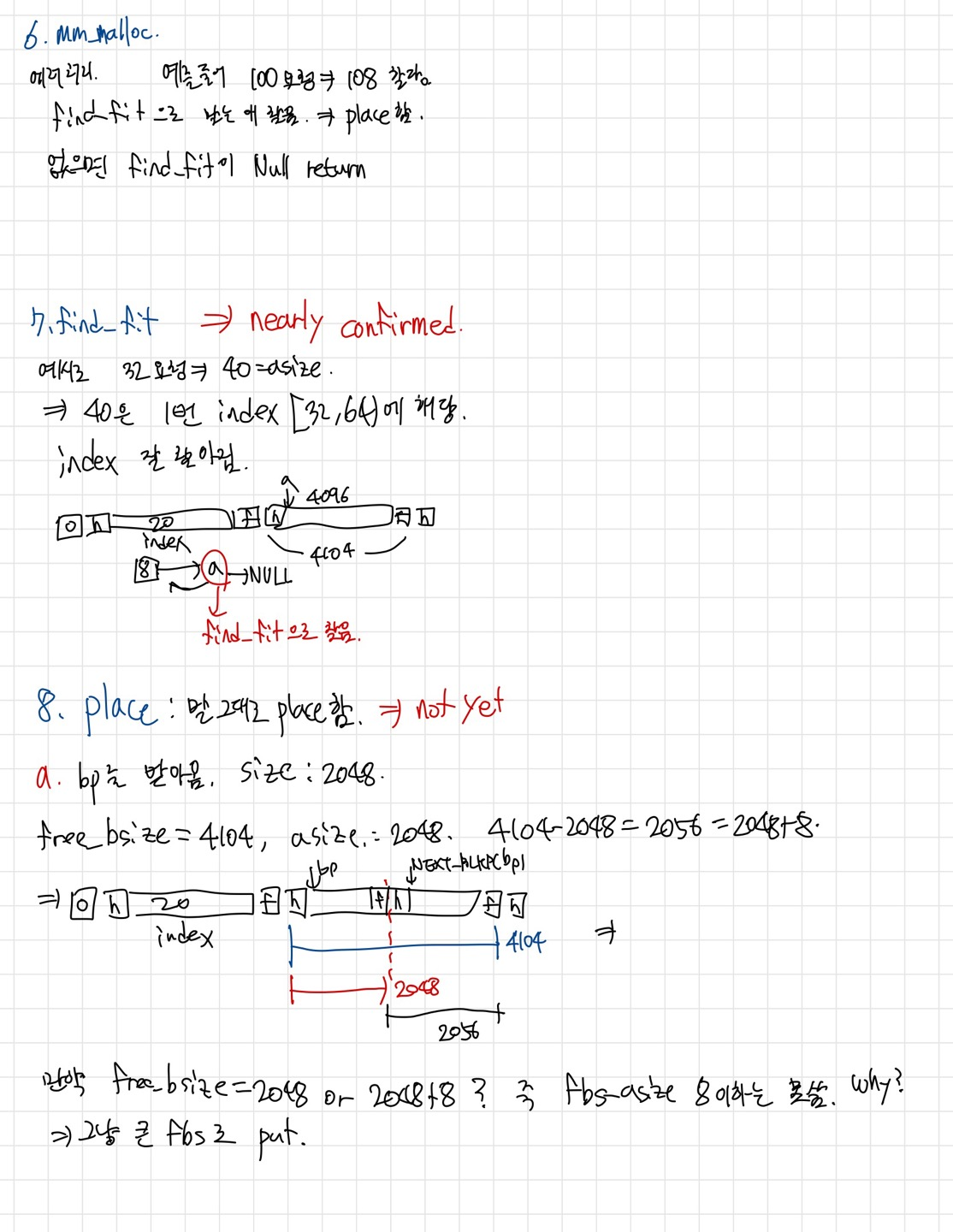


이제 segmentation 오류는 거의 해결된 것인가 싶었는데, 이제 error message들이 나오기 시작했다. 따라서 다시 한 번 코드를 쭉 리뷰해보면서 오류가 왜 발생하는지 분석해보았다.



그러다 보니, realloc을 제대로 구현하지 않아 생기는 문제일 수도 있다고 생각하여, realloc을 우선 free후 malloc하는 것으로 일단 바꾸어 프로그램을 동작시켜 보았지만, 결과는 위 화면과 같았다. 두 번째 명령부터 payload overlap error가 나는 것 같았다. 이에, 함수 별로 분석하여 예외 상황이 있는지, 오류가 어디서 발생할 수 있을지 분석해보았다. 아래는 시행 흐름에 따라 각 함수별로 분석하며 코드를 수정 및 보완했던 기록이다.

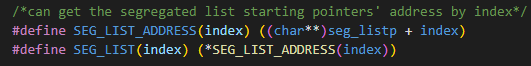




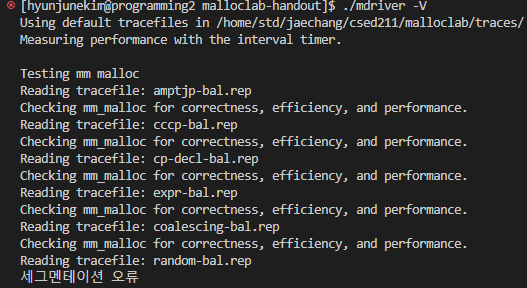
이 과정에서 다양하게 각 함수들에서 예외 상황을 컨트롤하여 수정을 진행하였다. 하지만 수정 이후에는 segmentation fault가 발생하기 시작하였다. 따라서 printf로 malloc의 각 부분에 표지를 해 놓고 실행시켜 디버깅을 진행하였다.

그 결과, find\_fit 함수에서 문제가 발생된 것으로 추정되었고, 확인해본 결과 SEG\_LIST(i)가 48이 alloc될 때 모두 NULL로 되어 있어야 하는데, 아래와 같이 예전 block의 list index인 7이 남아 있는 것을 볼 수 있었다. 즉, delete\_block에서 list의 초기화가 제대로 이뤄지지 않았던 것이었다. 확인해보니 ptr에 그냥 NULL을 assign하여 발생한 문제였고, SEG\_LIST(i)=NULL로 다시 처리해주었다.

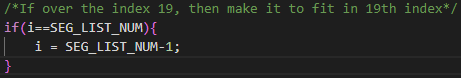
이 밖에도 디버깅을 진행하면서 몇 가지 잘못된 부분들을 모두 고쳐주었다. 그 결과 잘 실행이 되는가 했는데, 잘 되다가 막판에 segmentation fault가 뜨는 부분이 나왔고, debuging하다 보니, 처음에 계획한 것은 list에서 첫 블록의 PRED 포인터에는, 리스트 자체의 주소를 저장하고자 하였는데, 그 부분에서 오류가 생기는 것을 확인하였다. 따라서 계획한 대로 구현하기 위해 SEG\_LIST\_ADDRESS(i)라는 macro를 만들어서 segregated list의 index를 받아 해당 list의 시작 주소를 얻도록 하였다.



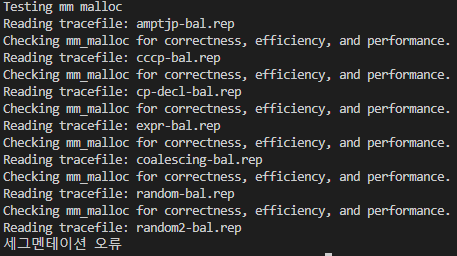
SEG\_LIST(i)도 SEG\_LIST\_ADDRESS(i)를 활용하는 것으로 수정해주었고, 이를 활용하여 계획한 대로 insert\_seg\_list와 delete\_seg\_list 수정을 마쳤다. 그 결과 mdriver를 실행하니 아래의 결과를 얻을 수 있었다.



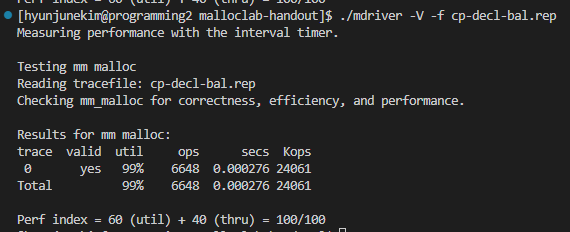
random-ball trace 파일에서 세그멘테이션 오류가 생겼는데, random-bal이 다른 trace들과 다른 점이 무엇일까 살펴보니, 할당 size 자체가 굉장히 큰 것을 확인할 수 있었다. 따라서 insert\_seg\_list에서 index를 찾는 부분에서, i가 20을 넘어가게 될 수 있다는 것을 확인하고, 20을 넘어갈 경우 19의 index로 들어갈 수 있도록 i를 다시 19로 낮춰주는 코드를 작성하였다.



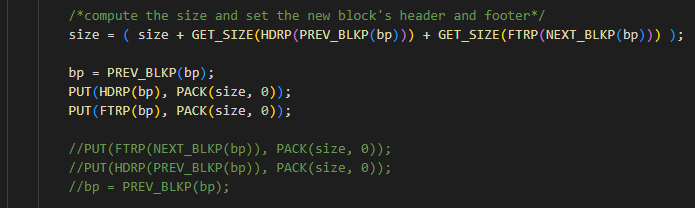
그러고 나서 다시 mdriver를 동작하니 random-bal은 통과하고 random2-bal에서 segmentation fault가 발생하였다.



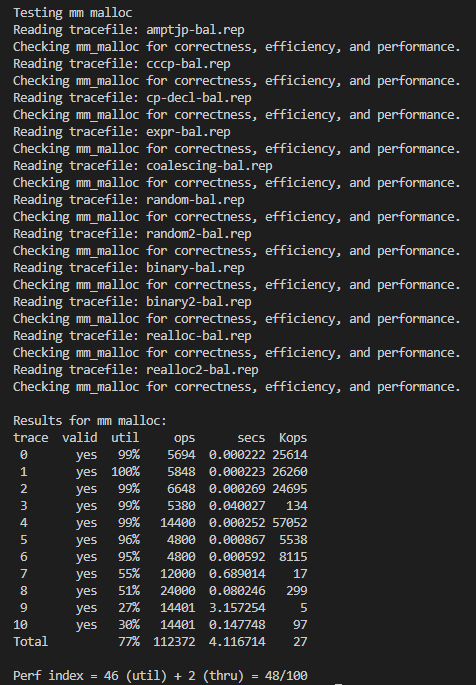
따라서 이를 해결하기 위해 random2-bal.rep만 따로 -f option으로 mdriver를 동작시켜 디버깅을 진행하였고, 4801번째 줄의 free에서 segmentation fault가 일어난다는 사실과, coalesce함수에서 오류가 일어난다는 사실을 알게 되었다.



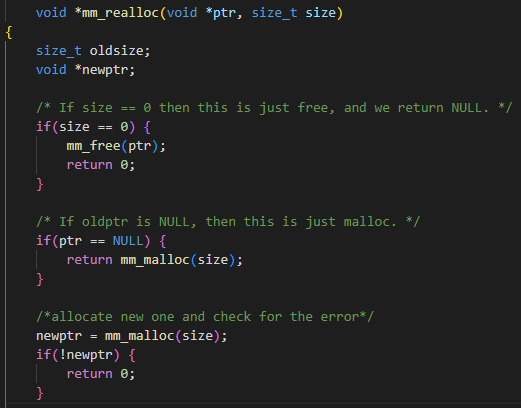
그런데, random2-bal.rep 이외의 trace들을 각각 돌려보았을 때에는 괜찮은 결과들이 나왔다. 나머지 trace들에서는 모두 valid한 동작을 보여주었다. 여러 가지로 코드를 바꿔보다, PUT(HDRP(PREV\_BLKP(bp), PACK(size, 0)); 구문에서 segmentation이 일어나는 것을 확인하고, 아래와 같은 방식으로 바꾸어 작동시키자 제대로 동작하였다. 주석 처리된 것이 원래의 코드이다.

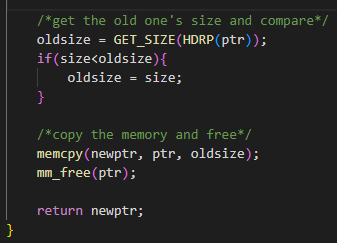


완성한 코드를 -V mode로 동작시켜 보니 아래와 같은 결과를 얻을 수 있었다.



0~6번 정도의 trace에서는 높은 퍼센테이지를 보여주었지만, realloc이 사용되는 9, 10번 trace에서는 저조한 수치를 보여주었다. 따라서, 현재 간단하게만 구현이 되어 있었던 mm\_realloc 함수를 더 발전시켜 보고자 하였다.





현재 realloc은 위와 같은 형태로 fragmentation이 많이 발생할 수 있는 구조였기 때문에 stand-alone realloc을 만들어보려고 하였다. 그러나 segregated list 구현을 하면서 디버깅에 너무 많은 시간을 소요하였기 때문에 제 시간에 stand alone realloc을 완전히 구현하지 못하고, 위의 코드 대로 제출하게 되었다.

Size와 oldsize를 비교하여 각각 size가 커서 범위를 넓혀야 할 때에는 인접한 블록의 free 여부를 판별하여 재할당하는 구조, 같을 때는 간단하게 ptr를 다시 리턴하는 구조, 작을 때에는 남은 block을 다시 free 블록으로 만들어서 segregated list로 넣는 과정을 만들어보려고 했었다. 아쉽지만 이대로 제출하였고, 이로써 malloc-lab을 완성하였다.