M1522.000800 System Programming
Fall 2018

# System Programming MemoryLab Report

Kim Bochang
2014-16757

## 1. <lab> Memory Lab

메모리랩의 목적은 직접 dynamic memory allocator를 구현하면서 직접 힙을 구성하고, 어떤방식으로 free block과 allocate block을 만들것인지, malloc할때 어떤 free block을 사용할지(placing), 할당하고 남은 블록은 어떻게 할것인지 (splicing), free 할때 생겨난 freeblock을 어떻게 다른 freeblock과 합칠것인지 (coalescing) 등을 고민해보면서
실제 동적 메모리 할당이 어떻게 이루어 지는지 이해하는 것이다.

## 2. Implementation

Segregated Free list에서 Segregated fit 방식을 사용하였다.

| | 힙 시작 | Prologue block | | | | | | blocks | Epilogue block | 힙 끝 |
|---|---|---|---|---|---|---|---|---|---|---|
| 역할 | to fit align | size set 저장 | | | | | endpoint | ... | mark heap end | |
| 구조 | zero | Header | | sizeset point | | sizeset point | 0x1 | ... | Header | |
| 크기 | 4byte | (Sizesetnum + 2) * 4byte | | 4byte | ... | 4byte | 4byte | ... | 0 | |
| allocated? | No use | yes | yes | yes | yes | yes | yes | | yes | |

<heap architecture>

alignment 요구사항이 8바이트 이기 때문에, payload들이 8byte align이 되도록 다음과 같이 힙의 prologue block 과 epilogue block을 구성하였다.
Sizeset startpointer 들은 prologue block의 헤더 뒤에 위치하고 있으며, 각 sizeset의 처음 원소를 가리키고 있다. 초기값은 NULL로, size set의 원소가 존재하지 않음을 뜻한다.
각 sizeset의 크기는 16~31,32~63,64~127,…,32768~ 으로 총 12개의 sizeset을 사용한다.
sizeset의 크기는 각 sizeset에서 가지고 있을 수 있는 block의 크기를 나타낸다.
또한 endpoint는 sizeset들의 끝을 나타내며, 0x1의 값을 가지고 있다.
각 블록들은 4바이트 워드의 집합으로 이루어져있으므로 이 블록들의 pointer들의 0~1비트의 값에 절대로 1이 올수 없다. 따라서 sizeset startpoint 들이 끝나는 endpoint를 0x1으로 설정함으로써 sizeset startpoint들의 끝을 나타내었다.
Epilgue block의 헤더는 크기 0과 이 블록이 allocated 되었음을 나타내고 있다.

그 외 allocated block, free block, 그리고 각 sizeset의 구조는 다음과 같다.
각 block의 size가 최소 16byte이기 때문에, header의 0~3번째 비트를 사용할 수 있지만 이 구현에서는 0~1번째 비트만 각각 현재 block의 allocate 여부, 전 block의 allocate 여부로 쓴다.
allocated block은 footer가 없이 header와 payload로, free block은 heaer, prev free block pointer, next free block pointer, footer로 구성된다. sizeset은 각 free block들이 explicit free list의 구조를 이루고 있어 원소 서로간에 접근이 가능하고, sizeset startpoint를 통해 sizeset에 접근할 수 있다. 이로 인한 최소 블록사이즈는 16byte가 된다.

| | | | block pointer point here | |
|---|---|---|---|---|
| | allocated block | | ↓ | |
| 역할 | Header | | Payload | end |
| 구조 | 31~03 bits | 02, 01, 00 bit | user space | |
| 의미 | size | 02 : has footer? (don't used) | user data | |
| | | 01 : forward block is allocated? | | |
| 크기 | 4byte | 00 : this block is allocated? | user size (8byte align) | |

<allocated block architecture>

| | | | block pointer point here | | | | |
|---|---|---|---|---|---|---|---|
| | free block | | ↓ | | | | |
| 역할 | Header | | Prev Freeblock pointer | Next Freeblock pointer | no use | Footer | end |
| 구조 | 31~03 bits | 02, 01, 00 bit | 31~00 bits | 31~00bits | trash value | same as header | |
| 의미 | size | 02 : don't use | Prev Freeblock pointer | Next Freeblock pointer | none | same as header | |
| | | 01 : forward block is allocated? | | | | | |
| 크기 | 4byte | 00 : this block is allocated? | 4byte | 4byte | ??? | 4byte | |

<free block architecture>

| size set | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | NULL | ← | free block | ← | free block | ← | free block | | | |
| sizeset start pointer | | → | | → | | → | | → | NULL | |
| (if sizeset is empty, pointer's value is NULL) | | | (Start block) | | | | (End block) | | | |

<size set architecture>

### 2.1 Memory allocator

*<mm.c>*

내가 구현한 Memory allocator는 초기에 mm_init()을 호출하고, mm_malloc()과 mm_free()를 반복해서 호출하다, allocator가 종료될때 mm_exit()를 호출하는 구조다.

처음 메모리 할당기를 초기화 할때는 mm_init()을 이용해 Prologue, Epilogue 블록과 Sizeset 들을 만들어준다.

이후 mm_malloc() 을 통해 다음과 같은 과정을 진행하며 할당을 진행한다.

먼저 들어온 사이즈를 alignment 요건을 만족하고, minimum blocksize보다 크거나 같도록 변환한 다음, 해당 블록을 수용할 수 있는 free block이 sizeset에 존재하는지를 sizeset 내부에서 first-fit 방식으로 확인한다. 적당한 free block을 찾으면 placing을 진행하고, 아니라면 다음 sizeset으로 이동해서 같은 과정을 진행한다. 만일 힙안에 적당한 sizeset이 없으면, heap을 확장하고 다시 freeblock을 찾아 placing을 진행한다. (place() function)

placing은 다음과 같은 과정으로 진행된다. 먼저 freeblock을 sizeset에서 제거한다.

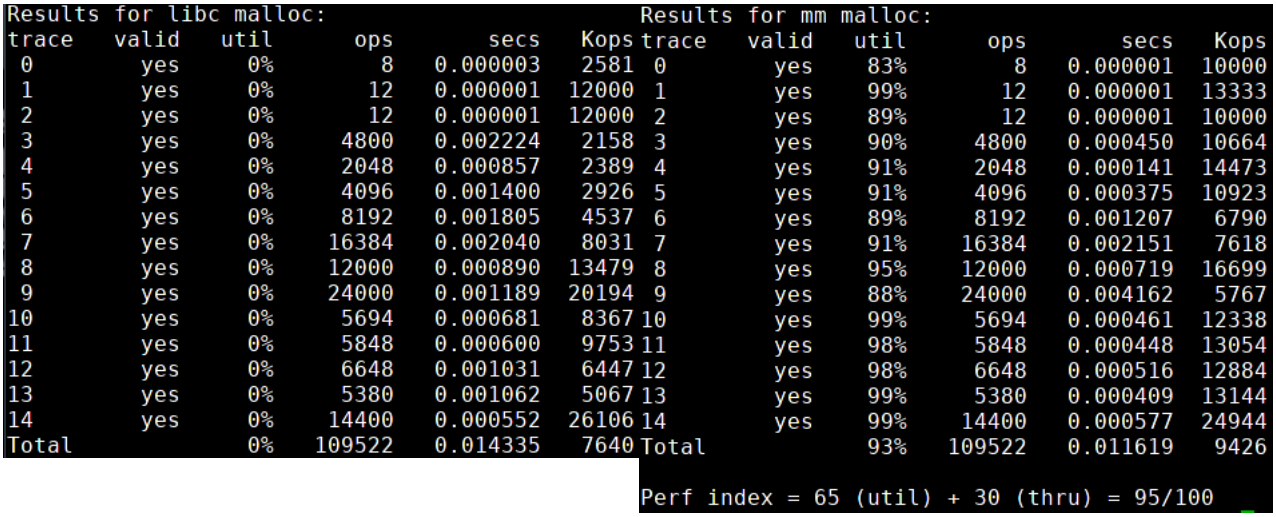
만약 freeblock의 사이즈가 allocation을 하고도 minimum blocksize 이상 남는다면 split을 진행하고, 아니라면 그냥 freeblock 전체를 allocateblock으로 사용한다.

split을 진행할때는, 새로 allocate될 블록의 사이즈가 특정값(코드에서 SMALLBLOCKSIZE 매크로)보다 작거나 같다면 allocateblock이 앞에 오도록 split을 하고, 아니라면 allocate block이 뒤로 가도록 split을 진행한다.

이렇게 함으로써 size가 작은 block들이 heap 앞으로가는 경향성이 생기게 되고, 따라서 heap 전체를 size가 작은 block들이 양단해서 생길 수 있는 external fragment를 상당히 줄일 수 있다.

place를 진행한 다음, 할당이 진행된 블록의 다음블록의 forward allocation field를 수정해주고, split 이 진행되었다면 새로 생긴 freeblock을 다시 sizeset에 추가해준다.
이때 coalescing은 진행할 필요가 없는데, freeblock들은 이미 모두 coalescing되어 있는 상태라서 앞과 뒤 블록이 allocated block인것이 보장되어 있기 때문이다.
mm_free()는 다음과 같은 과정을 통해 진행된다. 먼저 free하려는 블록이 allocated 블록인지 확인한 다음 (double free check), 실제로 free를 진행하게 된다.
Allocate block을 free block으로 만든다음, 이 block을 sizeset에 추가하고, coalescing을 진행하게 된다. (coalesce() funtion) 이때 굳이 block을 sizeset에 추가해주는 이유는, coalescing을 진행할때 각 freeblock이 header,footer,prev block pointer와 next block pointer를 갖고있는 완전한 상태여야 하도록 구현을 했기 때문이다.
coalescing을 할때, 먼저 앞의 블록과 뒤의 블록의 allocate 상태를 확인한다. 내 구현에서 Allocated block은 footer를 가지고 있지 않기때문에, 앞의 블록은 현재 블록의 forward allocation field를 참조해서 여부를 판단하고, 뒤 블록은 현재 블록의 size field를 통해, 포인터 연산으로 직접 뒤 블록의 헤더에 접근, allocated 여부를 확인한다.
앞이나 뒤의 블록이 free된 상태라면, 현재 블록과 합칠 블록들을 sizeset에서 제거하고,
블록을 하나로 합쳐준다음, 해당 size에 맞는 sizeset에 집어넣어 준다.
아니라면 sizeset 관련 연산을 진행하지 않는다.
그리고 주변 블록의 forward allocation field를 적당히 바꾸어 준 뒤 coalescing을 끝낸다.
sizeset에 새로운 free block을 집어넣을때는, sizeset의 first pointer를 삽입할 free block으로 바꾸어주고, sizeset의 first pointer가 가리키던 블록의 prev pointer를 삽입할 free block으로 수정해준다. first pointer가 NULL일때는 이과정은 생략한다. 이렇게 함으로써 상수시간에 sizeset 삽입을 완료한다.
sizeset에서 free block을 제거할때는, 제거할 free block이 갖고있는 pointer들을 이용해 prev free block과 next free block에 접근해서 각 block들의 pointer를 각각 제거할 블록의 next free block, prev free block로 바꿔준다. 제거할 free block의 prev free block pointer 혹은 next fre block pointer 가 NULL이면 각 과정에서 가리키는 값을 NULL로 바꿔주고, 각 free block에 접근하는것은 생략한다. 이렇게 함으로써 상수시간에 sizeset에서의 제거를 완료한다.
마지막으로 allocator가 종료할때는 mm_exit()를 호출해서 free되지 않은 allocated 블록들을 모두 free해주게 된다.
시간효율성을 분석해보면, malloc때는 sizeset에서 block을 찾을때 최대 해당 sizeset 원소 개수 + 다음 sizeset들의 원소개수의 총합 시간 + 힙 할당시간, 찾은 free block에 placing & spliting을 진행할때 상수시간이 들고, free때는 allocated block을 free block으로 바꿀때 상수시간, coalescing을 진행할때 상수시간이 드므로 전체적으로 상당히 빠르게 할당기가 실행된다.
공간 효율성도 괜찮은데, block이 split 될수 있거나 coalescing 될수 있으면 모두 나누거나 합치고, allocated block이 footer를 사용하지 않기때문에 internal fragment정도가 낮고,

external fragment도 placing 루틴을 이용해 개선하여 뛰어난 효율성을 보인다.



Results for libc malloc:

| trace | valid | util | ops | secs | Kops |
|---|---|---|---|---|---|
| 0 | yes | 0% | 8 | 0.000003 | 2581 |
| 1 | yes | 0% | 12 | 0.000001 | 12000 |
| 2 | yes | 0% | 12 | 0.000001 | 12000 |
| 3 | yes | 0% | 4800 | 0.002224 | 2158 |
| 4 | yes | 0% | 2048 | 0.000857 | 2389 |
| 5 | yes | 0% | 4096 | 0.001400 | 2926 |
| 6 | yes | 0% | 8192 | 0.001805 | 4537 |
| 7 | yes | 0% | 16384 | 0.002040 | 8031 |
| 8 | yes | 0% | 12000 | 0.000890 | 13479 |
| 9 | yes | 0% | 24000 | 0.001189 | 20194 |
| 10 | yes | 0% | 5694 | 0.000681 | 8367 |
| 11 | yes | 0% | 5848 | 0.000600 | 9753 |
| 12 | yes | 0% | 6648 | 0.001031 | 6447 |
| 13 | yes | 0% | 5380 | 0.001062 | 5067 |
| 14 | yes | 0% | 14400 | 0.000552 | 26106 |
| Total | | 0% | 109522 | 0.014335 | 7640 |

Results for mm malloc:

| trace | valid | util | ops | secs | Kops |
|---|---|---|---|---|---|
| 0 | yes | 83% | 8 | 0.000001 | 10000 |
| 1 | yes | 99% | 12 | 0.000001 | 13333 |
| 2 | yes | 89% | 12 | 0.000001 | 10000 |
| 3 | yes | 90% | 4800 | 0.000450 | 10664 |
| 4 | yes | 91% | 2048 | 0.000141 | 14473 |
| 5 | yes | 91% | 4096 | 0.000375 | 10923 |
| 6 | yes | 89% | 8192 | 0.001207 | 6790 |
| 7 | yes | 91% | 16384 | 0.002151 | 7618 |
| 8 | yes | 95% | 12000 | 0.000719 | 16699 |
| 9 | yes | 88% | 24000 | 0.004162 | 5767 |
| 10 | yes | 99% | 5694 | 0.000461 | 12338 |
| 11 | yes | 98% | 5848 | 0.000448 | 13054 |
| 12 | yes | 98% | 6648 | 0.000516 | 12884 |
| 13 | yes | 99% | 5380 | 0.000409 | 13144 |
| 14 | yes | 99% | 14400 | 0.000577 | 24944 |
| Total | | 93% | 109522 | 0.011619 | 9426 |

Perf index = 65 (util) + 30 (thru) = 95/100

<libc malloc (라이브러리) vs mm malloc (사용자)>

## 3. Conclusion

segeregated fit방식의 dynamic memory allocator를 직접 짜보면서, 실제 Heap에서 메모리를 할당하는 작업이 일어나는 과정과, allocator를 어떻게 최적화 하는지 알게되었다. 처음에 만든 할당기는 시간적으로나 공간적으로나 약간 비효율적인 점이 있었는데, Allocated block이 가지는 불필요한 footer를 없애서 memory utilization을 올리고, mdriver 프로그램을 실행하면서 내가 구현한 alloctator의 최적화를 진행해 나갔다.

처음에는 단순히 segeregated free list를 적용해 구현한 버전은 throughtput에서 30점, utilization에서 59점을 받았었다. 이때 utilization을 증가시키기 위해 split을 진행할때 작은 블록을 앞에, 큰 블록을 뒤에 가도록 하는 아이디어를 생각했는데, 하지만 이 과정에서 불필요하게 실행되는 연산들이 너무 많아서 utilization은 올라갔지만 throughput이 25점대로 떨어지게 되었다. 그 뒤로 throughput을 올리기 위해 자주 실행되는 함수의 4중 if문을 2단의 if else 문으로 바꾸기도 하고, 변수를 두거나 제거하기도 하면서 불필요한 연산을 제거해 나갔다.

또한 작은 블록을 앞에 가도록 사이즈를 계산하는 과정에서 뺄셈이 일어나 시간이 너무나 많이 걸렸으므로, 작은블록(코드에서 SMALLBLOCKSIZE 매크로 값)을 크기 128이하인 블록으로 정의하고 상수와의 비교연산을 통해 (bitwise 연산) 작은 블록인지의 여부를 빠르게 파악할 수 있게 하면서 throughput을 최대치로 끌어올렸다.

그 뒤에 작은 블록의 크기를 96으로 조금 더 줄여서 bitwise연산도 적용할 수 있게 하면서(64 + 32), 너무 크지 않은값으로 조정하였다.

이와 같은 값으로 작은 블록을 조정하면, 대부분의 변수나 구조체, 짧은 배열 하나를 선언하는등의 작은 값들은 대부분 힙의 앞에 할당되어 external fragmentation을 줄일 수 있다.

메모리랩을 하면서, 포인터를 다루는 작업의 위험성을 너무나 절실히 느끼게 되었다. 포인터 연산에서 실수하지 않으려고 매크로들을 만들고 사용했는데, 블록의 헤더를 집어넣어야 하는 매크로에 헤더를 집어넣지 않거나, 블록 포인터가 가리키는 내용(포인터)가 가리키는 곳의 내용을 바꿔야하는데 블록 포인터가 가리키는 내용을 바꾼다던가 하는 실수를 정말 많이 했다. 덕분에 segmentation fault 메시지를 정말 많이 봤는데, 문제가 있어도 gdb를 이용해 힙 내부구조를 보는것이 꽤 만만치 않은 일이라 어디서 문제가 일어났는지를 알지 못해 디버깅이 어려웠다. mm_check()함수를 구현하고, 쓰면서 이러한 오류를 고칠수 있었는데, 만약 mm_check()함수를 구현하지 않았다면 오류를 고치는데 굉장히 많은 시간이 걸렸을것이다.

메모리랩을 하면서 놀라웠던점은, 간단히 작은 블록을 앞에다가 몰아준다는 아이디어만 적용했을 뿐인데 생각보다 효과가 뛰어났다는 점이 놀라웠다. 간단한 아이디어 하나일 뿐인데 external fragmentation의 빈도가 정말 많이 줄어들었다. 컴퓨터 분야에서는 이러한 하나하나 사소한 아이디어들이 모여서 크나큰 발전이 이루어질 수 있다는것을 느낄 수 있었다.