# 2021 SYSTEM PROGRAMMING
# Lab3 Report – Malloc Lab

자유전공학부 2012-13311 안 효 지

## 0. Intro

슬프게도 이번 과제는 제대로 마치지 못하였다. 컨디션 난조로 과제를 너무 늦게 시작해 시간이 촉박했던 것이 패인이다. 마음만 조급한 나머지 탄탄한 알고리즘을 짜지 못하였고, 그래서 주먹구구식으로 디버깅을 하는 데에 시간을 너무 많이 소비하여 하는 수 없이 내가 설계하던 segregated list는 구현에 실패하였다. Implicit list를 구현한 후, explicit list를 먼저 구현할 지, 아니면 바로 내가 구상하고 있는 segregated list를 구현할 지 고민했다. Explicit list를 먼저 구현하는 경우, 그래도 implicit 보다는 성능이 좋을 것이라 예상되므로 보험용으로 갖고 있을 수 있었지만, explicit과 segregated 둘 다 구현할 수는 없을 것 같아 곧바로 segregated를 구현하기 시작했다. 하지만 Short1, 2는 통과했으나 다른 것들까지 통과하도록 디버깅할 시간이 없어 일단 제출한다. 결론적으로, 제출한 코드는 교재에 나와있는 implicit list로 구현을 하였다. Implicit list는 교재에도 설명이 잘 나와있으므로 교재에 없는 realloc, findfit, place 함수만 이 보고서에는 기술할 예정이다. 그리고 원래 내가 구현하고자 했던 segregated list를 자세히 기술할 것이다.

## 1. How to implement

### 1) realloc function in implicit list

Realloc(void* bp, size_t size)은 (1) 기존에 alloc되어 있는 사이즈보다 작거나 같은 경우, (2) 기존 사이즈보다 큰 경우로 나눌 수 있다. 그 전에 `size == 0`인 경우에는 mm_free(bp)를 대신 호출하도록 한다.

(1) 기존의 사이즈보다 작거나 같은 경우, oldbp의 위치를 옮기지 않고, oldbp의 header와 footer에 adjsize(align이 된 새로운 사이즈)를 넣는다. 그리고 next block의 header, footer에 oldsize-adjsize를 넣고, alloc bit를 0으로 맞추어준다. 그 후 next block을 coalescing시킨다.

(2) 새로 realloc 요청된 사이즈가 기존보다 더 큰 경우에는, ① 뒤에 충분한 공간이 있는 경우와 ② 그렇지 않은 경우로 나뉜다. ① 뒤에 충분한 공간이 있는 경우에는 oldbp를 그대로 리턴하기 위한 작업을 한다. 현재 블록의 ftr를 0으로 리셋하고, next 블록의 footer 사이즈를 기존 블록에게 침범당하는 만큼 줄여준다. 그리고 next block의 헤더도 없애준다. 그 후 old block의 hdr와 ftr의 값을 new size의 값으로 업데이트 해주고, 마지막으로 next blk의 header까지 업데이트 해주면 된다. ② 뒤에 충분한 공간이 없는 경우는, 새로운 곳에 malloc을 해주고, free를 하기 전에 memcpy를 통해 기존 블록에 있던 내용들을 새로운 블록으로 옮겨준다. 그 후에

free를 한다.

### 2) find_fit(size_t adjsize)

이 함수는 free blocks들 중 어느 위치에 alloc이 될 지 결정하는 함수이다. 일단 가장 간단한 first fit으로 구현을 하였었는데 이것으로 제출할 줄 예상하지 못하였다. Heap_listp를 시작으로 하여, epilog hdr의 size는 0이므로, 사이즈가 0인 블록이 아닐 동안 계속 다음을 bp를 서칭한다. 만약 해당 블록의 사이즈가 내가 원하는 사이즈보다 큰 경우, 그리고 alloc되지 않은 블록인 경우에 그 블록을 그냥 선택한다.

### 3) place(void* bp, size_t adjsize)

findfit으로 찾은 블록(bp)에 adjsize를 위치시킨다. 만일, 블록 사이즈보다 adjsize가 16byte 이상으로 작은 경우에는 나머지 블록을 split하고 남는 블록을 free block으로 놔둔다. 16byte인 이유는 header 4byte, footer 4byte, 1byte를 할당해도 alignment를 맞추어야 하기 때문에 최소 8byte가 payload에 할당되기 때문이다. 만일 16byte 미만(0byte, 8byte)으로 남는다면 internal fragment를 감수하고 그냥 alloc 블록에 포함을 시킨다.

## 2. Segregated list(failed)

### 1) mm_init()

```
//                heap_listp        free_listp
//--------------------|-----------------|-----------------------------------
// unused | prolH 8/1 | prolF 8/1 | Hdr | surface | deep | Ftr | epilH 0/1 |
//-------------------------------------------------------------------------
```

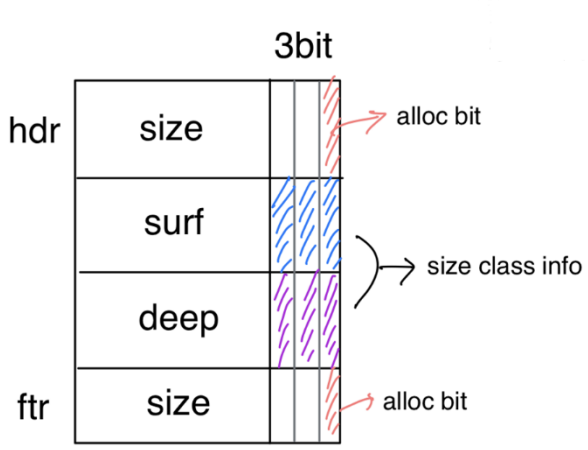
Heap을 initialize한다. 맨 처음은 unused block, 그 다음은 prologue header, prologue footer이다. 이들의 값은 size 8, alloc 1로서, 최소 블록 사이즈가 16인 regular block과 차별화되어 heap의 시작을 알려준다. 그 다음에는 free_listp의 header가 온다. header에는 여느 regular block과 동일하게 size/alloc bit이 들어가 있다. 이 16 byte의 free_listp 블록은 linked list의 dummy block과 동일하다. Segregated의 size class list의 주소를 매번 저장하여 메모리를 낭비할 필요가 없이, 저 free_listp의 surface 부분이 main stream의 첫 free block을 가리킨다. 보다 자세한 내용은 아래에서 설명하겠다.

```
    void* rgbp;     //reg blk ptr
    /* Extend the empty heap with a free block of CHUNKSIZE bytes */
    if ((rgbp = extend_heap(CHUNKSIZE/WSIZE)) == NULL) {return -1;}

//heap_listp  free_listp """""""""""""""""| rgbp
//----|-----------|------"--------------v-------------------------------
//    | prolF 8/1 | surface | deep | hdr | regular blk | ftr | epilH 0/1 |
//-------------------------------------------------------------------------
// free_listp의 dummy block의 surface가 freep를 가리키도록 한다.
```

그 후에는 기본적으로 사용이 될 chunk size의 heap을 할당한다. Freelistp의 surface에는 regular block의 pointer가 들어간다. 이 역시 아래에 기술하겠다.

### 2) free block의 구조




Free block은 free인 상태에서는 payload가 없기 때문에 이 공간은 자유롭게 사용해도 무관하다.
Header와 footer의 구조는 수업시간에 배운 것과 동일하다. 마지막 1 bit만 alloc bit으로 사용한다.
① surf : surf는 main stream 방향을 가리키는 edge들을 일컫는다.
② deep : deep은 중첩된 linked list의 아래쪽을 가리키는 edge를 일컫는다.

③ 파란색과 보라색으로 색칠된 "size class info" : 원래 계획은 log함수를 사용하여 free block의 사이즈만 가지고 바로 size class의 대소를 비교하려고 하였다. 하지만 `<math.h>`를 include 해도 log(x)의 x 부분에 값이 이미 정해져 있는 상수가 아닌 변수가 들어오는 경우에는, 컴파일을 할 때 `-lm` 옵션을 꼭 넣어야 한다는 것을 발견하였다. 하지만 mm.c 파일 이외에 수정할 수 있는 방법이 없어 하는 수 없이 hard coding으로 클래스를 나누어주었다.

```
if (adjsize ≤ 1≪6){ return 0; }
else if (adjsize ≤ 1≪7){ return 1; }
else if (adjsize ≤ 1≪8){ return 2; }
else if (adjsize ≤ 1≪9){ return 3; }
else if (adjsize ≤ 1≪10){ return 4; }
else if (adjsize ≤ 1≪11){ return 5; }
else if (adjsize ≤ 1≪12){ return 6; }
else if (adjsize ≤ 1≪13){ return 7; }
else if (adjsize ≤ 1≪14){ return 8; }
else if (adjsize ≤ 1≪15){ return 9; }
else if (adjsize ≤ 1≪16){ return 10; }
else if (adjsize ≤ 1≪17){ return 11; }
else if (adjsize ≤ 1≪18){ return 12; }
else if (adjsize ≤ 1≪19){ return 13; }
  // 최대 입력 숫자는 6자리임.
else { return 14; }
```

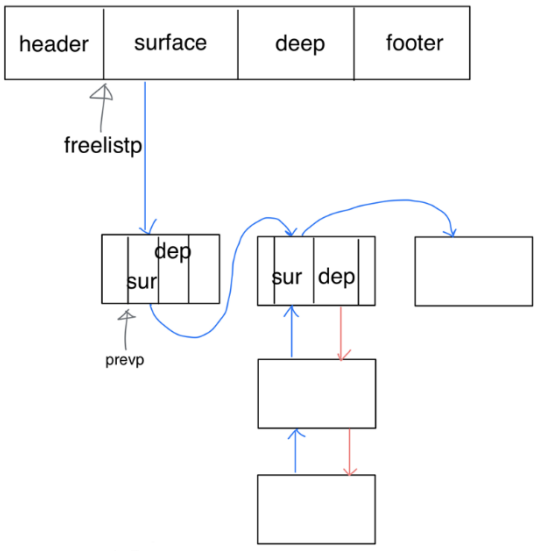
테스트케이스를 다 살펴본 결과, 가장 큰 size는 6자리수였다. 따라서 2^19 정도까지만 클래스를 만들면 되었다.

Surf와 deep 영역에는 address가 저장되어 있는데, 모든 블록의 alignment는 8byte이기 때문에 LSB 3 bit는 사용되지 않는다. 이를 이용하여 size class information을 파란색과 보라색의 lsb 3bit에 넣기로 하였다.

총 6bit이므로 2^6인 64개의 클래스를 감당할 수 있다. 원래 코드를 완성하면 사이즈 클래스의 개수도 변경해가면서 무엇이 더 효율적인지 실험해보려고 하였으나 실패로 돌아갔다.

④ size class 해석방법 : 6bit 중, surface는 LSB 3bit, deep은 MSB 3bit을 의미한다. 따라서, 매크로를 이용해 ` #define GET_CLSINFO(bp)  ((GET(bp) & 0x7) + (((GET((char*)bp + WSIZE)) & 0x7)<<3))` 식으로 만들어주면 bp만 넣어도 해당 블록이 어떤 사이즈 클래스에 속하는 지 알 수가 있다.

### 3) free list 구조




Init할 때 생기는 16byte의 surface 부분에서 시작한다. 각 free block은 블록 안에 사이즈 클래스의 정보를 가지고 있다. 따라서, 그림에서 가로로 위치해있는 main stream을 돌면서 자신의 size class가 존재하는지 확인하고, 존재하면 그 아래(deep)로 들어가 링크를 연결해준다. 존재하지 않는 경우에는 좌측은 작은 크기, 우측은 큰 크기로 정렬을 할 것이기 때문에 자신보다 큰 사이즈 클래스를 만나면 그 앞에 main으로 insert를 한다.

- Insert/ delete: Main과 deep은 서로 insert와 delete를 하는 방식이 약간 다르다. 따라서, header와 footer의 alloc bit 앞에 main bit를 두어(사진상에는 표

시하지 못하였다), 이 블록이 main stream에서 연결된 것인지, 아니면 main stream에 딸려있는 deep에 속한 블록인지 판단하도록 하였다. Insert 시에는 무조건 main stream으로 삽입이 되도록 하였다. Delete 시에는 이미 delete 될 블록이 정해진 상태로 delete 함수가 호출되므로 `#define IS_MAINST(bp)  ((GET(HDRP(bp)) & 2) >> 1)` 과 같은 매크로를 사용하여 main 인지 아닌지 판단해서 해당하는 방법으로 delete를 수행해주면 된다.

- 또한, main에 속하는 블록을 수정할 때에는 (main 입장에서) doubly linked list가 아니기 때문에 prevp가 필요하다. 어짜피 클래스는 많아봤자 현재 설계 상 16개이므로 loop를 돌면서 prev를 받아도 오버헤드가 심하지 않는다고 판단하였다. 나중에 class를 더 많이 두어도 max 64개 정도면 마찬가지로 오버헤드가 별로 심하지 않을 것이라 생각했다. 원래는 findfit을 한 후에 delete가 호출되면 findfit을 하는 과정에서 prevp가 결정되기 때문에 static으로 prevp를 정의해놓았으나, delete 이전에 항상 findfit이 불리는 것이 아니라는 것을 깨닫고, 그냥 loop를 돌면서 prevp를 구하는 방식으로 수정하였다.

#### 4) find fit

원래 구현이 쉬울 것이라 생각한 함수였는데 은근히 신경쓸 게 많았다. 예를 들어, 한 클래스에 크기가 3개 정도만 들어가도록 하여도, 그 3개 중 가장 작은 크기가 들어오면 아무거나 선택을 할 수 있는 반면에, 가장 큰 크기가 들어오면 본인과 크기가 동일한 freed block이 존재해야 find가 가능하다. 만약 본인과 동일한 사이즈 클래스에 본인보다 작은 freed block들만 존재한다면, main stream으로 다시 나가 그 다음 크기의 size class에 있는 블록들을 선택해야 한다.

Throughput을 줄이기 위해 크기가 본인보다 작지 않은 이상 main stream의 블록들을 선택하도록 코드를 짰다.

#### 5) 다른 함수들

기본적으로 malloc, free, realloc, place, extend heap 함수의 구조는 implicit과 크게 다르지 않다. 하지만 중첩된 linked list이고, main bit, size class bit 등, 블록에 추가된 정보 bit가 많다 보니 insert, delete, extend 등등, 모든 작업을 할 때마다 저 bit의 정보들을 정확히 update 해주어야 했다. 완벽하게 디버깅을 하지 못해서 확언할 수는 없으나, 아마 버그의 이유는 저런 정보들이 내가 의도하지 않은 부분에서 삭제되었기 때문이라고 추측한다.

## 3. What was difficult & surprising

원래는 red-black tree를 사용해서 구현을 하려고 하였다. 하지만 헤더와 푸터를 제외하고 2 words만 사용할 수 있다고 생각을 하였다. 왜냐하면 최소 블록 크기가 24byte이면 internal fragment가 너무 많이 생길 것이라 생각했기 때문이다. 또한, Insert/delete가 일어난 후에 fix를 하려면 parent ptr을 타고 root까지 올라가야 하는데 parent ptr이 없으면 불가능하다고 판단하여 16byte로는 구현을 못할 것이라 예상하고 상기한 방법으로 구현을 시작하였다. 하지만 알고 보니

insert/delete location을 찾을 때 재귀적으로 구현을 하면 parent node의 주소가 stack에 저장되도록 할 수 있다는 것을 나중에 알았다. 이미 제출 시간은 다가오고, 레드블랙트리로 방향을 틀 시간이 없어 내가 구현한 방법을 밀어붙였다. 하지만 결국 디테일한 버그를 잡지 못하여 segregated도 완성하지 못하였다.

Main bit이 작성되어 있는 상태에서, PUT(bp, PACK(size, 1))과 같은 매크로를 사용하는 경우에, main bit이 제거되는 불상사를 맞닥뜨렸다. 그래서 main bit의 유지가 필요한 경우에는 main인지 아닌지 판단한 후, main이면 PUT을 수행한 이후에 다시 main bit을 써주었다.

처음으로 gdb를 이용해보았다. Gdb에 대한 악평을 많이 들어왔던 터라 평소엔 그냥 프린트문만 이용해서 디버깅을 했었는데, 어느 부분에서 segfault가 벌어지는지 알 수 있어서 편하긴 했다. 하지만 보다 편리한 메모리트래커가 있으면 좋겠다고 생각했다. 이미 존재하지만 내가 모르는 것일 가능성이 농후한 듯하다.

종강하고 반드시 말록랩을 다시 구현할 것이다.

## 4. Result screen shot

```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.009540   597
 1       yes   99%    5848  0.008267   707
 2       yes   99%    6648  0.015739   422
 3       yes  100%    5380  0.011801   456
 4       yes   66%   14400  0.000126114558
 5       yes   92%    4800  0.010113   475
 6       yes   92%    4800  0.008899   539
 7       yes   55%   12000  0.142320    84
 8       yes   51%   24000  0.312651    77
 9       yes   80%   14401  0.000237 60815
10       yes   46%   14401  0.000123116891
Total          80%  112372  0.519816   216

Perf index = 48 (util) + 14 (thru) = 62/100
```