Malloclab 实验报告

实验目标

实现以下 4 个函数:`int mm_init(void)`,`void *mm_malloc(size_t size)`, `void mm_free(void *ptr)`,`void *mm_realloc(void *ptr, size_t size)`.

其中 `void *mm_malloc(size_t size)` 的作用、参数和返回值和 C 中的 malloc 完全相同. 动态地从内存堆区中申请一块指定大小的内存块区域, 并返回指向这块区域的指针.

`void mm_free(void *ptr)` 的作用、参数和 C 中的 free 也完全相同, 作用是通过一个指针释放掉之前申请了的内存.

`void *mm_realloc(void *ptr, size_t size)` 的作用、参数和返回值和 C 中的 realloc 也完全相同. 作用是为一块内存重新分配一块内存(改变大小). 需要注意的是要确保内存中的内容也要复制到新的内存中, 因为新的内存比旧的内存小导致的内容丢失不用考虑.

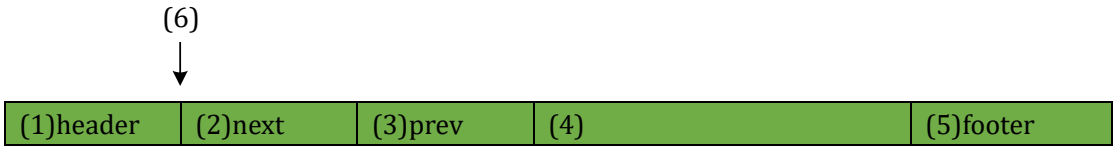
`int mm_init(void)` 的作用是做一些必要的初始化工作, 为后续的工作搭建基础. 这个函数只会在一次运行的最开始被调用一次.

内存管理结构

目前完成的实现是:

1. 显式双向空闲队列(Explicit free list);
2. 在堆区的最开始有一个哨兵;
3. Boundary tag 用于储存块大小和状态码. 二进制最后一位为 1 表示此块被占用. 否则此块空闲.

a. 用一个双向链表组织起空闲块. 空闲块的结构为:



(1) header, 4 字节, 储存空闲块的大小(size). 并记录块的状态. 这里取 8 对齐, 则 size 的二进制表示的最后两位为 0. 对于分配块, 将让其最后一位变为 1.

(2) next, 4 字节, 储存指向下一个空闲块的指针, 如果空闲队列中只有一个空闲块, 则它将指向自己.

(3) prev, 4 字节, 储存指向前一个空闲块的指针, 如果空闲队列中只有一个空闲块, 则它也将指向自己.

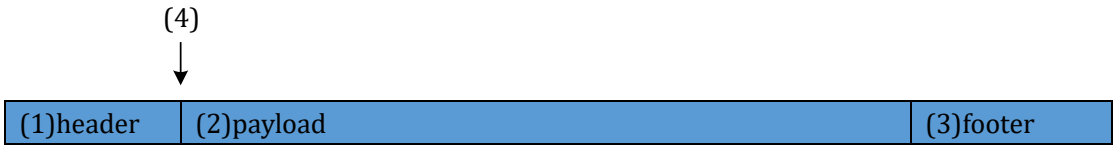
(4) 空闲的空间, 这里是可以没有的.

(5) footer, 4 字节, 其值和 header 完全相同. 作用是帮助空闲块的合并.

(6) 无论是空闲块还是分配块, 都让指向一个块的指针指向(6)所指向的位置.

可以发现一个空闲块最小的大小为 16 字节.包含(1)(2)(3)(5).

b. 占用块的结构为



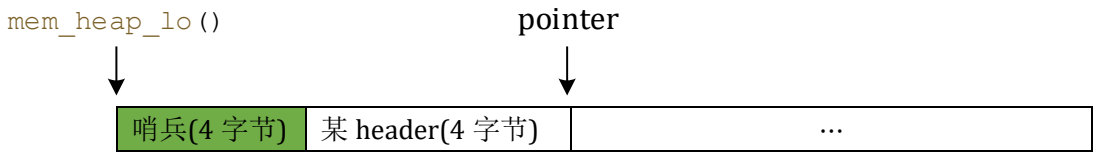
header 和 footer 的作用是储存占用块的大小(包括 header 和 footer 在内). 它们二进制表示的最后一位被置为 1 来标识这个块是占用块.

这里的实现保证 payload 至少为 8 字节. 故一个占用块的大小至少为 16 字节.

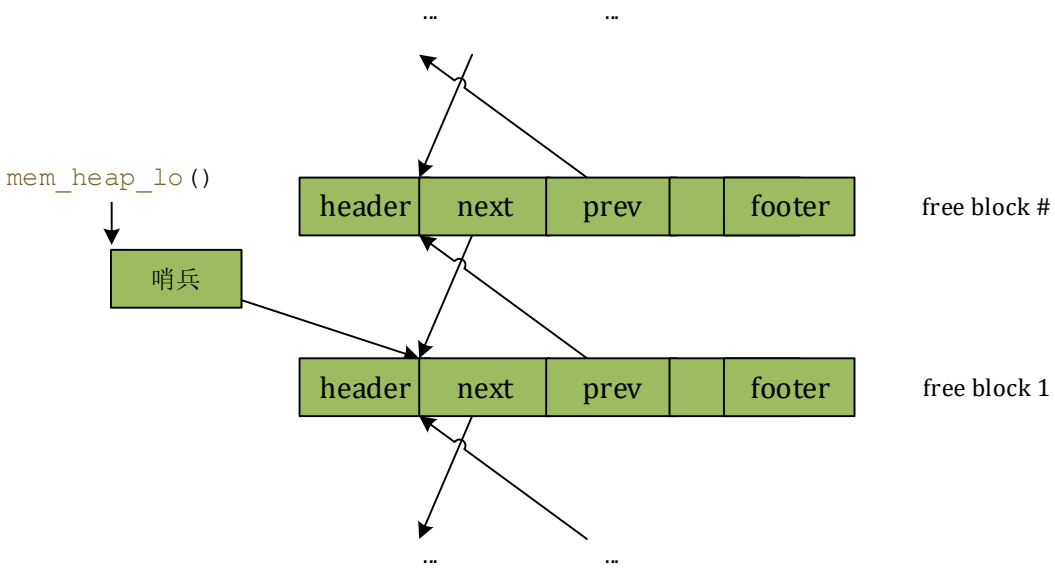
c. 在 `int mm_init(void)` 中定义了一个哨兵:

```
1  int mm_init(void){
2      size_t **p = mem_sbrk(4);
3      *p = 0;
4      return 0;
5  }
```

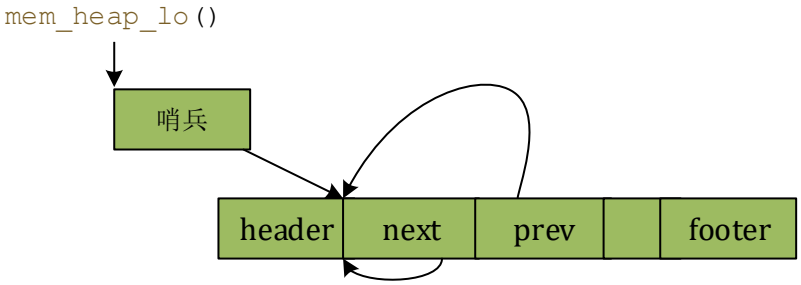
哨兵实质上是一个指针, 当其非 0 时, 指向的是 free list 中的一块. 其作用一是方便找到 free list, 二是占用 4 字节的空间. 在实际操作中发现 `mem_heap_lo()` 总是返回 8 的倍数. 在哨兵后面置放内存块时, 由于内存块 header 要占用掉 4 字节, 这样能让指向内存块的指针也都是 8 的倍数. 恰好实现了 8 对齐:



d. free list 的结构, 如下图所示.



free list 中只有一个空闲块的情况:



实现策略

1. `mm_malloc(size_t size)`

   其策略为，先遍历所有的空闲块(在 free list 中)，从中找大小最合适的空闲块(即 Best fit)．如果存在大小合适的，则返回指向这个空闲块的指针．这部分对应的函数为

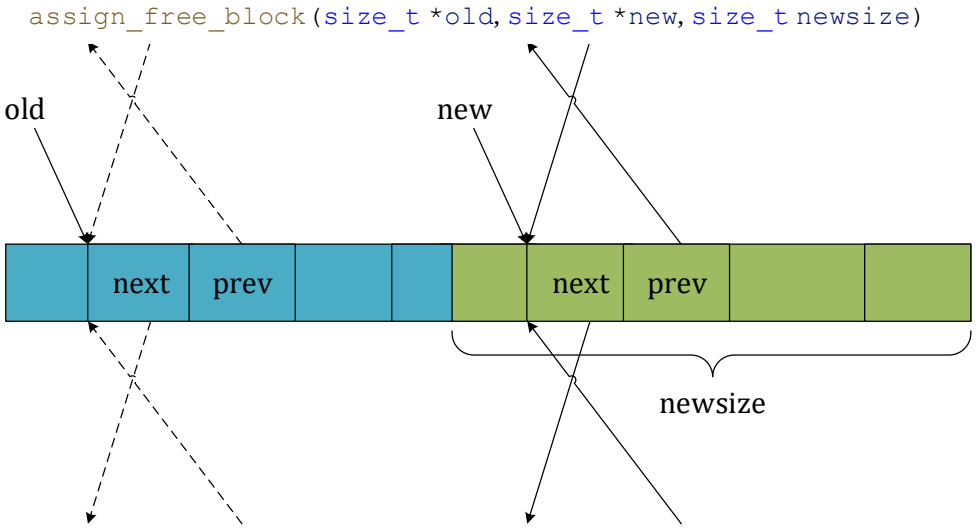
   `size_t *scan_free_block(size_t newsize);`

   当发现一个空闲块的大小比所需要的大小大不超过 16 字节，或者大小刚好相等时，立即将这整个块设置好占用块的 boundary tag 并将其返回．这是因为这个块已经足够合适，剩余的空间不足以形成一个空闲块(这里之前没有考虑到，导致出错)．返回这个块的同时要把原来的空闲块从 free list 中删除．删除时要注意哨兵，如果删除了哨兵指向的空闲块，则哨兵要做相应改变．从 free list 中删除一个块的函数为:

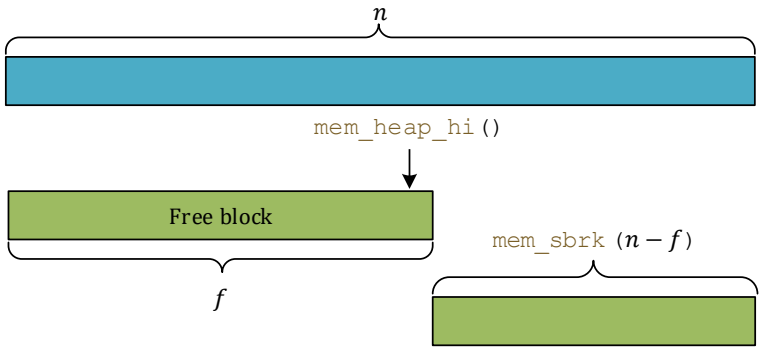
   `void delete_free_block(size_t *ptr);`

   在其他找到 Best fit 的情形下(空闲块的大小和所需大小相差最小，且差距大于 16 字节)，空闲块中的前一部分作为分配块被设置好 boundary tag 并返回．而 free list 中这原有的空闲块要变成剩余的那部分空闲块．包括 header, footer, next, prev．而且要考虑到哨兵指向的恰好是这一空闲块的情况．相关函数为:

   `void assign_free_block(size_t *old, size_t *new, size_t newsize)`，这个函数可以通过在链表中删除 old 块，再插入 new 块完成，但可以另外实现减少指针操作的数量．如图所示，绿色部分是被分配出去的块，白色是剩余的空闲块．只要移动 4 个指针即可．(如果哨兵指向了 old 块，则哨兵也要改变).



以上是在 free list 中找空闲块的情况. 当找不到合适的空闲块(free list 为空或其中的空闲块都太小)时, `scan_free_block` 将返回 0. 这时候就必须要通过 `mem_sbrk` 函数来实现内存分配. 但是这里还有可以优化的空间. 如果发现堆区最后一块是空闲块, 那么 `mem_sbrk` 的大小就不需要是整个所需的大小. 如果堆区最后一个块是空闲块, 设其大小为f字节, 所需大小为n字节, 那么通过调用 `mem_sbrk`$(n-f)$就可以将空闲块扩大到刚好合适的大小. 将其设置好占用块的 boundary tag 并返回, 从 free list 中删除这一空闲块即可.



如果堆区的最后一块不是空闲块, 只能调用 `mem_sbrk`(n), 并将这一块设置好占用块的 boundary tag 并返回.

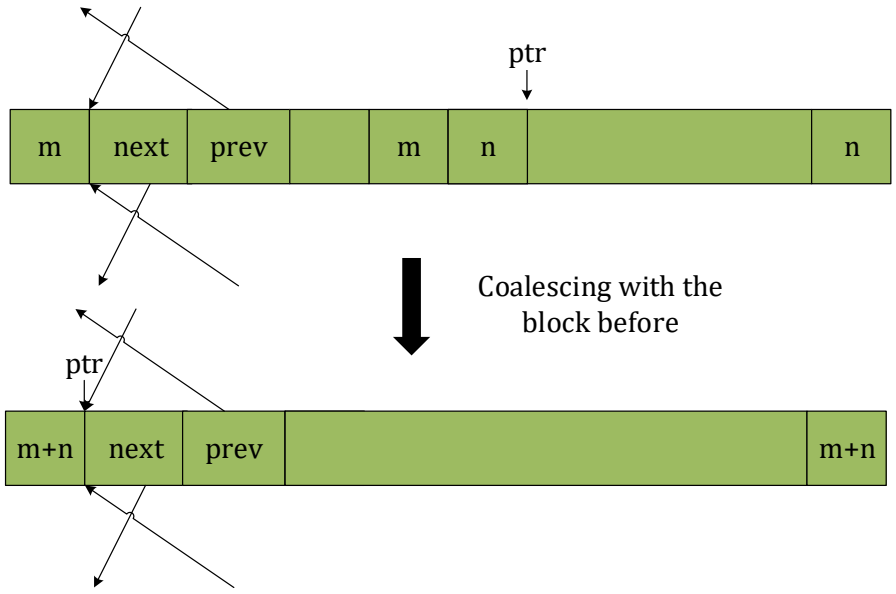
以上就是 `mm_malloc` 的策略.

2. `mm_free(void *ptr)`

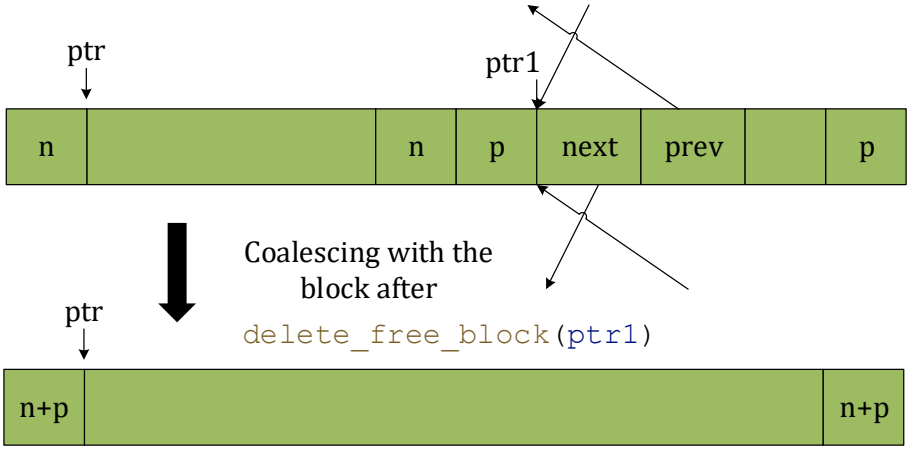
释放一个块时. 要考虑和前一块和后一块的合并. 通过检查前一块的 footer 来判断前一块是否为空闲块, 通过检查后一块的 header 来判断后一块是否为空闲块.

指向被 free 的块的指针为 ptr.

如果可以, 先和前一块合并. 和前一块合并时, free list 不用发生变化, 只需要改变前一块的 header 和 footer 即可, 并将 ptr 移动到指向合并后的空闲块的位置.

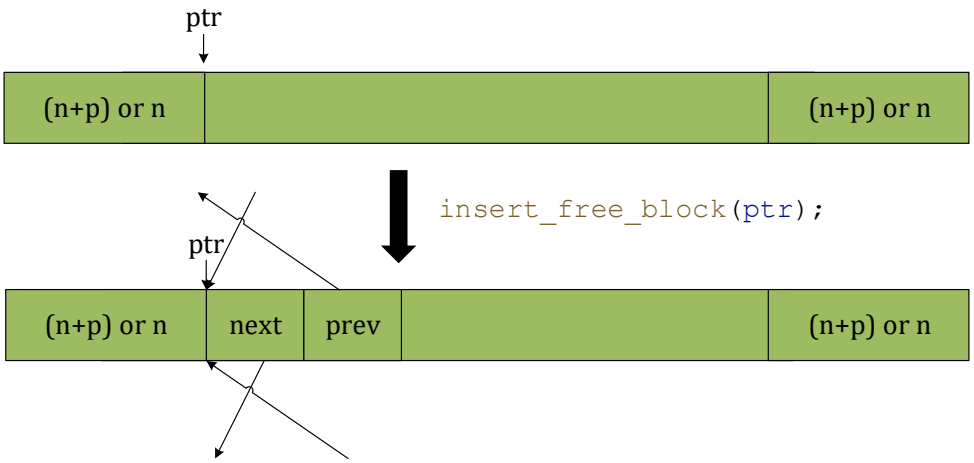


然后再看能否和后一块合并. 如果可以, 和后一块合并时, 要将后一空闲块从 free list 中删除.



最后，如果没有发生和前一块的合并(是否与后一块合并无影响)，则要将 ptr 指向的块加入到 free list 中. 相关的函数为:

```
void insert_free_block(size_t *ptr);
```



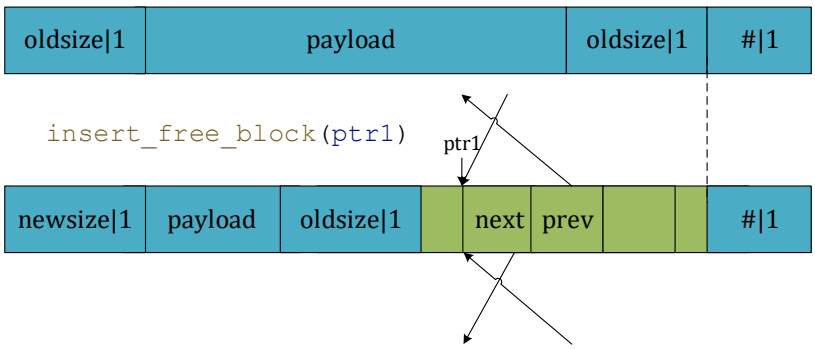
以上就是 mm_free 的策略.

3. mm_realloc(void *ptr, size_t size)

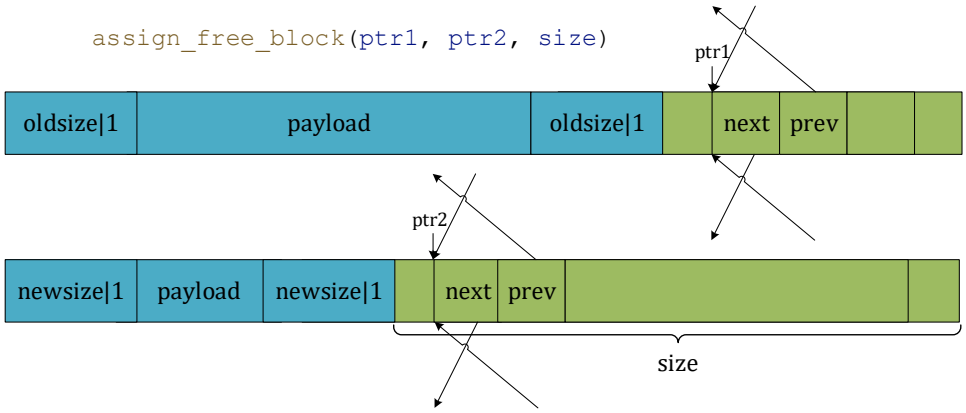
设原来的内存块大小为 oldsize，重新分配的内存块大小为 newsize.(包括 header 和 footer，单位是字节).分情况讨论. 注意 newsize 不等于参数中的 size.

a. newsize <= oldsize – 16.

这时候，重新设置好分配块的 boundary tag. 然后会多一块空闲块，这时如果后一块也是空闲块，则要将它们合并. (使用之前的 assign_free_block 函数). 如果后一块不是空闲块，则将多的空闲块插入到 free list 中. 后一块是占用块的情况:



后一块是空闲块的情况:



b. newsize > oldsize – 16 && newsize <= oldsize - 8.

这时候，多出来的 8 个字节的块虽然不能单独作为一个空闲块，但是如果后一块是空闲块，则可以将它们合并(图见上). 如果可以，合并即可. 剩余的块返回. 如果不可以，无需进行任何操作，原样返回指针.

c. newsize > oldsize – 8 && newsize <= oldsize.

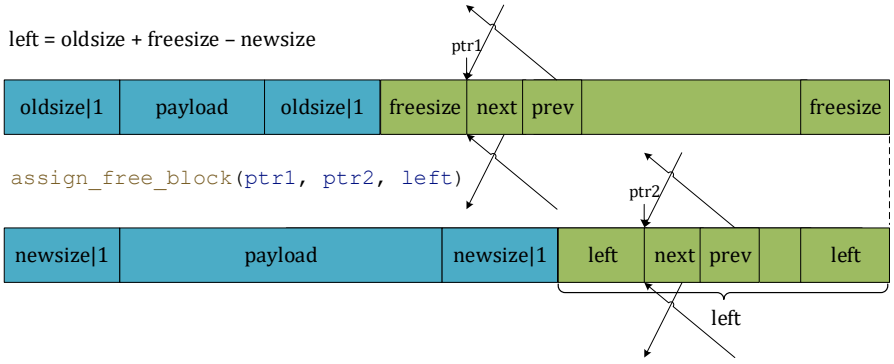
这时候原样返回指针即可.

d. newsize > oldsize.

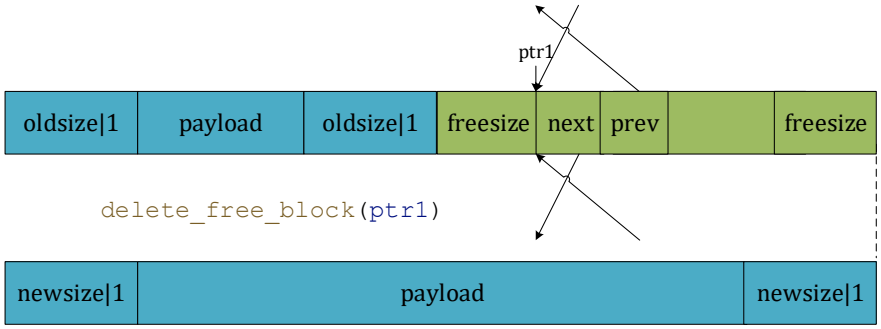
这时候要对内存进行扩，又要分情况讨论.

d.1 如果 ptr 所指向的内存块之后一块是空闲块，设空闲块的大小为 freesize.

d.1.1 如果 newsize <= oldsize + freesize – 16.，把空闲块的前一部分内存划分给 ptr 块即可. 空闲块还有剩余. 改变 free list 中的这一空闲块即可. (使用之前的 `assign_free_block` 函数).



d.1.2 如果 newsize > oldsize + freesize – 16 && newsize <= oldsize + freesize，空闲块全部划分给 ptr，没有剩余(剩余的空间不够形成空闲块)，因此要从 free list 中删除这一空闲块.

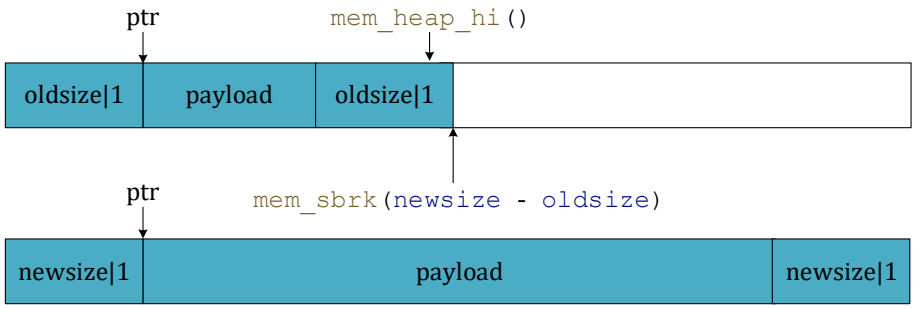


d.1.3 newsize > oldsize + freesize

d.2 后一块是占用块.

以上两种情况，都必须通过 `mm_malloc(newsize)`、内存拷贝(要自己实现，对应函数 `mm_copy(size_t *old, size_t *new)`)，和 `mm_free(old)` 来完成. 简而言之就是把内存整个搬到一个更大的地方去.

当然在这之前，还要考虑一种情况，就是要扩大内存时，realloc 的内存块刚好位于堆区的最后一块. 这时候只要调用 `mem_sbrk(newsize - oldsize)` 即可扩大内存. 如图所示.



以上就是 `mm_realloc` 的策略.

实验结果

为了方便是在本地写好的 mm.c，通过 scp 上传到服务器端编译、调试和运行.

运行的结果如下:

```
[2018202135@VM_0_46_centos malloclab-handout]$ make
gcc -Wall -g -O2 -m32 -std=gnu99   -c -o mm.o mm.c
gcc -Wall -g -O2 -m32 -std=gnu99 -o mdriver mdriver.o mm.o memlib.o fsecs.o fcyc.o clock.o ftimer.o
[2018202135@VM_0_46_centos malloclab-handout]$ ./mdriver -v
Team Name:invictus_gaming
Member 1 :Liguoxiang:2018202135@ruc.edu.cn
Using default tracefiles in /home/handin-malloc/traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.000156 36500
 1       yes  100%    5848  0.000183 31939
 2       yes   99%    6648  0.000199 33323
 3       yes  100%    5380  0.000133 40390
 4       yes  100%   14400  0.000132109339
 5       yes   96%    4800  0.001478  3247
 6       yes   95%    4800  0.001433  3351
 7       yes   55%   12000  0.017760   676
 8       yes   51%   24000  0.066953   358
 9       yes  100%   14401  0.000147 98233
10       yes   87%   14401  0.000154 93392
Total          89%  112372  0.088728  1266

Perf index = 54 (util) + 40 (thru) = 94/100
```

Performance index 为 94. 从分数分布上看，可见 Best fit 的速度很快. 但是也会导致空间利用率较低的情况(测试文件 7 和 8). 由于时间关系没有再往下优化，如果使用 Segregated free list 可能可以提高平均空间利用率从而获得更好的性能得分.

总结与反思

这次实验没有参考任何别人的代码和思路，主要都是自己完成的. 有个 bug 请教了别的同学才调试出来(就是一个空闲块大小不能小于 16 字节，bug 是因为把不足 16 字节的块也当做空闲块，导致空闲块设置指针和 Boundary tag 时导致把后面字节的块覆盖了).

没有使用任何全局变量和结构体，但是也没有使用任何宏定义. 没有宏定义不是很好的编程习惯，这一点要尽量在后面的实验中改正.