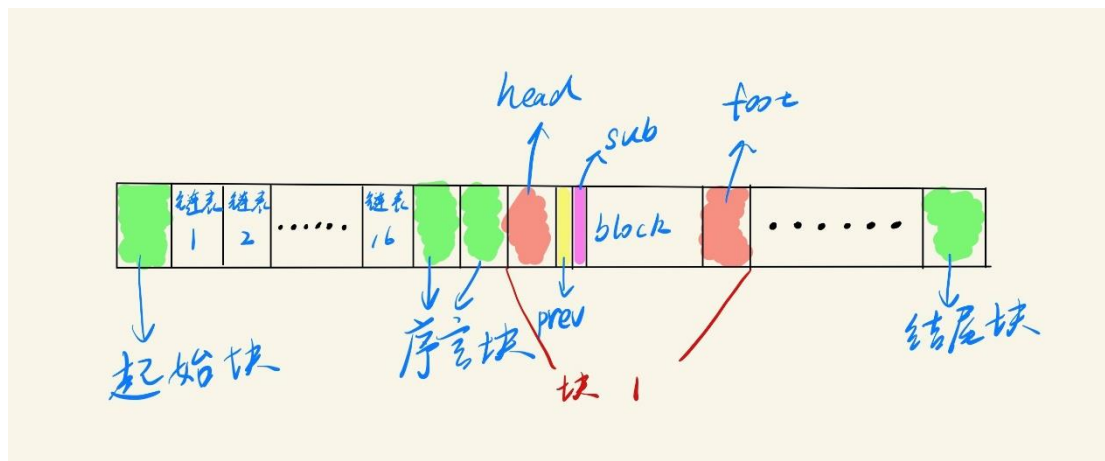


本次实验要求我们完成一个简易的内存分配器，我采用了显示的分离式链表管理内存，下面为实现该内存管理器所在 mm.c 中构建的函数。

我采用起始块到序言块之间的 16×4 个字节用于存放空闲链表头，每一类的块的大小范围为 $2^{(n-1)} \sim 2^n$ ，其中 n 为链表的序号，至于为什么为 16 个，经过测试发现，最大的空闲块出现在第九个测试 trace，小于 2^{16} ，因此 16 个类别足矣。每一个块的头标签后的前两个 WSIZE，第一个用于存放前一个空闲块的 bp 指针，第二个用于存放后一个空闲块的 bp 指针，即前驱和后继，这样构建了一个基本的链表。

下图为堆的结构图



各个基本函数的介绍

`int mm_init(void)`

该函数用于初始化堆，申请 $20 \times \text{WSIZE}$ 字节用于堆的构建，具体结构为上图所示随后我们初始化每一个基本结构，随后进行一次默认的拓展堆，完成对于堆的初始化。我们可以注意到，bp 即有效载荷指针指向的是 preve。

`void *extend_heap(size_t words)`

该函数用于拓展堆，首先我们要对于传入的 words 数进行双字对齐，随后申请对应大小的堆空间，最后封装好新申请的空间为空闲块，并封装新的结尾块，将新空闲块插入空闲链表，返回该空闲块的有效载荷指针。

`void *coalesce(void *bp)`

该函数用于合并空闲块，首先获得前后块的分配情况，然后当周围有空闲块时合并周围的空闲块，最后插入到空闲链表，然后返回该新空闲块的指针。

`void insert(void *bp)`

该函数用于插入空闲块，我们首先获得空闲块的大小和通过 search 函数获得空闲块所对应的链表序号，随后根据块大小将该块插入到合适的位置，使得所有链表的块都是从小到大排序的，有利于提高空间利用率。

`void delete(void *bp)`

该函数用于删除指定的空闲块，即更新该块前驱的后继和后继的前驱。

`int search(size_t v)`

该函数用于计算 v 大小对应的链表序号。步骤为下

确定 v 的大小类：通过一系列的位移操作，代码将输入值 v 的二进制表示中第一个非零位之前的位全部置零，然后将得到的值用于确定 v 的大小类。这是通过将不同位上的比特位移到高位，然后进行按位或运算的方式来实现的。

计算 x ：根据确定的大小类，计算 x 的值。 x 被初始化为 $(int)r - 4$ ，其中 r 是通过上述位移操作得到的大小类。

调整 x ：根据 x 的大小调整 x ，使得 x 落于设定好的范围内。

这个方法可以在 $O(\lg(N))$ 运算中求出 N 位整数的以 2 为底的对数，方法来自于 <https://graphics.stanford.edu/~seander/bithacks.html#IntegerLogLookup>

`void *mm_malloc(size_t size)`

该函数用于分配内存，主要步骤为对齐 $size$ ，寻找合适的空闲块，若没有合适的空闲块就拓展堆，最后调用 `place` 函数填充块，返回 `bp` 指针。

`void *find_fit(size_t asize)`

该函数用于寻找合适的空闲块，找到后返回对应的指针，否则返回 `NULL`

`void place(void *bp, size_t asize)`

该函数用于分离空闲块，以减少内存碎片。

`void mm_free(void *ptr)`

释放内存

`void *mm_realloc(void *ptr, size_t size)`

该函数用于重新分配内存，为了提高内存的利用率，添加了向后合并的算法。

函数时间复杂度分析

`mm_init`:

时间复杂度: $O(1)$

初始化函数分配了一个固定大小的堆，并初始化了数据结构。时间复杂度是常数。

`extend_heap`:

时间复杂度: $O(1)$

此函数通过从操作系统请求额外的内存来扩展堆。时间复杂度是常数，因为它涉及单个内存分配。

`coalesce`:

时间复杂度: $O(1)$

coalesce 函数在释放操作后合并相邻的空闲块。它涉及更新自由链表中的指针。由于受影响的空闲块数量是常数, 时间复杂度为 $O(1)$ 。

insert:

时间复杂度: $O(1)$

insert 函数将一个块插入到分离的自由链表中, 保持基于块大小的排序顺序。时间复杂度是 $O(1)$, 因为它涉及更新链表中的指针。

delete:

时间复杂度: $O(1)$

delete 函数从分离的自由链表中删除一个块。由于它涉及更新链表中的指针, 因此时间复杂度是 $O(1)$ 。

search:

时间复杂度: $O(1)$

search 函数用于确定空闲块的大小类。由于其操作是基于位移和常数级别的运算, 时间复杂度是 $O(1)$ 。

mm_malloc:

时间复杂度: 取决于 find_fit 和 place 函数

mm_malloc 函数分配内存, 其中 find_fit 的时间复杂度取决于搜索合适的空闲块, 而 place 的时间复杂度是 $O(1)$ 。因此, 整体的时间复杂度取决于 find_fit 函数。

find_fit:

时间复杂度: $O(k)$, 其中 k 是大小类的数量

find_fit 函数在不同的大小类中查找合适的空闲块。它的时间复杂度取决于搜索过程, 最坏情况下可能需要遍历整个链表。因此, 时间复杂度是 $O(k)$, 其中 k 是大小类的数量。

place:

时间复杂度: $O(1)$

place 函数用于将内存块分割或标记为已分配。由于它涉及更新自由链表中的指针, 时间复杂度是 $O(1)$ 。

mm_free:

时间复杂度: $O(1)$

mm_free 函数释放内存块并调用 coalesce 函数以合并空闲块。由于 coalesce 的时间复杂度是 $O(1)$, 因此整体的时间复杂度是 $O(1)$ 。

mm_realloc:

时间复杂度: 取决于内部操作

mm_realloc 函数重新分配内存块, 其中一些情况下需要调用 mm_malloc 和 mm_free。整体的时间复杂度取决于内部的操作, 可能是 $O(1)$ 或 $O(k)$ 的级别, 取决于具体的情况。

函数执行过程

`mm_init`:

分配一块固定大小的堆空间。

初始化各个大小类的链表头指针为 `NULL`。

创建序言块和结尾块，并设置为已分配状态。

调用 `extend_heap` 函数扩展堆空间，初始化堆的数据结构。

`mm_malloc`:

调用 `find_fit` 函数在合适的大小类中查找空闲块。

如果找到合适的块，调用 `place` 函数分割或标记该块为已分配状态。

如果未找到合适的块，调用 `extend_heap` 函数扩展堆，并再次调用 `place` 函数。

`mm_free`:

将指定内存块的头部和尾部标记为未分配状态。

调用 `coalesce` 函数合并相邻的空闲块，以避免碎片化。

`mm_realloc`:

如果 `ptr` 为 `NULL`，等同于 `mm_malloc(size)`，返回新分配的内存块。

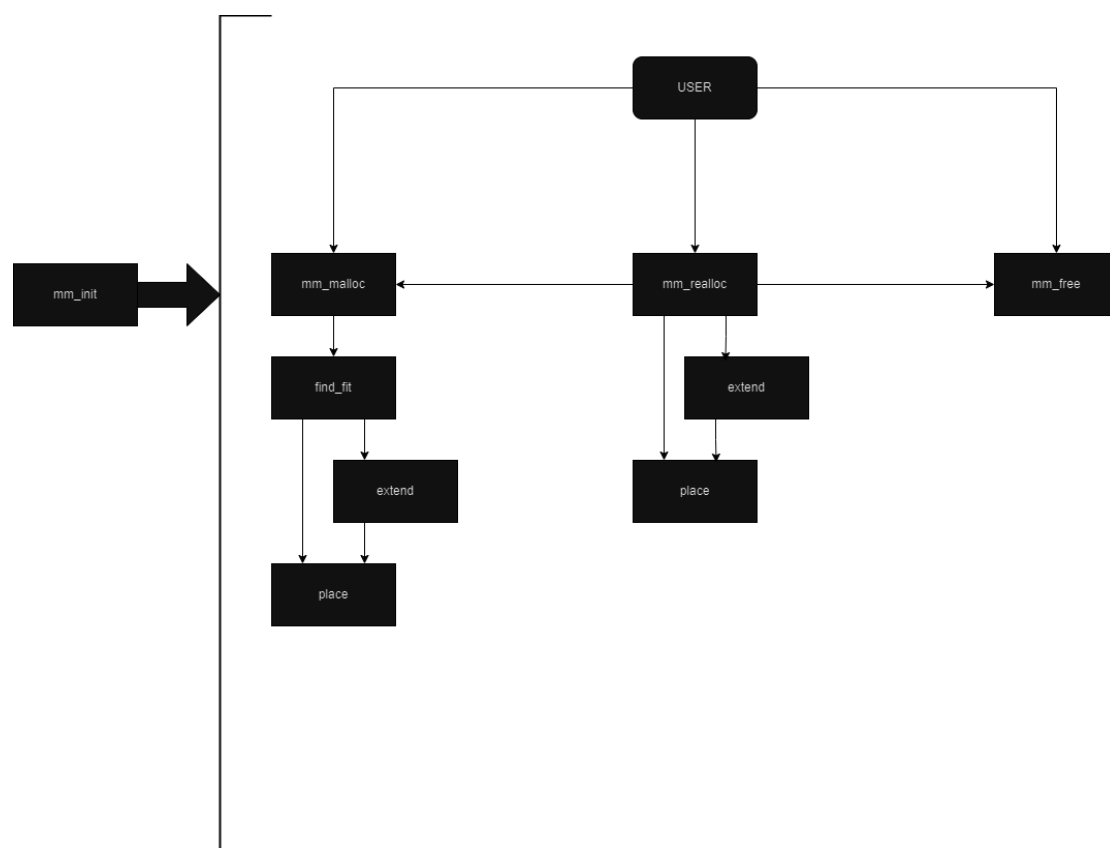
如果 `size` 为 0，等同于 `mm_free(ptr)`，释放原先分配的内存块。

如果新大小等于原先大小，直接返回原先的内存块。

根据先前块和后继块的情况，可能会进行内存块的合并、分割或扩展。

最终调用 `mm_malloc` 分配新大小的内存，将数据从旧内存复制到新内存，然后调用 `mm_free` 释放旧内存。

执行过程图如下



对于 util 的小优化

我们知道内存利用率主要取决于内存碎片和单个块的大小，优化单个块的大小较为困难，这需要修改内存管理器整体的逻辑，本次 lab 主要以减少内存碎片为优化目标。

链表的管理：

我们可以在插入新的空闲块时，将新插入的块根据 size 插入，使得每个链表的空闲块都是从小到大排序的，这样做可以使得我们在 find_fit 函数中，遇到的第一个大小大于等于所需大小的块就是所有块里面的最优块，这样即可尽量减小内存碎片。

Place 的逻辑：

我们在 place 时，会将较大的空闲块分割为两块来进行操作，那么如何设定分割的阈值对于减小较小的内存碎片是很有影响的，经过实验发现，当阈值设定为 $32 \times \text{DSIZE}$ 时对于我们的输入 trace 样例效果最优，但是事实上这个阈值是和输入数据的特征密切相关的。

realloc 内存的逻辑：

realloc 内时，我们可以检测前后的内存是否为空闲块，若合并完周围空闲块后可以满足 realloc 的大小需求，我们可以合并前后的块，并移动数据，但是由于前合并需要改变指针和移动数据，并且 trace 输入中对于前移的 realloc 极少，对于实验结果几乎没有影响，因此没有完成前合并。

而对于后合并，还有三种情况，一种为后面为结尾块，我们直接扩展所差的大小再分配即可，另两种为后面为空闲块，一种为空闲块的大小够重新分配，直接合并并分配，另一种为后面的空闲块不够，但是空闲块后面为结尾块，扩展所差大小再分配即可。

修改初始化堆大小:

初始化堆时, 我们不需要初始化过于大的内存块, 因为这样容易造成大量的内存被空闲, 我们将 CHUNKSIZE 修改为 $1 < 8$ 即可得到良好的效果。

本次实验对于 debug 的要求有些困难, 因为一个函数的错误往往会引起其他函数的内存访问错误, 因此需要对于函数的调用栈和整体的逻辑有较为清晰的理解。

最 终 实 验 完 成 如 下

```
Results for mm malloc:
trace valid util ops secs Kops
0 yes 99% 5694 0.000597 9539
1 yes 100% 5848 0.000354 16501
2 yes 99% 6648 0.000697 9538
3 yes 100% 5380 0.000531 10126
4 yes 96% 14400 0.000982 14661
5 yes 96% 4800 0.001375 3490
6 yes 95% 4800 0.001313 3654
7 yes 55% 12000 0.001098 10926
8 yes 51% 24000 0.002114 11355
9 yes 100% 14401 0.000701 20552
10 yes 81% 14401 0.000454 31741
Total 88% 112372 0.010217 10999

Perf index = 53 (util) + 40 (thru) = 93/100
harry@ubuntu22:~/Code/CSAPP-LAB/malloclab-handout$ ./mdriver
Team Name:SJF
Member 1 :SJF:10225501403@stu.ecnu.edu.cn
Using default tracefiles in /home/harry/Code/CSAPP-LAB/malloclab-handout/trace
s/
Perf index = 53 (util) + 40 (thru) = 93/100
harry@ubuntu22:~/Code/CSAPP-LAB/malloclab-handout$
```

对于要取得更高的分数, 可能需要从块的结构和空闲内存的管理入手, 即删除不必要的结构, 如已分分配块的 foot, 转而用 head 的第 31 位记录前一块的分配情况, 再或者是用堆来管理空闲内存。

调试函数设置

int checkBlock(void *bp)

检查块是否双字对齐, 检车是否头脚标记相同

void checkFreeList()

检查所有空闲块是否可以通过 checkBlock

void checkHeap()

检查整个堆是否可以通过 checkBlock

void printBlock(void *bp)

打印特定的块的信息

```
void printFreeList()
```

打印空闲链表