Malloclab实验报告

实验过的算法及实验过程分析

第一版

第一版是仿照书上的算法写的。整个堆的开头设有序言块，结尾设有结尾块。空闲块的组织形式为隐式空闲链表，每个块都有头部和尾部的tag来记录块大小和当前块是否已分配。在分配的时候用的是first fit来找可用空闲块，且分配的时候有分割的操作，在释放一个块的时候有合并的操作。

用到的宏定义

因为代码中存在很多指针加减、取值、赋值的操作，把这些操作提前用宏定义写好，在后面的代码部分就不容易出错。

```
#define WSIZE 4 //字
#define DSIZE 8 //双字
#define CHUNKSIZE (1<<12) //每次向堆申请的内存大小

#define MAX(x,y) ((x)>(y)? (x):(y))
#define PACK(size, alloc) ((size) | (alloc)) //标记该块是否已分配
#define GET(p) (*(unsigned int*)(p)) //得到p地址处存的内容
#define PUT(p, val) (*(unsigned int*)(p)=(val)) //往p地址处存入值
#define GET_SIZE(p) (GET(p) & ~0x7) //得到块的大小
#define GET_ALLOC(p) (GET(p) & 0x1) //得到该块是否已分配

//bp指向块的有效载荷的第一个字节
#define HDRP(bp) ((char*)(bp) - WSIZE) //得到块的头部tag地址
#define FTRP(bp) ((char*)(bp) + GET_SIZE(HDRP(bp)) - DSIZE) //得到块的尾部tag地址
#define NEXT_BLKP(bp) ((char*)(bp) + GET_SIZE(((char*)(bp) - WSIZE))) //得到下一个块的地址
#define PREV_BLKP(bp) ((char*)(bp) - GET_SIZE(((char*)(bp) - DSIZE))) //得到上一个块的地址
```

具体算法

`int mm_init(void)`在初始化的时候，调用Extend函数申请空间。在堆的开头放了一个序言块，在堆的末尾放了个结尾块，这两个边界块都设置为已分配块，这样在合并的时候就不用额外判断边界，也不会有越界的问题。

`void *Extend(size_t words)`调用mem_sbrk来申请空间。如果申请成功的话，设置申请到的空闲块的头尾tag和整个堆的新结尾块，并且调用Combine函数对新的空闲块进行合并，然后返回合并后的块的有效载荷的第一个字节。

`void *Combine(void *bp)`对空闲块进行合并的操作。判断前后块是否空闲，如果都不空闲则直接返回当前块的有效载荷的第一个字节。如果前后块有空闲的话，就修改合并后的块的头部和尾部tag里记录的块大小，并返回合并后的块的有效载荷的第一个字节。

`void *mm_malloc(size_t size)`先把size按8字节对齐进行处理，然后用Find函数找可用的空闲块。如果找不到的话就调用Extend函数申请额外的堆空间，找到的话就调用Place函数进行处理。

`void *Find(size_t asize)`用first fit找到隐式空闲链表里第一个可以用的空闲块。

`void Place(void *bp, size_t asize)`在分配的时候修改头尾tag里的大小和是否分配的标记位，当所选块的大小比所需大小大于两字的时候才进行分割，分割也只是改变两个块的头尾tag里记录的大小。

`void mm_free(void *bp)`修改头尾tag里的标记位为空闲，然后调用Combine函数进行合并。

`void *mm_realloc(void *ptr, size_t size)`判断size，如果是0则相当于free，如果等于当前块大小则直接返回原地址，其他情况都重新malloc，然后调用memcpy函数复制块里的内容。

遇到问题及解决方案

因为是在理解了书上的算法后写的，所以没有什么问题。

优化想法

1. 把隐式空闲链表改为显示空闲链表；
2. 把Find里找空闲块的方法改为best fit；
3. 分配块里只保留头部tag。

第二版

该版的空闲块组织形式是显示空闲链表。每个块还是都有头部和尾部tag，其中空闲块的头部tag后的第一个字和第二个字里存链表中前后块的地址。整个堆的开头设有序言块，结尾设有结尾块，形式跟第一版的类似，只是序言块的第一个字是作为显示空闲链表的头指针，存了链表里第一个空闲块的地址。同样有分割和合并的操作。

用到的宏定义

在第一版里的宏定义的基础上，为显示空闲链表的操作添加了两个宏定义。

```
#define PRE_BPFREE(bp) ((char*)(bp)) //链里上一个块的地址
#define NEXT_BPFREE(bp) ((char*)(bp) + WSIZE)//链里下一个块的地址
```

具体算法

`int mm_init(void)`思路与第一版相同，只是序言块的第一个字要作为链表头，用0进行初始化。

`void *Extend(size_t words)`思路与第一版相同，只是在设置完新的空闲块的头尾tag后，还需要初始化该块记录前后块地址处为0。

`void *Combine(void *bp)`做法与第一版相同，是把当前块与物理地址上的前后块合并。只是在合并的时候需要先调用Move函数，把可以合并的块从链表里移出，并且在合并结束后，调用Insert函数，把合并完的新块放入链表中。

`void Move(char *p)`把p指向的块从链表中移出。先获得链表中前后块的地址，根据当前块移出后的链表关系把前后块的地址存到前后块里相应的位置上，如果当前块是链表中的第一块或者最后一块的时候要注意存的位置。

`void Insert(char *p)`把p指向的块放入链表头，即作为链表的第一块。

`void *mm_malloc(size_t size)`与第一版算法相同。

`void *Find(size_t asize)`与第一版算法类似，在空闲块的第3个字里存了链表中下一块的地址，所以直接用GET(NEXT_BPFREE(bp))获得链表中下一个空闲块的地址。

`void Place(void *bp, size_t asize)`与第一版算法类似，只是一开始要先调用Move函数，把选中的当前块从空闲链表中移出，只有当所选块的大小比所需大小大于四字的时候才进行分割，分割的时候要初始化分出的空闲块里存前后块地址的地方。

`void mm_free(void *bp)`与第一版算法类似，只要初始化该块存前后块地址的地方。

`void *mm_realloc(void *ptr, size_t size)`与第一版算法相同。

遇到的问题及解决方法

1. 根据原先的优化的想法，是打算把分配块里的尾部tag去掉，然后在每个块的头部tag里记录前一块是否已分配。在处理的过程中发现很麻烦，容易出错，效果也不是很好，所以就还是保留了分配块的尾部tag。
2. 在用显示空闲链表的时候需要在空闲块里存入链表中前后块的地址。一开始直接想做类型转换，到网上找到了地址和int相互转换的函数，但是感觉不太靠谱，就没敢用。后来想到可以用相对的地址，即把两个地址相减后的值当成指针存入空闲块。但是因为类型转换的问题，在地址加减时出现问题，一直出现segmentation fault的报错。在用gdb调试的时候，发现前后两次调用Move函数移出链表里唯一一个块的时候，第一次调用的结果正确，但第二次错误。分析代码本身是没有问题的，打印出链表里存的内容的时候发现是类型有问题。后来看到实验指导的提示里写了可以用宏定义实现指针转换，才想到可以直接用宏定义里的PUT来直接把指针存入空闲块。
3. 为了提高利用率，本来打算是把整个堆的序言块简化，并去掉结尾块，然后在有边界的地方额外加入判断。后来发现这样还是会越界。在简化序言块的时候发现，虽然在malloc里有对分配的大小进行调整，但还是会出现不对齐的报错，分析认为在这个程序里是否对齐是根据块的首地址来判断的，所以序言块的大小不能是任意的。并且再次分析后认为序言块和结尾块本身不大，对利用率的影响不大，没必要修改，所以还是按照原样保留。

优化

1. 把Find函数里找可用空闲块的方法从first fit改为best fit，分数从84上升到86。
2. 考虑改进为分离适配算法。

第三版（最终版）

该版直接在第二版的显示空闲链表的基础上写了分离适配算法，实际也就是把一个显示空闲链表改为多个显示空闲链表，一共分出了11个链表。每个块都有头部和尾部tag，其中空闲块的头部tag后的第一个字和第二个字里存链表中前后块的地址。整个堆的开头设有序言块，结尾设有结尾块。其中序言块里有11个字是作为11个链表的头指针，里面存相应链表里的第一块的地址。在Find里寻找可用的空闲块和Insert里把空闲块放入链的时候，先判断块的大小，然后到相应的链里找。

本来觉得分离适配算法会比显示空闲链表好很多，但是分数居然还是一样的。分析后认为，第二版中用的是best fit，找到的块其实跟分离适配里找到的块是基本一样的，所以提升的只有吞吐量，而原本的显示空闲链表的版本的吞吐率得分已经满了，所以得分没有提升。

优化方法

1. 想到了改进一直没有改过的realloc的算法。realloc的算法一直没有改进是因为觉得原版把realloc的功能转化为malloc和free，可以直接利用写好的mm_malloc和mm_free函数，只要改进malloc和free函数就好。但是想到实际的realloc函数中不会把所有块都重新分配，只有空间不够了才会给该块重新分配另外一个空间。所以在改进的时候就利用这一点，先考虑当前块的物理位置上的前后块是否空闲，可否直接扩展空间，空间不够时才额外分配。改进后得分上升到88。
2. 对于原版的算法实在不知道怎么改了，所以只能针对测试集来进行改进。发现在realloc.rep和realloc2.rep这两个测试集里，都是一直只对一个块进行realloc，而且realloc的大小是一直递增的。这种情况下在realloc的时候就没有必要分割，反正后面也用得到这些空间。所以就把realloc函数里分割的操作去掉了，后面两个测试集的分数就上升了，总得分上升到93。

最终的具体算法

`int mm_init(void)` 在初始化的时候，调用Extend函数申请空间。在堆的开头放了一个序言块，在堆的末尾放了个结尾块。且在序言块里放了各链表的链表头，作为存链表中第一个块的地址的地方，都初始化为0。

`void *Extend(size_t words)` 调用mem_sbrk来申请空间。如果申请成功的话，设置申请到的空闲块的头尾tag和整个堆的新结尾块，并且调用Combine函数对新的空闲块进行合并，然后返回合并后的块的有效载荷的第一个字节。

`void *Combine(void *bp)` 对空闲块进行合并的操作。判断前后块是否空闲，如果都不空闲则直接返回当前块的有效载荷的第一个字节。如果前后块有空闲的话，就修改合并后的块的头部和尾部tag里记录的块大小。在合并的时候需要先调用Move函数，把可以合并的块从链表里移出，并且在合并结束后，调用Insert函数，把合并完的新块放入链表中。

`void Move(char *p)` 把p指向的块从链表中移出。先获得链表中前后块的地址，根据当前块移出后的链表关系把前后块的地址存到前后块里相应的位置上，如果当前块是链表中的第一块的话还要调用Find_Line函数来判断是哪一条链的第一块。

`void Insert(char *p)` 先调用Find_Line函数找到当前块的大小应属于哪个链表，然后把p指向的块放入对应链表头，即作为链表的第一块。

`int Find_Line(size_t size)` 判断size大小的块是属于哪条链。

`void *mm_malloc(size_t size)` 先把size按8字节对齐进行处理，然后用Find函数找可用的空闲块。如果找不到的话就调用Extend函数申请额外的堆空间，找到的话就调用Place函数进行处理。

`void *Find(size_t asize)` 调用Find_Line函数找到对应大小的链表，然后用best fit在该条链表里找到可用的空闲块。如果该条链找不到的话，就到更上一级的链表里找，直到找到或者所有链都找完。

`void Place(void *bp, size_t asize)` 在分配的时候修改头尾tag里的大小和是否分配的标记位，当所选块的大小比所需大小大于四字的时候才进行分割，分割也只是改变两个块的头尾tag里记录的大小，然后初始化分割后的空闲块里记录前后块地址的位置。

`void mm_free(void *bp)` 修改头尾tag里的标记位为空闲，初始化该块存前后块地址的地方，然后调用Combine函数进行合并。

`void *mm_realloc(void *ptr, size_t size)`判断size，如果是0则相当于free。不是0的话先把size进行对齐，再进行后面的判断。如果小于等于当前块大小则直接返回原地址。如果size大于当前块大小，调用Find_Realloc函数来判断该块前后有无足够的可以扩展的空间并进行扩展，并返回扩展情况。Find_Realloc的返回值为1则是没有足够的空间，需要重新分配，并返回新分配的块的指针；返回值为2则是空间直接往后扩展了，不需要复制内容，块的地址也没有变；返回值为3则是空间有向前扩展，需要先把内容复制到合并后的块的对应位置里，块的地址也变为前一个块的地址。

`int Find_Realloc(void* ptr, size_t newsize)`跟Combine函数类似，只是如果前后块空闲，还得再判断如果合并的话，大小是否足够。如果没法合并或者大小不够的话，返回1；如果只向后合并的话，返回2；如果有向前合并的话，返回3。

遇到的问题及解决办法

在主要的几种算法都写过后，不知道要怎么再优化了。解决方法就是不只是盯着大的算法来改进，还可以考虑之前没有改变过的地方。在所有地方都考虑过后，可以针对测试集进行改进。

反思总结

1. 先考虑优化大的算法，不要一开始就考虑把序言块、结尾块等这些简化掉，这些部分本身占的空间不大，而且又是可以简化操作、防止出错的，自己写的判断不会比这些设置好的边界好用。
2. 先想好靠谱的方法再往下写，不然可能写完了才发现有调不好的根本性的bug。
3. 调bug的时候不能只反复看自己代码的逻辑问题，有可能是某个处理方法的问题而不是代码逻辑的问题。
4. 把学过的主要的算法都试过后，优化的时候就要考虑没有改动过的地方。所有地方都优化过后，就得针对测试集进行优化。