malloclab实验报告

实验目的

实现自己的动态内存分配器，提高峰值利用率、分配速度，减少内部碎片和外部碎片。

实验记录

书中提到c语言malloc包是使用分离适配方式实现的，在使用首次适配搜索时即可达到对于整个堆的最佳适配搜索的内存利用率。以下分别根据空闲链表实现、malloc函数包的实现、辅助函数等三部分介绍我的实现。

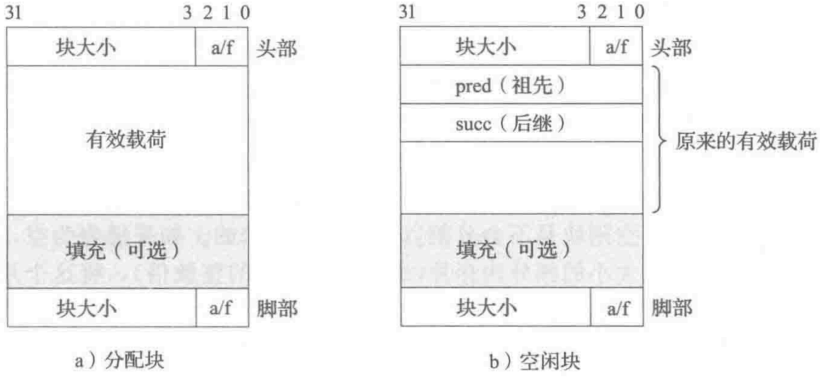
空闲链表

在我实现的分离适配中，第n个大小类中空闲块中字的个数为2的n次方到2的n+1次方减一。一共有16个大小类，每个大小类中空闲块大小如下表所示：

分离适配大小类中空闲块大小

n	空闲块大小	n	空闲块大小	n	空闲块大小	n	空闲块大小
0	1	4	16～31	8	256～511	12	4096～8191
1	2～3	5	32～63	9	512～1023	13	8192～16383
2	4～7	6	64～127	10	1024～2047	14	16384～32767
3	8～15	7	128～255	11	2048～4095	15	>32767

分配块与空闲块的结构如下图所示。已分配块由头部、有效载荷与填充、尾部组成，与书中结构一致。空闲块由头部、前驱、后继、有效载荷与填充、尾部组成，是双向空闲链表的形式，这是方便链表的插入、删除结点，以及在链表中搜索。



已分配块与空闲块的形式

插入节点的策略是：遍历每个大小类的对应空闲链表头节点，每次将空闲节点的大小右移一位。当空闲节点的大小不大于1时，或遍历到第15个大小类时退出，此时就找到了待插入节点所属的

大小类的对应空闲链表头节点。遍历这个空闲链表，找到第一个大小大于待插入节点大小的空闲块，在其前面插入。这样得到的空闲链表中，空闲块的大小是递增的。在插入时也根据插入空链表、在链表头插入、在链表中插入、在链表尾插入等四种情况分别处理。

删除节点的策略是：如果待删除节点的前驱是空指针，则其一定是某个大小类对应的空闲链表头节点，通过在插入节点策略中所述的方法找到对应的大小类，将其对应的空闲链表头节点置为待删除节点的后继，如果后继不为空指针，则设置后继的前驱为空指针。如果待删除节点在空闲链表中，则根据在链表中删除和在链表尾删除等两种情况分别处理。

malloc函数包

malloc函数包分为mm_init，mm_malloc，mm_free，mm_realloc等四个函数。

mm_init的实现：初始化空闲链表，将所有大小类对应的空闲链表头置为空指针并初始化首言区。接下来使用辅助函数extend_heap来扩展堆的大小。最后使用辅助函数check_heap对堆进行检查。

mm_malloc的实现：若分配的字节大小为0，则返回空指针，否则使用ADJUST宏得到对齐之后的字节大小。根据对齐后的字节大小找到相应的大小类，在大小类对应的空闲链表中查找第一个大小大于对齐后的字节大小的空闲块。如果在相应的大小类对应的空闲链表中无法找到能够利用的空闲块，则在更大的大小类对应的空闲链表中进行查找。如果找不到能力利用的空闲块，则调用extend_heap函数来扩展堆的大小。最后调用辅助函数place进行放置。

mm_free的实现：将待释放块的头部和脚部都设置为空闲，调用辅助函数coalesce合并前后的空闲块。

mm_realloc的实现：若重新分配的字节大小为0，则返回空指针，否则使用ADJUST宏得到对齐之后的字节大小。如果原分配块的大小小于对齐之后的字节大小，则查看下一个块是否是空闲块，若是则将合并后的块的大小与对齐之后的字节大小比较，若大于则对合并后的块进行分割，放置重新分配之后的块和空闲块，保证空闲块的大小能够存放必要的信息并调用辅助函数coalesce合并前后的空闲块，若小于则释放合并后的块，调用mm_malloc重新分配空闲块并复制数据；若不是则释放当前块，调用mm_malloc重新分配空闲块并复制数据。如果原分配块的大小大于对齐之后的字节大小，则对分配块进行分割，放置重新分配之后的块和空闲块，保证空闲块的大小能够存放必要的信息并调用辅助函数coalesce合并前后的空闲块。

辅助函数

辅助函数包括下列函数：extend_heap，place，coalesce，check_heap，check_block，check_list，print_block，print_list等。

extend_heap的实现：与书中一致，使用memlib.c提供的mem_sbrk函数来在堆上分配空间，最后调用辅助函数coalesce来合并前后空闲块。

place的实现：我实现的place函数充分考虑前后空闲块的空间来提高内存使用率，**这也是本次lab的核心**。place函数输入一个空闲块指针和待放置块的大小，如果空闲块大小与待放置块的大小之差小于4个四字，则直接返回，因为不能存放空闲块必要的信息。如果空闲块大小与待放置块的大小之差大于4个四字，则可以将空闲块分为两部分，一部分放置待放置块，另一部分为空闲块。这时有空闲块在前和空闲块在后两种选择，**对此我使用了一种贪婪的策略**：

1. 在前后都是空闲块时，我让空闲块与前后空闲块中更大的相邻，这样能够立刻合成更大的空闲块。
2. 在前后有仅有一个为空闲块时，我让空闲块与前后块中空闲的那个相邻，这样能够立刻合成更大的空闲块。

3. 在前后都为分配块时，假设前后块被释放的可能性相同，我让空闲块与前后分配块中更大的相邻，这样在分配块被释放时，能够合成更大的空闲块。

经过试验验证，这样的放置策略要比固定地在前或在后放置空闲块要好，也比根据阈值来决定在前或在后放置空闲块要好。

coalesce的实现：与书中基本一致，分四种情况实现，但我增加了对前后块是否有效的检查。

check_heap：检查每个大小类对应的空闲链表，检查每个块。调用check_list与check_block。

print_list：以json格式在一行内打印空闲链表，如：

{"size": [2048,4095],"nodes": [{"pointer": 0xf6bf32b0,"header": {"size": 2240,"alloc": f},"pred": (nil),"succ": 0xf69cd930}{"pointer": 0xf69cd930,"header": {"size": 2992,"alloc": f},"pred": 0xf6bf32b0,"succ": 0xf6ce2d88}{"pointer": 0xf6ce2d88,"header": {"size": 3856,"alloc": f},"pred": 0xf69cd930,"succ": (nil)}]}

print_block：以json格式在一行内打印块，如：

{"pointer": 0xf6969c98,"header": {"size": 14400,"alloc": a},"footer": {"size": 14400,"alloc": a}}

实验结果

直接运行./mdriver –v，得到如下输出：

```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.000327 17434
 1       yes   99%    5848  0.000329 17753
 2       yes   99%    6648  0.000385 17272
 3       yes   99%    5380  0.000306 17582
 4       yes   99%   14400  0.002248  6405
 5       yes   96%    4800  0.000686  6993
 6       yes   95%    4800  0.000648  7409
 7       yes   60%   12000  0.001209  9926
 8       yes   52%   24000  0.002789  8606
 9       yes   99%   14401  0.000491 29300
10       yes   98%   14401  0.000457 31484
Total          91%  112372  0.009876 11378

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

得到了94分的成绩。可以看到第7和第8个测试文件的分数较低，而其他测试文件上的分数较好。同时我也发现分配速度的要求较低，基本上都是满分。

总结

在本次malloclab中，我实现了分离适配，经过各种debug后终于得到了正确的程序，同时也了解了详细的动态内存分配的实现。在写程序的过程中，由于我使用的电脑无法debug，我只能通过在mm.c中定义DEBUG和VERBOSE宏来输出各种信息和检查堆，进而进行debug。这个过程比较麻烦，而使用gdb也不是非常方便。最后测试时要注意去掉DEBUG和VERBOSE宏的定义。