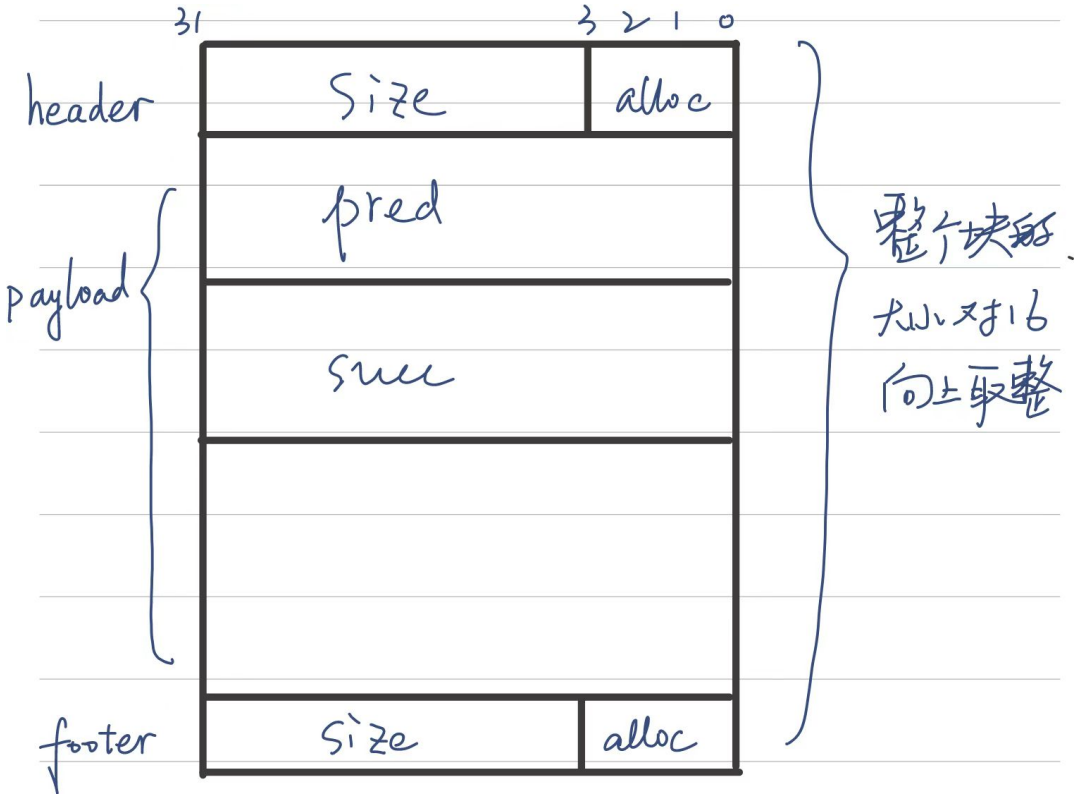
Malloc Lab实验 REPORT

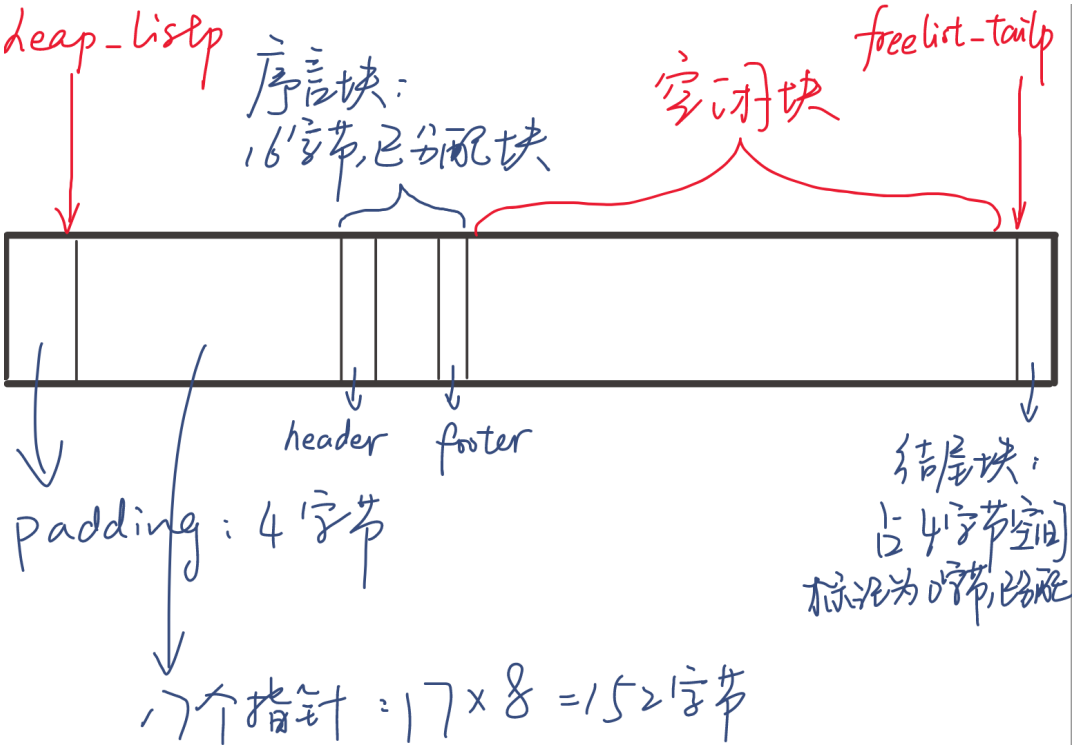
计15 宋驰 2021010797

实验思路

- 在对于隐式空闲链表和显式空闲链表的理解基础上，先实现显式空闲链表，后为了提升性能，将显式空闲链表改为分离空闲链表，并对一些细节进行优化。
- 显式空闲链表的块结构：
  - 为了节约空间，头部和脚部都是4字节，payload中至少含有两个指针，因此payload的大小至少是16字节。由于需要16字节对齐，整个块的大小需要是16的倍数，因此payload加上头部和脚部的大小后，需要对16向上取整。因此得到一个正常的块大小至少是32字节（序言块和结尾块除外）。



  - 在显式空闲链表中，将所有空闲块组织成为一个链表，每次搜索采用首次适配，插入新的空闲块时直接将该空闲块插入在链表的头部。
- 分离空闲链表：
  - 分离空闲链表的块结构和显式空闲链表的块结构相同，不同点在对于空闲块的组织上。由于最小块的大小是32，按照2的次幂的大小来划分等价类，{32}，{33-64}，{65-128}，...{2^20+1，正无穷}，共计17个等价类。
  - mm_init函数中，需要一开始初始化17个指针，每个都对应一个等价类的头指针，考虑到16字节对齐，在所有指针的前面还需要4字节的padding。具体见下图：



- mm_malloc函数中，先根据参数size计算出需要分配的块的大小asize。然后调用find_fit函数找到合适的空闲块，调用place函数将该空闲块进行分配。若find_fit失败，则调用extend_heap函数扩展堆，随后再调用place函数。
- mm_free函数中，只需将ptr指向的块设置为空闲块，然后调用合并函数coalesce。
- mm_realloc函数中，对于新块小于等于旧块的情况很好处理，而对于新块大于旧块的情况，先判断该旧块的后面一个块是否是空闲块，如果是的话判断两个块的大小之和是否满足比新块大，如果能满足的话就不需要重新找空闲块了。若不行则另找空闲块。
- 其他的辅助函数中，search是通过块的大小来判断属于哪一个等价类；insert和delete是用于在分离空闲链表中，空闲块插入链表和从链表中删除；extend_heap是扩展堆的函数；coalesce是用于检查一个空闲块的前后是否存在可以合并的空闲块，并且将合并后的空闲块插入到链表中；place是用于在指针bp指向的空闲块中放置一个大小为asize的分配块；find_fit用于在链表的等价类中首次适配合适的空闲块。

数据分析和原因分析

- 在实现了基本的分离空闲链表后，得到了一个还可以的分数，在85分左右。分离空闲链表可以有效优化如binary的两个trace，以binary2-bal.rep为例，采用显式空闲链表时，在malloc128的时候，会遍历所有的已释放的112的空闲块，而且无法分配。这种遍历是无用的，消耗了大量的时间。采用了分离空闲链表后，112和128在16字节对齐后的块大小不在同一个等价类中，这样直接避免了上述的遍历过程，极大地提高了吞吐率。
- 但分离空闲链表的数据中，有几个rep的利用率较为低下，因此进行以下的优化。
  - 在coalescing-bal.rep这个trace中，不停地malloc两个4095大小的块，然后分别释放，然后再malloc8190大小的块，然后再释放，如此循环，会发现这一trace的空间利用率低的原因是4095加上头部、脚部以及padding，需要分配的大小超过了4096，而4096是一次extend_heap的大小，这就导致为了放下这两个4095的块，需要extend_heap三次，开辟的空间是3 * 4096，导致空间利用率在66%左右。因此采用的优化手段是在init初始化的时候，extend_heap的大小比4096略大，这样两个4095的块就能放得下，省去第三次extend_heap。

```
  if (extend_heap((CHUNKSIZE + 5 * ALIGNMENT) / WSIZE) == NULL)
    return -1;
  return 0;
```

- 在realloc的两个trace中，空间利用率也同样低下，这两个trace的malloc、free、realloc的过程比较复杂，造成了大量的碎片。首先对于place函数进行优化，在原来的place中，若可以切分出一个多余的小块，默认将这个小块放在已分配块的后面，这样导致了在realloc的trace中块的合并受到问题，采用的优化手段是当分配块的大小如果大于100时，将切分出来的小空闲块放在这个分配块的前面。如果小于则和之前一样，这样的方式有利于块的合并。

```
    if (asize >= 100) {          /* if aszie is big */
      PUT(HDRP(bp), PACK(csize - asize, 0));
      PUT(FTRP(bp), PACK(csize - asize, 0));

      p = NEXT_BLKP(bp);

      PUT(HDRP(p), PACK(asize, 1));
      PUT(FTRP(p), PACK(asize, 1));

      coalesce(bp);
      return p;
    }
```

- 然后，realloc的函数也进行优化，在改变了place函数后，realloc的trace中有一个块在一直增长并realloc，这个块会一直放在毗邻结尾块的位置，因此当我们需要realloc的时候，我们不必将这个很大的分配块挪至别的地方，只需判断它和结尾块相邻的前提下，直接做一次extend_heap，这样用多少extend多少，减少了碎片，提升了空间利用率。

```
    else if(!GET_SIZE(HDRP(nextptr))) {   // if nextptr is epilogue,
just extend_heap
        size_t extend_size = asize - oldsize;
        if((long)(mem_sbrk(extend_size)) == -1)
            return NULL;

        PUT(HDRP(ptr), PACK(oldsize + extend_size, 1));
        PUT(FTRP(ptr), PACK(oldsize + extend_size, 1));
        PUT(HDRP(NEXT_BLKP(ptr)), PACK(0, 1));
        return ptr;
    }
```

- 优化后的数据表现见下图：

```
Results for mm malloc:
trace            name     valid  util     ops      secs   Kops
 1     amptjp-bal.rep       yes   99%    5694  0.000370  15373
 2       cccp-bal.rep       yes   98%    5848  0.000415  14078
 3    cp-decl-bal.rep       yes   98%    6648  0.000475  13984
 4       expr-bal.rep       yes   99%    5380  0.000377  14274
 5 coalescing-bal.rep       yes   97%   14400  0.000583  24704
 6     random-bal.rep       yes   94%    4800  0.000486   9875
 7    random2-bal.rep       yes   91%    4800  0.000492   9746
 8     binary-bal.rep       yes   91%   12000  0.000567  21160
 9    binary2-bal.rep       yes   81%   24000  0.001061  22618
10    realloc-bal.rep       yes   99%   14401  0.000341  42232
11   realloc2-bal.rep       yes   99%   14401  0.000191  75477
Total                            95%  112372  0.005360  20966

Score = (57 (util) + 40 (thru)) * 11/11 (testcase) = 97/100
```

- 可以看到，在优化之后，coalescing-bal.rep、realloc-bal.rep、realloc2-bal.rep的空间利用率明显提高，原因在上面的优化过程中已经做了分析。
- 我们还可以注意到，binary的两个trace的空间利用率还存在进一步提高的可能，其中binary2-bal.rep的空间利用率是81%，这个trace是反复malloc16和112，之后释放所有112的块，然后插入很多128的块，这里的空间利用率较低的原因可能是由于头部和脚部导致的，因为malloc的块的数量实在太多，对于大小为16的块而言，加上头部和脚部的大小，再16字节对齐，导致需要占据32字节的空间，对于空间利用率有一定的负面影响。

困难、心得、技巧

- 困难
  - 在实验刚开始的时候，由于我之前对于宏定义和指针的操作不熟练，导致理解csapp上的示例代码就花费了一定的时间，但好在自己很快适应并理解了这些操作的具体含义。
  - 调试segmentation fault的时候，尝试在代码中加入printf来确定错误的位置，一开始发现没有printf出来任何语句，这一问题困扰了我很长时间，后来在帮助下才意识到在触发了segmentation fault之后，printf中的语句没有加'\n'就不会输出，了解了这一细节后自己debug的问题迎刃而解。
  - 一开始自己在完成了从显式空闲链表到分离空闲链表的转化后，分数不增反降，这一问题困扰了我很久，在尝试了很多再改进的措施后，发现是自己一开始划分的等价类数目太少了，导致了很多无效的遍历。
- 心得和技巧
  - 这次malloclab对于自己的代码能力和对于malloc的理解深度都提升很大，自己也觉得这应该是最难的一次实验。由于之前对于宏定义和指针的不熟练，让我在开始的时候就遭遇困难，在完成实验的过程中无论是debug还是优化性能，都花了很长的时间，但在这个过程中自己对于malloc的整体过程和实现细节的理解逐渐深入，有很大的收获。
  - 在做本次malloclab的过程中，我也体会到了宏定义操作的好处，之前很少使用宏定义的我，在做这次lab的过程中能明显感受到宏定义让很多代码显得清晰易懂。同时，自己也感受到malloc函数背后的实现细节以及优化性能的思想是非常深奥的。