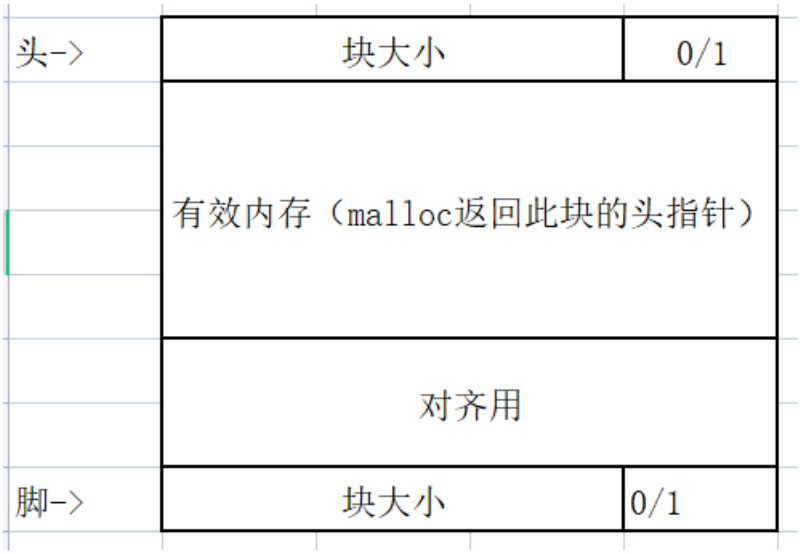
Malloc Lab实验报告

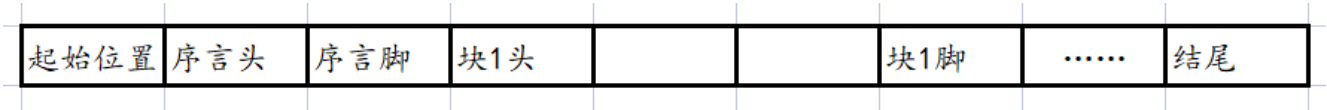
致理-信计11 慎梓戎 2021013420

实验要求

Malloc lab的实验目的是实现一个内存分配器，支持分配和回收内存。因为它需要模拟内存系统，所以要考虑如何自己实现回收内存、处理内部外部碎块，而不能调用系统中释放内存和收缩堆的函数。

隐式空闲列表

- 思路
  - 用（头，内容，脚）来表示一个块，头和脚上记录块的大小和是否被占用，如下：
  - 

  - 一系列连续的块构成内存列表。在init阶段，申请一个序言块（不需要内容）和结尾块（不需要脚），这样就方便处理边界情况了，如下：
  - 

  - 每次malloc时，遍历所有的块，寻找有没有大小足够且空闲的块，如果没有的话就在堆里申请一块新内存。将找到的块的地址记为bp。
  - 如果bp的大小比size大很多，就将bp切分为两个块，一个大小为size，剩下的则变成新的空闲块。
  - 如何解决相邻的空闲块：每当产生新的空块，就尝试向前后延伸合并，这样可以保证任意时刻的空闲块都是极大的。（见coalesce函数）
  - 注意，因为每个块需要一个头和一个脚，所以实际申请的内存大小至少是`size+2*WSIZE`。此外还要注意对齐问题。
- 需要用到的宏

```
//简化指针操作
#define GET(p) (*(size_t *)(p))
#define PUT(p,val) ((*(size_t *)(p)) = (val))

//块的大小&是否被占用
#define GET_SIZE(p) (GET(p) & ~0x7)
#define GET_ALLOC(p) (GET(p) & 0x1)

//用于访问块的头和脚
#define HDRP(bp) ((char *)(bp)-WSIZE)
#define FTRP(bp) ((char *)(bp)+GET_SIZE(HDRP(bp))-DSIZE)

//上，下一个紧邻的块
#define PREV_BLKP(bp) ((char *)(bp) - GET_SIZE( ((char *)(bp)-DSIZE) ))
#define NEXT_BLKP(bp) ((char *)(bp) + GET_SIZE( ((char *)(bp)-WSIZE) ))
```

- init

```
heap_listp=mem_sbrk(4*WSIZE);
if(heap_listp==(void*)-1) return -1;
PUT(heap_listp,0);
PUT(heap_listp+1*WSIZE,PACK(DSIZE,1));
PUT(heap_listp+2*WSIZE,PACK(DSIZE,1));
PUT(heap_listp+3*WSIZE,PACK(0,1));
```

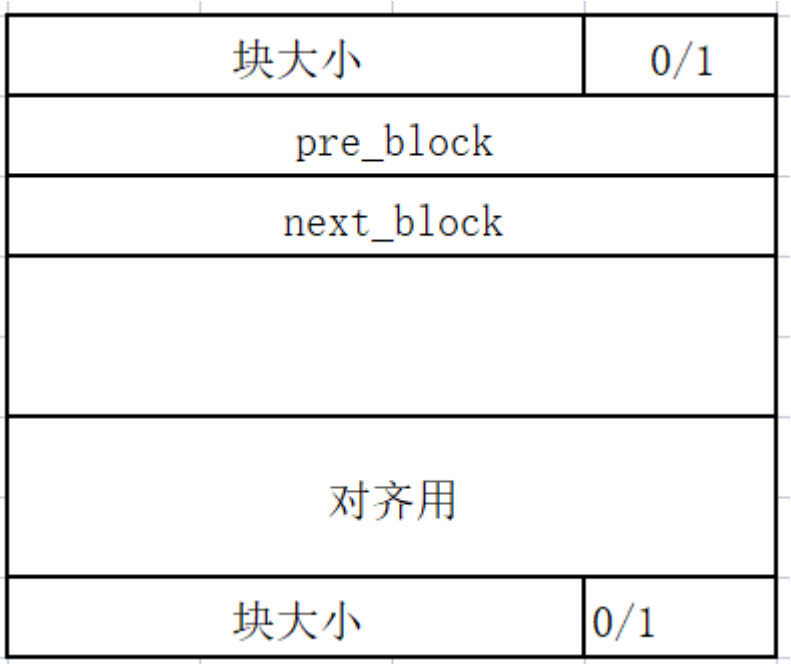
- malloc

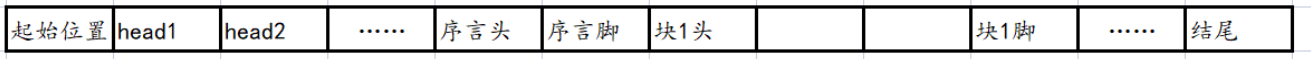
```
size+=DSIZE;
asize=DSIZE*( (size+DSIZE-1)/(DSIZE) );
//对齐完毕
bp=find_fit(asize);
if(bp!=NULL)
{
    bp=place(bp, asize,1);
    return bp;
}
//找不到，扩大堆
extendsize=MAX(asize,CHUNKSIZE);
if((bp=extend_heap(extendsize/WSIZE)) == NULL) return NULL;

bp=place(bp, asize,1);
```

- realloc
  - 一个简单的想法是无论如何都申请一块大小为新size的内存，然后memcpy以前的数据。但这种方法没有充分利用已有的空间，所以内存利用率和吞吐率都非常低。在优化部分将会仔细叙述优化后的做法。
- 此时可以拿到45的分数，主要问题是吞吐率太低以及realloc的内存利用率太低。

分离的显式空闲列表

- 思路:
  - 如果一个块已经被占用，那么不管他是大是小，都不可能再被分配给新需求。因此，在malloc寻找恰当的块时，可以只遍历当前空闲的块。
  - 考虑使用空块的空闲内存来实现一个空闲列表。因为空块也要有头和脚且需要对齐，所以内部至少有两个WSIZE大小的空间，正好可以记录两个指针，分别指向前驱和后继空闲块的位置。

    - 

    - 假设现在需要一个大小为10000的块，那么大小小于10000的块根本没必要考虑，但上述做法却会将这些块都遍历一遍，是非常浪费时间的。我们可以以块大小为关键字，将块分为几个类别，建立多个链表以提高查询的速度。在init时多申请常数个空间，用于储存每个链表的头指针。
    - 可以以二进制进行分组，如`(16,32],(32,64],(64,128]`，每次直接定位到合适的大小区间去寻找空闲块（注意，这个区间未必含有符合要求的块，此时可以进入更大的区间查找）
    - 

- 新加入的宏：
    - ```
      #define GET_FIRST(size) ((size_t *)(long)(GET(heap_listp+WSIZE*size)))
      //上、下一个空闲的块
      #define PRE_BLOCK(bp) ((size_t *)(long)(GET(bp)))
      #define NEX_BLOCK(bp) ((size_t *)(long)(GET((size_t *)(bp)+1)))
      ```
- insert(生成了一个新的空闲块)
    - ```
      void insert(void* bp)
      {
          size_t size=GET_SIZE(HDRP(bp));
          int num=fit_size(size);
          if(GET_FIRST(num)==NULL)
          {
              PUT(heap_listp+num*WSIZE,(size_t)bp);
              PUT(bp,0);
              PUT((size_t *)bp+1,0);
          }
          else
          {
              PUT((size_t *)bp+1,(size_t)GET_FIRST(num));
              PUT(GET_FIRST(num),(size_t)bp);
              PUT(bp,0);
              PUT(heap_listp+num*WSIZE,(size_t)bp);
          }
      }
      ```
- delete（删除了一个空闲块）

```
void delete(void* bp)
{
    size_t size=GET_SIZE(HDRP(bp));
    int num=fit_size(size);
    if(PRE_BLOCK(bp)==NULL)
        PUT(heap_listp+num*WSIZE,(size_t)NEX_BLOCK(bp));
    if(PRE_BLOCK(bp)!=NULL)
        PUT(PRE_BLOCK(bp)+1,(size_t)NEX_BLOCK(bp));
    if(NEX_BLOCK(bp)!=NULL)
        PUT(NEX_BLOCK(bp),(size_t)PRE_BLOCK(bp));
}
```

- coalesce(合并空闲块，但要注意修改空闲列表，不能出现重复块)

```
size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
size_t size=GET_SIZE(HDRP(bp));

if(prev_alloc && next_alloc)
{
    insert(bp);
    return bp;
}
else if(prev_alloc && !next_alloc)
{
    delete(NEXT_BLKP(bp));
    size+=GET_SIZE(HDRP(NEXT_BLKP(bp)));
    PUT(HDRP(bp),PACK(size,0));
    PUT(FTRP(bp),PACK(size,0));
}
else if(!prev_alloc && next_alloc)
{
    delete(PREV_BLKP(bp));
    size+=GET_SIZE(HDRP(PREV_BLKP(bp)));
    PUT(FTRP(bp),PACK(size,0));
    PUT(HDRP(PREV_BLKP(bp)),PACK(size,0));
    bp=PREV_BLKP(bp);
}
else
{
    delete(NEXT_BLKP(bp));
    delete(PREV_BLKP(bp));
    size+=GET_SIZE(HDRP(PREV_BLKP(bp)))+GET_SIZE(FTRP(NEXT_BLKP(bp)));
    PUT(HDRP(PREV_BLKP(bp)),PACK(size,0));
    PUT(FTRP(NEXT_BLKP(bp)),PACK(size,0));
    bp=PREV_BLKP(bp);
}
insert(bp);
return bp;
```

- 此时可以得到47分，看起来优化不大，但主要是因为realloc太慢了使吞吐率非常低。如果单独测试前九个点，已经可以取得不错的性能。

优化

- 减少零碎的空闲块
  - 思路：很多数据中出现了大块小块交替出现的情况。这时会产生一个问题，即某个小块一直没有free，两边的大空闲块free后无法合并，再申请时发现两个块分别的大小都不够，只得申请新空间，造成大量浪费。

- 例：`000000000000x000000000x00000000`
- 简单来说，就是小块虽然占据的空间很少，却可能把大块空间卡住。因此可以想到一种优化的思路，就是将小块尽量放在一起，大块尽量放在一起，以免互相干扰。
- 在place时（在bp指向的空闲块中切分出size大小），可以找到一个flag值，比flag小的就尽量往左放，比flag大的就尽量往右放。此处我选择flag=96，取得了不错的结果，使两个binary测试点的空间利用大幅提高。
- ```
  if(size<=96)
  {
      PUT(HDRP(bp),PACK(size,1));
      PUT(FTRP(bp),PACK(size,1));
      void* old_bp=bp;
      bp=NEXT_BLKP(bp);
      PUT(HDRP(bp),PACK(block_size-size,0));
      PUT(FTRP(bp),PACK(block_size-size,0));
      insert(bp);
      return old_bp;
  }
  else
  {
      PUT(HDRP(bp),PACK(block_size-size,0));
      PUT(FTRP(bp),PACK(block_size-size,0));
      void* old_bp=bp;
      bp=NEXT_BLKP(bp);
      PUT(HDRP(bp),PACK(size,1));
      PUT(FTRP(bp),PACK(size,1));
      insert(old_bp);
  }
  ```

- realloc的优化
  - 原来的realloc过于粗糙，无论如何都申请新内存复制数据，显然有很大优化空间，按原size和新size的大小关系分情况讨论：
    - 如果原来的size比新的还大，那么只要把原块切割一下就可以了，不用申请新内存也不用复制，同时提高两项指标；
    - 如果原来的size比较小，但原块后有足够大的空闲块，那么只要将这两个块合并即可，不用再专门复制了。
    - 另一种情况是原块的前面有空间，此时可以先free原块，和前块合并后作为新块，此时也可以省下一些空间。
    - 需要注意的是，free原块可能导致原块的开头两个数据被篡改（这两个位置free后变成记录指针的了），所以需要在free前额外记录。此外，place新块可能导致新块的头和脚覆盖掉一些数据，解决方法比较简单，就是先复制数据再place新块，需要对malloc函数做一点小改动。
  - 数据和现实中都可能遇到这样一种情况：某个程序不断动态申请内存，而大部分程序基本只有申请和释放需求。对于这种情况，可以有一种特殊的优化方法：
    - 在init阶段申请一小块内存，专门用于处理小数据的频繁修改需求。（类似place时的分类讨论）
    - ```
      int mm_init(void)
      {
          ……
          if(extend_heap(10)==(void*)-1) return -1;
          mm_malloc(1);
          //防止这一块空间被合并掉
          return 0;
      }
      ```
    - 频繁realloc的块有权力多保留一些空间，每次extend_heap后的空间不再进行切割分配，而是全部合并进这个块，以减少频繁切割块的时间损耗。因为每次extend其实只有2K左右，所以并不会造成很大的

浪费。此外，这还可以防止其它块频繁插入到这个块后面，导致下次realloc时进行大规模的数据移动。

```
void *mm_realloc(void *ptr, size_t size)
{
  size_t old_size=GET_SIZE(HDRP(ptr))-DSIZE;
  if(ptr == NULL) return mm_malloc(size);
  if(size==0)
  {
      mm_free(ptr);
      return NULL;
  }
  if(old_size>=size) return ptr;
  size_t temp_prev=GET(ptr);
  size_t temp_succ=GET((size_t*)ptr+1);
  mm_free(ptr);
  void* new_ptr=re_malloc(size);
  if(new_ptr==NULL) return NULL;
  if(new_ptr!=ptr)
      memcpy((size_t*)new_ptr+2,(size_t*)ptr+2,old_size-DSIZE);
  size=DSIZE*( (size+DSIZE+DSIZE-1)/(DSIZE) );
  new_ptr=place(new_ptr,size,0);
  PUT(new_ptr,temp_prev);
  PUT((size_t *)new_ptr+1,temp_succ);
  return new_ptr;
}
```

最终成果

- 97分

```
u2021013420@hp:~/malloclab-handout$ ./mdriver -v
Team Name:2021013420
Member 1 :慎梓戎:shenzr21@mails.tsinghua.edu.cn
Using default tracefiles in ./traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace            name     valid  util     ops      secs   Kops
 1     amptjp-bal.rep       yes   97%    5694  0.000406  14014
 2       cccp-bal.rep       yes   97%    5848  0.000421  13881
 3    cp-decl-bal.rep       yes   97%    6648  0.000520  12775
 4       expr-bal.rep       yes   98%    5380  0.000410  13119
 5 coalescing-bal.rep       yes   96%   14400  0.000588  24473
 6     random-bal.rep       yes   93%    4800  0.000505   9505
 7    random2-bal.rep       yes   90%    4800  0.000501   9573
 8     binary-bal.rep       yes   91%   12000  0.000592  20277
 9    binary2-bal.rep       yes   81%   24000  0.001605  14952
10    realloc-bal.rep       yes   99%   14401  0.000307  46863
11   realloc2-bal.rep       yes   96%   14401  0.000253  56831
Total                            94%  112372  0.006110  18390

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

心得

- 困难
  - csapp课本里的宏是基于32位机器来写的，我一开始写的时候只考虑了对齐要改成16位，没有改指针的类型 `unsigned int *` 。在加入显式空闲列表的时候，我遇到了一些很奇怪的内存问题，比如一个块的前驱突然变成了自己（是因为保存指针的部分被错误覆盖了）。后来才意识到是因为64位机器下指针是8位的，不能

用`unsigned int`存下。全部改成`size_t *`后解决了问题（另一个好处是可以通过修改size_t来方便地适配两种机器）

- 收获：
  - 对程序的内存管理问题更了解了。
  - 对64位机器和32位机器的差异有了更直观的认识，学习了如何使用宏来简化指针操作。
- 交流：在完成malloclab的过程中，和朱悦宁同学、黄嘉盛同学交流讨论了realloc优化的一些可能方案。