Malloclab

李梓童 2017202121

1. 所完成的部分

个人完成本次实验。

2. 遇到的问题及解决方案

- **宏定义里没有加括号出错。**
  详述：用宏定义链表操作时，如#define READ_LIST_HEAD(x) ((mem_heap_lo())+(x)*DSIZE)，一开始没有在 x 两边加上括号，导致程序运行出错。出错原因是宏只是简单的替换，如 #define double(x) 2*x，如果在代码中调用 double(x+1) 得到的结果会变成 2x+1 而不是 2x+2。由于宏不会进行拼写和类型检查，所以出错的话比较难调试。
  解决方案：在宏定义的每个变量两边都加上括号。
- **进行链表操作时，对空链表的情况考虑不周。**
  详述：一开始写链表操作时，只考虑了开头插入、中间插入、结尾插入的情况，认为空链表时可以直接在开头插入。而实际上空链表插入和开头插入是有区别的，开头插入要重设 PRED_PTR 和 SUCC_PTR，而空链表插入则需把两者都设为 NULL。
  解决方案：在 add_node 里另开一项（cursor1==cursor2==NULL），处理空链表。
- **指针出错。**
  详述：实验中用到了大量指针，而指针使用也是实验中很容易出 bug 的一个地方。例如链表操作中使用 void *来存储有关地址，试图对改地址解引用时会有毛病。又如在给指针赋值时会傻傻分不清楚，究竟是把指针放到了堆里还是把指针指向的值放到了堆里。
- **Coalesce 函数。**
  详述：coalesce 函数在课本中已给出示例，要考虑四种情况。采用分离链表后，需要把四种情况改成链表操作的版本。其实四种情况的原理都是一样的：如果前后有 free 块，就先从链表里把这个 free 块删掉，把空闲块的 size 增加，然后把修改 size 后的 free 块放到链表里。
  编写代码的时候，容易忽略对返回 ptr 的更新。一开始全都忘了更新，没有把返回的 ptr 指向新的空闲块头。另外，需要考虑不同情况对 ptr 的更新是不一样，前面块 allocated 的话是不用更新的。
- **Realloc 函数。**
  详述：realloc 函数需要考虑的细节比较多。
  首先申请的 size 太小（<=DSIZE），要将 size 调整为最小块的大小，这一步一开始忽略了，上来就 ALIGN（size+DSIZE）。这一步虽然对结果没什么影响，没有也能过，但经别人提醒觉得还是写一下比较好。

如果申请的 size 小于已有的大小，则返回原指针。

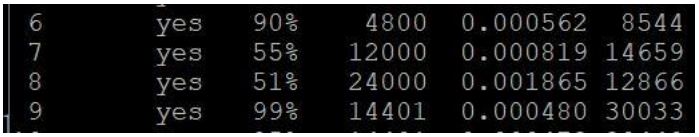
第二步是检查附近有没有连续的可以用的 free 块。如果刚好有的话，不仅要在链表里删除用掉的 free 块，还要设置新块的头尾。一开始忘了设置新块的头尾，导致后面出现了 bug，这个也不太好发现。

考虑了上面两方面，最后才是使用 malloc 和 memcpy 来解决。

Realloc 的实现其实不困难，难的是面面俱到，把各种可能的情况考虑清楚，必要的话可以在纸上列出来讨论。

- **策略完善**。

  详述：初步完成显式空闲链表和分离适配的时候提交测试，发现 trace7 和 trace8 的性能不好。



6	yes	90%	4800	0.000562	8544
7	yes	55%	12000	0.000819	14659
8	yes	51%	24000	0.001865	12866
9	yes	99%	14401	0.000480	30033

  查看了相关的输入文件后发现，这两个文件的输入是 malloc 一小块、malloc 一大块、malloc 一小块、malloc 一大块……这样如果 free 的时候先把大块给 free 了，再来个更大块的时候就会产生较大的外部碎片（因为大块中间都被小块分隔了，小块都还是 allocated）。

  解决方法：打算把放置策略调整为“当块大小超过一定值的时候，从另一边开始堆垒”。这样的话，大块聚集在一边，小块聚集在一边，可以解决上面提到的问题。根据两个测试文件中最大块和最小块的尺寸，这个界限值应该在 64-112 之间，测试之后选了 110（实际上，80、90、100 都没什么区别，对性能影响不大）。修改之后提交，看到 util 高了，但是吞吐量降下来了，最终得分其实没什么变化。

- **一些代码层面上的优化**。

  详述：把原本的静态链表操作函数 read_list_head 和 set_list_head 转化成宏，希望提高运行效率，但是实际上没有什么影响。

  把“/=2”变成了位运算“>>=1”，实际上并没有什么影响。

- **关于为什么不能使用全局变量的问题**。

  不同的进程使用的虚存不一样，不是说每次都能读到全局变量。全局变量应该放到所申请的堆里，不然不同的应用程序不能保证。在本次实验中，用指向堆底部的指针来定位，即把所需要的链表头头放到堆开始的地方，这样进行读取和修改。

3. 实现的内存分配管理算法

显式空闲链表和分离适配。

4. 反思总结

- 实验中，首先阅读了课本给出的简单适配器代码（隐式空闲链表+边界标记合并策略），参考 http://csapp.cs.cmu.edu/3e/ics3/code/vm/malloc/mm.c

```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.008421   676
 1       yes   99%    5848  0.007765   753
 2       yes   99%    6648  0.013175   505
 3       yes  100%    5380  0.009628   559
 4       yes   66%   14400  0.000112128571
 5       yes   91%    4800  0.007192   667
 6       yes   92%    4800  0.006950   691
 7       yes   55%   12000  0.097243   123
 8       yes   51%   24000  0.328500    73
 9       yes   27%   14401  0.073197   197
10       yes   34%   14401  0.002669  5395
11       yes   66%      12  0.000000 60000
12       yes   89%      12  0.000000 40000
Total          75%  112396  0.554852   203

Perf index = 45 (util) + 1 (thru) = 45/100
```

课本代码出来的结果是 45/100，从而帮助我们确定自己的策略：显式空闲链表+分离适配。阅读优秀的代码后，我的大概印象是：好的代码就像自然语言一样，读起来很通顺地就表达出程序员想要做的事。后来编写自己的分配器时也在努力朝这个方向靠近，希望能以函数健全为前提在表意层面上编写代码。

由于已经有了示例代码，这次实验并不是从零开始，因此也降低了难度。

- 很多时候程序不能正常运行，原因是程序员自己没有把问题想清楚，是原理上的错误。在这次实验中，需要我们自己确定策略。程序跑出来经常会挂，总结挂掉的原因，什么指针转化这样的 bug 还是次要，大头是“这里忘了添加链表节点、那里忘了删除节点”这类原因，是由于编写代码的人没有把解决问题的思路一步一步理清楚、把问题的各个方面考虑清楚引起的错误。
- 之前使用 malloc 的时候，不会去想它背后有什么内涵。通过本次实验，我对内存分配有了更进一步的洞观。下面是自己对一些概念的通俗解释，作为本次实验基础知识的巩固：
  - 首次适配：从头开始查找，放得下就 ok。
    下一次适配：从上次查找的地方开始找，放得下就 ok。
    最佳适配：找最小的放得下的。
  - 隐式空闲链表：空闲块通过头部的大小字段隐含地连接，头部隐含空闲块信息。
    显式空闲链表：全是空闲块。
  - 分离空闲链表 -> 简单分离存储：每个大小类的空闲链表包含大小相同的空闲链表，不分割，不合并。
    分离空闲链表 -> 分离适配：先找范围（非固定值），再找块。
    分离空闲链表 -> 伙伴系统：遇到一个已分配的伙伴则停止。