MallocLab实验报告

姓名	学号
卢虹宇	2023202269

实验结果

94/100

```
Team Name:ICSGG
Member 1 :Lu Hongyu:2976218320@qq.com
Using default tracefiles in ../traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   98%    5694  0.000153 37338
 1       yes   97%    5848  0.000165 35421
 2       yes   99%    6648  0.000179 37244
 3       yes   99%    5380  0.000142 37887
 4       yes   66%   14400  0.000255 56382
 5       yes   94%    4800  0.000226 21267
 6       yes   91%    4800  0.000230 20888
 7       yes   95%   12000  0.000213 56417
 8       yes   88%   24000  0.000402 59642
 9       yes  100%   14401  0.000192 74888
10       yes   64%   14401  0.000171 84020
Total          90%  112372  0.002328 48274

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

实验功能分析

本实验实现的动态内存分配器采用了 **分离空闲链表 (Segregated Free Lists)** 的策略来管理内存。这种方法通过将不同大小的空闲块组织在不同的链表中，以期提高查找和分配效率。

1. 核心数据结构

- **分离空闲链表 (Segregated Free Lists):**

    - 使用一个数组 `seg_list_head` (大小为 `LISTLIMIT` = 16) 作为分离空闲链表的头部指针数组。每个元素指向一个特定大小范围的空闲块链表的头节点。
    - 空闲块内部使用显式链表结构，通过块内存储的前驱 (PRED) 和后继 (SUCC) 指针形成双向链表。
    - 需要注意的是头部指针数组被存储在堆中，通过一个全局指针`seg_list_head`来访问，所以不存在定义符合全局的复合数据结构。
- **数据块：**
    - **分配块**：包含一个头部 (Header) 和一个脚部 (Footer)，用于存储块的大小和分配状态 (通过最低位标识，1为已分配，0为空闲)。
    - **空闲块**：除了头部和尾部之外还包含一个前驱指针`PRED`和一个后驱指针`SUCC`指向链表中前后的空闲块。
    - 所有块都按8字节对齐。
    - 定义了最小已分配块大小 (MIN_ALLOCATED_BLOCK_SIZE = 16字节) 和最小空闲块大小 (MIN_SPARE_BLOCK_SIZE = 24字节，需要容纳头部、脚部以及前驱和后继指针)。
- **堆的边界管理:**
    - 使用序言块 (Prologue Block) 和结尾块 (Epilogue Block) 来简化堆的边界条件处理，避免在合并或查找时出现特殊情况。

2. 采用的优化和技术（实现细节在下面一小节）

- **分离空闲链表:** 核心策略，通过按大小分类空闲块，减少了搜索特定大小空闲块的时间。
- **LIFO 插入策略:** 将新释放或合并产生的空闲块插入到对应链表的头部，操作简单高效。
- `place` **中的启发式分割:** `SPLIT_BACK` 宏使`place`函数在特定情况下将分配块插入到空闲块的尾部（正常情况下是头部）。
- `mm_realloc` **的优化:** 尝试通过与物理相邻的空闲块合并来原地扩展（原地向前向后合并空闲块），提高了效率。

3. 操作流程

- **初始化 (`mm_init`):**
    1. 为分离空闲链表的头指针数组 `seg_list_head` 分配空间并将其所有指针初始化为 `NULL`。
    2. 在堆的起始位置设置对齐填充、序言块（包含头部和脚部，标记为已分配）和结尾块（仅头部，标记为已分配）。
    3. 调用 `extend_heap` 函数扩展初始的堆空间 (`CHUNKSIZE`)，并将返回的空闲块加入到相应的分离空闲链表中。
- **分配内存 (`mm_malloc`):**
    1. 对请求的大小 `size` 进行调整，加上头部和脚部开销，并进行8字节对齐，得到实际需要分配的大小 `asize`。确保 `asize` 不小于 `MIN_ALLOCATED_BLOCK_SIZE`。
    2. 使用 `get_list_index(asize)` 确定从哪个分离空闲链表开始查找。
    3. **查找策略:** 从该索引对应的链表开始，向后（即向更大size的链表）遍历分离空闲链表。在每个被检查的链表中，采用 **首次适应 (First-Fit)** 策略，即遍历链表中的空闲块，找到第一个大小不小于 `asize` 的空闲块。
    4. **放置 (`place`):**
        - 如果找到合适的空闲块，调用 `place(bp, asize)` 函数。

        - `place` 函数首先将选中的空闲块 `bp` 从其所在的空闲链表中移除 (`remove_from_list`)。
        - **分割策略:** 判断分配 `asize` 后，剩余部分 `csize - asize` 是否足够大 (不小于 `MIN_SPARE_BLOCK_SIZE`) 以形成一个新的空闲块。
            - 如果可以分割，代码采用了一种启发式的分割策略 (由 `SPLIT_BACK` 宏定义)：若请求分配的 `asize` 较小 (小于96字节) 或者分割后剩余的下一块较小 (小于48字节)，则优先从原空闲块的 **前部** 分割出 `asize` 进行分配，剩余部分形成新的空闲块。否则，从原空闲块的 **后部** 分割出 `asize` 进行分配，原空闲块的前部调整大小后作为新的空闲块。无论是哪种分割，新产生的空闲块都会通过 `add_to_list` 添加回相应的空闲链表。
            - 如果不能分割，则整个空闲块 `bp` 都被分配。
        - 更新分配块的头部和脚部，标记为已分配。
    5. **扩展堆:** 如果遍历完所有相关链表后仍未找到合适的空闲块，则调用 `extend_heap` 扩展堆空间（扩展大小为 `MAX(asize, CHUNKSIZE)`），然后从新扩展的空闲块中调用 `place` 进行分配。
- **释放内存 (`mm_free`):**
    1. 检查指针的有效性。
    2. 获取要释放块的大小，并更新其头部和脚部，将其标记为未分配状态 (0)。
    3. 调用 `coalesce(ptr)` 函数尝试与前后物理相邻的空闲块合并。
- **合并 (`coalesce`):**
    1. 检查目标块 `bp` 前一个物理块和后一个物理块的分配状态。
    2. **Case 1 (前后都空闲):** 将前、中、后三个块合并。从各自的空闲链表中移除前块和后块，更新合并后大块（起始于前块地址）的大小，然后将此大块加入空闲链表。
    3. **Case 2 (仅前块空闲):** 合并前块和当前块。从空闲链表中移除前块，更新合并后块（起始于前块地址）的大小，加入空闲链表。
    4. **Case 3 (仅后块空闲):** 合并当前块和后块。从空闲链表中移除后块，更新合并后块（起始于当前块地址）的大小，加入空闲链表。
    5. **Case 4 (前后都已分配):** 无需合并，直接将当前块 `bp` 加入空闲链表。
    6. 空闲块通过 `add_to_list` 函数以 **LIFO (Last-In, First-Out)** 顺序添加到相应分离链表的头部。
- **重新分配内存 (`mm_realloc`):**
    1. 处理边界情况：若 `ptr` 为 `NULL`，则等效于 `mm_malloc(size)`；若 `size` 为 0，则等效于 `mm_free(ptr)`。
    2. 计算新的对齐后大小 `new_asize`。
    3. **缩小或不变:** 如果 `new_asize` 不大于原块大小 `old_asize`，直接返回原指针 `ptr` (代码未实现原地缩小并释放多余部分的功能，仅返回原指针)。
    4. **扩大:** 如果 `new_asize` 大于 `old_asize`：
        - **优化1 (向后合并):** 检查紧邻的下一个物理块是否为空闲块，并且其大小与当前块合并后是否足够容纳 `new_asize`。如果是，则从空闲链表中移除该后续空闲块，合并两块。更新当前块的头部和脚部为新总大小并标记为已分配。如果合并后的总大小 `combined_size` 比 `new_asize` 大且剩余部分满足 `MIN_SPARE_BLOCK_SIZE`，则将多余部分分割出来，标记为空闲，并调用 `coalesce` 将其加入空闲链表。

- **优化2 (向前合并):** 检查紧邻的上一个物理块 (且不是序言块) 是否为空闲块，并且其大小与当前块合并后是否足够容纳 `new_asize`。如果是，则从空闲链表中移除该前序空闲块。将原块的数据通过 `memcpy` 拷贝到前序空闲块的起始位置。然后更新前序块的头部和脚部为新总大小并标记为已分配。类似地，如果有多余空间，也进行分割和 `coalesce`。
- **最后手段:** 如果以上两种优化均不适用，则调用 `mm_malloc(new_asize)` 分配一块全新的内存。将原内存块中的数据 (取 `old_payload_size` 和 `new_requested_size` 中的较小者) `memcpy` 到新分配的内存块中。最后，调用 `mm_free(ptr)` 释放原内存块。

实验感想

这个实验主要优化的点在于空间利用率，吞吐量很容易就顶满了。第一遍实现的是一个naive的分离空闲链表，甚至分还不如给的示例代码高，主要调优的点在于realloc的策略和对空闲块进行分割的策略，realloc实现原地合并空闲块之后分数来到了90，正常手段再也调不动了，于是乎开始面向trace编程，发现效果最差的trace是在反复对一个块进行realloc，于是乎加了一个宏调整了一下place函数放置分配块的位置，让某些分配块被free掉之后可以与空闲块连起来，空间利用效率得到提升（这个思路受到王安栋同学启发，最初的思路是realloc的时候直接一步到位分配最后所需的大小，但这样有点太刻意了🤡）。