哈爾濱工業大學

实验报告

实 验（八）

题　　目 Dynamic Storage Allocator

动态内存分配器

专　　业 计算机

学　　号 1170300825

班　　级 1730008

学 生 姓 名 李大鑫

指 导 教 师 郑贵滨

实 验 地 点

实 验 日 期

计算机科学与技术学院

目 录

第 1 章 实验基本信息

1.1 实验目的

- 理解现代计算机系统虚拟存储的基本知识
- 掌握 C 语言指针相关的基本操作
- 深入理解动态存储申请、释放的基本原理和相关系统函数
- 用 C 语言实现动态存储分配器，并进行测试分析
- 培养 Linux 下的软件系统开发与测试能力

1.2 实验环境与工具

1.2.1 硬件环境

X64 CPU；2GHz；2G RAM；256GHD Disk 以上

1.2.2 软件环境

Windows7 64 位以上；VirtualBox/Vmware 11 以上；Ubuntu 16.04 LTS 64 位/优麒麟 64 位

1.2.3 开发工具

Windows7 64 位以上；VirtualBox/Vmware 11 以上；Ubuntu 16.04 LTS 64 位/优麒麟 64 位

1.3 实验预习

■ 上实验课前，必须认真预习实验指导书（**PPT 或 PDF**）

■ 了解实验的目的、实验环境与软硬件工具、实验操作步骤，复习与实验有关的理论知识。

■ 熟知 C 语言指针的概念、原理和使用方法

■ 了解虚拟存储的基本原理

■ 熟知动态内存申请、释放的方法和相关函数

■ 熟知动态内存申请的内部实现机制：分配算法、释放合并算法等

第 2 章 实验预习

总分 20 分

2.1 动态内存分配器的基本原理（5 分）

动态内存分配器维护着一个进程的虚拟内存区域，称为堆。分配器将堆视为一组不同大小的块的集合来维护。每个块就是一个连续的虚拟内存片，要么是已分配的，要么是空闲的。已分配的块显式地保留为供应用程序使用。空闲块可用来分配。空闲块保持空闲，直到它显式地被应用所分配。一个已分配的块保持已分配状态，直到它被释放，这种释放要么是应用程序显式执行的，要么是内存分配器自身隐式执行的。

分配器分为两种基本风格：显式分配器、隐式分配器。

显式分配器：要求应用显式地释放任何已分配的块。

隐式分配器：要求分配器检测一个已分配块何时不再使用，那么就释放这个块，自动释放未使用的已经分配的块的过程叫做垃圾收集。

2.2 带边界标签的隐式空闲链表分配器原理（5 分）

一）堆及堆中内存块的组织结构：

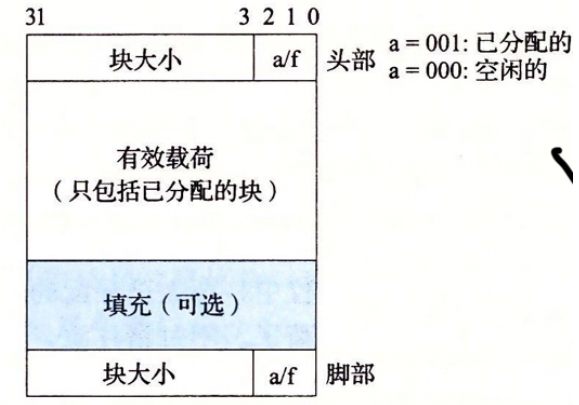


图 9-39 使用边界标记的堆块的格式

在内存块中增加 4B 的 Header 和 4B 的 Footer，其中 Header 用于寻找下一

个 blcok，Footer 用于寻找上一个 block。Footer 的设计是专门为了合并空闲块方便的。因为 Header 和 Footer 大小已知，所以我们利用 Header 和 Footer 中存放的块大小就可以寻找上下 block。

二）隐式链表

所谓隐式空闲链表，对比于显式空闲链表，代表并不直接对空闲块进行链接，而是将对内存空间中的所有块组织成一个大链表，其中 Header 和 Footer 中的 block 大小间接起到了前驱、后继指针的作用。

三）空闲块合并

因为有了 Footer，所以我们可以方便的对前面的空闲块进行合并。合并的情况一共分为四种：前空后不空，前不空后空，前后都空，前后都不空。对于四种情况分别进行空闲块合并，我们只需要通过改变 Header 和 Footer 中的值就可以完成这一操作。

2.3 显示空间链表的基本原理（5 分）

将空闲块组织成链表形式的数据结构。堆可以组织成一个双向空闲链表，在每个空闲块中，都包含一个 pred（前驱）和 succ（后继）指针，如下图：

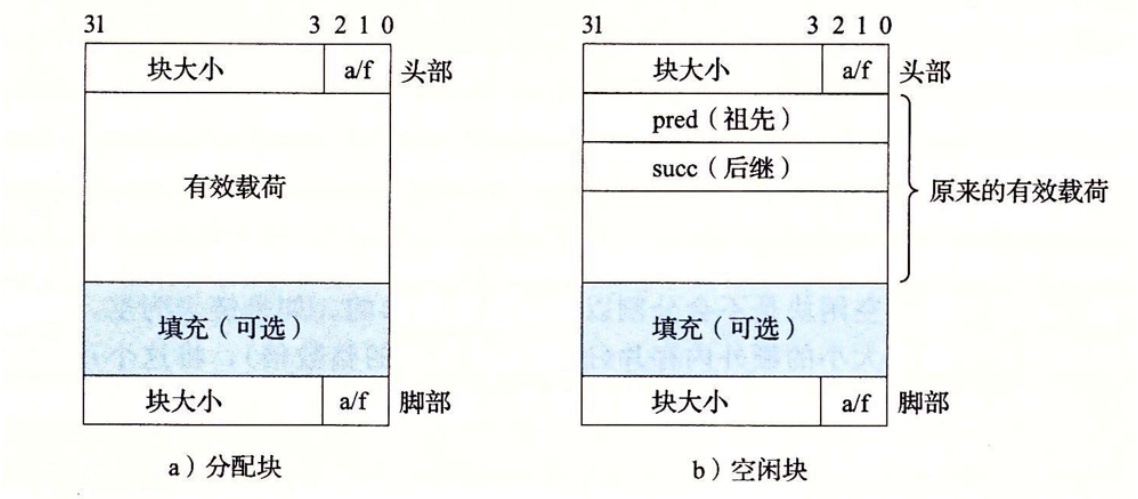


图 9-48　使用双向空闲链表的堆块的格式

使用双向链表而不是隐式空闲链表，使首次适配的分配时间从块总数的线性时间减少到了空闲块数量的线性时间。

维护链表的顺序有：后进先出（LIFO），将新释放的块放置在链表的开始处，使用 LIFO 的顺序和首次适配的放置策略，分配器会最先检查最近使用过的块，在这种情况下，释放一个块可以在线性的时间内完成，如果使用了边界标记，那么

合并也可以在常数时间内完成。按照地址顺序来维护链表，其中链表中的每个块的地址都小于它的后继的地址，在这种情况下，释放一个块需要线性时间的搜索来定位合适的前驱。平衡点在于，按照地址排序首次适配比 LIFO 排序的首次适配有着更高的内存利用率，接近最佳适配的利用率。

2.4 红黑树的结构、查找、更新算法（5 分）

红黑树，本质上来说就是一棵二叉查找树，但它在二叉查找树的基础上增加了着色和相关的性质使得红黑树相对平衡，从而保证了红黑树的查找、插入、删除的时间复杂度最坏为 O(log n)。

但它是如何保证一棵 n 个结点的红黑树的高度始终保持在 h = logn 的呢？这就引出了红黑树的 5 条性质：

1）每个结点要么是红的，要么是黑的。
2）根结点是黑的。
3）每个叶结点（叶结点即指树尾端 NIL 指针或 NULL 结点）是黑的。
4）如果一个结点是红的，那么它的俩个儿子都是黑的。
5）对于任一结点而言，其到叶结点树尾端 NIL 指针的每一条路径都包含相同数目的黑结点。

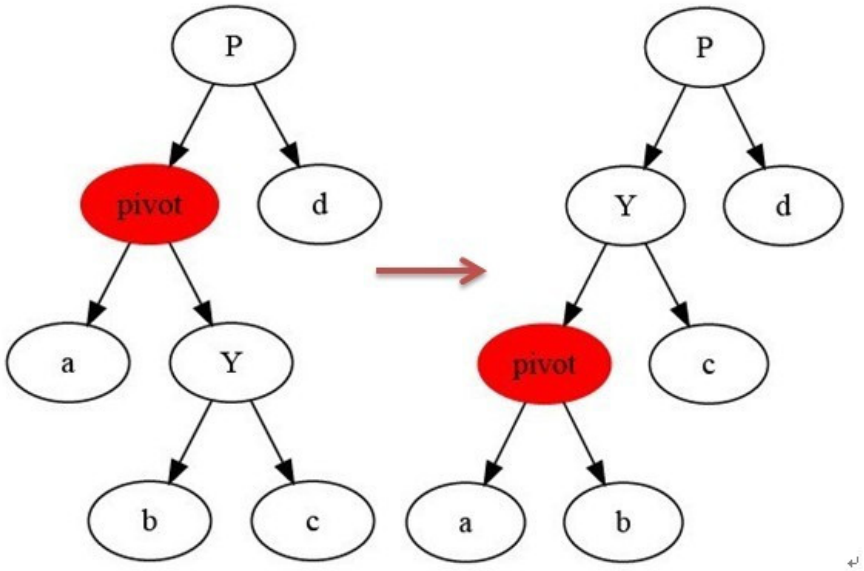
树的旋转知识

当我们在对红黑树进行插入和删除等操作时，对树做了修改，那么可能会违背红黑树的性质。

为了继续保持红黑树的性质，我们可以通过对结点进行重新着色，以及对树进行相关的旋转操作，即修改树中某些结点的颜色及指针结构，来达到对红黑树进行插入或删除结点等操作后，继续保持它的性质或平衡。

树的旋转，分为左旋和右旋。

1.左旋



如上图所示：

当在某个结点 pivot 上，做左旋操作时，我们假设它的右孩子 y 不是 NIL[T]，pivot 可以为任何不是 NIL[T]的左孩子结点。

左旋以 pivot 到 y 之间的链为“支轴”进行，它使 y 成为该孩子树新的根，而 y 的左孩子 b 则成为 pivot 的右孩子。

左旋操作的参考代码如下所示（以 x 代替上述的 pivot）：

```
 LEFT-ROTATE(T, x)
1  y ← right[x] ▷ Set y.
2  right[x] ← left[y]      ▷ Turn y's left subtree into x's right subtree.
3  p[left[y]] ← x
4  p[y] ← p[x]             ▷ Link x's parent to y.
5  if p[x] = nil[T]
6     then root[T] ← y
7     else if x = left[p[x]]
8             then left[p[x]] ← y
9             else right[p[x]] ← y
10  left[y] ← x             ▷ Put x on y's left.
11  p[x] ← y
```

2.右旋

原理类似左旋，此处省略。

红黑树的插入（查找、更新）

红黑树的插入相当于在二叉查找树插入的基础上，为了重新恢复平衡，继续做了插入修复操作。**红黑树的插入操作中，我们首先进行了查找，然后进行了更**

新（旋转）操作。

假设插入的结点为 z，红黑树的插入伪代码具体如下所示：

```
RB-INSERT(T, z)
 1  y ← nil[T]
 2  x ← root[T]
 3  while x ≠ nil[T]
 4      do y ← x
 5         if key[z] < key[x]
 6            then x ← left[x]
 7            else x ← right[x]
 8  p[z] ← y
 9  if y = nil[T]
10     then root[T] ← z
11     else if key[z] < key[y]
12             then left[y] ← z
13             else right[y] ← z
14  left[z] ← nil[T]
15  right[z] ← nil[T]
16  color[z] ← RED
```

第 3 章 分配器的设计与实现

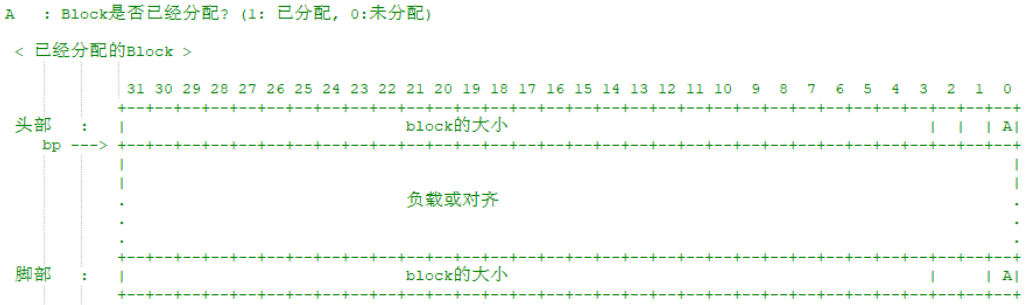
总分 50 分

3.1 总体设计（10 分）

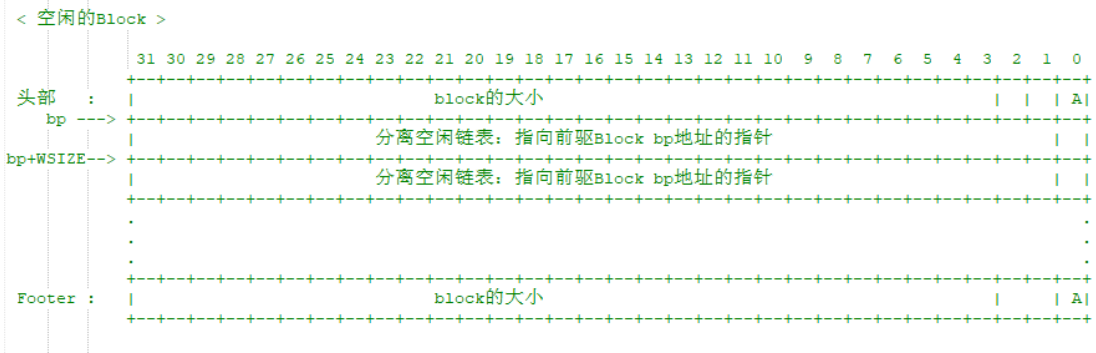
介绍堆、堆中内存块的组织结构，采用的空闲块、分配块链表/树结构和相应算法等内容。

堆及堆中内存块的组织结构

已经分配的 Block 的组织结构：



空闲 Block 的组织结构：



分配块空闲块结构链表：

分配块、结构块使用隐式链表相连，具体相连的方式同 mm-implicit.c。

采用分离的空闲链表来链接所有的空闲块，使用分离适配的方法对链表进行分组，划分大小类如下：{1},{2},(2^1,2^2],(2^2,2^3]…，观察数据，所有 tracesfile 中出现的最大块大小为 614784，所以设置 LISTLIMIT=20 作为链表头指针数目的最大值。

一条链表之中，块按照大小进行递增排序，每次插入与删除链表节点时维护

链表的递增序。

算法

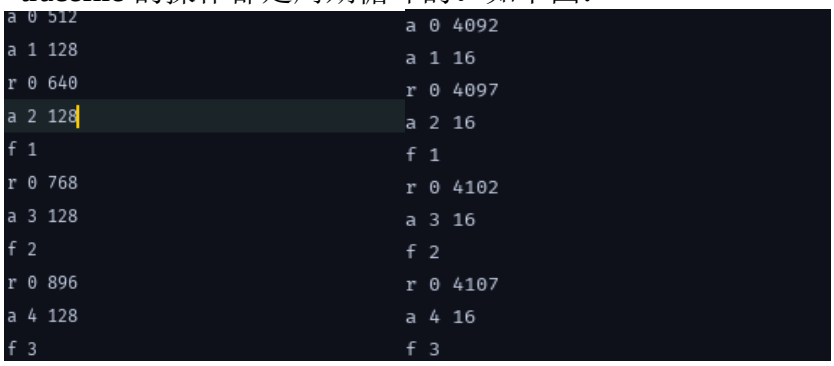
因为链表是按照空间递增序维护的，所以采用首次适配的方法。

放置的时候进行了**重要的优化**，如下：

优化是针对后两个测试 mm_realloc 数据而言，观察数据之后发现：

1） 两个 tracefile 之中 mm_realloc 的调用都是“r 0 size”类型，而且没有对 0 的 free。其中 size 从小到大依次递增，查看调用程序 mdriver.c 之后发现，程序维护一个 trace->blocks[]，存放每次 mm_malloc 之后指向 block 负载的指针 ptr，index 0 代表第一次使用 mm_malloc 开辟的 Block。所以可以得出，第一次调用 mm_alloc “a 0 size”产生的 Block 是不会被 free 的，而且每次 realloc 都会这个 Block 进行拓展。

2） 两个 tracefile 的操作都是周期循环的。如下图：



(realloc-bal.rep)　　（realloc2-bal.rep）

可以发现除去第一个 Block，在一个循环内，只存储了两个相同大小的 block。

mm_realloc：INTIALSIZE 为提前申请的堆空间的大小（除去开始 block 和结束 block），CHUNKSIZE 为每次 mm_malloc 和 mm_realloc 申请堆空间时的最小值。

使用合适的放置策略优化：首先在 mm_init 函数中拓展 INITIALSIZE 大小的堆空间。place“分配”函数中，在放置 asize 大小的块时，**如果 asize 超过一定阈值（因为两个 tracefile 中第一个 Block 的 size 都是大于其他 block 的，可以这样来识别第一个 Block）我们就将这个块放在空闲 block 的后部**，否则放在空闲 block 前部。通过设置合适的阈值，我们可以使第一个 Block 放在空闲块的最后，因为 tracefile 中数据的周期性质，通过**合理设置**，可以保证后面的数据总是放在前面 INITIALSIZE+第一个 Block 未占用的前部空间之中，这样就保证了第一个 block 始终位于整个堆空间的最大地址处。在进行 mm_realloc 的时候，特殊判断，如果当前块的后面是结束 block（证明是第一个 Block），则直接申请堆空间拓展 Block 大小。

合理设置：在 realloc2-bal.rep 中，第一个 Block 为 4092B，每次 realloc 的 size

递增，其他数据每条大小 16B；在 realloc2-bal.rep 中，第一个 Block 为 512B，每次 realloc 的 size 递增，其他数据每条大小 128B。综上，可以设置 INITIALSIZE 为 48，CHUNKSIZE 为 4096（我测试了一下，没有更好的），对于 realloc2-bal.rep，以后的数据都存放在前 48B 之中（可以设置为 44B 但是影响不大），对于 realloc-bal.rep，第一块 Block 申请 CHUNKSIZE（4096）空间，占据后部 512B，以后数据都会放在 48B+该空间的前部。

辅助函数

一） extend_heap：作用是拓展 size 大小的堆空间。1）首先对齐 size 2）然后调用 mem_sbrk 申请堆空间 3）设置 block 的 Header 和 Footer，重新设置重点 block 4）调用 coalesce 函数合并空闲块。

二） insert_node：向显式分离空闲链表中添加空闲 ptr 指向的大小为 size 的 block。1）寻找合适大小的链表 2）在链表之中寻找合适的插入位置 3）将 ptr 指向的空闲 block 插入到空闲链表之中，链表基操，插入的时候注意前驱后继是否为 NULL。

三） delete_node：在显式分离空闲链表之中删除 ptr 指向的 block。1）选择合适大小的链表，找到指针 list 2）将 ptr 指向的 block 在链表之中删除，需要注意前驱后继是否为 NULL 的情况，同时注意是否需要改变链表头指针 list。

四） coalesce：在关键函数设计中。

五） place：“分配”函数，在 ptr 指向的 block 之中分配 asize 大小的空间。1）调用 delete_node 在空闲链表之中删除 ptr 指向的 block。2）如果剩余大小不足 16B 则不进行切分 3）如果 asize 大于等于阈值，则将在 block 的后部分分配空间（原因在算法中），需要将 block 切割成前后两部分 4）如果 asize 小于阈值，则在 block 的前部分分配空间。需要将 block 切割成前后两部分。

3.2 关键函数设计（40 分）

3.2.1 int mm_init(void)函数（5 分）

函数功能：初始化整个分配器。

处理流程：

1） 初始化分离空闲链表，将每个链表设置为 NULL

2） 设置开始 Block，结束 Block：申请堆空间，设置开始 Block 的 Header，Footer，设置结束 Block 的 Footer。

3） 拓展初始大小：拓展堆空间，拓展大小为 INITIALSIZE（48B）。

4） mm_check：堆的一致性检查。

要点分析：

拓展初始堆空间：这里对应的是对于 realloc2-bal.rep 这个 tracefile 的优化，拓展初始化大小之后可以使第一块 Block 之后的数据始终保存在前 48B 之中，同时保证了第一块 Block 始终位于堆空间的最高地址，保证了两个 tracefile 之中 realloc 的简单与空间利用率。

3.2.2 void mm_free(void *ptr)函数（5 分）

函数功能：释放 ptr 指向的 block。
参　　数：void *ptr 代表指向需要释放的 block 有效负载的指针。
处理流程：

1） 设置隐式链表信息：将 block 的 HDR 和 FTR 都设置为空闲状态(size,0)。
2） 插入显示空闲链表：调用 insert_node 函数，将 ptr 指向的 block 插入到分离空闲链表之中。
3） 合并空闲 block：调用 coalesce 进行空闲 block 合并。
4） mm_check：堆的一致性检查。

要点分析：

空闲块：在释放分配块之后则该 block 已经空闲下来了，需要将该 block 加入到显式分离空闲链表之中，添加之后因为位于堆之中，地址前后的块可能存在空闲块，所以我们需要调用 colesce 进行空闲块的合并，colesce 之中包括如果发生合并产生的必要的链表操作逻辑。

3.2.3 void *mm_realloc(void *ptr, size_t size)函数（5 分）

函数功能：将 ptr 指向的 block 拓展为 size 大小（size 比原值小则不改变）。
参　　数：

void*ptr 代表指向需要释放的 block 有效负载的指针;size_t size 代表新 block 的大小。

处理流程：

1） 将 new_size 对齐：如果 new_size<=DSIZE，则手动对齐，否则调用 ALIGN 函数进行对齐至 8B 的倍数。
2） 根据新的 new_size 与 old_size 对比，判断是否为拓展。
3） 如果不为拓展则不改变。
4） 如果为拓展的情况：再次进行分类
   a) 如果后一个是结束 block：根据在算法之中的阐述，这里的判断等效于当前的 block 就是第一个 Block：按照缺少的大小申请堆空间（满足申请堆空间的最小值为 CHUNKSiZE）、在分离空闲链表之中删除结束 block、重新设置当前第一个 Block 的大小为新大小。
   b) 如果后一个是空闲块，则判断是否将当前分配块与后一个空闲块合并

之后是否足够拓展要求

i. 如果足够，则删除后一个空闲块，改变当前 block 的大小。

ii. 如果不足够，则直接进行 mm_malloc 动态分配内存，将内容复制转移到新的 block，然后释放当前的 block。

c) 如果以上都不是，则直接进行(ii)操作。

5） mm_check：堆的一致性检查。

要点分析：

判断第一个 Block：同样根据上面算法的阐释，通过我们的维护，对于最后两个测试数据，我们始终使第一个 Block 位于堆空间的最后，这样的话，我们可以直接拓展的堆空间就正好位于第一个 Block 之后，拓展堆空间之后直接改变 Block 的大小即可完成空间拓展。

3.2.4 int mm_check(void)函数（5 分）

函数功能：进行堆的一致性检查。

处理流程：

1） 扫描所有的显式分离空闲链表，对所有链表，进行遍历，对于每个节点，获取前一个和后一个 block，1.检查是否有连续的空闲块没有被合并。

2） 只要能够正常遍历，说明获得的 prev 和 succ 指针没有错误，则证明 2.空闲链表均指向有效的空闲块。

3） 遍历链表的时候检查每个块是否已经被占用，如果没有则证明 3.空闲列表的每个块都标为 free。

4） 遍历链表统计所有的空闲块，遍历堆内存统计所有的空闲块，比较两个空闲块如果相等则证明 4.每个空闲块都在空闲链表。

5） 如果能够成功遍历，则证明 5.每个堆块中的指针都指向有效的堆地址。

要点分析：

获取指向堆地址的开始指针 heap_start 和获取指向所有链表的指针 segregated_free_lists。可以利用操作是否能够完成来进行程序检查，像：空闲链表中的指针是否均指向有效的空闲块，我们用是否能够完整遍历整个链表来检查是否成立，因为指针的 pred 和 succ 都十分“脆弱”，如果改变则遍历出错的可能性很大，所以可以用这种方法大致检查程序是否存在错误。

3.2.5 void *mm_malloc(size_t size)函数（10 分）

函数功能：动态分配大小为 size B 的内存

参　　数：size_t size 代表需要动态分配的内存的大小。

处理流程：

1） 数据对齐：如果 size<=DSIZE，则手动对齐，否则调用 ALIGN 函数进行对齐至 8B 的倍数。
2） 寻找合适链表：通过要求的块的大小，在分离空闲链表之中进行查找，查找到包含块大小的链表。
3） 遍历该链表：因为链表是大小递增的、同时我们使用的是首次适配的算法，所以当寻找到第一个符合大小条件的空闲块的时候则返回 ptr。
4） 如果没有找到合适的空闲块，则拓展堆大小（满足申请堆空间的最小值为 CHUNKSiZE）。获得 ptr。
5） 分配：通过调用“分配”函数 place，在 ptr 指向的空闲块之中分配指定大小的空间。
6） mm_check：堆的一致性检查。

要点分析：
1） 链表操作：因为链表代表的块空间的大小，所以有巧妙的索引方式：对 searchsize 进行循环/2，最终<=1 时就找到了正确链表。
2） Segregated_free_lists 中始终存放的是链表的尾，所以遍历的时候每次取得前驱，而访问顺序符合要求的地址递增原则。
3） 放置函数 place：place 函数之中的“分配”方法，已经在上面的算法中有了详细介绍。

3.2.6 static void *coalesce(void *bp)函数（10 分）

函数功能：进行隐式链表之中相邻地址空闲块的合并。
参　　数：void* bp，指向需要进行相邻空闲块合并的 block 中有效负载的指针。
处理流程：
1） 如果前后都已经分配：则直接返回。
2） 如果前分配，后未分配：在显式分离链表中删除后面的 block 和当前 block，合并两个 block。
3） 如果前未分配，后分配：在显式分离链表中删除前面的 block 和当前 block，合并两个 block。
4） 如果前后都未分配：在显式分离链表中删除三个 block，将三个 block 合并。
5） 将新合成的空闲块 block 插入到显式分离链表中。

要点分析：
1） 合并空闲块：合并空闲块为一个大的空闲块的时候只需要改变空闲块最前方的 Header 和最后面的 Footer。
2） 删除原有显式分离空闲链表之中的节点：需要先删除节点之后再把最后合成的大空闲块加入。
3） 需要返回的指针：对于前未分配的情况，需要返回的指针已经改变，此时返回指向前一个 block 有效负载的指针。

第 4 章测试

总分 10 分

4.1 测试方法

使用实验包中给定的测试函数。

1） make clean。清除已经有的 make 信息（在 Makefile 中有定义）。

2） make。链接、编译成可执行程序 mdriver。其中 mdriver.c 是 mm.c 的调用程序，整个测试程序的执行逻辑存放其中。

3） ./mdriver -t traces/ -v 。测试 traces 文件夹下的所有的轨迹文件并输出结果。

4.2 自测试结果

```
linda:malloclab-handout>./mdriver -t ./traces/ -v
Team Name:Bomb Squad
Member 1 :ldx:@DX
Using default tracefiles in ./traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.000399 14264
 1       yes   97%    5848  0.000386 15135
 2       yes   99%    6648  0.000410 16203
 3       yes   99%    5380  0.000319 16865
 4       yes   99%   14400  0.000498 28916
 5       yes   94%    4800  0.000614  7812
 6       yes   91%    4800  0.000428 11218
 7       yes   95%   12000  0.000478 25120
 8       yes   88%   24000  0.000869 27631
 9       yes   99%   14401  0.000289 49882
10       yes   98%   14401  0.000151 95056
Total          96%  112372  0.004842 23209

Perf index = 58 (util) + 40 (thru) = 98/100
```

4.3 测试结果评价

对于前面 8 个数据而言，使用显式分离链表+维护大小递增+首次适配算法可以

达到较好的效果。

但对于后两个程序而言是不够的，此时需要应用上述专门针对数据的优化方法。在这种针对性背后，程序仍然具有很好的普适性，因为后两个的测试数据只是符合程序的一个特殊情况罢了。

对于不符合这种特殊情况的 realloc 程序同样进行了优化：尝试合并后一个空闲块，这虽然在测试中没有得到体现但是也是个不错的优化思路。

当然其中有很大的功劳源于对参数的合适设置。这种参数设置的启发源于测试数据。

第 5 章 总结

5.1 请总结本次实验的收获

* 动态内存分配器的原理
* 显式分离空闲链表的基本实现方法
* 针对特殊数据，进行特殊优化

5.2 请给出对本次实验内容的建议

注：本章为酌情加分项。