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 计算机类

学　　号 1170300821

班　　级 1703008

学 生 姓 名 罗瑞欣

指 导 教 师 郑贵滨

实 验 地 点 G721

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； Ubuntu 16.04 LTS 64 位/优麒麟 64 位

1.2.3 开发工具

VirtualBox/Vmware 11 以上；

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。系统之间细节不同但是不失通用性，假设堆是一个请求二进制零的区域，它紧接在未初始化的数据区域后开始，并向上生长（向更高的地址）。对于每个进程，系统内核维护着一个变量 brk，它指向堆的顶部。

分配器将堆视为一组大小不同的块的集合来维护。每个快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。

显式分配器必须在一些相当严格的约束条件下工作：

处理任意请求序列。一个应用可以有任意的分配请求和释放请求序列，只要满足约束条件就必须对相应指令进行响应。

立即响应请求。分配器必须立即响应分配请求。因此，不允许分配器为了提高性能重新排列或者缓冲请求。

只使用堆。为了使分配器是可扩展的，分配器使用的任何非标量数据结构都必须保存在堆里。

对齐。

不修改已分配的块。对已经分配的。分配器就不能再对它做出更改或者移动。

简单的分配器会把堆组织成一个大的字节数组，还有一个指针 p，初始指向这个数组的第一个字节。为了分配 size 个字节，malloc 将 p 的当前值保存在栈里，将 p 增加 size，并将 p 的旧值返回到调用函数，free 只是简单地返回到调用函数，而不做其他任何事情。然而针对更多数目的 malloc 和 free 指令后，还会出现，防止，分割，合并的问题。

2.2 带边界标签的隐式空闲链表分配器原理（5 分）

一个块是由一个字的头部、有效载荷，以及可能的一些额外的填充组成的。头部编码了这个块的大小（把偶偶头部和所有的填充），以及这个块是已分配的还是空闲的。

头部后面就是应用 malloc 时请求的有效载荷。有效载荷后面是一片不使用的填充块，其大小可以是任意的。需要填充有很多原因。比如，填充可能是分配器策略的一部分，用来对付外部碎片。或者也需要用它来满足对齐要求。

我们称这种结构为隐式空闲链表，是因为空闲块是通过头部中的大小字段隐含地连接着的。分配器通过遍历堆中所有的块，从而间接地遍历整个空闲块的集合。

带边界标签的思想就是在每个块的结尾处添加一个脚部，其中脚部就是一个头部的副本。如果每个块包括这样一个脚部，那么分配器就可以通过检查它的脚部，判断前一个块的起始位置和状态，这个脚部总是在距当前块开始位置一个字的距离。

2.3 显示空间链表的基本原理（5 分）

可以将空闲块组织成为某种形式的显式数据结构。因为根据定义，程序不需要一个空闲块的主体，所以实现这个数据结构的指针可以存放在这些空闲块的主体里面。例如，堆可以组织成一个双向空闲链表，在每个空闲块中，都包含一个前驱指针，和一个后继指针，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。不过，释放一个块的时间可以是线性的，也可以是个常数。这取决于我们所选择的空闲链表中块的排序策略。

2.4 红黑树的结构、查找、更新算法（5 分）

红黑树是一种特定类型的二叉树，它是在计算机科学中用来组织数据比如数字的块的一种结构。所有数据块都存储在节点中。这些节点中的某一个节点总是担当起始位置的功能，它不是任何节点的儿子，我们称之为根节点或根。它有最多两个"儿子"，都是它连接到的其他节点。所有这些儿子都可以有自己的儿子，以此类推。这样根节点就有了把它连接到在树中任何其他节点的路径。

如果一个节点没有儿子，我们称之为叶子节点，因为在直觉上它是在树的边缘上。子树是从特定节点可以延伸到的树的某一部分，其自身被当作一个树。在

红黑树中，叶子被假定为 null 或空。

由于红黑树也是二叉查找树，它们当中每一个节点的比较值都必须大于或等于在它的左子树中的所有节点，并且小于或等于在它的右子树中的所有节点。这确保红黑树运作时能够快速的在树中查找给定的值。红黑树对树有如下要求

性质 1. 节点是红色或黑色。

性质 2. 根节点是黑色。

性质 3 每个叶节点（NIL 节点，空节点）是黑色的。

性质 4 每个红色节点的两个子节点都是黑色。(从每个叶子到根的所有路径上不能有两个连续的红色节点)

性质 5. 从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。

红黑树背后的思想是用标准的二叉查找树（完全由 2-结点构成）和一些额外的信息（替换 3-结点）来表示 2-3 树。

我们将树中的链接分为两种类型：红链接将两个 2-结点连接起来构成一个 3-结点，黑链接则是 2-3 树中的普通链接。确切地说，我们将 3-结点表示为由一条左斜的红色链接相连的两个 2-结点。

这种表示法的一个优点是，我们无需修改就可以直接使用标准二叉查找树的 get()方法。对于任意的 2-3 树，只要对结点进行转换，我们都可以立即派生出一颗对应的二叉查找树。我们将用这种方式表示 2-3 树的二叉查找树称为红黑树。

红黑树的另一种定义是满足下列条件的二叉查找树：

⑴红链接均为左链接。

⑵没有任何一个结点同时和两条红链接相连。

⑶该树是完美黑色平衡的，即任意空链接到根结点的路径上的黑链接数量相同。

如果我们将一颗红黑树中的红链接画平，那么所有的空链接到根结点的距离都将是相同的。如果我们将由红链接相连的结点合并，得到的就是一颗 2-3 树。

相反，如果将一颗 2-3 树中的 3-结点画作由红色左链接相连的两个 2-结点，那么不会存在能够和两条红链接相连的结点，且树必然是完美平衡的。

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。

我们的分配器使用实验所提供的 memlib.c 所提供的一个内存系统模型。目的在于允许我们在不干涉已存在的系统层 malloc 包的情况下，运行分配器。包中提供的 mem_init 函数将对于堆来说可用的虚拟内存模型虚拟化为一个大的、双字对齐的字节数组。在 mem_heap 和 mem_brk 之间的字节表示已分配的虚拟内存。mem_brk 之后的字节表示为分配的虚拟内存。分配器通过调用 mem_sbrk 函数来请求额外的堆内存这个函数和系统的 sbrk 函数有相同的接口和语义。Mm_init 函数初始化分配器，如果成功就返回 0，否则返回-1，分配器在本次实验中，采用显式空闲链表和最佳适配的方式来进行分配，因此块的结构与之前所述结构相同。

在该分配器中，我们将第一个定义为一个双字对齐不使用的填充字。填充后面紧跟着一个特殊的序言块，这是一个 8 字节的已分配块，只有一个头部和一个脚部组成。序言块是在初始化的时候创建的，并且永不释放，在序言块后紧跟的是 0 个或者多个由 malloc 或者 free 调用创建的普通块。堆总是以一个特殊的结尾块来结束，这个块是一个大小为 0 的已分配块，只有一个头部组成。序言块和结尾块是一种消除合并时边界条件的技巧分配器使用一个单独的私有全局变量，它总是指向序言块。

主要算法设计：

链表是按照空间递增序维护的，采用首次适配的方法。

放置的优化针对后两个测试 mm_realloc 数据。一方面，在两个 tracefile 之中没有对 0 的 free，mm_realloc 的调用都是“r 0 size”类型。其中 size 从小到大依次递增。

```
a 262 16
f 261
r 0 5402
a 263 16
f 262
r 0 5407
a 264 16
f 263
r 0 5412
a 265 16
f 264
r 0 5417
a 266 16
f 265
r 0 5422
a 267 16
f 266
```

```
 5 a 0 512
 6 a 1 128
 7 r 0 640
 8 a 2 128
 9 f 1
10 r 0 768
11 a 3 128
12 f 2
13 r 0 896
14 a 4 128
15 f 3
16 r 0 1024
17 a 5 128
18 f 4
19 r 0 1152
20 a 6 128
21 f 5
22 r 0 1280
23 a 7 128
24 f 6
25 r 0 1408
26 a 8 128
```

根据调用程序 mdriver.c 中的函数设计发现，程序 mdriver 维护一个 trace->blocks[]，blocks[]用来存放每次调用 mm.c 中的 mm_malloc 之后指向 block 负载的指针 ptr。其中 index 0 代表第一次使用 mm_malloc 开辟的 Block。

```
    for (j = 0; j < oldsize; j++) {
        if (newp[j] != (index & 0xFF)) {
            malloc_error(tracenum, i, "mm_realloc
                "data from old block");
            return 0;
        }
    }
    memset(newp, index & 0xFF, size);

    /* Remember region */
    trace->blocks[index] = newp;
    trace->block_sizes[index] = size;
    break;

    case FREE: /* mm_free */
...
    /* Remove region from list and call st
    p = trace->blocks[index];
    remove_range(ranges, p);
    mm_free(p);
```

由此可以得出，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。

另外两个 tracefile 的操作都是周期循环的，除去第一个 Block，在一个循环内，只存储了两个相同大小的 block。

（1）mm_realloc：

INTIALSIZE ：提前申请的堆空间的大小（不包括开始 block 和结束 block）

CHUNKSIZE ：每次 mm_malloc 和 mm_realloc 申请堆空间时的最小值。

优化放置策略：

首先在 mm_init 函数中拓展 INITIALSIZE 大小的堆空间。在两个 tracefile 中第一个 Block 的 size 在 block 中式最大的的，由此可以判断是否是第一个 Block。place“分配”函数中，在放置 asize 大小的块时，如果 asize 超过一定阈值，我们就将这个块放在空闲 block 的后部，否则放在空闲 block 前部。

如果上述阈值设置足够合理，我们可以使第一个 Block 放在空闲块的最后，始终位于整个堆空间的最大地址处。依据 tracefile 中数据的周期性质，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。

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。

设置阈值：在 realloc-bal.rep 和 realloc2-bal.rep 每次 realloc 的 size 递增。其中，在 realloc-bal.rep 中，第一个 Block 为 512B，其他数据每条大小 128B，realloc2-bal.rep 第一个 Block 为 4092B，其他数据每条大小 16B。综上，可以设置 INITIALSIZE 为 48，CHUNKSIZE 为 4096，测试 realloc-bal.rep 时第一块 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；测试

realloc2-bal.rep 时，以后的数据都存放在前 48B 之中。

其他函数设计：

（1）extend_heap ：拓展 size 大小的堆空间。

首先对齐 size，然后调用 mem_sbrk 申请堆空间，然后设置 block 的 Header 和 Footer，重新设置重点 block，最后调用 coalesce 函数合并空闲块。

（2）insert_node ：向显式分离空闲链表中将 block 添加到空闲 ptr 处

首先寻找合适大小的链表，然后在链表之中寻找合适的插入位置，最后将 ptr 指向的空闲 block 插入到空闲链表之中，插入的时候注意前驱后继是否为 NULL。

（3）delete_node ：在显式分离空闲链表之中删除 ptr 指向的 block。

首先选择合适大小的链表，找到指针 list，然后将 ptr 指向的 block 在链表之中删除（注意前驱后继是否为 NULL，以及是否需要改变链表头指针 list）

（4）place ：在 ptr 指向的 block 之中分配 asize 大小的空间。

首先调用 delete_node ，在空闲链表之中删除 ptr 指向的 block。如果剩余大小不足 16B 则不进行切分。然后如果 asize 大于等于阈值，在 block 的后部分分配空间，并将 block 切割成前后两部分；如果 asize 小于阈值，则在 block 的前部分分配空间。

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) 调用 mm_check：检查堆的一致性。

要点分析：

(1) 拓展初始堆空间：对应对于 realloc2-bal.rep 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，
(2) 同时保证了第一块 Block 始终位于堆空间的最高地址，同时保证两个 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 ，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) 检查堆的一致性： mm_check
要点分析：
(1) 需要将空闲块 block 加入到显式分离空闲链表之中。
(2) 在添加之后 block 位于堆之中，地址前后的块可能存在空闲块。
(3) 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：拓展 ptr 指向的 block 为 size 大小。
参　　数：
(1) void*ptr ：指向需要释放的 block 有效负载的指针;
(2) size_t size：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。如果为拓展的情况：再次进行分类。
   a) 后一个是结束 block：根据在算法之中的阐述，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) 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。如果不足够，则直接进行动态分配内存 mm_malloc，将内容复制转移到新的 block，然后释放当前的 block。 如果足够，则删除后一个空闲块，改变当前 block 的大小。
(3) 检查堆的一致性：mm_check 。
要点分析：
(1) 判断第一个 Block：对于最后两个测试数据，始终使第一个 Block 位于堆空间的最后。
(2) 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：堆的一致性检查。

处理流程：

(1) 检查是否有连续的空闲块没有被合并：扫描所有的显式分离空闲链表，对遍历所有链表，获取每个节点前一个和后一个 block，检查是否有连续的空闲块没有被合并。
(2) 检查空闲链表均指向有效的空闲块：只要能够正常遍历，说明获得的 prev 和 succ 指针没有错误，证明空闲链表均指向有效的空闲块。
(3) 检查空闲列表的每个块都标为 free：遍历链表的时候检查每个块是否已经被占用，如果没有则证明空闲列表的每个块都标为 free。
(4) 检查每个空闲块都在空闲链表：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，则证明每个空闲块都在空闲链表。
(5) 检查每个堆块中的指针都指向有效的堆地址：如果能够成功遍历，则证明每个堆块中的指针都指向有效的堆地址。

要点分析：

(1) 利用操作是否能够完成，来检查获取指向堆地址开始的指针 heap_start 和指向所有链表的指针 segregated_free_list 的程序。
(2) 用是否能够完整遍历整个链表来检查空闲链表中的指针是否均指向有效的空闲块。
(3) 指针的 pred 和 succ 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 ：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、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（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) 检查堆的一致性：mm_check。

要点分析：

(1) 链表操作：因为链表代表的块空间的大小，对 searchsize 进行循环除以二，最终小于等于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。

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) 若*bp 前后都已经分配：则直接返回。
(2) 若*bp 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) 若*bp 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) 若*bp 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 mm.o，进一步生成可执行程序 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 测试结果评价

(1) 显式分离链表+维护大小递增+首次适配算法，对于前八个有比较好的效果

(2) 对于两个的测试数据应用上述专门针对数据的优化方法。

(3) 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。

(4) 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。

(5) 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。

4.3 自测试结果

```
1170300821@luoruixin:~/hitics/lab8/malloclab-handout$ ./mdriver -t ./traces/ -v
Team Name:1170300821
Member 1 :Luo Ruixin:3102595709@qq.com
Using default tracefiles in ./traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   97%    5694  0.000401 14207
 1       yes   99%    5848  0.000375 15599
 2       yes   99%    6648  0.000345 19247
 3       yes   99%    5380  0.000394 13658
 4       yes   99%   14400  0.000449 32100
 5       yes   94%    4800  0.000366 13129
 6       yes   91%    4800  0.000450 10679
 7       yes   95%   12000  0.000439 27360
 8       yes   88%   24000  0.003762  6380
 9       yes   99%   14401  0.000264 54632
10       yes   98%   14401  0.000194 74385
Total          96%  112372  0.007436 15111

Perf index = 58 (util) + 40 (thru) = 98/100
1170300821@luoruixin:~/hitics/lab8/malloclab-handout$
```

第 5 章 总结

5.1 请总结本次实验的收获

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

5.2 请给出对本次实验内容的建议

注：本章为酌情加分项。

参考文献

为完成本次实验你翻阅的书籍与网站等

[1] 林来兴. 空间控制技术[M]. 北京：中国宇航出版社，1992：25-42.

[2] 辛希孟. 信息技术与信息服务国际研讨会论文集：A 集[C]. 北京：中国科学出版社，1999.

[3] 赵耀东. 新时代的工业工程师[M/OL]. 台北：天下文化出版社，1998 [1998-09-26]. http://www.ie.nthu.edu.tw/info/ie.newie.htm（Big5）.

[4] 谌颖. 空间交会控制理论与方法研究[D]. 哈尔滨：哈尔滨工业大学，1992：8-13.

[5] KANAMORI H. Shaking Without Quaking[J]. Science，1998，279（5359）：2063-2064.

[6] CHRISTINE M. Plant Physiology: Plant Biology in the Genome Era[J/OL]. Science，1998，281：331-332[1998-09-23]. http://www.sciencemag.org/cgi/collection/anatmorp.