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 计算机专业

学　　号 1190200501

班　　级 1903002

学　　生 林燕燕

指 导 教 师 郑贵滨

实 验 地 点 G709

实 验 日 期 2021.06.11

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；1.6GHz；8G RAM；256G SSD Disk；1T HDD Disk

1.2.2 软件环境

Windows10 64 位；Vmware 14pro；Ubuntu 20.04.2 LTS 64 位

1.2.3 开发工具

Visual Studio Code 64 位；vim/gpedit+gcc

1.3 实验预习

- 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。

分配器有两种基本风格：显式分配器和隐式分配器。两种风格都要求应用显式地分配块。它们的不同之处在于由哪个实体来负责释放已分配的块。

显式分配器：要求应用显式地释放任何已分配的块。例如 C 程序通过调用 malloc 函数来分配一个块，通过调用 free 函数来释放一个块。

隐式分配器：要求分配器检测一个已分配块何时不再被程序所使用，那么就释放这个块，也叫做垃圾收集器。例如，诸如 Lisp、ML、以及 Java 之类的高级语言就依赖垃圾收集来释放已分配的块。

2.2 带边界标签的隐式空闲链表分配器原理（5 分）

对于带边界标签的隐式空闲链表分配器，一个块是由一个字的头部、有效载荷、可能的一些额外的填充，以及在块的结尾处的一个字的脚部组成的。头部编码了这个块的大小（包括头部和所有的填充），以及这个块是已分配的还是空闲的。如果我们强加一个双字的对齐约束条件，那么块大小就总是 8 的倍数，且块大小的最低 3 位总是 0。因此，我们只需要内存大小的 29 个高位，释放剩余的 3 位来编码其他信息。在这种情况中，我们用其中的最低位（已分配位）来指明这个块是已分配的还是空闲的。

头部后面就是应用调用 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。

我们将对组织为一个连续的已分配块和空闲块的序列，称为隐式空闲链表，是因为空闲块是通过头部中的大小字段隐含地连接着的。分配器可以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。

2.3 显式空间链表的基本原理（5 分）

因为根据定义，程序不需要一个空闲块的主体，所以实现空闲链表数据结构的指针可以存放在这些空闲块的主体里面。

显式空闲链表结构将堆组织成一个双向空闲链表，在每个空闲块的主体中，都包含一个 pred（前驱）和 succ（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，也可能是个常数，这取决于空闲链表中块的排序策略。

一种方法是用后进先出（LIFO）的顺序维护链表，将新释放的块放置在链表的开始处。另一种方法是按照地址顺序来维护链表，其中链表中每个块的地址都小于它后继的地址。

2.4 红黑树的结构、查找、更新算法（5 分）

1、红黑树的结构：

红黑树是一种近似平衡的二叉查找树，它能够确保任何一个节点的左右子树的高度差不会超过二者中较低那个的一陪。具体来说，红黑树是满足如下条件的二叉查找树（binary search tree）：

1) 每个节点要么是红色，要么是黑色。
2) 根节点必须是黑色
3) 红色节点不能连续，即：红色节点的孩子和父亲都不能是红色。
4) 对于每个节点，从该点至 null（树尾端）的任何路径，都含有相同个数的黑色节点。

在树的结构发生改变时，往往会破坏上述条件 3 或 4，需要通过调整使得查找树重新满足红黑树的条件。

2、红黑树的查找：

红黑树是一种特殊的二叉查找树，他的查找方法也和二叉查找树一样，不需要做太多更改。但是由于红黑树比一般的二叉查找树具有更好的平衡，所以查找起来

更快。红黑树的主要是想对 2-3 查找树进行编码，尤其是对 2-3 查找树中的 3-nodes 节点添加额外的信息。红黑树中将节点之间的链接分为两种不同类型，红色链接，他用来链接两个 2-nodes 节点来表示一个 3-nodes 节点。黑色链接用来链接普通的 2-3 节点。特别的，使用红色链接的两个 2-nodes 来表示一个 3-nodes 节点，并且向左倾斜，即一个 2-node 是另一个 2-node 的左子节点。这种做法的好处是查找的时候不用做任何修改，和普通的二叉查找树相同。

3、红黑树的更新：

1) 与 2-3 树对应关系

如果将一棵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节点合并，得到的就是一棵 2-3 树。

2) 旋转

修复红黑树，使得红黑树中不存在红色右链接或两条连续的红链接。

左旋：将红色的右链接转化为红色的左链接

右旋：将红色的左链接转化为红色的右链接，代码与左旋完全相同，只要将 left 换成 right 即可。

3) 插入结点

在插入新的键时，我们可以使用旋转操作帮助我们保证 2-3 树和红黑树之间的一一对应关系，因为旋转操作可以保持红黑树的两个重要性质：有序性和完美平衡性。也就是说，我们在红黑树中进行旋转时无需为树的有序性或者完美平衡性担心。下面我们来看看应该如何使用旋转操作来保持红黑树的另外两个重要性质：不存在两条连续的红链接和不存在红色的右链接。如以下一些情况：

1. 向树底部的 2-结点插入新键

一棵只含有一个键的红黑树只含有一个 2-结点。插入另一个键之后，我们马上就需要将他们旋转。如果新键小于老键，我们只需要新增一个红色的节点即可，新的红黑树和单个 3-结点完全等价。如果新键大于老键，那么新增的红色节点将会产生一条红色的右链接。我们需要使用 parent = rotateLeft(parent);来将其旋转为红色左链接并修正根结点的链接，插入才算完成。两种情况均把一个 2-结点转换为一个 3-结点，树的黑链接高度不变。

2. 向一棵双键树（即一个 3-结点）中插入新键

这种情况又可分为三种子情况：新键小于树中的两个键，在两者之间，或是大于树中的两个键。每种情况中都会产生一个同时链接到两条红链接的结点，而我们的目标就是修正这一点。

三者中最简单的情况是新键大于原树中的两个键，因此它被链接到 3-结点的右链接。此时树是平衡的，根结点为中间大小的键，它有两条红链接分别和较小和较大的结点相连。如果我们将两条链接的颜色都由红变黑，那么我们就得到了一棵由三个结点组成，高为 2 的平衡树。它正好能够对应一棵 2-3 树。其他两种情况最终也会转化为这两种情况。

如果新键小于原书中的两个键，它会被链接到最左边的空链接，这样就产生了两条连续的红链接。此时我们只需要将上层的红链接右旋转即可得到第一种情况。

如果新键介于原书中的两个键之间，这又会产生两条连续的红链接，一条红色左链接接一条红色右链接。此时我们只需要将下层的红链接左旋即可看得到第二种情况。

3. 根结点总是黑色

颜色转换会使根结点变为红色，我们在每次插入操作后都会将根结点设为黑色。

4. 向树底部的 3-结点插入新键

现在假设我们需要在树的底部的一个 3-结点下加入一个新结点。前面讨论过的三种情况都会出现。颜色转换会使指向中结点的链接变红，相当于将它送入了父结点。这意味着在父结点中继续插入一个新键，我们也会继续用相同的办法解决这个问题。

5. 将红链接在树中向上传递

2-3 树中的插入算法需要我们分解 3-结点，将中间键插入父结点，如此这般知道遇到一个 2-结点或是根结点。总之，只要谨慎地使用左旋，右旋，颜色转换这三种简单的操作，我们就能保证插入操作后红黑树和 2-3 树的一一对应关系。在沿着插入点到根结点的路径向上移动时在所经过的每个结点中顺序完成以下操作，我们就能完成插入操作：

如果右子结点是红色的而左子结点是黑色的，进行左旋转

如果左子结点是红色的且她的左子结点也是红色的，进行右旋

如果左右子结点均为红色，进行颜色转换。

4) 删除操作

要描述删除算法，首先要回到 2-3 树。和插入操作一样，我们也可以定义一系列局部变换来在删除一个结点的同时保持树的完美平衡性。这个过程比插入一个结点更加复杂，因为我们不仅要在（为了删除一个结点而）构造临时 4-结点时沿着查找路径向下进行变换，还要在分解遗留的 4-结点时沿着查找路径向上进行变换（同插入操作）。

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。

1. 堆：

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。对于每个进程，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。

2. 堆中内存块的组织结构：

一个块是由一个字的头部、有效载荷、可能的一些额外的填充，以及在块的结尾处的一个字的脚部组成的。头部编码了这个块的大小（包括头部和所有的填充），以及这个块是已分配的还是空闲的。如果我们强加一个双字的对齐约束条件，那么块大小就总是 8 的倍数，且块大小的最低 3 位总是 0。因此，我们只需要内存大小的 29 个高位，释放剩余的 3 位来编码其他信息。在这种情况中，我们用其中的最低位（已分配位）来指明这个块是已分配的还是空闲的。

3. 采用的空闲块、分配块链表：

使用隐式的空闲链表，使用立即边界标记合并方式，最大的块为 2^{32} =4GB，代码是 64 位干净的，即代码能不加修改地运行在 32 位或 64 位的进程中。

4. 相关算法：

int mm_init(void)函数；

void mm_free(void *ptr)函数；

void *mm_realloc(void *ptr, size_t size)函数；

int mm_check(void)函数；

void *mm_malloc(size_t size)函数。

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：初始化带空闲块的堆
处理流程：

1. 在使用 mm_malloc、mm_realloc 或 mm_free 之前，首先要调用该函数进行初始化，将分离空闲链表全部初始化为 NULL。
2. mm_init 函数从内存中得到四个字，并将堆初始化，创建一个空的空闲链表。
3. 调用 extend_heap 函数，这个函数将堆扩展 INITCHUNKSIZE 字节，并且创建初始的空闲块。

要点分析：

分配器使用最小块的大小为 16 字节，创建空闲链表之后需要使用 extend_heap 来扩展堆。

3.2.2 void mm_free(void *ptr)函数（5 分）

函数功能：释放参数“ptr”指向的已分配内存块
参　　数：指向请求块首字的指针 ptr
处理流程：

1. 通过调用 GET_SIZE(HDRP(bp))函数获得请求块的大小
2. 调用 PUT(HDRP(bp), PACK(size, 0))、PUT(FTRP(bp), PACK(size, 0))，将请求块的头部和脚部的已分配位设置为 0，表示为 free。
3. 调用 coalesce(bp)，将释放的块 bp 与相邻的空闲块合并起来。

要点分析：

1. ptr 是之前调用 mm_malloc 或 mm_realloc 返回的值，并且没有释放过；
2. 在将请求块的 bp 标记位 free 之后，要将它与其相邻的空闲块合并。

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

函数功能：向 ptr 所指的块重新分配一个具有至少 size 字节的有效负载的块
参　　数：指向请求块首字的指针 ptr，需要分配的字节 size
处理流程：

1. 首先检查请求的真假，在检查完请求的真假后，分配器调整请求块的大小，为头部和脚部留有空间，满足双字对齐的要求。强制最小块的大小是 16 字节，8 字节用于满足对齐的要求，另外 8 字节用来放头部和脚部。对于超过 8 字节的请求，一般的规则是加上开销字节，向上舍入到最接近的 8 的整数倍。
2. 调用 copySize = GET_SIZE(HDRP(ptr))得到 ptr 块大小，如果 size 比 copySize 小，则更新 copySize 为 size，释放 ptr。

要点分析：

要调整 size 的大小，当 size 小于原来的 ptr 指向的块大小时，要更新 copySize 的值。

3.2.4 int mm_check(void)函数（5 分）

函数功能：检查堆的一致性
处理流程：
1. 定义指针 bp，将其初始化为序言块的全局变量 heap_listp。
2. 检查序言块，当序言块不是 8 字节的已分配块，打印 Bad prologue header。
3. 调用 checkblock 函数，检查是否都为双字对齐，通过获得 bp 所指块的头部和脚部指针，判断两者是否匹配，不匹配的话就返回错误信息。
4. 检查所有 size 大于 0 的块，若 verbose 不为 0，执行 printblock 函数。
5. 检查结尾块，当结尾块不是大小为 0 的已分配块，打印 Bad epilogue header。

要点分析：
checkheap 函数主要检查了堆序言块和结尾块，每个 size 大于 0 的块都需要检查是否为双字对齐和头部脚部 match，并且打印块的头部和脚部信息。

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

函数功能：申请有效载荷至少是 size 字节的内存块，返回该内存块地址首地址
参　　数：申请的块大小 size 字节
处理流程：
1. 检查请求的真假后，分配器调整请求块的大小，为头部和脚部留有空间，满足对双字对齐的要求。强制最小块的大小是 16 字节，8 字节用于满足对齐的要求，另外 8 字节用来放头部和脚部。对于超过 8 字节的请求，一般的规则是加上开销字节，向上舍入到最接近的 8 的整数倍。
2. 调整请求大小后，搜索空闲链表，寻找合适空闲块。若有合适空闲块，分配器就放置这个请求块，分割出多余的部分，返回新分配块的地址。

要点分析：
该函数时为了更新 size 来满足要求的大小，在分离空闲链表数组里找到合适的请求块，找不到就使用新的空闲块扩展堆。

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

函数功能：将要回收的空闲块和临近的空闲块（如果有的话）合并成一个大的空闲块，返回合并后的空闲块指针
处理流程：
1. 获得前一块和后一块的已分配位，调用 GET_SIZE 函数获得 bp 指向的块的大小
2. 根据相邻块，分四种情况：
   1) 前后均为 allocated 块，不合并，返回 bp；
   2) 前面的块是 allocated，后面的块时 free，将两个 free 块合并；
   3) 前面的块是 free，后面的块是 allocated，将两个 free 块合并，更新 bp；
   4) 前后均为 free，将三个块合并，更新 bp；
3. 返回 bp

要点分析：

在合并 free 块后注意根据情况更新 bp。

第 4 章测试

总分 10 分

4.1 测试方法与测试结果(3 分)

1. 测试方法：
   1) 生成可执行评测程序文件的方法
      linux>make
   2) 评测方法:
      mdriver [-hvVa] [-f <file>]
      选项：
      -a 不检查分组信息
      -f <file> 使用 <file>作为单个的测试轨迹文件
      -h 显示帮助信息
      -l 也运行 C 库的 malloc
      -v 输出每个轨迹文件性能
      -V 输出额外的调试信息
   3) 轨迹文件：指示测试驱动程序 mdriver 以一定顺序调用 mm_malloc, mm_realloc 和 mm_free
   4) 性能评分：性能分 pindex 是空间利用率和吞吐率的线性组合
   5) 获得测试总分 linux>./mdriver -av -t traces/
2. 测试结果：

```
lyy@ubuntu:/mnt/hgfs/CSAPP/Lab/Lab08_malloc/malloclab-handout-hit$ ./mdriver -av -t traces/
Using default tracefiles in traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.006857   830
 1       yes   99%    5848  0.006363   919
 2       yes   99%    6648  0.010763   618
 3       yes  100%    5380  0.008031   670
 4       yes   66%   14400  0.000096149378
 5       yes   92%    4800  0.006754   711
 6       yes   92%    4800  0.006109   786
 7       yes   55%   12000  0.143984    83
 8       yes   51%   24000  0.285567    84
 9       yes   27%   14401  0.061129   236
10       yes   34%   14401  0.002101  6854
Total          74%  112372  0.537753   209

Perf index = 44 (util) + 14 (thru) = 58/100
```

4.2 测试结果分析与评价（3 分）

仅用隐式空闲链表实现，未作优化，得分 58 分，测试结果不理想。

4.3 性能瓶颈与改进方法分析（4 分）

仅用隐式空闲链表实现，性能不优，可以使用显式空闲链表、基于边界标签的空闲块合并、首次适配来优化性能，或使用红黑树作为最优方案，还可以使用宏函数实现一些指针的算术运算。

第 5 章 总结

5.1 请总结本次实验的收获

明白了动态内存分配的原理，了解了堆的运行规则。

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

[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.