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 1190200122

班　　级 1903001

学 生 姓 名 袁野

指 导 教 师 郑贵滨

实 验 地 点 G709

实 验 日 期 2021-6-15

计算机科学与技术学院

目 录

第 1 章 实验基本信息 - 4 -

1.1 实验目的 - 4 -
1.2 实验环境与工具 - 4 -
1.2.1 硬件环境 - 4 -
1.2.2 软件环境 - 4 -
1.2.3 开发工具 - 4 -
1.3 实验预习 - 4 -

第 2 章 实验预习 - 6 -

2.1 动态内存分配器的基本原理（5 分） - 6 -
2.2 带边界标签的隐式空闲链表分配器原理（5 分） - 6 -
2.3 显式空间链表的基本原理（5 分） - 7 -
2.4 红黑树的结构、查找、更新算法（5 分） - 7 -

第 3 章 分配器的设计与实现 - 11 -

3.2.1 INT MM_INIT(VOID)函数（5 分） - 11 -
3.2.2 VOID MM_FREE(VOID *PTR)函数（5 分） - 12 -
3.2.3 VOID *MM_REALLOC(VOID *PTR, SIZE_T SIZE)函数（5 分） - 12 -
3.2.4 INT MM_CHECK(VOID)函数（5 分） - 12 -
3.2.5 VOID *MM_MALLOC(SIZE_T SIZE)函数（10 分） - 13 -
3.2.6 STATIC VOID *COALESCE(VOID *BP)函数（10 分） - 13 -

第 4 章测试 - 14 -

4.1 测试方法与测试结果(3 分) - 14 -
4.2 测试结果分析与评价（2 分） - 14 -
4.4 性能瓶颈与改进方法分析（5 分） - 14 -

第 5 章 总结 - 16 -

5.1 请总结本次实验的收获 - 16 -
5.2 请给出对本次实验内容的建议 - 16 -

参考文献 - 18 -

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

Legion Y7000P 2019 PG0
CPU:Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz (12 CPUs), ~2.6GHz
RAM: 16384MB

1.2.2 软件环境

Windows 10 家庭中文版 64-bit
Ubuntu 20.04.2 LTS
VMware® Workstation 16 Player 16.1.0 build-17198959

1.2.3 开发工具

Microsoft Visual Studio Community 2019 版本 16.9.2
Microsoft Visual 1.54.3
GCC 9.3.0

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者着一个进程的虚拟内存区域，称为堆（heap）。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，内核维护者一个变量 brk（读作“break”），它指向堆的顶部。

分配器将堆视为一组不同大小的块（block）的集合来维护。每个块就是一个连续的虚拟内存片（chunk），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应用程序使用。空闲块可以用来分配。空闲块保持空闲，直到它显式地被应用所分配。一个已分配地块保持已分配状态，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。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地块。

显式分配器：要求应用显式地释放任何已分配的块。例如，C 标准库提供一种叫做 malloc 程序包地显式分配器。C 程序通过调用 malloc 函数来分配一个块，并通过调用 free 函数来释放一个块。C++中的 new 和 delete 操作符与 C 中的 malloc 和 free 相当。

隐式分配器，另一方面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。隐式分配器也叫做垃圾收集器，而自动释放未使用的已分配的块的过程叫做垃圾收集。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）

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。

我们称这种结构称为隐式空闲链表，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。注意：此时我们需要某种特殊标记的结束块，可以是一个设置了已分配位而大小为零的终止头部。

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）

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)。

红黑树具有如下五条性质：

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，那么它的两个儿子都是黑的。

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

（一）红黑树的查找

红黑树的查找与一般二叉查找树相差不大，伪代码如下：

```
1.  RB-SEARCH(T, z)
2.    y ← NIL
3.    x ← root[T]
4.    while x ≠ NIL
5.      do y ← x
6.        if z < key[x]
7.          then x ← left[x]
8.        else if z > key[x]
9.          x ← right[x]
10.       else
11.         return x
12.   return NIL
```

（二）红黑树的插入和插入修复

插入前面亦无不同

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

```
13.            then left[y] ← z
14.            else right[y] ← z
15.   left[z] ← nil[T]
16.   right[z] ← nil[T]
17.   color[z] ← RED
18.   RB-INSERT-FIXUP(T, z)
```

当遇到下述 3 种情况时：

插入修复情况 1：如果当前结点的父结点是红色且祖父结点的另一个子结点（叔叔结点）是红色

插入修复情况 2：当前结点的父结点是红色,叔叔结点是黑色，当前结点是其父结点的右子

插入修复情况 3：当前结点的父结点是红色,叔叔结点是黑色，当前结点是其父结点的左子

要进行调整呢，具体如下所示：

```
1.   RB-INSERT-FIXUP（T,z）
2.    while color[p[z]] = RED
3.       do if p[z] = left[p[p[z]]]
4.             then y ← right[p[p[z]]]
5.                  if color[y] = RED
6.                     then color[p[z]] ← BLACK                ▷ Case 1
7.                          color[y] ← BLACK                   ▷ Case 1
8.                          color[p[p[z]]] ← RED               ▷ Case 1
9.                          z ← p[p[z]]                        ▷ Case 1
10.                    else if z = right[p[z]]
11.                            then z ← p[z]                   ▷ Case 2
12.                                 LEFT-ROTATE(T, z)          ▷ Case 2
13.                            color[p[z]] ← BLACK             ▷ Case 3
14.                            color[p[p[z]]] ← RED            ▷ Case 3
15.                            RIGHT-ROTATE(T, p[p[z]])        ▷ Case 3
16.            else (same as then clause
17.                          with "right" and "left" exchanged)
18. color[root[T]] ← BLACK
```

（三）红黑树的删除和删除修复

单纯删除结点的总操作

```
1.   RB-DELETE-FIXUP(T, x)
2.    while x ≠ root[T] and color[x] = BLACK
3.       do if x = left[p[x]]
```

```
4.          then w ← right[p[x]]
5.              if color[w] = RED
6.                then color[w] ← BLACK                    ▷ Case 1
7.                     color[p[x]] ← RED                   ▷ Case 1
8.                     LEFT-ROTATE(T, p[x])                ▷ Case 1
9.                     w ← right[p[x]]                     ▷ Case 1
10.             if color[left[w]] = BLACK and color[right[w]] = BLACK
11.               then color[w] ← RED                       ▷ Case 2
12.                    x ← p[x]                            ▷ Case 2
13.               else if color[right[w]] = BLACK
14.                      then color[left[w]] ← BLACK        ▷ Case 3
15.                           color[w] ← RED                ▷ Case 3
16.                           RIGHT-ROTATE(T, w)            ▷ Case 3
17.                           w ← right[p[x]]               ▷ Case 3
18.                    color[w] ← color[p[x]]               ▷ Case 4
19.                    color[p[x]] ← BLACK                  ▷ Case 4
20.                    color[right[w]] ← BLACK              ▷ Case 4
21.                    LEFT-ROTATE(T, p[x])                 ▷ Case 4
22.                    x ← root[T]                          ▷ Case 4
23.       else (same as then clause with "right" and "left" exchanged)
24. color[x] ← BLACK
```

第 3 章 分配器的设计与实现

总分 50 分

3.1 总体设计（10 分）

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。

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：

创建一个带初始空闲块的堆。

处理流程：

调用 mm_init 函数来初始化分离空闲链表和堆。mm_init 函数从内存中得到四个字，并将它们初始化，创建一个空的空闲链表。调用 extend_heap 函数，将堆扩展 CHUNKSIZE 字节，并且创建初始的空闲块。此刻，分配器已初始化了，并且准备好接受来自应用的分配和释放请求。

要点分析：

要注意堆的初始结构分布，添加序言块。要将扩展的空闲块加入的空闲链表。为了保持对齐，extend_heap 将请求大小向上舍入为最接近 2 字（8 字节）的倍

数， 然后向内存系统请求额外的堆空间。

3.2.2 void mm_free(void *ptr)函数（5 分）

函数功能：释放所请求的块
参　　数：指向请求块首字的指针 ptr
处理流程：

调用 GET_SIZE(HDRP(bp))来获得请求块的大小。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置为 0，表示为 free。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：

注意 free 块需要和与之相邻的空闲块使用边界标记合并。

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：

ptr：指向的内存块。

size：新的大小。

处理流程：

首先检查请求的真假，在检查完请求的真假后，分配器必须调整请求块的大小。从而为头部和脚部留有空间，并满足双字对齐的要求。然后对 size 进行判断：如果它小于原来块的大小，那么直接返回原来的块；如果不是，那么先检查加上前/后的块是否足够大，那么就可以直接利用，否则的话就要申请新的空闲块，然后进行数据的移动操作。
要点分析：

要点分析：

当需要重新分配的大小 size 小于原来的 ptr 指向的块的大小时，注意更新 copy_sized 的值。

3.2.4 int mm_check(void)函数（5 分）

函数功能：检查堆是否一致
处理流程：

1.首先定义指针 bp，将其初始化为指向序言块的全局变量 heap_listp。其后的操作大多数都是在 verbose 不为零时才执行的。最开始检查序言块，当序言块不是 8 字节的已分配块，就会打印 Bad prologue header。

2.对于 checkblock 函数：作用为检查是都为双字对齐，然后通过获得 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。

3.检查所有 size 大于 0 的块，如果 verbose 不为 0，就执行 printblock 函数。

4.检查结尾块。当结尾块不是一个大小为 0 的已分配块，就会打印 Bad epilogue header。

要点分析：

注意检查要比较详细。

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

函数功能：向内存请求大小为 size 字节的块

参　　数：块大小 size

处理流程：

首先检查请求的真假，在检查完请求的真假后，分配器必须调整请求块的大小。从而为头部和脚部留有空间，并满足双字对齐的要求。然后在空闲链表中寻找大小区间匹配的空闲块。如果没有找到的话就对堆进行扩展。

要点分析：

要首先，寻找大小区间匹配的空闲链，然后在这个链中找到大小合适的 free 块，如果没有就在更大的区间找一个最小的。最后才扩展栈。

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。

处理流程：

根据 ptr 所指向的块前后相邻块的情况，可以分为四种可能性：1.前面的块和后面的块都是已分配的，这种情况下直接返回。2.前面的块是已分配的，后面的块是空闲的，这种情况下将后面的块和这个块合并。3.前面的块是空闲的，后面的块是已分配的，这种情况下将前面的块和这个块合并。4.前面的和后面的块都是空闲的，这种情况将三个块合并。最后将合并好的块加入空闲表中。

要点分析：要对四种情况进行分析，合并前后注意从链表中删除和添加到链表中。

第 4 章测试

总分 10 分

4.1 测试方法与测试结果(3 分)

生成可执行评测程序文件的方法：
linux>make
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
轨迹文件：指示测试驱动程序 mdriver 以一定顺序调用
性能分 pindex 是空间利用率和吞吐率的线性组合
获得测试总分 linux>./mdriver -av -t traces/

4.2 测试结果分析与评价（3 分）

没有时间写其他的函数，所以效果一般。

4.4 性能瓶颈与改进方法分析（4 分）

我们可以采用显式空闲链表+分离空闲表+分离适配的方式以及按块大小递减的排序策略。

显式空闲链表：将空闲块组织为某种形式地显式数据结构。根据定义，程序不需要一个空闲块地主体，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，堆可以组成一个双向空闲链表，在每个空闲块中，都包含一个 pred 和 succ 指针。

分离空闲表：将所有可能的块大小分成一些等价类，也叫做大小类。分离器维护者一个空闲链表数组，每个大小类一个空闲链表，按照大小的升序排列，当分配器需要一个大小为 n 的块时，它就搜索相应的空闲链表。如果不能找到合适的块与之匹配，它就搜索下一个链表，以此类推。

分离适配：分配器维护着一个空闲链表的数据。每个空闲链表是一个大小类相关联的，并且被组织成某种类型的显式或隐式链表。每个链表包含潜在的大小不同的块，这些块的大小是大小类的成员。为了分配一个块，必须确定请求的大小类，并且对适当的空闲链表做首次适配，查找一个合适的块。如果找到了一个，那么就分割它，并将剩余的部分插入到适当的空闲链表中。如果找不到合适的块，那么就搜索下一个更大的大小类的空闲链表。如此重复，，直到找到一个合适的块。如果空闲链表中没有合适的块，那么就向操作系统请求额外的堆内存，从这个新的堆内存中分配出一个块，将剩余的部分放置在适当的大小类中，要释放一个块，我们执行合并，并将结果放置到相应的空闲链表中。

由于时间紧张不能实现完全。

第 5 章 总结

5.1 请总结本次实验的收获

1.明白了动态内存分配的原理。
2.知道的堆的运行规则。

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

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.