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 1190201423

班　　级 1903004

学　　生 顾海耀

指 导 教 师 史先俊

实 验 地 点 G709

实 验 日 期 2021/6/9

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

Visual Studio 2010 64 位以上；TestStudio；Gprof;Valgrind 等

1.3 实验预习

填上实验课前，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。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。

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 标准库提供一种叫做 malloc 程序包的显式分配器。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 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，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。

Knuth 提出了一种边界标记技术，允许在常数时间内进行对前面快的合并。这种思想是在每个块的结尾处添加一个脚部，其中脚部就是头部的一个副本。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，判断前面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）

红黑树的结构

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：

每个结点要么是红的，要么是黑的。

根结点是黑的。

每个叶结点（叶结点即指树尾端 NIL 指针或 NULL 结点）是黑的。

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。

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。

```
//查找获取指定的值
public override TValue Get(TKey key)
{
    return GetValue(root, key);
}

private TValue GetValue(Node node, TKey key)
{
    if (node == null) return default(TValue);
    int cmp = key.CompareTo(node.Key);
    if (cmp == 0) return node.Value;
    else if (cmp > 0) return GetValue(node.Right, key);
    else return GetValue(node.Left, key);
}
```

红黑树的更新

左旋:

```
/**************对红黑树节点 x 进行左旋操作 ******************/
/*
 * 左旋示意图：对节点 x 进行左旋
 *     p                       p
 *    /                       /
 *   x                       y
 *  / \                     / \
 * lx  y      ----->       x  ry
 *    / \                 / \
 *   ly ry               lx ly
 * 左旋做了三件事：
 * 1. 将 y 的左子节点赋给 x 的右子节点,并将 x 赋给 y 左子节点的父节点(y 左子节点非空时)
 * 2. 将 x 的父节点 p(非空时)赋给 y 的父节点，同时更新 p 的子节点为 y(左或右)
 * 3. 将 y 的左子节点设为 x，将 x 的父节点设为 y
 */
private void leftRotate(RBNode<T> x) {
    //1. 将 y 的左子节点赋给 x 的右子节点，并将 x 赋给 y 左子节点的父节点(y 左子节点非
空时)
    RBNode<T> y = x.right;
    x.right = y.left;

    if(y.left != null)
        y.left.parent = x;

    //2. 将 x 的父节点 p(非空时)赋给 y 的父节点，同时更新 p 的子节点为 y(左或右)
    y.parent = x.parent;

    if(x.parent == null) {
        this.root = y; //如果 x 的父节点为空，则将 y 设为父节点
    } else {
        if(x == x.parent.left) //如果 x 是左子节点
            x.parent.left = y; //则也将 y 设为左子节点
        else
            x.parent.right = y;//否则将 y 设为右子节点
    }

    //3. 将 y 的左子节点设为 x，将 x 的父节点设为 y
    y.left = x;
    x.parent = y;
}
```

右旋:

```
/**************对红黑树节点 y 进行右旋操作 ******************/
/*
 * 左旋示意图：对节点 y 进行右旋
 *        p                   p
 *       /                   /
 *      y                   x
 *     / \                 / \
 *    x  ry   ----->      lx  y
 *   / \                     / \
 * lx  rx                   rx ry
 * 右旋做了三件事：
 * 1. 将 x 的右子节点赋给 y 的左子节点,并将 y 赋给 x 右子节点的父节点(x 右子节点非空时)
 * 2. 将 y 的父节点 p(非空时)赋给 x 的父节点，同时更新 p 的子节点为 x(左或右)
 * 3. 将 x 的右子节点设为 y，将 y 的父节点设为 x
 */
private void rightRotate(RBNode<T> y) {
    //1. 将 y 的左子节点赋给 x 的右子节点，并将 x 赋给 y 左子节点的父节点(y 左子节点非
空时)
    RBNode<T> x = y.left;
    y.left = x.right;

    if(x.right != null)
        x.right.parent = y;

    //2. 将 x 的父节点 p(非空时)赋给 y 的父节点，同时更新 p 的子节点为 y(左或右)
    x.parent = y.parent;

    if(y.parent == null) {
        this.root = x; //如果 x 的父节点为空，则将 y 设为父节点
    } else {
        if(y == y.parent.right) //如果 x 是左子节点
            y.parent.right = x; //则也将 y 设为左子节点
        else
            y.parent.left = x;//否则将 y 设为右子节点
    }

    //3. 将 y 的左子节点设为 x，将 x 的父节点设为 y
    x.right = y;
    y.parent = x;
}
```

插入

```
/********************** 向红黑树中插入节点 **********************/
public void insert(T key) {
    RBNode<T> node = new RBNode<T>(key, RED, null, null, null);
    if(node != null)
        insert(node);
}

//将节点插入到红黑树中，这个过程与二叉搜索树是一样的
private void insert(RBNode<T> node) {
    RBNode<T> current = null; //表示最后 node 的父节点
    RBNode<T> x = this.root; //用来向下搜索用的

    //1. 找到插入的位置
    while(x != null) {
        current = x;
        int cmp = node.key.compareTo(x.key);
        if(cmp < 0)
            x = x.left;
        else
            x = x.right;
    }
    node.parent = current; //找到了位置，将当前 current 作为 node 的父节点

    //2. 接下来判断 node 是插在左子节点还是右子节点
    if(current != null) {
        int cmp = node.key.compareTo(current.key);
        if(cmp < 0)
            current.left = node;
        else
            current.right = node;
    } else {
        this.root = node;
    }

    //3. 将它重新修整为一颗红黑树
    insertFixUp(node);
}
```

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 232 =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：创建一个空堆

处理流程:

1.初始化分离空闲链表。根据申请的链表数组，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，并且将堆初始化，创建一个空的空闲链表。其中创建一个空的空闲链表分为以下 3 步:
a.第一个字是一个双字边界对齐不使用的填充字。
b.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，创建的时候使用 PUT(heap_listp + (1 * WSIZE), PACK(DSIZE, 1)); PUT(heap_listp + (2 * WSIZE), PACK(DSIZE, 1)); 两个语句将头部和脚部指向的字中，填充大小并且标记为已分配。
c.堆的结尾以一个特殊的结尾块来结束，使用 PUT(heap_listp + (3 * WSIZE), PACK(0, 1)); 语句来表示这个块是一个大小为零的已分配块。
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:
将请求块的 bp 标记位 free 之后，将它插入到分离的空闲链表中，注意 free 块需要和与之相邻的空闲块使用边界标记合并。

3.2.2 void mm_free(void *ptr)函数（5 分）

函数功能：释放一个块
参　　数：void *ptr
处理流程:
1.通过 GET_SIZE(HDRP(bp))来获得请求块的大小。并且使用 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 。
2.调用上文介绍的插入分离空闲链表函数 InsertNode(bp, size); 将 free 块插入到分离空闲链表中。
3.调用 coalesce(bp); 使用边界标记合并技术将释放的块 bp 与相邻的空闲块合并起来。
要点分析:
将请求块的 bp 标记位 free 之后，将它插入到分离的空闲链表中，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：void *ptr, size_t 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。第 12~13 行强制了最小块大小是 16 字节；8 字节用来满足对齐要求，而另外 8 个用来放头部和脚部。对于超过 8 字节的请求，一般的规则是加上开销字节，然后向上舍入到最接近的 8 的整数倍。
2., 执行 copySize = GET_SIZE(HDRP(ptr));得到 ptr 的块大小, 如果 size 比 copy_size 小，则更新 copy_size 为 size，然后释放 ptr。
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：检查堆的一致性
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:
checkheap 主要检查了堆序言块和结尾块，所有 size 大于 0 的块都需要检查是否双字对齐和头部脚部 match，并且打印块的头部和脚部的信息。

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：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。第 12~13 行强制了最小块大小是 16 字节；8 字节用来满足对齐要求，而另外 8 个用来放头部和脚部。对于超过 8 字节的请求，一般的规则是加上开销字节，然后向上舍入到最接近的 8 的整数倍。
2.分配器调整了请求的大小后，就会搜索空闲链表，寻找一个合适的空闲块。如果

有合适的，那么分配器就放置这个请求块，并可选址地分割出多余的部分，然后返回新分配块的地址。
要点分析：
mm_malloc 函数是为了更新 size 来满足要求的大小，然后在分离空闲链表数组里面找到合适的请求块，找不到的话就使用一个新的空闲块来扩展堆。

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

函数功能：使用边界标记合并技术将之与邻接的空闲块合并起来。
参　　数：bp
处理流程：
1.首先获得前一块和后一块的已分配位，并且调用 GET_SIZE(HDRP(bp))获得 bp 所指向的块的大小。

```
size_t  prev_alloc = GET_ALLOC(HDRP(PREV_BLKP(bp)));
size_t  next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
size_t size = GET_SIZE(HDRP(bp));
```

2.然后根据 bp 所指向的相邻块，可以得到四种情况：
(1).前后均为 allocated 块，不做合并，直接返回

```
    if (prev_alloc && next_alloc)     /*case1*/
    {
        return bp;
    }
```

(2).前面的块是 allocated，但是后面的块是 free 的，这时将两个 free 块合并

```
    else if (prev_alloc && !next_alloc)     /*case2*/
    {
        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size, 0));
    }
```

(3).后面的块是 allocated，但是前面的块是 free 的，这时将两个 free 块合并

```
    else if (!prev_alloc && next_alloc)     /*case3*/
    {
        size += GET_SIZE(HDRP(PREV_BLKP(bp)));
        PUT(FTRP(bp), PACK(size, 0));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);
}
```

(4).前后两个块都是 free 块，这时将三个块同时合并

```
    else     /*case4*/
```

```
    {
        size += GET_SIZE(HDRP(PREV_BLKP(bp))) +
GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);
}
```

3.四种操作后更新的 bp 所指向的块插入分离空闲链表

要点分析:

使用的空闲链表格式允许我们忽略潜在的麻烦边界情况，也就是请求块 bp 在堆的起始处或者堆的结尾处，如果没有这些特殊块，代码将混乱的多，更加容易出错，并且更慢，因为我们将不得不在每次释放请求时，都去检查这些并不常见的边界情况。

第 4 章测试

总分 10 分

4.1 测试方法

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：linux>./mdriver -av -t traces/

4.2 测试结果评价

使用显式分离链表+维护大小递增+首次适配算法。总体有个不错的测试结果，即使是首次适配，也能几乎达到最佳适配一样的效果。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。

4.3 自测试结果

```
ghy1190201423@ubuntu:/mnt/hgfs/hitics/malloclab-handout-hit/malloclab-handout-hit$ ./mdriver -t traces/ -v
Team Name:ghy
Member 1 :ghy:guhaiyao56@gmail.com
Using default tracefiles in traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   97%    5694  0.000266 21422
 1       yes   99%    5848  0.000261 22380
 2       yes   99%    6648  0.000304 21897
 3       yes   99%    5380  0.000242 22222
 4       yes   99%   14400  0.000372 38710
 5       yes   94%    4800  0.000332 14467
 6       yes   91%    4800  0.000333 14436
 7       yes   95%   12000  0.000366 32805
 8       yes   88%   24000  0.002830  8480
 9       yes   99%   14401  0.000214 67200
10       yes   98%   14401  0.000162 89060
Total          96%  112372  0.005681 19780

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

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

建议老师讲这个实验的要求再深入讲一下。

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.