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 1180300105

班　　级 1803001

学 生 姓 名 吴雨伦

指 导 教 师

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

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

1.2 实验环境与工具

1.2.1 硬件环境

Ryzen5 3550H

1.2.2 软件环境

gcc

1.2.3 开发工具

vs code

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）

调用 sbrk 向内核申请堆空间。建立数据结构来在这个空间上管理空闲/使用块，对 malloc 请求分配空间，对 free 请求回收空间。当自己维护的空间不足以支持 malloc 请求时，调用 sbrk 申请新的堆空间以满足要求。

2.2 带边界标签的隐式空闲链表分配器原理（5 分）

边界标签保存了这个块是否空闲、这个块的大小。

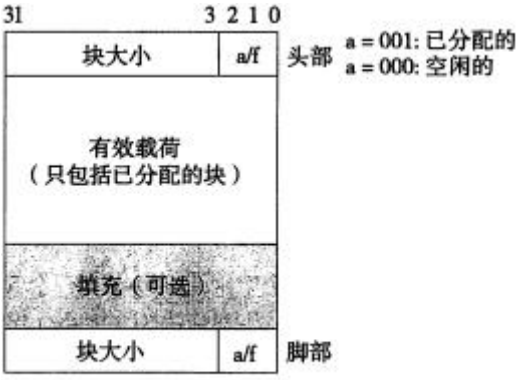


图 9-39 使用边界标记的堆块的格式

malloc 时，找到一个足够大的空闲块，然后分割成两个块，一个分配掉，一个仍然空闲；

free 时，共四种情况：

- 前驱和后继都空闲。此时直接合并三个空闲块。
- 前驱空闲，后继已分配。此时合并前驱和目标块。
- 前驱已分配，后继空闲。此时合并目标块和后继。
- 前驱和后继都已分配。此时仅目标块改为空闲块。

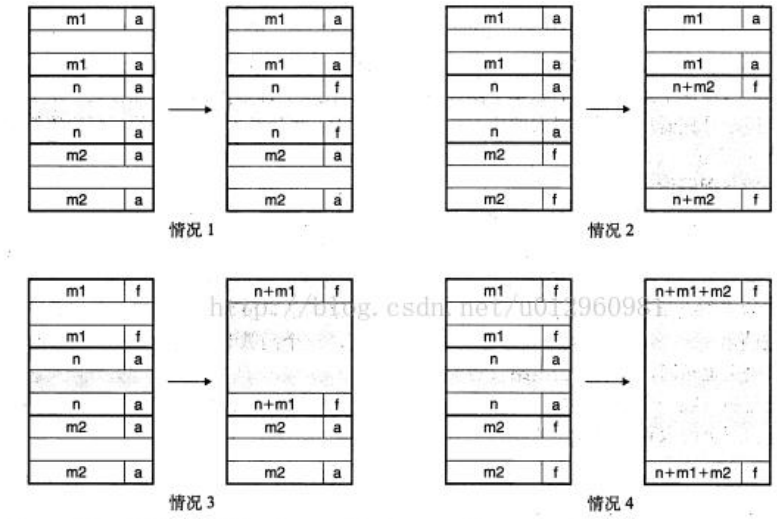


图 9-40 使用边界标记的合并。情况 1：前面的和后面块都已分配。情况 2：前面的块已分配，后面的块空闲。情况 3：前面的块空闲，后面的块已分配。情况 4：后面的块和前面的块都空闲

2.3 显示空间链表的基本原理（5 分）

利用链表来组织所有空闲块。

由于空闲块内没有 payload，故可以在空闲块内存放前驱和后继指针，分别指向前一个空闲块、后一个空闲块。

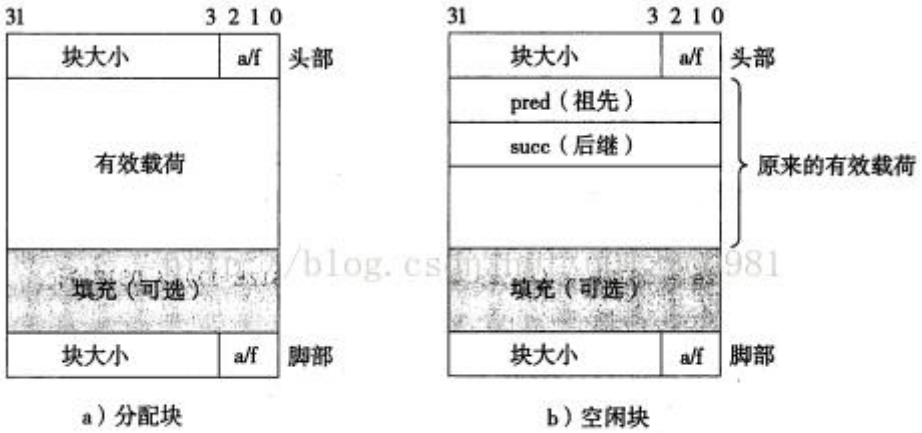
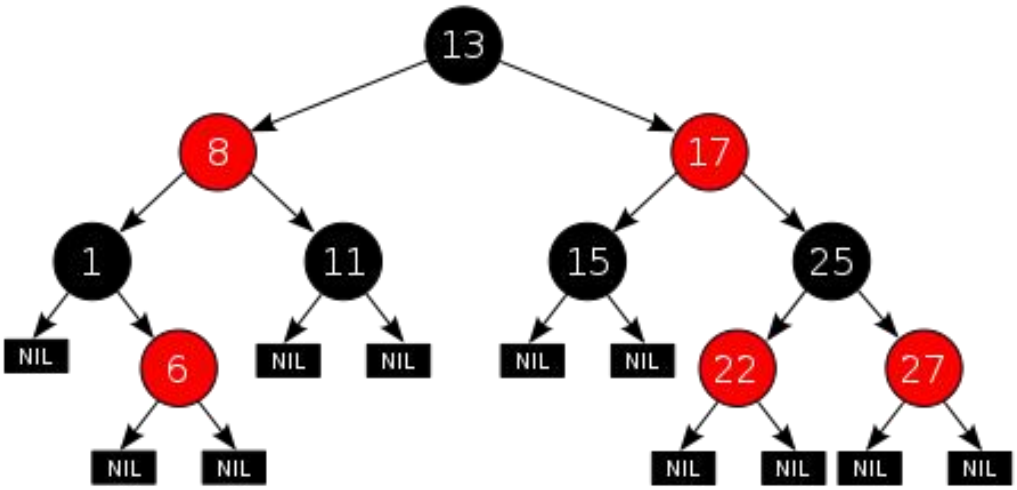


图 9-48 使用双向空闲链表的堆块的格式

2.4 红黑树的结构、查找、更新算法（5 分）

红黑树作为平衡树，需要满足

1. 节点是红色或黑色。
2. 根是黑色。
3. 所有叶子都是黑色（叶子是 NIL 节点）。
4. 每个红色节点必须有两个黑色的子节点。（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。）
5. 从任一节点到其每个叶子的所有简单路径都包含相同数目的黑色节点。



```
void insert_case(Node *p){
    if(p->parent == NULL){
        root = p;
        p->color = BLACK;
        return;
    }
    if(p->parent->color == RED){
        if(p->uncle()->color == RED) {
            p->parent->color = p->uncle()->color = BLACK;
            p->grandparent()->color = RED;
            insert_case(p->grandparent());
        } else {
            if(p->parent->rightTree == p && p->grandparent()->leftTree == p->parent) {
                rotate_left(p);
                p->color = BLACK;
                p->leftTree->color = p->rightTree->color = RED;
            } else if(p->parent->leftTree == p && p->grandparent()->rightTree == p->parent) {
                rotate_right(p);
                p->color = BLACK;
                p->leftTree->color = p->rightTree->color = RED;
            } else if(p->parent->leftTree == p && p->grandparent()->leftTree == p->parent) {
                p->parent->color = BLACK;
                p->grandparent()->color = RED;
                rotate_right(p->parent);
            } else if(p->parent->rightTree == p && p->grandparent()->rightTree == p->parent) {
                p->parent->color = BLACK;
                p->grandparent()->color = RED;
                rotate_left(p->parent);
            }
        }
    }
}
```

```
177    void delete_case(Node *p){
178        if(p->parent == NULL){
179            p->color = BLACK;
180            return;
181        }
182        if(p->sibling()->color == RED) {
183            p->parent->color = RED;
184            p->sibling()->color = BLACK;
185            if(p == p->parent->leftTree)
186                //rotate_left(p->sibling());
187                rotate_left(p->parent);
188            else
189                //rotate_right(p->sibling());
190                rotate_right(p->parent);
191        }
192        if(p->parent->color == BLACK && p->sibling()->color == BLACK
193                && p->sibling()->leftTree->color == BLACK && p->sibling()->rightTree->color == BLACK) {
194            p->sibling()->color = RED;
195            delete_case(p->parent);
196        } else if(p->parent->color == RED && p->sibling()->color == BLACK
197                && p->sibling()->leftTree->color == BLACK && p->sibling()->rightTree->color == BLACK) {
198            p->sibling()->color = RED;
199            p->parent->color = BLACK;
200        } else {
201            if(p->sibling()->color == BLACK) {
202                if(p == p->parent->leftTree && p->sibling()->leftTree->color == RED
203                        && p->sibling()->rightTree->color == BLACK) {
204                    p->sibling()->color = RED;
205                    p->sibling()->leftTree->color = BLACK;
206                    rotate_right(p->sibling()->leftTree);
207                } else if(p == p->parent->rightTree && p->sibling()->leftTree->color == BLACK
208                        && p->sibling()->rightTree->color == RED) {
209                    p->sibling()->color = RED;
210                    p->sibling()->rightTree->color = BLACK;
211                    rotate_left(p->sibling()->rightTree);
212                }
213            }
214            p->sibling()->color = p->parent->color;
215            p->parent->color = BLACK;
216            if(p == p->parent->leftTree){
217                p->sibling()->rightTree->color = BLACK;
218                rotate_left(p->sibling());
219            } else {
220                p->sibling()->leftTree->color = BLACK;
221                rotate_right(p->sibling());
222            }
223        }
224    }
```

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。

堆的前 4 个字节是无意义填充，用于对齐。

接下来 8 个字节，是一个序言块，仅有一个头部和一个尾部，没有 payload. heap_listp 指向序言块。

接下来是普通块，每个块都有头部和尾部。只要保证 payload+padding 是 8 字节对齐的，就能保证 malloc 返回值总是 8 的倍数。

在所有块的末尾，有一个结尾块，只有头部，块大小为 0，标记为已分配，作为结束标志。

malloc 采用首次适配，从整个堆内从前往后寻找第一个能放下 payload 的块。如果找不到，则申请更大的堆空间，并更新结尾块。

free 分类讨论四种情况，对空闲块执行合并操作。

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：应用程序（例如轨迹驱动的测试程序 mdriver）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。例如申请初始堆区域。

处理流程：

- 先向系统申请 4 字的空间（用于存放初始 padding、序言块、结尾块）
- 放置 1 字的初始 padding
- 放置序言块头部
- 放置序言块尾部

- 放置结尾块
- 将堆扩展一个页大小

要点分析：

若向系统申请空间失败，则返回-1.

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，没有返回值。指针值 ptr 应该是之前调用 mm_malloc 或 mm_realloc 返回的值，并且没有释放过。

参　　数：指向释放目标的指针

处理流程：

- 将这个目标设为空闲块
- 调用 coalesce 进行空闲块合并

要点分析：需要合并空闲块

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

函数功能：

- 如 ptr 是空指针 NULL,等价于 mm_malloc(size)
- 如果参数 size 为 0，等价于 mm_free(ptr)
- 如 ptr 非空, 它应该是之前调用 mm_malloc 或 mm_realloc 返回的数值 ,指向一个已分配的内存块。

参　　数：目标 ptr， 大小 size

处理流程：

- 尝试 malloc 分配 size 大小的空间
- 将 size 大小（最多原 payload 大小）的内容从 ptr 复制到新的空间
- 释放 ptr 指向的块

要点分析：注意处理 malloc 过程的异常（可能已经无法申请更多的内存）

3.2.4 int mm_check(void)函数（5 分）

函数功能：检查重要的不变量和一致性条件。当且仅当堆是一致的，才能返回非 0 值。

处理流程：

检查

- 空闲列表中的每个块是否都标识为 free（空闲）？
- 是否有连续的空闲块没有被合并？
- 是否每个空闲块都在空闲链表中？
- 空闲链表中的指针是否均指向有效的空闲块？
- 分配的块是否有重叠？
- 堆块中的指针是否指向有效的堆地址？

要点分析：

提交时注释掉 mm_check，以防影响吞吐率

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参数“size”指定大小的内存块，返回该内存块地址首地址（可以使用的区域首地址）。申请的整个块应该在对齐的区间内，并且不能与其他已经分配的块重叠。返回的地址应该是 8 字节对齐的（地址%8==0）。

参　　数：size

处理流程：

- 如果 size<=0，则什么也不做
- 确定块大小，以包含头部、尾部占用空间，且满足对齐要求
- 寻找第一个大小足够的空闲块，如果有则切割
- 如果没有找到，则向内核申请新的堆空间（至少一个页大小），然后切割新空间。

要点分析：内核可能不给新的空间，需要处理这个异常。

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：

- 得到前驱和后继的分配标记
- 分类讨论四种情况

要点分析：

要返回合并之后的空闲块指针

第 4 章测试

总分 10 分

4.1 测试方法

./mdriver -f short1-bal.rep
./mdriver -f short2-bal.rep
./mdriver -v -t traces/

4.2 测试结果评价

实现了程序要求的功能。

4.3 自测试结果

```
→ workspace git:(master) ✗ ./mdriver -v -t traces
Team Name:pion1eer
Member 1 :Wu Yulun:ruanxingzhi@gmail.com
Using default tracefiles in traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.004378  1301
 1       yes   99%    5848  0.004242  1379
 2       yes   99%    6648  0.006112  1088
 3       yes  100%    5380  0.004337  1240
 4       yes   66%   14400  0.000136105727
 5       yes   92%    4800  0.005746   835
 6       yes   92%    4800  0.005308   904
 7       yes   55%   12000  0.082550   145
 8       yes   51%   24000  0.158744   151
 9       yes   27%   14401  0.095493   151
10       yes   34%   14401  0.001972  7304
Total          74%  112372  0.369018   305

Perf index = 44 (util) + 20 (thru) = 65/100
→ workspace git:(master) ✗
```

第 5 章 总结

5.1 请总结本次实验的收获

学习了内存分配相关的知识。

5.2 请给出对本次实验内容的建议

希望不要把最后一次实验和大作业拖到期末：）

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] cppreference.com