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 计算机科学与技术

学　　号 1180300220

班　　级 1836101

学 生 姓 名 崔涵

指 导 教 师 郑贵滨

实 验 地 点 G712

实 验 日 期 2019.12.12

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

Xcode

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堆。堆是一个进程的虚拟内存区域。分配器会把堆分为若干大小不同的块，已经被应用程序使用的块为已分配的块，未被应用程序占用的块为空闲块。空闲块可以被分配给新的块使用。

分配器分为显式分配器和隐式分配器。显式分配器要求应用显式地释放已经分配给该应用的块。如 C 语言中需要通过调用 free 释放内存，通过 malloc 分配内存。Malloc 在调用时，先调用 sbrk，此函数会调用一段较大空间，malloc 会从这段空间中不断分配内存。当这段内存不够用时，malloc 会再次调用 sbrk。

隐式分配器，也叫做垃圾收集器。如 java 就利用隐式分配器释放已分配的内存。

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，一个块有一个字的头部、有效载荷、可能的一些额外的填充，结尾的一个字的脚部组成。头部负责记录这个块的大小和是否被分配的信息。如果有双字的对齐约束条件，块的大小就一定是 8 的倍数。这是，块大小的最低三位一定为 0，我们只需要存储前 29 位，剩余的三位用来编码其余信息，此块是否被分配就存在最后一位。

头部后面是有效载荷，大小可以是任意的，作用是填充，大多为了满足对其要求或解决外部碎片。

一个已经组织好的，连续的已分配块和空闲块的排列成为一个隐式空闲链表。空闲块是通过其头部的大小字段隐含的连接着的。分配器在分配内存时，将遍历所有的块，找到一个大小合适的空闲块。

2.3 显示空间链表的基本原理（5 分）

根据定义，程序并不需要一个空闲块的主体，所以实现空闲链表的指针可以存放在这些主体里边。

显式空闲链表结构将堆组织成一个双向空闲链表，每个空闲块的主题都包含一个前驱和后继指针。

双向链表与隐式链表相比，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。

2.4 红黑树的结构、查找、更新算法（5 分）

结构：

红黑树是每个节点都带有颜色属性的二叉查找树，颜色或红色或黑色。在二叉查找树强制一般要求以外，对于任何有效的红黑树我们增加了如下的额外要求:

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 4. 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。

查找：

递归：从根节点开始，如果树是空的，则返回未命中；如果不是空的，就将对应的子树的根节点放入查找函数继续向下查找。如果根节点的值比要查找的值大，就将左子树的根节点放入函数；如果比要查找的值小，就将右子树的根节点放入函数。

对于一个二叉查找树，当我们不断向下查找是，对应根节点下的树的大小也在不断减小。

更新：

更新的操作与查找类似，根本目的在于找到一个合适的位置。

将需要被插入的值与根节点进行比较。如果被插入的值大于根节点，则将根节点的右子树的根节点放入函数继续比较；如果右子树不存在，则将此要插入的值作为右子节点。如果被插入的值小于根节点，则将根节点的左子树的根节点放入函数继续比较，如果左子树不存在，则要插入的值作为根节点的左子节点

删除

在查找路径上进行和删除最小键相同的变换同样可以保证在查找过程中任意当前结点均不是 2-结点。如果被查找的键在树的底部，我们可以直接删除它。如果不在，我们需要将它和它的后继结点交换，就和二叉树一样。因为当前结点必然不是 2-结点，问题已经转化为在一颗根结点不是 2-结点子树中删除最小键，我

们可以在这个子树中使用前问所述的算法。和以前一样，删除之后我们需要向上回溯并分解余下的 4-结点。

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。

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。

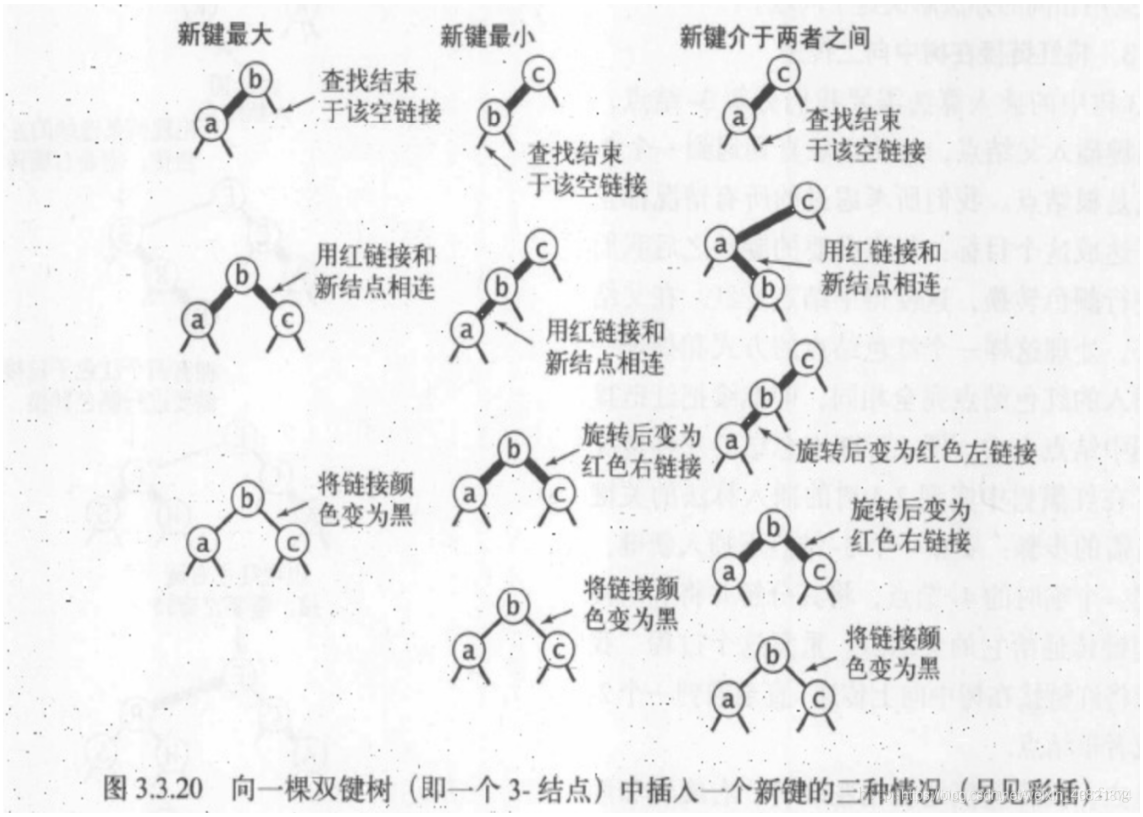

图 3.3.20 向一棵双键树（即一个 3-结点）中插入一个新键的三种情况（另见彩插）

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：

堆是一个进程的虚拟内存区域。分配器会把堆分为若干大小不同的块，已经被应用程序使用的块为已分配的块，未被应用程序占用的块为空闲块。空闲块可以被分配给新的块使用。

堆中内存块的组织结构：

堆被隐式空闲链表组织。对于带有边界标签的隐式空闲链表分配器，一个块又头部、有效载荷、填充、在结尾的脚。头部记录这个块的大小，这个块是否被使用。如果有双字的对齐约束条件，块的大小就一定是 8 的倍数。这是，块大小的最低三位一定为 0，我们只需要存储前 29 位，剩余的三位用来编码其余信息，此块是否被分配就存在最后一位。

空闲块和分配块链表：

采用分离的空闲链表。因为使用单向空闲块链表的分配去需要与空闲块数量呈线性关系的时间来分配块，而此堆的设计采用分离存储来减少分配时间，即维护多个空闲链表，每个链表中的块大小大致相同。

放置策略：首次适配。Malloc 在搜索时，将不会去所有空的块中寻找，而是去局部链表的空闲块中寻找。当分到对应大小的链表是，他的空间也会在大小类链表的范围中。这样做可以使空间利用率接近最佳适配。

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：初始化堆和分离空闲链表。

处理流程：

1. mm_init 函数从内存中的到四个字，并将堆初始化，创建一个空的空闲链表，具体步骤如下：
   a) 第一个字是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，其中包括一个头部和一个脚部。

```
PUT(heap_listp+WSIZE, PACK(OVERHEAD, 1));  /* prologue header */
PUT(heap_listp+DSIZE, PACK(OVERHEAD, 1));  /* prologue footer */
```

   这两句将头部和脚部指向字中，填充大小，并标记为已分配。

c) 堆的结尾以一个特殊的结尾块来结束，使用

```
PUT(heap_listp+WSIZE+DSIZE, PACK(0, 1));
```

此句表示这个块是一个大小为 0 的已分配块。

3. 调用 extend_heap 函数，这个函数将堆拓展 CHUNKSIZE 个字节，并创建初始的空闲块。

要点分析：

mm_init 函数初始化分配器时，分配器使用最小块的大小是 16 字节，空闲链表组织成一个隐式空闲链表，他的恒定形式是一个双字边界对齐不使用的填充字+8 字节的序言块+4 字节的结尾块。另外注意，空闲链表创建之后，堆的扩展由 extend_heap 完成。

3.2.2 void mm_free(void *ptr)函数（5 分）

函数功能：将已分配的块释放

参　　数：指向块的首地址的指针 ptr

处理流程：

1. 通过 get_size(HDRP(bp))得到请求块的大小。
2. 通过

```
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 free。

3. 调用 coalesce（bp）：使用此函数将刚刚释放的块和原本就空闲而且与此块相邻的块合并。

要点分析：

Free 一个块之后要将此块与相邻的块用 coaleasce 合并。

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

函数功能：调整 ptr 指向的块的空间为 size

参　　数：指向块的首地址的指针 ptr，新大小 size

处理流程：

```
void *mm_realloc(void *ptr, size_t size)
{
    void *newp;
    size_t copySize;

    if ((newp = mm_malloc(size)) == NULL) {
    printf("ERROR: mm_malloc failed in mm_realloc\n");
    exit(1);
    }
    copySize = GET_SIZE(HDRP(ptr));
    if (size < copySize)
      copySize = size;
    memcpy(newp, ptr, copySize);
    mm_free(ptr);
    return newp;
}
```

函数的思想很简单，就是新分配一个新的块，然后将原来的块 free 掉。

1. 首先分配一个大小为新大小 size 的块，如果分配失败就打印信息。
2. 如果分配成功，就将原来的块中的信息全部复制到新的块中。
3. 将原来的块 free 掉。
4. 返回新的块的指针

要点分析：

复制内存内容时需要比较原块大小和新块大小，防止超出内存范围影响后边的块。

3.2.4 int mm_checkheap(void)函数（5 分）

函数功能：检查堆是否一致。

处理流程：

1. 定义指针 bp 指向块的全局变量 heap_listp。检查序言块，如果不是 8 字节的已分配块，打印错误信息。

```
if ((GET_SIZE(HDRP(heap_listp)) != DSIZE) || !GET_ALLOC(HDRP(heap_listp)))
printf("Bad prologue header\n");
```

2. Checkblock 函数：

```
static void checkblock(void *bp)
{
    if ((size_t)bp % 8)
    printf("Error: %p is not doubleword aligned\n", bp);
    if (GET(HDRP(bp)) != GET(FTRP(bp)))
    printf("Error: header does not match footer\n");
}
```

此函数判断是否双字对齐，并且通过托多 bp 所指的块的头部和脚部指针，判断二者是否匹配，如果不匹配，打印错误信息。

3. 检查所有 size 大于 0 的块。如果 verbose 为 0，执行 printblock：

```
static void printblock(void *bp)
{
    size_t hsize, halloc, fsize, falloc;

    hsize = GET_SIZE(HDRP(bp));
    halloc = GET_ALLOC(HDRP(bp));
    fsize = GET_SIZE(FTRP(bp));
    falloc = GET_ALLOC(FTRP(bp));

    if (hsize == 0) {
    printf("%p: EOL\n", bp);
    return;
    }

    printf("%p: header: [%d:%c] footer: [%d:%c]\n", bp,
       hsize, (halloc ? 'a' : 'f'),
       fsize, (falloc ? 'a' : 'f'));
}
```

此函数先从 bp 所指的头部和脚部获得返回的大小和已分配位，然后打印信息。如果头部返回的大小为 0，则打印“eol”；如果不为 0，就打印头部和脚部的信息。

4. 检查结尾块，如果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 函数检查堆的序言块和结尾块，以及此外每个 size 大于 0 的块是否双字对齐，头脚是否 match，打印相关信息。

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：块大小 size

处理流程：

```
/* $begin mmmalloc */
void *mm_malloc(size_t size)
{
    size_t asize;      /* adjusted block size */
    size_t extendsize; /* amount to extend heap if no fit */
    char *bp;

    /* Ignore spurious requests */
    if (size <= 0)
    return NULL;

    /* Adjust block size to include overhead and alignment reqs. */
    if (size <= DSIZE)
    asize = DSIZE + OVERHEAD;
    else
    asize = DSIZE * ((size + (OVERHEAD) + (DSIZE-1)) / DSIZE);

    /* Search the free list for a fit */
    if ((bp = find_fit(asize)) != NULL) {
    place(bp, asize);
    return bp;
    }

    /* No fit found. Get more memory and place the block */
    extendsize = MAX(asize,CHUNKSIZE);
    if ((bp = extend_heap(extendsize/WSIZE)) == NULL)
    return NULL;
    place(bp, asize);
    return bp;
}
```

1. 检查非法大小。
2. 分配器调整请求块的大小，从而为头和脚预留空间，且需满足双字对齐的要求。此处强制了最小块大小是 16 字节；8 字节用来满足对齐要求，8 字节放头脚。多于 8 字节的请求，向上舍入到最近的 8 的整数倍。
3. 确定大小后，开始搜索空闲链表。如果有合适的空闲块，就将其放置在对应位置；如果没有，就先扩展堆，然后再在这个新的块中分割处理，在返回一个指针指向这个新的块。

要点分析：

1. 注意头脚空间的分配和对齐要求。
2. 分割函数被整合到 place 中。
3. 搜索方式在 find_fit 中调整。

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

函数功能：边界标记合并。
处理流程：指向块的首地址的指针 bp
1.

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

获得前一块和后一块的已分配位，并获得 bp 所指的块的大小。

2. 分四种情况：

a) 前后的块都不空闲。此时不需要合并，直接返回当前的块。

```
if(prev_alloc && next_alloc)
```

（if 中无内容）

b) 前边的块不空闲，后边的块空闲。先把当前块和后边的块从分离空闲链表中删除，然后将两个块合并，更新的内容为头脚内容，具体操作：

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

c) 前边的块空闲，后边的块不空闲。这种情况与上一种情况相同，先删除前边的块和当前块，然后将两个块合并，更新的内容为头脚内容，具体操作：

```
else if(!prev_alloc && next_alloc){
    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);
```

此处 bp 指向前边的块。

d) 前后的块都空闲：将前边块，后边块，当前块都删除，然后合并所有的三个块形成一个单独的空闲块，用三个块的大小之和更新头脚，具体操作如下：

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

3. 将之前的操作得到的新块插入分离空闲链表
4. 此处可以做一个优化，将适配方式改为下一次适配。在宏定义中修改 NEXT_FIT 的值，让搜索从上一次结束的地方开始。实验证明，效果有明显提升：

```
/*
 * If NEXT_FIT defined use next fit search, else use first fit search
 */
#define NEXT_FIT 1
```

```
    }
    else if(!prev_alloc && !next_alloc){
        size += GET_SIZE(HDRP(PREV_BLKP(bp))) + GET_SIZE(FTRP(NEXT_BLKP(bp)
        PUT(HDRP(PREV_BLKP(bp)),PACK(size,0));
        PUT(FTRP(NEXT_BLKP(bp)),PACK(size,0));
        bp = PREV_BLKP(bp);
    }
    #ifdef NEXT_FIT
            rover = bp;
    #endif
    return bp;
}
```

要点分析：
注意四种情况分别处理，原理基本相同。头脚为最前边的块的头，最后边的块的脚，大小为三个块的和。通过对 NEXT_FIT 的宏定义可以更改适配方式。

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

获得测试总分 linux>./mdriver -av -t traces/

4.2 测试结果评价

直接用了书上的函数，又将适配方式改为下一次适配。如果只运行书上函数，结果大约为 59，优化后结果有显著提升。但 trace9 和 trace10 的结果很不理想。

4.3 自测试结果

```
oclab-handout$ ./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   86%    5694  0.001900  2997
 1       yes   89%    5848  0.001147  5097
 2       yes   92%    6648  0.003144  2114
 3       yes   95%    5380  0.003172  1696
 4       yes   66%   14400  0.000108133210
 5       yes   87%    4800  0.003943  1217
 6       yes   84%    4800  0.004402  1090
 7       yes   55%   12000  0.014933   804
 8       yes   51%   24000  0.007660  3133
 9       yes   26%   14401  0.061598   234
10       yes   30%   14401  0.002235  6444
Total          69%  112372  0.104242  1078

Perf index = 42 (util) + 40 (thru) = 82/100
```

第 5 章 总结

5.1 请总结本次实验的收获

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.