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 1190201308

班　　级 1903006

学 生 姓 名 陈东鑫

指 导 教 师 史先俊

实 验 地 点 G709

实 验 日 期 6 月 9 日

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；1.60GHz；8G RAM；256GHD Disk。

1.2.2 软件环境

Windows10 64 位。

1.2.3 开发工具

VM VirtualBox 6.1；Ubuntu 20.04 LTS 64 位；
Visual Studio 2019 64 位；CodeBlocks 17.12 64 位；vi/vim/gedit+gcc；
cpu-z；Gprof；Valgrind。

1.3 实验预习

见第二章

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（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，内核维护着一个变量 brk(读做“break”），它指向堆的顶部。

分配器将堆视为一组不同大小的块（clock）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。

2.2 带边界标签的隐式空闲链表分配器原理（5 分）

隐式空闲链表简单形式（带边界标签在下面）

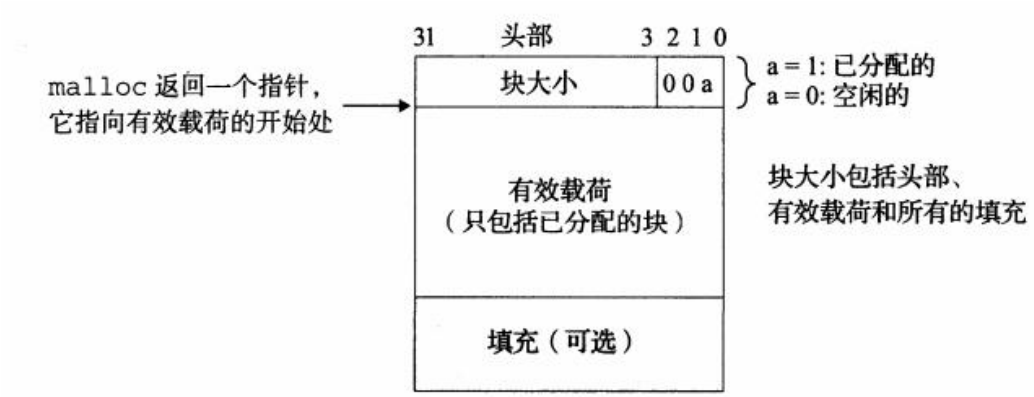


图 9-35　一个简单的堆块的格式

这种情况下，一个块是由一个字的头部、有效载荷，以及可能的填充组成。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。

头部后面是应用 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。

块的格式如图所示，我们可以将堆组织为一个连续的已分配块和空闲块的序列。

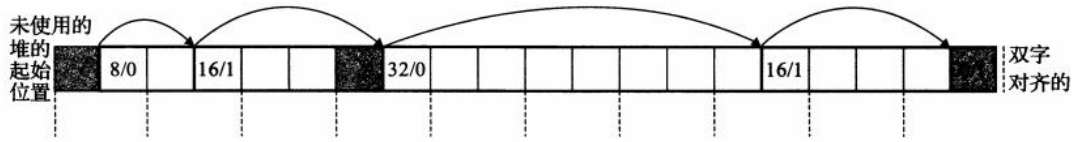


图 9-36　用隐式空闲链表来组织堆。阴影部分是已分配块。没有阴影的部分是空闲块。头部标记为(大小(字节)/已分配位)

我们称这种结构为隐式空闲链表，是因为空闲块是通过头部中的大小字段隐含地连接着的。

2.2.1 放置已分配的块

当一个应用请求一个 k 字节的块时，分配器搜索空闲链表，查找一个足够大可以放置所请求的空闲块。分配器执行这种搜索方式是由放置策略确定的，一些常见策略是首次适配、下一次适配和最佳适配。

2.2.2 分割空闲块

一旦分配器找到一个匹配的空闲块，就必须做一个另策决定，那就是分配这个块多少空间。

分配器通常会选择将这个空闲块分割为两部分。第一部分变为了已分配块，而剩下的变成一个新的空闲块。

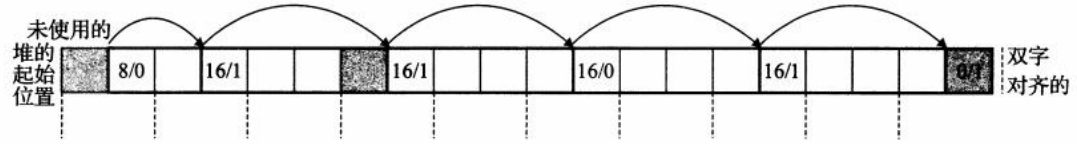


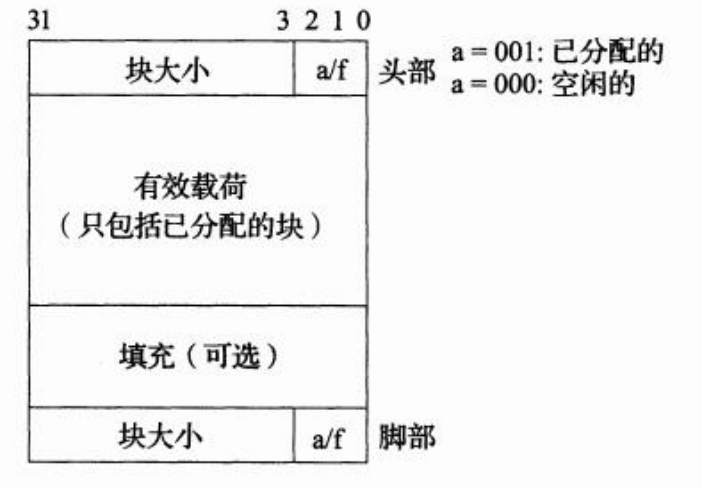
图 9-37　分割一个空闲块，以满足一个 3 个字的分配请求。阴影部分是已分配块。没有阴影的部分是空闲块。头部标记为(大小(字节)/已分配位)

2.2.3 获取额外的堆内存

如果分配器不能为请求块找到空闲块，一个选择是合并那些在物理内存上相邻的空闲块来创建一个更大的空闲块。然而如果这样还是不能生成一个足够大的块，或者空闲块已经最大程度地合并了，分配器会通过调用 sbrk 函数，向内核请求额外的内存。分配器将额外的内存转化成一个大的空闲块，将这个块插入到空闲链表中，然后将被请求的块放置在这个新的空闲块中。

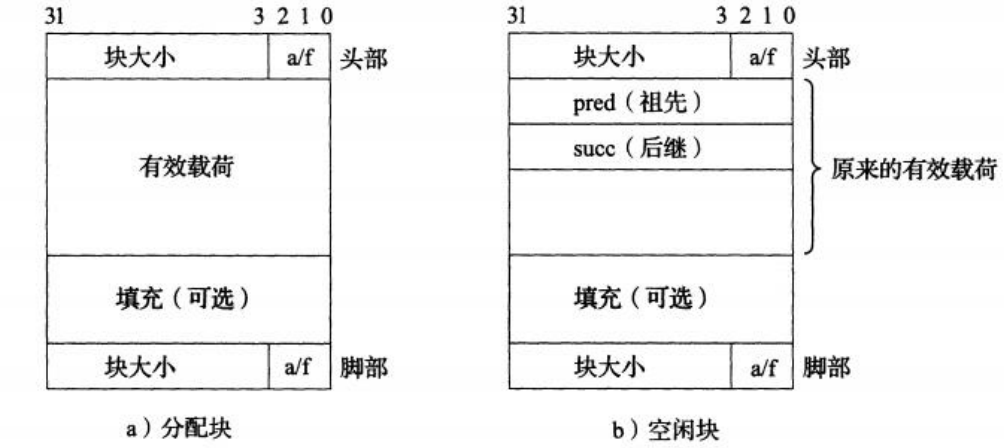
2.2.4 合并空闲块

Knuth 提出了一种聪明而通用的技术，叫做边界标记(boundary tag)，在每个块的结尾处添加一个脚部（footer，边界标记），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）

一种更好的方法是将空闲块组织为某种形式的显式数据结构。因为根据定义，程序不需要一个空闲块的主体，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，堆可以组织成一个双向空闲链表，在每个空闲块中，都包含一个 pred(前驱）和 succ（后继）指针。



2.4 红黑树的结构、查找、更新算法（5 分）

2.4.1 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倍。具体来说，红黑树是满足如下条件的二叉查找树（binary search tree）：

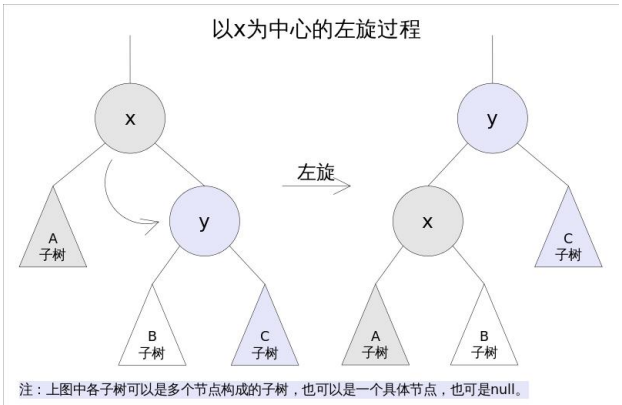
1. 每个节点要么是红色，要么是黑色。
2. 根节点必须是黑色。
3. 红色节点不能连续（也即是，红色节点的孩子和父亲都不能是红色）。
4. 对于每个节点，从该点至 null（树尾端）的任何路径，都含有相同个数的黑色节点。

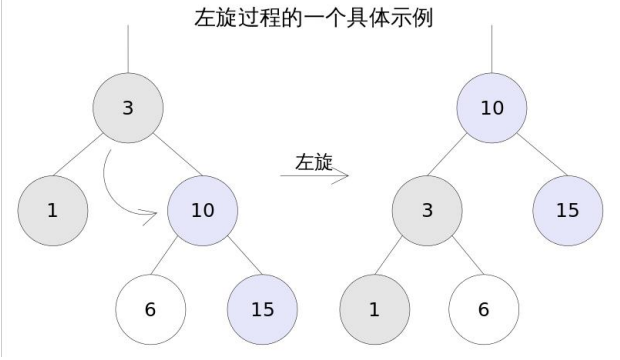
在树的结构发生改变时（插入或者删除操作），往往会破坏上述条件 3 或条件 4，需要通过调整使得查找树重新满足红黑树的条件。

红黑树的条件被破坏时，需要通过调整使得查找树重新满足红黑树的条件。调整可以分为两类：一类是颜色调整，即改变某个节点的颜色；另一类是结构调整，集改变检索树的结构关系。结构调整过程包含两个基本操作：左旋（Rotate Left），右旋（RotateRight）。

（1）左旋

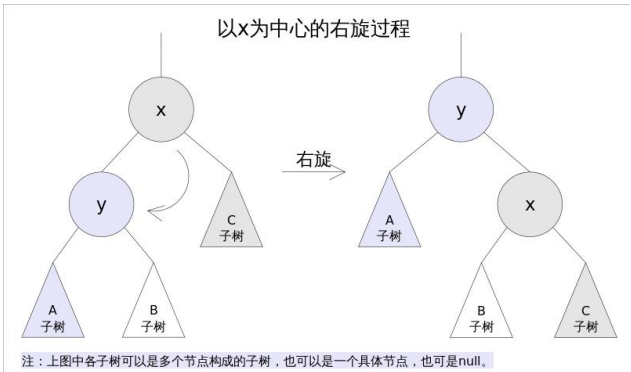
左旋的过程是将 x 的右子树绕 x 逆时针旋转，使得 x 的右子树成为 x 的父亲，同时修改相关节点的引用。旋转之后，二叉查找树的属性仍然满足。

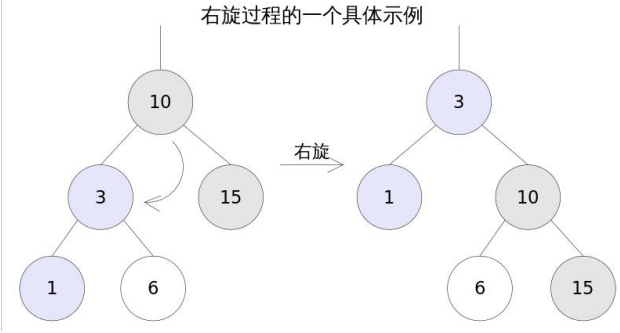




（2）右旋

右旋的过程是将 x 的左子树绕 x 顺时针旋转，使得 x 的左子树成为 x 的父亲，同时修改相关节点的引用。旋转之后，二叉查找树的属性仍然满足。





2.4.2 查找算法

类似二分查找，在通过左右旋维护好红黑树的基本结构后，红黑树为相对平衡的二叉查找树，通过二分查找可以使复杂度最坏为 O(logn)。

2.4.3 更新算法

在我们分离链表中应用则更新为删除和插入：都以查找算法为基础，删除或插入后通过左右旋转保持红黑树平衡。

第 3 章 分配器的设计与实现

总分 50 分

3.1 总体设计（10 分）

总体上使用显示双向空闲链表、基于边界标签的空闲块合并、首次适配放置策略的分配器。

3.1.1 堆结构

堆结构类似书上给的隐式空闲链表堆结构，我将原结构中填充字（padding）改为指向显示空闲链表的第一个空闲块的指针(root)。其余不变。

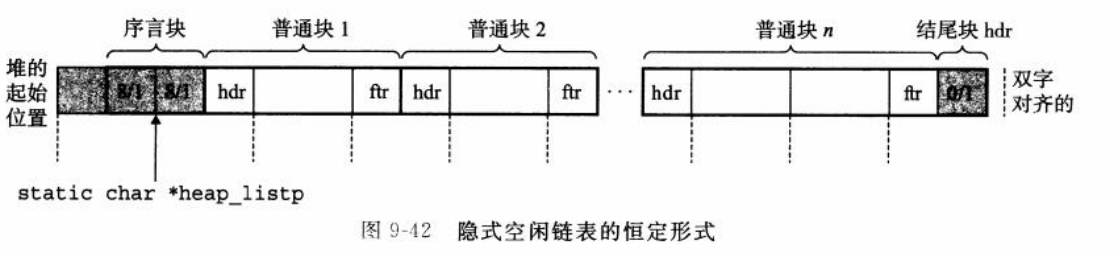
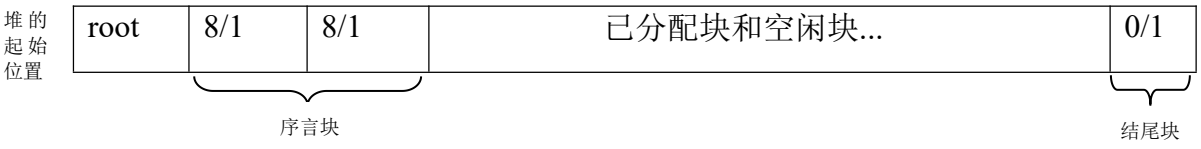


图 9-42 隐式空闲链表的恒定形式



3.1.2 内存块结构

内存块包括空闲块和已分配块，按照书上给的带边界标记的双向空闲链表结构。已分配块的开始和结尾处有头部和脚部；空闲块除了头部和脚部外，在有效载荷的前一个字和第二个字存放祖先和后继指针，分别指向空闲链表中的前驱和后继空闲块。由于实验在 32 位环境下进行，指针和地址为 4 字节，虽然空闲块内比已分配块多出两个必须使用的 8 字节，但块大小仍然至少为 16 字节。

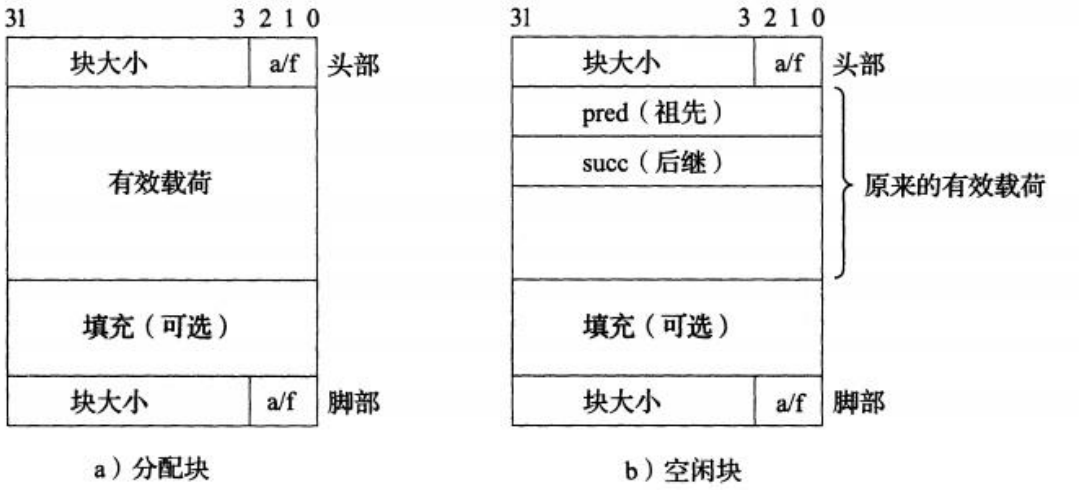


图 9-48　使用双向空闲链表的堆块的格式

3.1.3 首次适配

由于使用显示双向空闲链表，我们可以使用指针操作遍历整个空闲链表进行寻找合适的空闲块，选择第一个适配的空闲块。

3.1.4 辅助操作函数

（1）fbcheckout：将指定空闲块脱离空闲链表。脱离后要保持原链表中脱离块的前后关系，同时要考虑到脱离后链表为空的情况，共需要考虑四种情况，即对链表中前后邻接空闲块是否存在的四种情况分别进行处理。

（2）fbcheckin：将指定空闲块插入到空闲链表头部。插入时需要考虑链表是否为空。

（3）extend_heap：扩展堆，向内核请求额外的指定大小内存，其中大小需要进行对齐然后调用 mem_sbrk 进行申请。分配器将额外的内存转化成一个大的空闲块，将这个块插入到空闲链表中。

（4）place：使用指定空闲块分配指定大小的空闲块，若剩余大小不足块最小大小 16 字节，则改空闲块全部分配，将该块调用 fbcheckout 脱离空闲链表；若剩余大小大于 16 字节，则进行分割，需要分四种情况调整空闲块剩余部分与原块在空闲链表中的临近块之间的前后关系。

（5）find_fit：首次适配寻找合适大小的空闲块，若找到，则返回空闲块地址；若没找到（链表为空或链表中空闲块大小都不够），则返回 NULL。

（6）printblock：打印块包含的信息，若不是结尾快，则打印块地址、头部和脚部的扩大小、是否为空闲块的标记位；若是结尾快，则只打印块地址。

（7）checkblock：检查块是否字节对齐，并检查块的头部和脚部包含的信息是否匹配。

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。

处理流程：从内存系统得到四个字，每个字分别按照 3.1.1 中的堆结构安排为指向空闲链表第一个块指针 root，两个序言快，和一个结尾快。其中由于初始 时空闲链表为空，root 被设置指向 NULL；两个空闲块被设置为 大小为 8 的已分配块的头部和脚部；结尾快为大小为 0 的已分配块，只有一个头部。结构初始化完毕后拓展初始堆。拓展失败返回-1，拓展成功返回 0。

要点分析：由于实验要求不能定义全局或静态的复合型数据类型，于是任何形式的显示空闲链表都要在堆中进行定义，而不能设置全局变量。在堆中定义时，要保证 8 字对齐，并且由于实验在 32 位环境下进行，指针和地址为 4 字节，即一个字。

3.2.2 void mm_free(void *ptr)函数（5 分）

函数功能：释放一个已分配块，并在释放后与邻接的空闲块进行合并。

参　　数：需要释放的已分配块的有效载荷首地址。

处理流程：首先将这个已分配块的头部和脚部的分配标记设为 0，这个已分配块变为伪空闲块（祖先和后继指针没有设置，需要在合并后一并设置），然后调用函数 coalesce 与邻接的空闲块进行合并。

要点分析：函数重点在 coalesce 函数，函数内前将已分配块的头部和脚部内的标记设置为 0，是为了与扩展堆之后进行空闲块合并的情况统一。

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

函数功能：对已分配块进行重新分配大小。

参　　数：需要重新分配大小的已分配块的有效载荷首地址 ptr 与更改后的块大小 size 字节。

处理流程：重新申请一个 size 字节大小的块，并将原数据复制过去，释放原来的块的内存。若申请失败，则退出程序；若申请成功则返回重新申请的块地址。

要点分析：不要更改原已分配块中有效载荷中的内容。

3.2.4 int mm_check(void)函数（5 分）

函数功能：检查堆的一致性，若出现错误则打印提示信息。

处理流程：首先检查序言块是否按照标准设置，然后遍历所有块，查看每个块是否 8 字节对齐、头部脚部对应，最后检查结尾快是否按照标准设置。

要点分析：需要遍历所有块，而不只是空闲块。

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

函数功能：申请一个 size 字节大小的块。

参　　数：要申请的块大小 size。

处理流程：首先调用 find_fit 在空闲链表中通过首次适配寻找大小足够的空闲块，然后调用函数 place，分割出多余部分；若没有找到大小合适的空闲块，则直接扩展堆，然后分割扩展堆新分配的空闲块。返回分配的块的地址。

要点分析：要通过申请的字节数根据对齐调整请求块大小，为头部和脚部留有空间并满足双字对齐的要求。对于小于 8 字节的请求，最小的块大小为 16 字节（包括头部和脚部）；对于大于 8 字节的请求，应向上舍入到最近的 8 的整数倍并加上头部和脚部的 8 字节。

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

函数功能：将待合并的空闲块与其邻接的空闲块合并。

参　　数：待合并的空闲块有效载荷首地址。

处理流程：有四种情况分别进行处理：（1）当待合并块上下两个临界块均为已分配块时，直接将空闲块插入空闲链表头（2）当待合并块上一邻接块为已分配块、下一邻接块为空闲块时，将下一空闲块从空闲链表脱离，与待合并块合并，再将新的空闲块插入到空闲链表头（3）当待合并块上一邻接块为空闲块、下一邻接块为已分配块时，将上一空闲块从空闲链表脱离，然后将地址转向上一块地址，与待合并块合并，再将新的空闲块插入空闲链表头（4）当待合并块上下邻接块都为空闲块时，将前后都脱离空闲链表，然后将地址转向上一块地址，将两个空闲块与待合并块合并，再将新生成的空闲块插入到空闲链表头。

要点分析：要分四种情况分别处理；由于合并时邻接的空闲块在空闲　链 表中，需要先调用 fbcheckout 函数将其脱离空闲链表，这时 fbcheckout 函数内需要考虑空闲链表为空的情况；合并时，需要更改头部和脚部，这时需要转移到合适的地址进行操作，即转移到待合并的几个块中最前面的块的有效载荷的首地址，将头部和脚部分别改为 几个待合并块的大小之和；合并后，调用 fbcheckin 函数将合并生成的新空闲块插入到空闲链表头，这时也需要考虑空闲链表为空的情况。

第 4 章测试

总分 10 分

4.1 测试方法

（1）在命令行输入 make 生成可执行评测程序文件；

（2）使用命令./mdriver -v -t traces/ 用测试驱动程序对用 traces 文件夹下所有轨迹文件进行测试；

文件包括： amptjp-bal.rep
cccp-bal.rep
cp-decl-bal.rep
expr-bal.rep
coalescing-bal.rep
random-bal.rep
random2-bal.rep
binary-bal.rep
binary2-bal.rep
realloc-bal.rep
realloc2-bal.rep

4.2 测试结果评价

realloc 使用最基本的 malloc+free，即申请新块并将原块内容复制过去，没有考虑使用附近的碎片，故对 realloc 的测试结果较差。

另外，使用显示双向空闲链表，在链表中查找块的需要进行遍历，速度不如分离链表，故花费时间降不下来，导致 thru 较低。

4.3 自测试结果

如图，测试结果得分在 77~83 之间波动。

```
cdx1190201308@cdx1190201308-VirtualBox:~/Hitics/malloclab-handout-hit$ ./mdriver -v -t traces/
Team Name:双向空闲链表&基于边界标签的空闲快合并&首次适配
Member 1 :Dave OHallaron:droh
Using default tracefiles in traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   92%    5694  0.001850  3079
 1       yes   94%    5848  0.001009  5795
 2       yes   96%    6648  0.001539  4321
 3       yes   97%    5380  0.001548  3475
 4       yes   66%   14400  0.000555 25937
 5       yes   89%    4800  0.001522  3154
 6       yes   85%    4800  0.001074  4470
 7       yes   55%   12000  0.010234  1173
 8       yes   51%   24000  0.005608  4280
 9       yes   26%   14401  0.139340   103
10       yes   34%   14401  0.003644  3952
Total          71%  112372  0.167923   669

Perf index = 43 (util) + 40 (thru) = 83/100
```

第 5 章 总结

5.1 请总结本次实验的收获

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。

5.2 请给出对本次实验内容的建议

时间安排在考试周，各学科任务太多，没时间更深入研究其他形式空闲链表和分配算法。

注：本章为酌情加分项。

参考文献

[1] yfceshi. 查找（一）史上最简单清晰的红黑树解说[EB/OL]. https://www.cnblogs.com/yfceshi/p/6930915.html，2017-06-01/2021-06-20.