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 1190202105

班　　级 1903002

学　　生 傅浩东

指 导 教 师 郑贵滨

实 验 地 点 G709

实 验 日 期 2021.06.11

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

Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz；16GB RAM；1 TB SSD

1.2.2 软件环境

Windows 10 21H1；VirtualBox；Ubuntu 20.04 LTS

1.2.3 开发工具

EDB；GDB；CodeBlocks；vi/vim/gpedit+gcc；VSCode

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

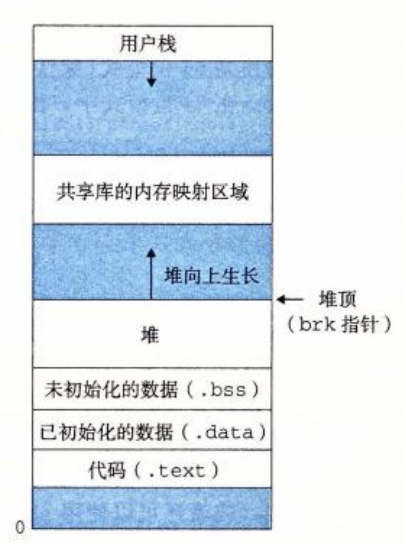
总分 20 分

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，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（explicit allocator），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++中的符与 C 中的 malloc 和 free 相当。

隐式分配器（implicit allocator），另一方面，要求分配器检测一个已分配块何

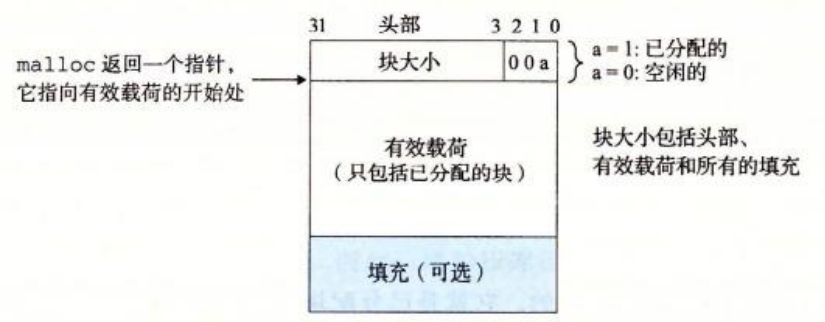
时不再被程序所使用，那么就释放这个块。隐式分配器也叫做垃圾收集器（garbage collector), 而自动释放未使用的已分配的块的过程叫做垃级收集（garbage collection）。例如，诸如 Lisp、ML 以及 Java 之类的高级语言就依赖垃圾收集来释放已分配的块。

2.2 带边界标签的隐式空闲链表分配器原理（5 分）

任何实际的分配器都需要一些数据结构，允许它来区别块边界，以及区别已分配块和空闲块。大多数分配器将这些信息嵌人块本身。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。

我们可以将堆组织为一个连续的已分配块和空闲块的序列，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。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。

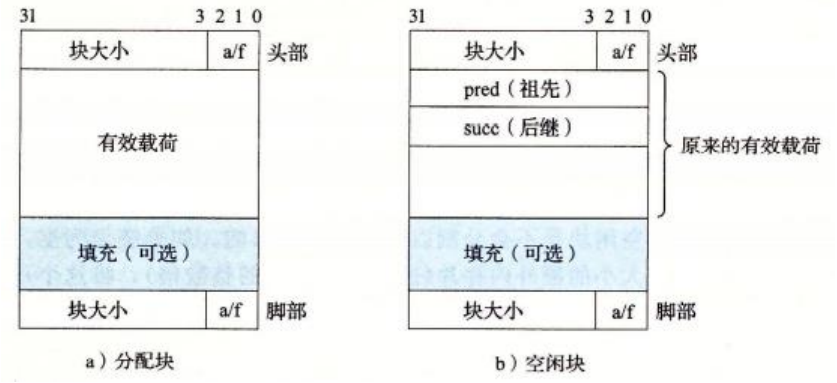


2.3 显示空间链表的基本原理（5 分）

隐式空闲链表为我们提供了一种介绍一些基本分配器概念的简单方法。然而，因为块分配与堆块的总数呈线性关系，所以对于通用的分配器，隐式空闲链表是

不适合的（尽管对于堆块数量预先就知道是很小的特殊的分配器来说它是可以的）。

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 siicc（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，这取决于我们所选择的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。使用 LIFO 的顺序和首次适配的放置策略，分配器会最先检查最近使用过的块。在这种情况下，释放一个块可以在常数时间内完成。如果使用了边界标记，那么合并也可以在常数时间内完成。

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。在这种情况下，释放一个块需要线性时间的搜索来定位合适的前驱。平衡点在于，按照地址排序的首次适配比 LIFO 排序的首次适配有更高的内存利用率，接近最佳适配的利用率。

一般而言，显式链表的缺点是空闲块必须足够大，以包含所有需要的指针，以及头部和可能的脚部。这就导致了更大的最小块大小，也潜在地提高了内部碎片的程度。

2.4 红黑树的结构、查找、更新算法（5 分）

1. 红黑树的结构

红黑树是每个节点都带有颜色属性的二叉查找树，颜色为红色或黑色。在二叉查找树强制一般要求以外，增加了如下的额外要求：

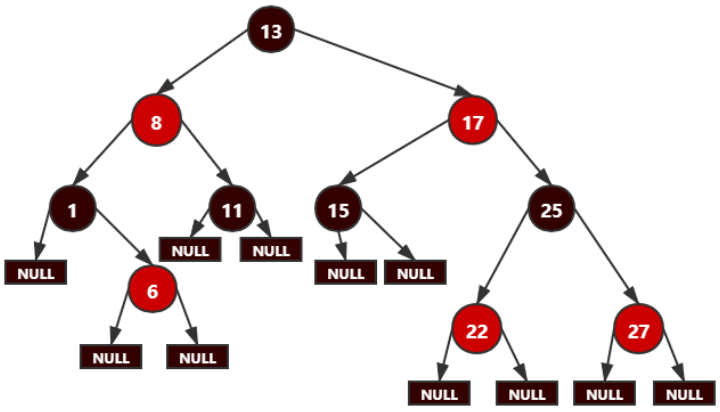
(1) 节点是红色或黑色。

(2) 根是黑色。

(3) 所有叶子都是黑色（叶子是 NULL 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。



2. 红黑树的查找

由于红黑树是一种特殊的二叉树，所以它的查找方式与二叉树是一样的，从根节点开始，大于节点的往右子树查找，小于节点的往左子树查找，直到与节点相等则查找成功，或者查到 NULL 则查找失败。

3. 红黑树的更新

只有插入和删除的时候才涉及更新，对于更改某一个值也可以当作插入和删除，所以分别对这两种算法情况进行介绍。

插入

首先以二叉查找树的方法增加节点并标记为红色，可能导致出现连续两个红色节点的冲突，可以通过颜色调换和树旋转进行调整。

(1) Case 1：新节点位于树的根上，没有父节点，把它重绘为黑色。

(2) Case 2：新节点的父节点是黑色，性质依然满足。

(3) Case 3：如果父节点和叔父节点都是红色，此时新插入节点做为的左子节点或右子节点，则我们可以将父节点和叔父节点重绘为黑色并重绘祖父节点为红色。现在我们的新节点有了一个黑色的父节点。这样就保证了路径上黑色节点数没变。但是，红色的祖父节点可能是根节点，也有可能祖父节点的父节点是红色的，因此把祖父节点当成是新加入的节点递归地进行各种情形的检查。

(4) Case 4：如果，父节点是红色而叔父节点是黑色或为 NULL，并且新节点是其父节点的右子节点而父节点又是祖父节点的左子节点。进行一次左旋转调换新节点和其父节点的位置，就将情况处理为 **Case 5，**接下来调整红黑树。

(5) Case 5：如果，父节点是红色而叔父节点是黑色或 NULL，新节点是其父节点的左子节点，而父节点又是祖父节点的左子节点。进行针对祖父节点的一次右旋转，产生的树中，以前父节点的位置现在是新节点和以前祖父节点的位置是父节点，祖父节点则在右子树开始。切换以前的父节点和祖父节点的颜色。

删除

首先，将要删除的点成为 P，它的一个儿子为 N，儿子兄弟为 S。

(1) Case 1：删除的节点是唯一节点，直接指向 NULL 就结束。

(2) Case 2：S 是红色。这种情形下在 N 的父亲上做左旋转，把红色兄弟转换成 N 的祖父，对调 N 的父亲和祖父的颜色。之后所有路径上黑色节点的数目没有改变，但现在 N 有了一个黑色的兄弟和一个红色的父亲，接下去按 **Case 4**、**Case 5** 或 **Case 6** 来处理。

(3) Case 3：N 的父亲、S 和 S 的儿子都是黑色的。重绘 S 为红色，通过 S 的所有路径，即以前不通过 N 的那些路径，都少了一个黑色节点。但删除 N 的初始的父亲使通过 N 的所有路径少了一个黑色节点，路径黑色节点数平衡。但通过 P 的所有路径现在比不通过 P 的路径少了一个黑色节点，为修正这个问题，从 **Case 1** 开始，从 P 做平衡处理。

(4) Case 4：S 和 S 的儿子都是黑色，但是 N 的父亲是红色。这种情形下，交换 P 和 S 的颜色。这不影响不通过 N 的路径的黑色节点的数目，但是它在通过 N 的路径上对黑色节点数目增加了一，添补了在这些路径上删除的黑色节点。

(5) Case 5：S 是黑色，S 的左儿子是红色，S 的右儿子是黑色，而 N 是它父亲的左儿子。在这种情形下我们在 S 上做右旋转，这样 S 的左儿子成为 S 的父亲和 N 的新兄弟。我们接着交换 S 和它的新父亲的颜色。所有路径仍有同样数目的黑色节点，但是现在 N 有了一个黑色兄弟，他的右儿子是红色的，所以我们进入了 **Case 6**。

(6) Case 6：S 是黑色，S 的右儿子是红色，而 N 是它父亲的左儿子。在这种情形下我们在 N 的父亲上做左旋转，这样 S 成为 N 的父亲（P）和 S 的右儿子的父亲。我们接着交换 N 的父亲和 S 的颜色，并使 S 的右儿子为

黑色。N 现在增加了一个黑色祖先：要么 N 的父亲变成黑色，要么它是黑色而 S 被增加为一个黑色祖父。所以，通过 N 的路径都增加了一个黑色节点。

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。

1. 介绍堆

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（向更高的地址）。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（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

用隐式空闲链表来组织堆，具体组织的算法在 mm_init 函数中。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。

3. 采用的空闲块、分配块链表：采用分离的空闲链表。

4. 相应算法如下介绍。

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：初始化内存模型

处理流程：

1. 初始化分离空闲链表。根据申请的链表数组，将分离空闲链表初始化为 NULL。
2. 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。
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：

(1) 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。
(2) 闲链表创建之后需要使用 extend_heap 函数来扩展堆。

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
参　　数：void *ptr 指向块的指针
处理流程：

1. GET_SIZE(HDRP(bp))获得请求释放块的大小。
2. 并且使用 PUT(HDRP(bp)/ FTRP(bp), PACK(size, 0))将请求块的头部和脚部的已分配位置为 0。
3. coalesce(bp)将释放的块 bp 与相邻的空闲块合并。

要点分析：
(1) free 块需要和相邻的空闲块合并。

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

函数功能：将 ptr 指向的内存块（旧块）大小变为 size，并返回新内存块地址
参　　数：void *ptr 指向块的指针, size_t size 新字节大小
处理流程：

1. 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。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。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. GET_SIZE(HDRP(ptr))得到 ptr 块大小，如果 size 比 copy_size 小更新 copy_size 为 size，接着释放 ptr。

要点分析：
(1) 返回的地址与原地址可能相同，也可能不同，这依赖于算法的实现、旧块内部碎片大小、参数 size 的数值。
(2) 新内存块中，前 min(旧块 size，新块 size)个字节的内容与旧块相同，其他字节未做初始化。

3.2.4 int mm_check(void)函数（5 分）

函数功能：堆的一致性检测
处理流程：

1. 先定义指针 bp，初始化为指向序言块的全局变量 heap_listp。后面的操作大多数都是在 verbose 不为零时执行的。最初是检查序言块，若序言块不是 8 字节的已分配块，则打印 Bad prologue header。
2. checkblock 函数主要功能是检查是否双字对齐，并且通过获得 bp 所指块的头头部和脚部指针，判断二者是否匹配，不匹配则返回错误信息。检查所有 size 大于 0 的块，如果 verbose 不是 0，则执行 printblock 函数。

3. printblock 函数先获得从 bp 所指的块的头部和脚部分别返回的大小和已分配位，然后打印信息。
4. 检查结尾块。结尾块不是一大小为 0 的已分配块，则打印 Bad epilogue header。

要点分析：

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

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_t size 请求块大小为 size
处理流程：

1. 在检查完请求的真假之后，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。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，然后向上舍人到最接近的 8 的整数倍。
2. 一旦分配器调整了请求的大小，它就会搜索空闲链表，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，并可选地分割出多余的部分，然后返回新分配块的地址。
3. 如果分配器不能够发现一个匹配的块，那么就用一个新的空闲块来扩展堆，把请求块放置在这个新的空闲块里，可选地分割这个块，然后返回一个指针，指向这个新分配的块。

要点分析：

(1) 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：边界标记合并，将邻接的空闲块合并，返回合并块地址。
处理流程：

1. 获得前面块和后面块的已分配位，和 bp 指向块的大小。
2. 根据前后相邻四种情况进行合并
   (1) 前面的块和后面的块都是已分配的，不合并。
   (2) 前面的块是已分配的，后面的块是空闲的，合并当前和后面的块。
   (3) 前面的块是空闲的，而后面的块是已分配的，合并当前和前面的块。
   (4) 前面的和后面的块都是空闲的，合并三块。
3. 更新 bp 所指向的块插入分离空闲链表。

要点分析：

(1) 注意处理四种不同相邻情况。
(2) 忽略潜在的边界情况。

第 4 章测试

总分 10 分

4.1 测试方法与测试结果（3 分）

清除 make 信息：make clean
链接、编译生成可执行文件：make
评测方法:./mdriver -t traces/ -v

```
lfu@lfu-VirtualBox:malloclab-handout-hit$ ./mdriver -t traces/ -v
Team Name:1190202105
Member 1 :Fu Haodong:1190202105@qq.com
Using default tracefiles in traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.006484   878
 1       yes   99%    5848  0.005544  1055
 2       yes   99%    6648  0.009177   724
 3       yes  100%    5380  0.006998   769
 4       yes   66%   14400  0.000131109924
 5       yes   92%    4800  0.005973   804
 6       yes   92%    4800  0.005848   821
 7       yes   55%   12000  0.134024    90
 8       yes   51%   24000  0.245769    98
 9       yes   27%   14401  0.054912   262
10       yes   34%   14401  0.002060  6992
Total          74%  112372  0.476919   236

Perf index = 44 (util) + 16 (thru) = 60/100
```

4.2 测试结果分析与评价（3 分）

测试结果符合预期但不理想。
对于 trace 0~3 以及 5、6，util 结果还算较好，整体的 secs 结果都不算太优。

4.3 性能瓶颈与改进方法分析（4 分）

隐式空闲链表提供了基本分配器的简单方法，但是因为块分配与堆块的总数呈线性关系，所以对于通用的分配器，隐式空闲链表是不适合的，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。一个使用单向空闲块链表的分配器需要与空闲块数量呈线性关系的时间来分配块，所以分配越多、空闲块越少、所消耗时间越长。等等原因导致结果较差。

基于上述分析可以采用显示分离链表、维护大小递增、首次适配算法等来优化。

第 5 章 总结

5.1 请总结本次实验的收获

理解了动态存储申请、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

5.2 请给出对本次实验内容的建议

暂无

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.