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 计算机系

学　　号 1190201421

班　　级 1936603

学　　生 张瑞

指 导 教 师 刘宏伟

实 验 地 点 G709

实 验 日 期 2021 年 6 月 10 日

计算机科学与技术学院

目 录

第 1 章 实验基本信息 - 3 -

1.1 实验目的 - 3 -
1.2 实验环境与工具 - 3 -
1.2.1 硬件环境 - 3 -
1.2.2 软件环境 - 3 -
1.2.3 开发工具 - 3 -
1.3 实验预习 - 3 -

第 2 章 实验预习 - 4 -

2.1 动态内存分配器的基本原理 - 4 -
2.2 带边界标签的隐式空闲链表分配器原理 - 4 -
2.3 显式空间链表的基本原理 - 6 -
2.4 红黑树的结构、查找、更新算法 - 6 -

第 3 章 分配器的设计与实现 - 8 -

3.1 总体设计 - 8 -
3.2 关键函数设计 - 9 -
3.2.1 int mm_init(void)函数 - 9 -
3.2.2 void mm_free(void *ptr)函数 - 9 -
3.2.3 void *mm_realloc(void *ptr, size_t size)函数 - 9 -
3.2.4 int mm_check(void)函数 - 10 -
3.2.5 void *mm_malloc(size_t size)函数 - 10 -
3.2.6 static void *coalesce(void *bp)函数 - 11 -

第 4 章测试 - 12 -

4.1 测试方法与测试结果 - 12 -
4.2 测试结果分析与评价 - 12 -
4.3 性能瓶颈与改进方法分析 - 13 -

第 5 章 总结 - 14 -

5.1 请总结本次实验的收获 - 14 -
5.2 请给出对本次实验内容的建议 - 14 -

参考文献 - 15 -

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

CodeBlocks；Valgrind 等

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

2.1 动态内存分配器的基本原理

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

一个块是由一个字的头部、有效载荷，以及可能的一些额外的填充组成的。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。

我们称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 提出了边界标记的技术，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，判断前面的一个块的起始位置和状态，这个脚部总是在据当前块开始位置一个字的距离。

考虑当分配器释放当前块时可能存在的所有情况：

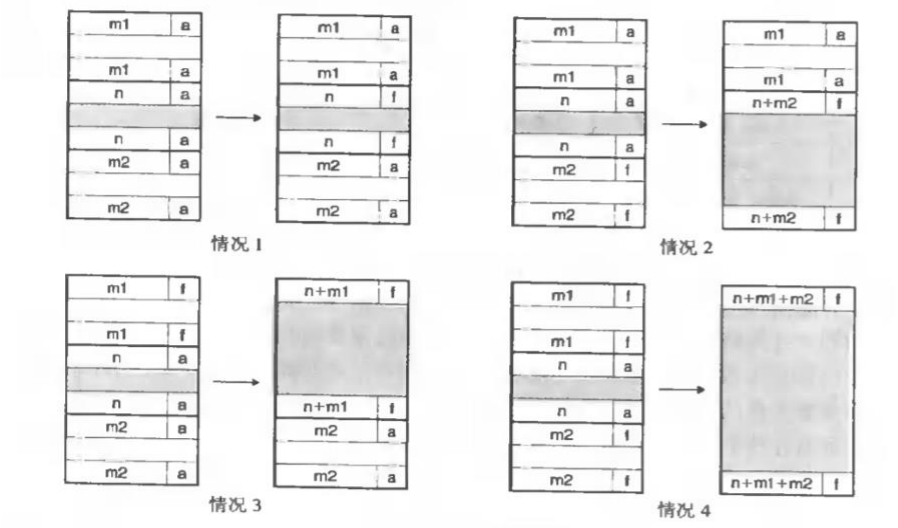
1）前面的块和后面的块都是已分配的。

2）前面的块是已分配的，后面的块是空闲的。

3）前面的块是空闲的，而后面的块是已分配的。

4）前面的和后面的块都是空闲的。

按照下图分类处理即可：

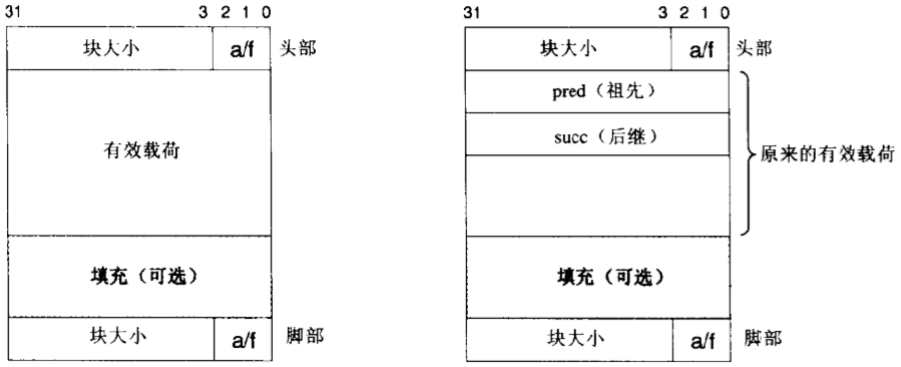


然而这种方法也存在一个潜在的缺陷。它要求每个块都保持一个头部和一个脚部，在应用程序操作许多个小块时，会产生显著的内存开销。

幸运的是，有一种非常聪明的边界标记的优化方法，能够使得在已分配块中不再需要脚部。把前面块的已分配位/空闲位存放在当前块中多出来的低位中，那么已分配的块就不需要脚部了，这样我们就可以将这个多出来的空间用作有效载荷了。不过空闲块仍然需要脚部。

2.3 显式空间链表的基本原理

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。例如，堆可以组织成一个双向空闲链表。在每个空闲块中，都包含一个前驱和后继指针。



（a）分配块　　（b）空闲块

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的线性时间。不过，释放一个块的时间也可以是线性的，也可能是某个常数，这取决于我们所选择的空闲链表中块的排序策略。

一种方法是使用后进先出（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。

另一种方式是按照地址顺序来维护链表，其中链表上每一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

红黑树的结构：

红黑树，一种二叉查找树，但在每个节点上增加一个存储位表示节点的颜色，可以是 Red 或 Black。通过对任何一条从根到叶子的路径上的各个节点着色方式

的限制，红黑树确保没有一条路径会比其他路径长出两倍，是接近平衡的。其结构有五个特点：

1.每个节点要么是红的要么是黑的。

2.根节点是黑的。

3.每个叶节点都是黑的（叶子是 NULL 节点）。

4.如果一个节点是红的，那么他的两个子节点都是黑的。

5.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。

红黑树的查找：（同一般二叉搜索树）

1.从根节点开始查找，如果树为空，就返回 NULL。

2.如果树不空，就让数据 X 和节点的数据 Data 作比较。

3.如果 X 的值大于节点的 Data，就往右子树中进行搜索；如果 X 的值小于节点的 Data，就往左子树中搜索。

4.如果 X 值等于 Data，就表示查找完成，返回该节点。

红黑树的更新：（旋转、涂色处理较为复杂，此处略去具体步骤）

对于插入情况，先按查找算法定位插入位置，将节点涂成红色插入：

1.若原树为空树，插入的是根节点，把节点改涂成黑色。

2.若插入的节点的父节点是黑色，红黑树没有被破坏，不用再调整。

3.若原树非空且父节点为红色，则需要根据具体情况进行调整（旋转、重新涂色）。

对于删除情况：

1.若删除节点左右子节点均为 NULL，不影响红黑树性质，无需调整。

2.若删除节点有一个子节点为 NULL，则让非 NULL 节点代替其位置。若出现两个红色节点相邻的情况，则根据具体情况进行调整（旋转、重新涂色）。

3.若删除节点有左右两个儿子，则需要从右子树中找到最小的节点，将其赋值到这个位置上，并将右子树中最小的节点删掉。若出现两个红色节点相邻，则根据具体情况进行调整（旋转、重新涂色）。

第 3 章 分配器的设计与实现

3.1 总体设计

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：

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。

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，以及可能的一些额外的填充组成的。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。

采用的空闲块、分配块链表结构：

基于隐式空闲链表，最大的块大小为 2^{32} =4GB。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。

相应算法：

使用立即边界标记合并方式，每个块有头部和脚部进行标记，一旦被释放立

即执行与相邻块的合并。放置策略实现了首次适配和下一次适配，具体采用哪一个取决于对 NEXT_FIT 的设置与否。

3.2 关键函数设计

3.2.1 int mm_init(void)函数

函数功能：创建带一个初始空闲块的堆

处理流程：

1.从内存系统得到四个字。

2.将它们初始化，生成一个序言块和一个结尾块，序言块包含一个头部和一个脚部，结尾块一会随着加入块会被替换掉。

3.将 heap_listp 指针指向序言块中间，（设置下一次适配所需的标记 rover）,调用 extend_heap 函数将堆扩展 CHUNKSIZE 字节，并且创建初始的空闲块。

要点分析：

1.理解好隐式空闲链表的恒定形式，明确分配器使用最小块的大小为 16 字节。

2.为了保持对齐，extend_heap 将请求大小向上舍入为最接近 2 字（8 字节）的倍数，然后向内存系统请求额外的堆空间。

3.2.2 void mm_free(void *ptr)函数

函数功能：释放一个块

参　　数：指向需要释放的块的首个指针 ptr

处理流程：

1.调用 GET_SIZE 函数来获得块的大小。

2.调用 PUT 将块的头部和脚部的已分配位设为 0，表示已 free。

3.调用 coalesce 将释放的块与相邻的空闲块合并。

要点分析：

将块标记为已 free 之后，还要将它加到空闲链表中，注意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

函数功能：将指定块的大小重新进行分配

参　　数：指向指定块的指针 ptr，需重新设定的块大小 size

处理流程：

1.调用 mm_malloc 申请一个 size 大小的内存块，如果失败则直接退出。

2.获取原有块的大小，将其和申请的改变大小 size 进行比较，取小值。

3.按得到的小值，使用 memcpy 函数复制原有内存块内容到新分配内存块中。

4.释放原有内存块，并将 ptr 指向的新分配的内存块。

要点分析：

重新分配大小会有两种情况，一个是改变后的大小大于等于原大小，这时 mm_malloc 之后直接把内容复制过来即可；另一个是改变后的大小小于原有大小，这时就只在原内存处截取改变后大小的部分即可。所以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 copySize 的值。

3.2.4 int mm_check(void)函数

（无上述函数，猜测应该是在说 void mm_checkheap(int verbose)这个函数吧……）

函数功能：检查隐式空闲链表所实现的堆的一致性

处理流程：

1.检查序言块是否正确，其大小需要是 DSIZE（8 字节），分配情况应该是已分配（1），否则输出提示信息。然后还要检查序言块是否为 8 字节对齐，头部和脚部是否匹配。

2.从堆的开始处开始，如果 verbose 不为 0，就逐个执行 printblock，再检查（checkblock）块是否为 8 字节对齐，头部和脚部是否匹配。

3. 如果 verbose 不为 0，就执行 printblock，再检查结尾块是否正确，其大小需要是 0，分配情况应该是已分配（1），否则输出提示信息。

要点分析：

checkheap 检查了堆所有的块，包括序言块、普通块和结尾块，所有 size 大于 0 的块都需要检查是否双字对齐和头部脚部匹配，并且打印块的信息。

3.2.5 void *mm_malloc(size_t size)函数

函数功能：从空闲列表分配一个块

参　　数：所需要分配的块大小 size

处理流程：

1.处理申请大小：若为 0，则直接返回 NULL；若小于等于 DSIZE（8 字节），则将 size 调整成 DSIZE+OVERHEAD（头部和尾部就需要一个 OVERHEAD）;若大于 8 字节，需加上头部和尾部的开销，然后向上舍入到最接近的 8 的倍数。

2.在空闲链表中寻找一个合适的空闲块，如果有合适的，则于该处放置这个请

求块，并且可选的分割出多余的部分（参见 place 函数），然后返回指向新分配块的指针。

3.若未搜索到合适的空闲块，则调用 extend_heap 函数扩展堆，若扩展失败，则直接返回 NULL。

4.将请求块放置在新的空闲块中，同上可选的分割这个块，然后返回指向这个新分配块的指针。

要点分析：

1.注意对齐要求，系统要求了双字对齐，最小分配块是 16 字节。

2.头部脚部占用 8 字节，还要向上舍入到最近的 8 的整数倍。

3.在隐式空闲链表里找不到合适块时，要重新向内存申请空间。

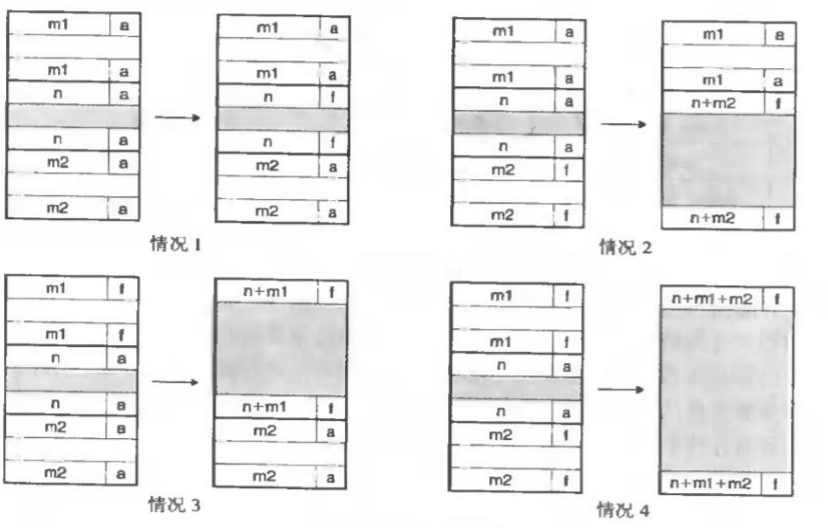
3.2.6 static void *coalesce(void *bp)函数

函数功能：将相邻的空闲块合并

处理流程：

1.获取前一个块的头部分配信息和后一个块的脚部分配信息。

2.根据下图的四种不同情况进行合并，并返回新的 bp 值：



（3.对于采用下一次适配的方案，需对标记 rover 进行检查，确保其不会指向刚合并好的空闲块的中央。）

要点分析：

1.对四种合并时的情况进行全面的考虑，分类处理。

2.不要忘了对下一次适配方案中涉及到的 rover 进行检查，因为块的合并可能使原本指向一个空闲块荷载中首字节的指针指向一个空闲块的中央，导致之后对块的查找出错。

第 4 章测试

4.1 测试方法与测试结果

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

实际是通过输入以下指令来实现的测试：

make

./mdriver -t traces -v

```
zr@ubuntu:~/shared/malloclab-handout-hit$ ./mdriver -t traces -v
Team Name:implicit first fit
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   99%    5694  0.009357   609
 1       yes   99%    5848  0.007823   748
 2       yes   99%    6648  0.013508   492
 3       yes  100%    5380  0.009857   546
 4       yes   66%   14400  0.000109131989
 5       yes   92%    4800  0.010868   442
 6       yes   92%    4800  0.008928   538
 7       yes   55%   12000  0.181579    66
 8       yes   51%   24000  0.349230    69
 9       yes   27%   14401  0.078726   183
10       yes   34%   14401  0.002785  5171
Total          74%  112372  0.672770   167

Perf index = 44 (util) + 11 (thru) = 56/100
```

4.2 测试结果分析与评价

测试时使用的是隐式空闲链表加上首次适配的放置策略，在实验中我主要是分析了几个关键的函数，并参照教材实现了 coalesce 空闲块合并函数。测试结果表

明函数都可以正常运行，但因为没有时间进行方案优化，性能评分很低，结果不算理想。

4.3 性能瓶颈与改进方法分析

遗憾的是，本次实验中未进行性能改进，但是按照 PPT 中的提示，采用显式空闲链表、基于边界标签的空闲块合并和首次适配的方式能改进性能，使用红黑树可以实现最优的性能。

第 5 章 总结

5.1 请总结本次实验的收获

在这次实验中，我对简单分配器的各个函数更加熟悉了，进一步理解了动态内存分配的原理，尤其是带边界标签的隐式空闲链表分配器原理。实现自己以为很简单的一看就会的东西，却能发现一些自己知识上的漏洞，进而及时地进行修补。

5.2 请给出对本次实验内容的建议

希望实验指导书在最后的验证这一块再详细一点，最好直接将需要用到的指令逐条列出，因为很多人对 linux 里 driver 这一类命令行上的操作都不太明白。

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.