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 计算学部

学　　号 1190200208

班　　级 1936602

学 生 姓 名 李旻翀

指 导 教 师 刘宏伟

实 验 地 点 G709

实 验 日 期 2021.6.10

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

1. 理解现代计算机系统虚拟存储的基本知识
2. 掌握 C 语言指针相关的基本操作
3. 深入理解动态存储申请、释放的基本原理和相关系统函数
4. 用 C 语言实现动态存储分配器，并进行测试分析
5. 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

Visual Studio 2010 64 位以上；CodeBlocks；vi/vim/gpedit+gcc

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称为堆的进程的虚拟内存区域。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，这种释放可以由应用程序显式执行或内存分配器自身隐式执行。

分配器有两种基本风格：显式分配器和隐式分配器。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 程序通过调用 malloc 函数来分配一个块，通过调用 free 函数来释放一个块。其中 malloc 采用的总体策略是：先系统调用 sbrk 一次，会得到一段较大的并且是连续的空间。进程把系统内核分配给自己的这段空间留着慢慢用。之后调用 malloc 时就从这段空间中分配，free 回收时就再还回来（而不是还给系统内核）。只有当这段空间全部被分配掉时还不够用时，才再次系统调用 sbrk。当然，这一次调用 sbrk 后内核分配给进程的空间和刚才的那块空间一般不会是相邻的。

隐式分配器：也叫做垃圾收集器，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）

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（把偶偶头部和所有的填充），同时标记这个块的分配情况。

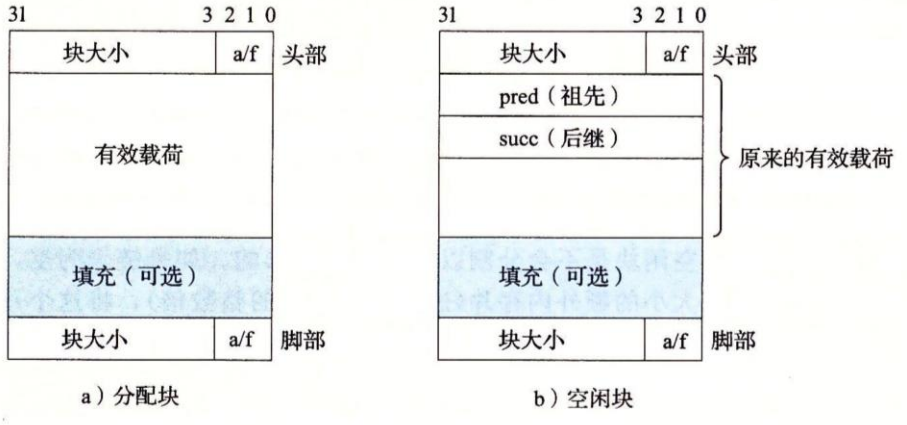
在一个字组成的头部之后是应用 malloc 时请求的有效载荷。有效载荷后面是一片不使用的填充块，其大小可以是任意的。需要填充有很多原因。比如，填充可能是分配器策略的一部分，用来对付外部碎片。或者也需要用它来满足对齐要求。

这种结构即为隐式空闲链表，其空闲块是通过头部中的大小字段隐含地连接着的。分配器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。

带边界标签的思想就是在每个块的结尾处添加一个脚部，其中脚部就是一个头部的副本。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）

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，如图：



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使线性的，也可以是一个常数，这取决于我们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，那么合并也可以在线性时间内完成。

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，显示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）

红黑树的结构：

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。具体而言，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。

在红黑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。

红黑树的查找：

红黑树是一种特殊的二叉查找树，查找方法同二叉查找树一样，不需要做更多更改。但是由于红黑树比一般的二叉查找树具有更好的平衡，所以查找起来更快。红黑树的主要是想对 2-3 查找树进行编码，尤其是对 2-3 查找树中的 3-nodes 节点添加额外的信息。红黑树中将节点之间的链接分为两种不同类型，红色链接，他用来链接两个 2-nodes 节点来表示一个 3-nodes 节点。黑色链接用来链接普通的 2-3 节点。特别的，使用红色链接的两个 2-nodes 来表示一个 3-nodes 节点，并且向左倾斜，即一个 2-node 是另一个 2-node 的左子节点。这种做法的好处是查找的时候不用做任何修改，和普通的二叉查找树相同。

红黑树的更新：

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。图 3.3.13

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。

插入结点

在插入新的键时，我们可以使用旋转操作帮助我们保证 2-3 树和红黑树之间的一一对应关系，因为旋转操作可以保持红黑树的两个重要性质：有序性和完美平衡性。也就是说，我们在红黑树中进行旋转时无需为树的有序性或者完美平衡性担心。下面我们来看看应该如何使用旋转操作来保持红黑树的另外两个重要性质：不存在两条连续的红链接和不存在红色的右链接。我们先用一些简单的情况热热身。

1.向树底部的 2-结点插入新键

一棵只含有一个键的红黑树只含有一个 2-结点。插入另一个键之后，我们马上就需要将他们旋转。如果新键小于老键，我们只需要新增一个红色的节点即可，新的红黑树和单个 3-结点完全等价。如果新键大于老键，那么新增的红色节点将会产生一条红色的右链接。我们需要使用 parent = rotateLeft(parent);来将其旋转为红色左链接并修正根结点的链接，插入才算完成。两种情况均把一个 2-结点转换为一个 3-结点，树的黑链接高度不变。

2.向一棵双键树（即一个 3-结点）中插入新键

这种情况又可分为三种子情况：新键小于树中的两个键，在两者之间，或是大于树中的两个键。每种情况中都会产生一个同时链接到两条红链接的结点，而我们的目标就是修正这一点。

三者中最简单的情况是新键大于原树中的两个键，因此它被链接到 3-结点的右链接。此时树是平衡的，根结点为中间大小的键，它有两条红链接分别和较小和较大的结点相连。如果我们将两条链接的颜色都由红变黑，那么我们就得到了一棵由三个结点组成，高为 2 的平衡树。它正好能够对应一棵 2-3 树，如图 3.3.20（左）。其他两种情况最终也会转化为这两种情况。

如果新键小于原书中的两个键，它会被链接到最左边的空链接，这样就产生了两条连续的红链接，如果 3.3.20（中）。此时我们只需要将上层的红链接右旋转

即可得到第一种情况。

如果新键介于原书中的两个键之间，这又会产生两条连续的红链接，一条红色左链接接一条红色右链接，如果 3.3.20（右）。此时我们只需要将下层的红链接左旋即可看得到第二种情况。

4.根结点总是黑色

颜色转换会使根结点变为红色，我们在每次插入操作后都会将根结点设为黑色。

5.向树底部的 3-结点插入新键

现在假设我们需要在树的底部的一个 3-结点下加入一个新结点。前面讨论过的三种情况都会出现，如图 3.3.22 所示。颜色转换会使指向中结点的链接变红，相当于将它送入了父结点。这意味着在父结点中继续插入一个新键，我们也会继续用相同的办法解决这个问题。

6.将红链接在树中向上传递

2-3 树中的插入算法需要我们分解 3-结点，将中间键插入父结点，如此这般知道遇到一个 2-结点或是根结点。总之，只要谨慎地使用左旋，右旋，颜色转换这三种简单的操作，我们就能保证插入操作后红黑树和 2-3 树的一一对应关系。在沿着插入点到根结点的路径向上移动时在所经过的每个结点中顺序完成以下操作，我们就能完成插入操作：

如果右子结点是红色的而左子结点是黑色的，进行左旋转

如果左子结点是红色的且她的左子结点也是红色的，进行右旋

如果左右子结点均为红色，进行颜色转换。

删除操作

要描述删除算法，首先要回到 2-3 树。和插入操作一样，我们也可以定义一系列局部变换来在删除一个结点的同时保持树的完美平衡性。这个过程比插入一个结点更加复杂，因为我们不仅要在（为了删除一个结点而）构造临时 4-结点时沿着查找路径向下进行变换，还要在分解遗留的 4-结点时沿着查找路径向上进行变换（同插入操作）。

1.自顶向下的 2-3-4 树

作为第一轮热身，我们先学习一个沿着查找路径既能向上也能向下进行变换的稍简单的算法：2-3-4 树的插入算法，2-3-4 树中允许存在我们以前见过的 4-结点。它的插入算法沿着查找路径向下进行变换是为了保证当前结点不是 4-结点（这样树底才有空间来插入新的键），沿着查找路径向上进行变换是为了将之前创建的 4-结点配平。

向下的变换和我们在 2-3 树中分解 4-结点所进行的变换完全相同。如果根结点是 4-结点，我们就将它分解成三个 2-结点，使得树高加 1。在向下查找的过程中，如果遇到一个父结点为 2-结点的 4-结点，我们将 4-结点分解为两个 2-结点并将中间键传递给他的父结点，使得父结点变为一个 3-结点；如果遇到一个父结点为 3-结点的 4-结点，我们将 4-结点分解为两个 2-结点并将中间键传递给它的父结点，使得父结点变为一个 4-结点；我们不必担心会遇到父结点为 4-结点的 4-结点，因为插入算法本身就保证了这种情况不会出现。到达树的底部之后，我们也只会遇到 2-结点或者 3-结点，所以我们可以插入新的键。要用红黑树实现这个算法，我们需要：

将 4-结点表示为由三个 2-结点组成的一颗平衡的子树，根结点和两个子结点都用红链接相连；

在向下的过程中分解所有 4-结点并进行颜色转换；

和插入操作一样，在向上的过程中用旋转将 4-结点配平。（因为 4-结点可以存在，所以可以允许一个结点同时链接两条红链接）。

令人惊讶的是，你只需要移动上面算法的 put()方法中的一行代码就能实现 2-3-4 树中的插入操作：将 colorFlip()语句（及其 if 语句）移动到递归调用之前（null 测试和比较操作之间）。在多个进程可以同时访问同一棵树的应用中这个算法优于 2-3 树。

2.删除最小键

在第二轮热身中我们要学习 2-3 树中删除最小键的操作。我们注意到从树底部的 3-结点中删除键是很简单的，但 2-结点则不然。从 2-结点中删除一个键会留下一个空结点，一般我们会将它替换为一个空链接，但这样会破坏树的完美平衡。所以我们需要这样做：为了保证我们不会删除一个 2-结点，我们沿着左链接向下进行变换，确保当前结点不是 2-结点（可能是 3-结点，也可能是临时的 4-结点）。首先根结点可能有两种情况。如果根是 2-结点且它的两个子结点都是 2-结点，我们可以直接将这三个结点变为一个 4-结点；否则我们需要保证根结点的左子结点不是 2-结点，如有必要可以从它右侧的兄弟结点“借”一个键来。

在沿着左链接向下的过程中，保证以下情况之一成立：

如果当前结点的左子结点不是 2-结点，完成；

如果当前结点的左子结点是 2-结点而它的亲兄弟结点不是 2-结点，将左子结点的兄弟结点中的一个键移动到左子结点中；

如果当前结点的左子结点和它的亲兄弟结点都是 2-结点，将左子结点，父结点中的最小键和左子结点最近的兄弟结点合并为一个 4-结点，使父结点由 3-结点

变为 2-结点或由 4-结点变为 3-结点。

3.删除操作

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。

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. 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。简单来说，动态分配器就是我们平时在 C 语言上用的 malloc 和 free,realloc，通过分配堆上的内存给程序，我们通过向堆申请一块连续的内存，然后将堆中连续的内存按 malloc 所需要的块来分配，不够了，就继续向堆申请新的内存，也就是扩展堆，这里设定，堆顶指针想上伸展（堆的大小变大）。

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. 空闲块和分配块链表
   采用分离的空闲链表。全局变量： void *Lists[MAX_LEN]; 因为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，而此堆的设计采用分离存储的来减少分配时间，就是维护多个空闲链表，每个链表中的块有大致相等的大小。将所有可能的块大小根据 2 的幂划分。

4. 分配器总体设计
   我们的分配器使用实验所提供的 memlib.c 所提供的一个内存系统模型。目的在于允许我们在不干涉已存在的系统层 malloc 包的情况下，运行分配器。包中提供的 mem_init 函数将对于堆来说可用的虚拟内存模型虚拟化为一个大的、双字对齐的字节数组。在 mem_heap 和 mem_brk 之间的字节表示已分配的虚拟内存。mem_brk 之后的字节表示为分配的虚拟内存。分配器通过调用 mem_sbrk 函数来请求额外的堆内存这个函数和系统的 sbrk 函数有相同的接口和语义。
   Mm_init 函数初始化分配器，如果成功就返回 0，否则返回-1，分配器在本次实

验中，采用显式空闲链表和最佳适配的方式来进行分配，因此块的结构与之前所述结构相同。
在该分配器中，我们将第一个定义为一个双字对齐不使用的填充字。填充后面紧跟着一个特殊的序言块，这是一个 8 字节的已分配块，只有一个头部和一个脚部组成。序言块是在初始化的时候创建的，并且永不释放，在序言块后紧跟的是 0 个或者多个由 malloc 或者 free 调用创建的普通块。堆总是以一个特殊的结尾块来结束，这个块是一个大小为 0 的已分配块，只有一个头部组成。序言块和结尾块是一种消除合并时边界条件的技巧分配器使用一个单独的私有全局变量，它总是指向序言块。

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。返回 0 表示正常，-1 表示发生错误

处理流程：
(1) 初始化分离空闲链表，将每个链表设置为 NULL
(2) 设置开始 Block，结束 Block：申请堆空间，设置开始 Block 的 Header，Footer，设置结束 Block 的 Footer。
(3) 拓展初始大小：拓展堆空间，拓展大小为 INITIALSIZE（48B）。
(4) 调用 mm_check：检查堆的一致性。

要点分析：
(1) 拓展初始堆空间：对应对于 realloc2-bal.rep 的优化，拓展初始化大小之后可以使第一块 Block 之后的数据始终保存在前 48B 之中，
(2) 同时保证了第一块 Block 始终位于堆空间的最高地址，同时保证两个 tracefile 之中 realloc 的简单与空间利用率。

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

参数：
void *ptr：指向已分配内存块

处理流程：
(1) 设置隐式链表信息：将 block 的 HDR 和 FTR 都设置为空闲状态(size,0)。
(2) 插入显示空闲链表：调用 insert_node 函数，将 ptr 指向的 block 插入到分离空

闲链表之中。
(3) 合并空闲 block：调用 coalesce 进行空闲 block 合并。
(4) 检查堆的一致性： mm_check

要点分析：
(1) 指针值 ptr 应是之前调用 mm_malloc 或 mm_realloc 返回的值，并且没有释放过
(2) 需要将空闲块 block 加入到分离空闲链表之中
(3) 在添加之后 block 位于堆之中，地址前后的块可能存在空闲块
(4) 需要调用 colesce 进行空闲块的合并，colesce 之中包括如果发生合并产生的必要的链表操作逻辑

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

函数功能：
将 ptr 所指向的旧内存块大小变为 size，并返回新内存块的地址

参数：
void *ptr：待处理的块第一个字的指针
size_t size：需要分配的字节

处理流程：
(1) 若 ptr 是空指针 NULL, 则函数操作等价于 mm_malloc(size)
(2) 若参数 size 等于 0，则函数操作等价于 mm_free(ptr)
(3) 若 ptr 非 NULL, 则它应该是之前调用 mm_malloc 或 mm_realloc 返回的数值，指向一个已分配的内存块。

要点分析：
(1) 先要实现内存对齐，调整 size 的大小，之后再寻找合适块。其中为了减少外部碎片，需要尽可能利用相邻的块，如果没有可以利用的连续未分配的块，此时只能申请新的而不连续的未分配块。
(2) 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。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：
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：
(1) 首先从堆的起始位置处检查序言块的头部和脚部是否都是双字, 如果序言块不是 8 字节的已分配块，则打印 Bad prologue header

(2) 从第一个块开始循环，直到结尾，依次检查当前块。通过 checkblock 函数检查是否双字对齐，通过 printblock 函数打印头部、脚部、已分配位的信息
(3) 检查结尾块是否大小为 0 且已分配, 如果结尾块不是一个大小位零的已分配块，则打印 Bad epilogue header

要点分析：
(1) 在检查堆的函数中调用检查每个块的函数，主要检查已分配块是不是双字对齐，头部和脚部是否匹配

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（8 字节对齐）。申请的整个块应该在恰当区间内，且不能与其他已经分配的块重叠。

参　　数：
size：申请内存块的大小

处理流程：
(1) 在检查完请求的真假之后，分配器调整请求块的大小，从而为头部和脚部留有空间，并且满足双字对齐的要求。最小块的大小是 16 字节，其中 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，然后向上舍入到最接近 8 的整数倍。
(2) 分配器调整了请求的大小后，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拓展堆，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) 链表操作：因为链表代表的块空间的大小，对 searchsize 进行循环除以二，最终小于等于时就找到了正确链表。
(2) Segregated_free_lists 中始终存放的是链表的尾，所以遍历的时候每次取得前驱，而访问顺序符合要求的地址递增原则。

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，并返回合并后的头指针。

处理流程：

(1) 前面的块和后面的块都是已分配的：不进行合并，当前块的状态只是简单地从分配变为空闲。同时，将其插入空闲链表的开头。

(2) 前面的块是已分配的，后面的块是空闲的：当前块与后面的块合并，用当前块和后面的块的大小的和来更新当前块的头部和后面块的脚部。同时，将新合成的块插入空闲链表的开头，将后面的块从原有空闲链表中删去。

(3) 前面的块是空闲的，后面的块是已分配的：前面的块和当前块合并，用两个块的大小的和来更新前面块的头部和当前块的脚部。同时，将新合成的块插入空闲链表的开头，将前面的块从原有空闲链表中删去。

(4) 前面的和后面的块都是空闲的：合并所有的三个块形成一个单独的空闲块，用三个块大小的和来更新前面块的头部和后面块的脚部。同时，将新合成的块插入空闲链表的开头，将前面、后面的块从原有空闲链表中删去。

要点分析：

(1) 获得了当前块的相邻块情况之后，主要的工作是对四种不同的情况进行处理。在合并前，首先要把待合并的块从分离空闲链表中删除，合并后注意更新总合并块的头部和脚部，大小为总合并块之和。最后需要把更新的 bp 所指的块插入到分离空闲链表中即可。

第 4 章测试

总分 10 分

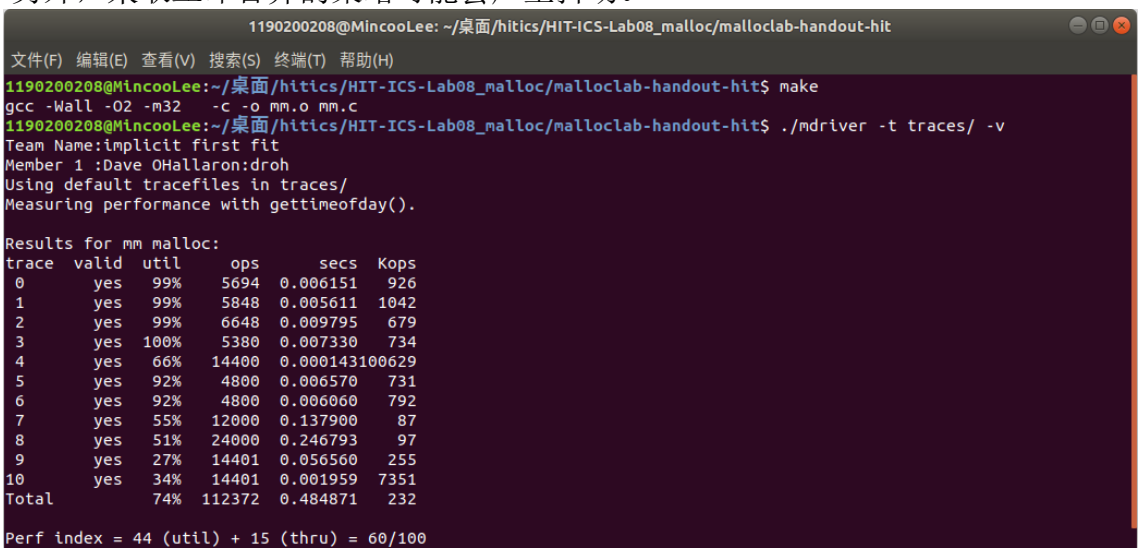
4.1 测试方法与测试结果(3 分)

按照 PPT 上所讲的方法进行测试。

先 make，通过链接、编译生成 mm.o，进一步生成可执行程序 mdriver。其中 mdriver.c 是 mm.c 的调用程序，整个测试程序的执行逻辑存放其中。再输入 ./mdriver -t traces/ -v ，对 traces 文件夹下的所有的轨迹文件测试并输出结果。

4.2 测试结果分析与评价（3 分）

采用隐式空闲链表+首次适配+立即合并，测试结果为：Perf index = 44(util)+7(thru)=60/100,实际查看可知，峰值利用率较高，但相对的吞吐量很小，其原因可能是隐式空闲链表每次进行 malloc 操作时需要从头开始遍历一次所有的块，另外，采取立即合并的策略可能会产生抖动。



4.4 性能瓶颈与改进方法分析（4 分）

性能瓶颈在于每次需要从头遍历一次及合并时的抖动。改进方法可以采用显式空闲链表+首次适配+立即合并的方法。显式空闲链表的结构，使得寻找空闲块时，遍历的是整个空闲块，而不是所有块，因此时间复杂度下降，吞吐量大大提高。

第 5 章 总结

5.1 请总结本次实验的收获

1. 深入理解了计算机系统虚拟存储的基本知识
2. 对 C 语言指针操作的理解进一步加深
3. 深入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 语言实现了动态存储分配器，并进行了测试分析

5.2 请给出对本次实验内容的建议

建议前几个实验安排轻松一点，不然后面的实验做着有些赶。

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.