

**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 1190200526

班　　 级 1903002

学 生 姓 名 沈城有

指 导 教 师 郑贵滨

实 验 地 点 G709

实 验 日 期 2021.6.11

**计算机科学与技术学院**

**目 录**

[第1章 实验基本信息 - 3 -](#_Toc74396373)

[1.1 实验目的 - 3 -](#_Toc74396374)

[1.2 实验环境与工具 - 3 -](#_Toc74396375)

[1.2.1 硬件环境 - 3 -](#_Toc74396376)

[1.2.2 软件环境 - 3 -](#_Toc74396377)

[1.2.3 开发工具 - 3 -](#_Toc74396378)

[1.3 实验预习 - 3 -](#_Toc74396379)

[第2章 实验预习 - 4 -](#_Toc74396380)

[2.1 动态内存分配器的基本原理（5分） - 4 -](#_Toc74396381)

[2.2 带边界标记的隐式空闲链表分配器原理（5分） - 4 -](#_Toc74396382)

[2.3 显式空闲链表的基本原理（5分） - 5 -](#_Toc74396383)

[2.4 红黑树的结构、查找、更新算法（5分） - 6 -](#_Toc74396384)

[第3章 分配器的设计与实现 - 8 -](#_Toc74396385)

[3.2.1 int mm\_init(void)函数（5分） - 8 -](#_Toc74396386)

[3.2.2 void mm\_free(void \*ptr)函数（5分） - 9 -](#_Toc74396387)

[3.2.3 void \*mm\_realloc(void \*ptr, size\_t size)函数（5分） - 9 -](#_Toc74396388)

[3.2.4 void mm\_checkheap(int verbose)函数（5分） - 9 -](#_Toc74396389)

[3.2.5 void \*mm\_malloc(size\_t size)函数（10分） - 10 -](#_Toc74396390)

[3.2.6 static void \*coalesce(void \*bp)函数（10分） - 10 -](#_Toc74396391)

[第4章测试 - 12 -](#_Toc74396392)

[4.1 测试方法 - 12 -](#_Toc74396393)

[4.2 测试结果评价 - 12 -](#_Toc74396394)

[4.3 自测试结果 - 12 -](#_Toc74396395)

[第5章 总结 - 13 -](#_Toc74396396)

[5.1 请总结本次实验的收获 - 13 -](#_Toc74396397)

[5.2 请给出对本次实验内容的建议 - 13 -](#_Toc74396398)

[参考文献 - 14 -](#_Toc74396399)

# 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

Visual Studio Code；gcc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

**总分20分**

## 2.1 动态内存分配器的基本原理（5分）

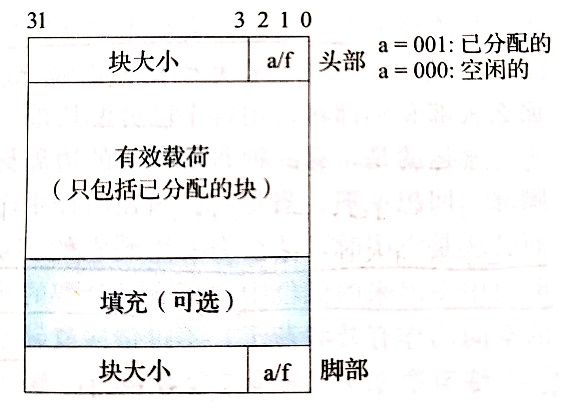
动态内存分配器维护着一个进程的虚拟内存区域，称为堆。分配器将堆视为一组不同大小的块的集合来维护，每个块就是一个连续的虚拟内存片，要么是已分配的，要么是空闲的。已分配的块显式地保留为供应用程序使用。空闲块可用来分配。空闲块保持空闲，直到它显式地被应用所分配。一个已分配的块保持已分配状态，直到它被释放，这种释放要么是应用程序显式执行的，要么是内存分配器自身隐式执行的。

分配器有两种基本风格：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：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，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 带边界标记的隐式空闲链表分配器原理（5分）

对于带边界标记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，如下图所示：



头部和脚部（为节省内存，已分配块可以没有脚部）编码了这个块的大小（包括头部和所有的填充）、分配情况等信息。假如存在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，我们只需要高29位保存块大小，剩余的3位用于编码其他信息。在上图中，最低位用来指明这个块是已分配的还是空闲的。头部后面是应用调用分配器时请求的有效载荷。有效载荷之后是填充（可选），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。

将这种格式的块组织为一个连续的已分配块和空闲块的序列，就称为带边界标记的隐式空闲链表。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。

带边界标记的隐式空闲链表的优点是简单，且合并空闲块仅需常数时间。但放置分配的块需要对空闲链表进行搜索，该搜索所需的时间与堆中所分配块和空闲块的总数呈线性关系。

## 2.3 显式空闲链表的基本原理（5分）

根据定义，程序不需要一个空闲块的主体，所以实现空闲链表数据结构的指针可以存放在这些空闲块的主体里面。

显式空闲链表将堆组织成一个双向空闲链表，在每个空闲块的主体中，都包含一个pred（前驱）和succ（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**

红黑树是一种不严格的平衡二叉查找树。要求如下：

1. 每个节点不是红色就是黑色的，根节点总是黑色的；
2. 每个叶子节点都是黑色的空节点(NIL)，也就是说，叶子节点不存储数据;（主要是为了简化红黑树的代码实现而设置，实际中所有叶子结点都指向同一个空黑节点）；
3. 任何相邻的节点都不能同时为红色，也就是说，红色节点是被黑色节点隔开的；
4. 每个节点，从该节点到达其可达叶子节点的所有路径，都包含相同数目的黑色节点。
5. **红黑树的查找算法**

红黑树的查找方法与其他二叉查找树类似，不需要做太多更改，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。具体过程如下：

从根结点开始，将要查找的值与当前结点值进行比较，若小于当前结点值，则与当前结点的左孩子结点进行比较；若大于当前结点值，则与当前结点的右孩子结点进行比较。重复以上过程直至到达叶结点（查找失败）或到达比较结果为相等的结点（查找成功）则结束。

示例Java源码：

RBNode<T> current = null;

RBNode<T> x = this.root;

while (x != null) {

    current = x;

    int cmp = node.key.compareTo(x.key);

    if (cmp < 0) {

        x = x.left;

    } else {

        x = x.right;

    }

}

1. **红黑树的更新算法**

红黑树的插入操作与其他二叉树类似，先使用查找算法找到插入的位置，然后再将节点插入。有三种情况会导致平衡被破坏：

* 插入节点的父节点和其叔叔节点（祖父节点的另一个子节点）均为红色；
* 插入节点的父节点是红色的，叔叔节点是黑色的，且插入节点是其父节点的右子节点；
* 插入节点的父节点是红色的，叔叔节点是黑色的，且插入节点是其父节点的左子节点。

红黑树的删除操作与二叉查找树类似，但需要在删除后进行平衡的修复。实际上，删除过程较复杂，很多情况下会采用在节点类中添加一个删除标记，并不是真正的删除节点，详细的删除过程此处略。

利用查找算法确定插入、删除位置并执行操作后，红黑树可能会不平衡，此时需要进行平衡修正。

红黑树主要通过三种方式对平衡进行修正：改变节点颜色、右旋和左旋。下面对三种方式进行简要介绍：

1. 改变结点颜色：按照红黑树结构的要求对节点颜色进行更改。
2. 右旋：首先要说明的是节点本身是不会旋转的，旋转改变的是节点之间的关系，选择一个节点作为旋转的顶端，如果做一次右旋，这个顶端节点会向下和向右移动到它右子节点的位置，它的左子节点会上移到它原来的位置。右旋的顶端节点必须要有左子节点。
3. 左旋：左旋的顶端节点必须要有右子节点，效果同理。

具体代码实现此处略去。

**注意**：改变颜色也是为了帮助我们判断何时执行什么旋转，而旋转是为了保证树的平衡。仅改变节点颜色是不能起到任何作用的，旋转才是关键的操作，在新增节点或者删除节点之后，可能会破坏二叉树的平衡，那么何时执行旋转以及执行什么旋转，这是需要重点关注的。

# 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。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。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，直到它被显式地被应用所释放（此动态内存分配器为显式分配器）。

1. **堆中内存块的组织结构**

堆中的一个内存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、尾部、有效载荷和所有填充），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。在此设计中，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。

1. **空闲块、分配块链表结构**

采用带边界标记的隐式空闲链表，使用立即边界标记合并方式。

1. **相关算法**

分配器使用首次适配（线性查找）、立即合并策略，具体算法见3.2节。

**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：创建有一个空闲块（大小为CHUNKSIZE）的初始堆。

处理流程：

1. 从内存中申请四个字，分别作为对齐填充、序言块头部、序言块脚部和结尾块；
2. 将私有全局变量heap\_listp指向序言块；
3. 调用extend\_heap()函数将堆扩展CHUNKSIZE字节并创建初始空闲块，将结尾块重新放置在堆顶；
4. 若在以上步骤中出现返回地址为空的情况，则需要立即停止初始化过程并返回-1提示错误；
5. 顺利完成以上步骤说明初始化完成，动态分配器已准备好接受来自应用的分配和释放请求。

要点分析：

1. 为保持对齐，序言块前要有一个字（4字节）的对齐填充；
2. 内存申请及堆扩展失败时需要停止初始化操作，并返回错误。

## 3.2.2 void mm\_free(void \*ptr)函数（5分）

函数功能：显式释放一个已分配的内存块。

参 数：void \*ptr 指向需释放内存块的有效载荷起始位置的指针

处理流程：

1. 使用宏函数获取要释放的块的大小；
2. 使用宏函数修改此内存块的头部和脚部信息（大小保持不变，已分配位置0表示空闲）；
3. 调用coalesce()函数检查是否需要合并空闲块，若需要则立即执行合并。

要点分析：

1. 释放内存块需修改已分配位标记；
2. 分配器采用立即合并策略，故在释放后要调用coalesce()函数检查空闲块并进行可能的合并。

## 3.2.3 void \*mm\_realloc(void \*ptr, size\_t size)函数（5分）

函数功能：将ptr所指的块重新分配为具有至少size字节的有效载荷的块，并复制原数据至新块。

参 数：void \*ptr 指向原内存块有效载荷起始位置的指针

size\_t size 需要分配的大小（字节）

处理流程：

1. 调用mm\_malloc()函数分配有效载荷至少为size字节大小的内存块；
2. 使用宏函数获取原块的大小；
3. 将原块数据写入新块，若原块大小大于新块，则截取前size字节写入新块；
4. 释放原内存块并返回新内存块的指针。

要点分析：

注意新内存块大小小于原内存块的情况(copySize > size)，需截断数据内容前size字节写入，避免越界访问。

## 3.2.4 void mm\_checkheap(int verbose)函数（5分）

函数功能：检查堆的一致性。

处理流程：

1. 可选输出堆中序言块地址；
2. 调用checkblock()检查序言块是否符合要求；
3. 调用printblock()输出其后每一个内存块的信息；
4. 调用checkblock()检查各内存块；
5. 可选输出堆中结尾块的信息；
6. 检查结尾块。

要点分析：

此函数从序言块开始，依次检查各块是否符合要求（大小、已分配位、头部和脚部信息是否一致、序言块和结尾块格式是否正确等），注意到此函数的使用会影响性能，确认堆一致性后应注释掉所有调用。

## 3.2.5 void \*mm\_malloc(size\_t size)函数（10分）

函数功能：分配一个有效载荷至少为size字节的内存块。

参 数：size\_t size 所申请内存块的有效载荷

处理流程：

1. 检查size合法性，若size为负则返回NULL；
2. 分情况计算应分配的块的大小：若size小于等于两字，则分配两字有效载荷（两字对齐）+头部+脚部，共四字；若size大于两字，则需计算头部+脚部+有效载荷向上取整到最接近的8字节（两字对齐）整数倍；
3. 使用首次适配策略找到满足条件的空闲块，若找到，则调用place()函数修改块头部、脚部信息，返回指向此块有效载荷的指针，完成分配；
4. 若未找到满足要求的块，则调用extend\_heap()函数扩展堆，调用place()函数分配块，返回指向此块有效载荷的指针，完成分配；
5. 假如扩展堆失败，说明内存已占满，返回NULL。

要点分析：

1. 注意检查size是否为负，避免非法操作，提高程序的健壮性；
2. 分配的块需满足两字对齐，有效载荷不足两字按两字计算，超过两字的要将块大小向上取整至最接近的两字整数倍；
3. 当前堆空间内无法找到适配的情况需扩展堆。
4. 分配块时需调用帮助函数place()来分割空闲块（若空闲块大小与分配大小之差大于等于最小块大小——16字节/4字）并设置块的头部、脚部信息。

## 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（若有）合并成一个大的空闲块。

处理流程：

1. 使用宏函数获取前一个块、后一个块的已分配位（用于判断情况）及此块的大小（存入size）；
2. 分情况执行：

* 假如前后两块均已分配，说明无法合并，直接返回原指针；
* 假如前块已分配，后块空闲，说明需合并后块与此块，将size加上后块大小（宏函数获取），修改此块头部与原后块的脚部信息（大小为size，已分配位置为0），返回原指针；
* 假如前块空闲，后块已分配，说明需合并前块与此块，将size加上前块大小（宏函数获取），修改此块脚部和原前块的头部信息（大小为size，已分配位置为0），并返回指向原前块有效载荷的指针；
* 假如前后两块均空闲，说明需将三个空闲块合并为一个，将size加上前后两块的大小（宏函数获取），修改前块的头部和后块的脚部信息（大小为size，已分配位置为0），并返回指向原前块有效载荷的指针。

要点分析：

1. 合并块时不需要修改每一个头部和脚部信息，只需修改合并后得到的块对应的头部和脚部即可，中间存在的信息会被忽略；
2. 注意分为四种情况分别处理，且前块空闲的两种情况需返回指向原前块（现为合并后空闲块）有效载荷的指针而非原指针。

# 第4章测试

**总分10分**

## 4.1 测试方法

1. 生成可执行评测程序文件：linux>make
2. 评测：linux>./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

-t <dir> 在dir目录寻找traces文件进行测试

1. 轨迹文件：

指示测试驱动程序mdriver以一定顺序调用mm\_malloc, mm\_realloc 和mm\_free。

1. 性能评分：

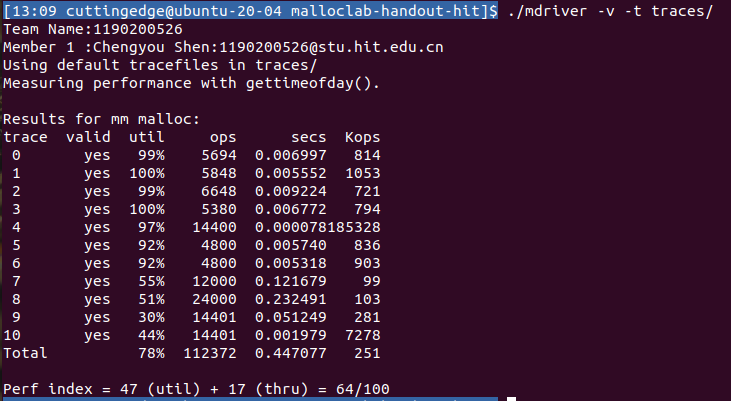
性能分pindex是空间利用率和吞吐率的线性组合。

命令：linux>./mdriver -v -t traces/

## 4.2 测试结果评价

由于使用的是简单的隐式空闲链表，且仅在提供代码的基础上进行少量的修改，故测试结果一般。

## 4.3 自测试结果



# 第5章 总结

## 5.1 请总结本次实验的收获

1. 掌握了带边界标记隐式空闲链表的相关知识；
2. 学会设计基于隐式空闲链表的简单动态内存分配器，完成初始化、分配、重分配、释放、合并空闲块等操作；
3. 学会使用宏函数进行一些简单的操作。

## 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

1. 深入理解计算机系统（原书第三版）机械工业出版社, 2016.
2. 红黑树. 博客.

<https://xlbpowder.cn/2021/02/25/%E7%BA%A2%E9%BB%91%E6%A0%91/>

1. 数据结构和算法（十一）：红黑树. 知乎专栏.

<https://zhuanlan.zhihu.com/p/37470948>