动态内存分配

Dynamic Memory Allocation

本章内容

Topic

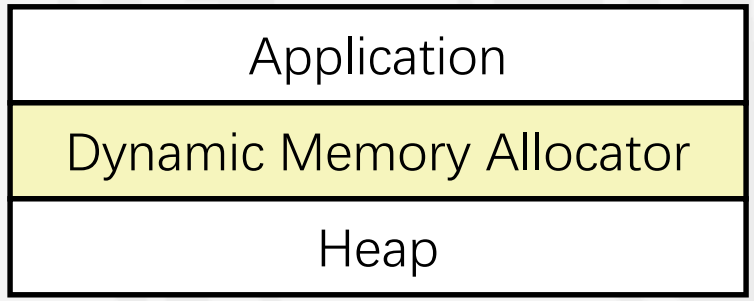
- 基本概念
- 隐式空闲链表
- 显式空闲链表
- 分离的空闲链表
- 垃圾收集
- 与内存相关的错误

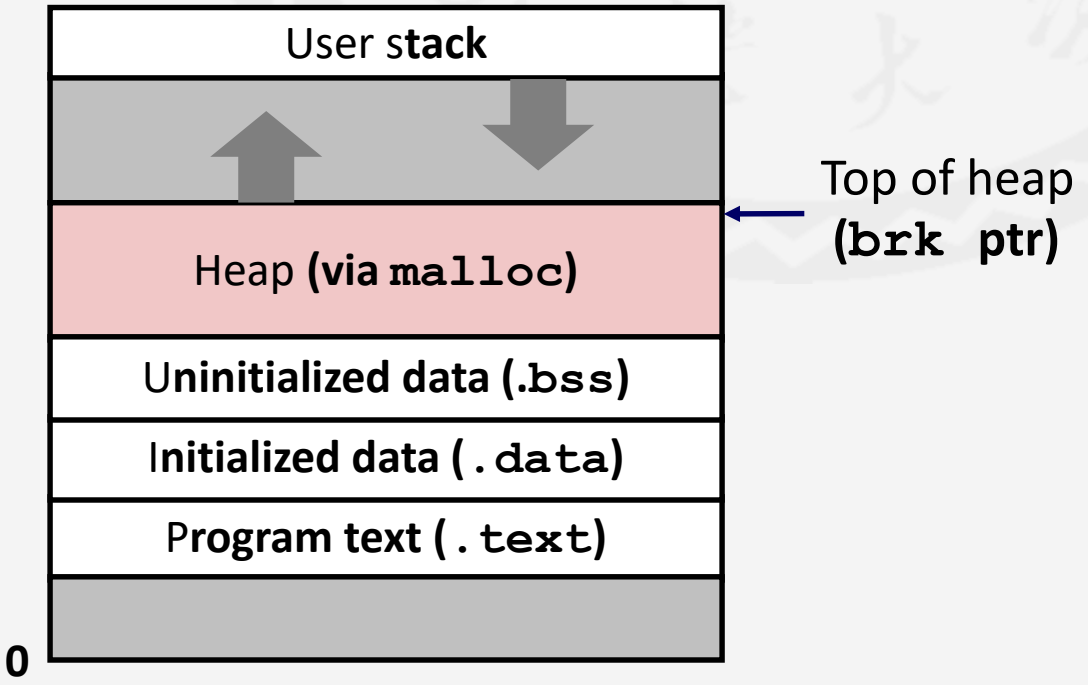
基本概念

Basic concepts

动态内存分配

- 程序员使用**动态内存分配器**（如malloc）在运行时获取虚拟内存
  - 特别是对于在运行时大小未知的数据结构。
- 动态内存分配器管理进程虚拟内存中的一个区域，被称为堆。





动态内存分配

- 分配器将堆维护为可变大小块的集合，这些块可以是已分配或空闲的。
- 分配器的类型有：
  - 显式分配器：应用程序分配并释放空间，例如C中的malloc和free。
  - 隐式分配器：应用程序分配空间，但不释放，例如Java、ML和Lisp中的垃圾回收。
- 接下来将讨论简单的显式内存分配

基本概念

Basic concepts

malloc和free

```
#include <stdlib.h>
```

```
void *malloc(size_t size)
```

- 成功时：
  - 返回一个指向至少大小字节的内存块的指针，对齐到8字节（x86）或16字节（x86-64）边界
  - 如果大小为0，则返回NULL
- 不成功时：
  - 返回NULL（0）并设置errno

```
void free(void *p)
```

- 将指针p指向的内存块返回到可用内存池中。
- p必须来自对malloc或realloc的先前调用。

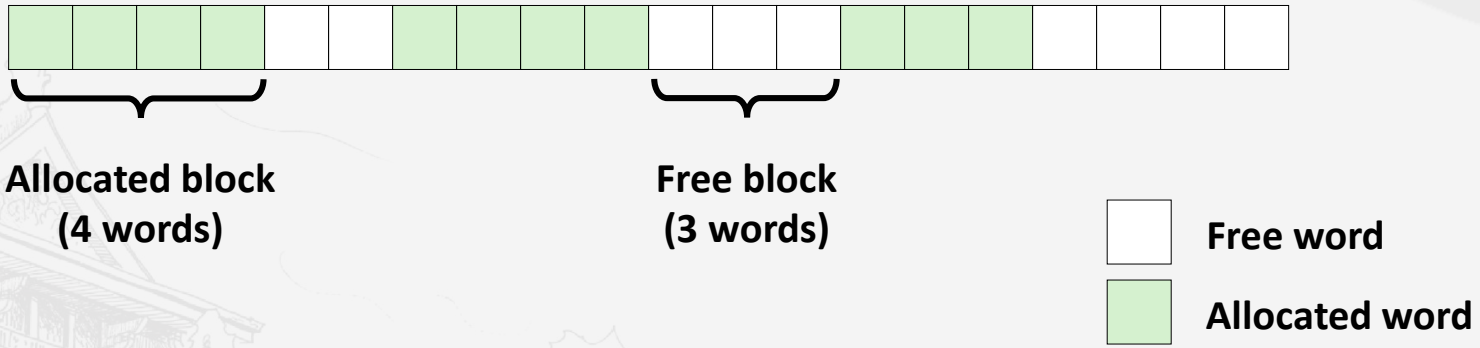
- 其他函数：
  - calloc：malloc的版本，将分配的块初始化为零。
  - realloc：更改先前分配块的大小。
  - sbrk：由分配器内部用于扩展或收缩堆。

malloc示例

```c
#include <stdio.h>
#include <stdlib.h>

void foo(int n) {
    int i, *p;

    /* Allocate a block of n ints */
    p = (int *) malloc(n * sizeof(int));
    if (p == NULL) {
        perror("malloc");
        exit(0);
    }

    /* Initialize allocated block */
    for (i=0; i<n; i++)
        p[i] = i;


    /* Return allocated block to the heap */
    free(p);
}
```

后续讨论的一些假设

- 为了简化后续讨论我们进行一些假设：
  - 内存是按字（word）寻址的
  - 字（word）的大小是int类型



示例

```
p1 = malloc(4)
```

```
p2 = malloc(5)
```

```
p3 = malloc(6)
```

```
free(p2)
```

```
p4 = malloc(2)
```

约束

- 应用程序：
  - 可以发出任意顺序的malloc和free请求
  - free请求必须针对一个malloc分配的块
- 分配器：
  - 无法控制分配块的数量或大小
  - 必须立即响应malloc请求，即不能重新排序或缓冲请求
  - 必须从空闲内存中分配块，即只能将分配块放置在空闲内存中
  - 必须对齐块，以满足所有对齐要求，如在Linux系统上是8字节（x86）或16字节（x86-64）对齐
  - 只能操作和修改空闲内存
  - 一旦malloc分配，就不能移动分配块，即不允许紧凑操作

性能目标：吞吐量

- 对于一系列malloc和free请求：
  - $R_0, R_1, ..., R_k, ..., R_{n-1}$
- 目标：最大化吞吐量和峰值内存利用率
  - 这些目标通常是相互冲突的
- 吞吐量：
  - 单位时间内完成的请求数量
  - 例如：
    - 在10秒内进行了5,000次malloc调用和5,000次free调用
    - 吞吐量为1,000次操作/秒

基本概念

Basic concepts

性能目标：峰值内存利用率

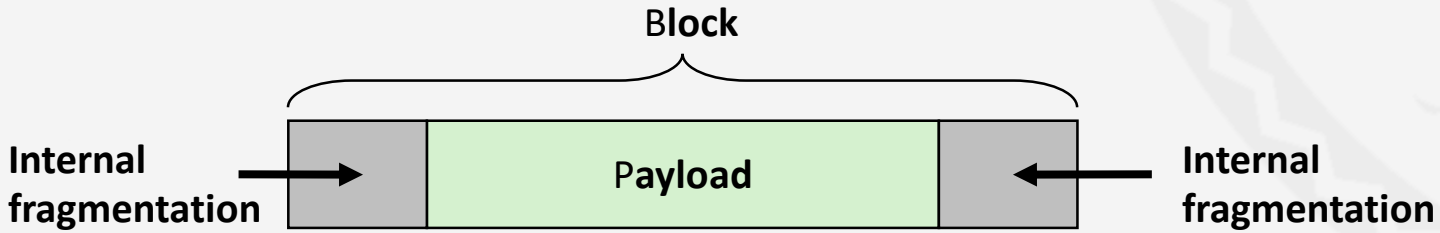
- 对于一系列malloc和free请求：
  - $R_0, R_1, ..., R_k, ..., R_{n-1}$
- 定义：聚集有效载荷P_k
  - malloc(p)会分配一个有效载荷为p字节的块
  - 在请求R_k完成后，聚集有效载荷P_k是当前分配有效载荷的总和
- 定义：当前堆大小H_k
  - 假设H_k是单调非减的， 仅当分配器使用sbrk时堆才会增长
- 定义：前k+1个请求的峰值内存利用率 $U_k = (max_{i<=k} P_i) / H_k$

碎片

- 不佳的内存利用率会导致内存碎片化
  - 内部碎片
  - 外部碎片

内部碎片

- 对于给定的块，如果有效载荷小于块大小，则产生内部碎片



- 引起内部碎片的原因包括:
  - 维护堆数据结构的开销
  - 为了对齐目的而进行的填充
  - 分配策略导致（例如，为了满足小请求而返回一个大块）
- 仅依赖于之前的请求，因此易于测量

外部碎片

- 当有足够的聚集堆内存，但没有足够大的单个空闲块时，发生外部碎片化

```
p1 = malloc(4)
```

```
p2 = malloc(5)
```

```
p3 = malloc(6)
```

```
free(p2)
```

```
p4 = malloc(6)
```

Oops! (这时会发生什么?)

- 外部碎片化取决于未来请求的模式，因此难以测量。

内存分配器实现时遇到的问题

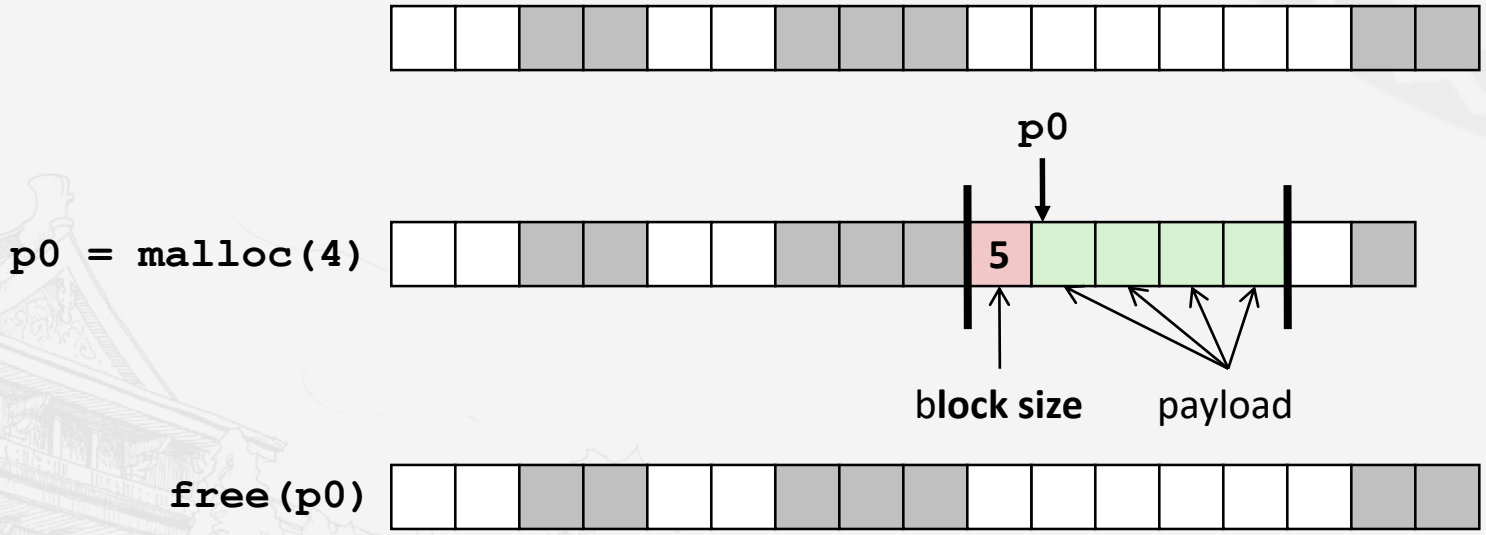
- 我们如何仅通过指针就能知道释放多少内存?
- 如何跟踪空闲块?
- 当分配的结构比放置它的空闲块小的时候，应该怎么处理额外的空间?
- 在多个块都适合分配时，如何选择要使用的块?
- 如何处理一个刚刚被释放的块?

基本概念

Basic concepts

如何知道需要释放多少内存?

- 标准方法:
  - 在块之前的字中保存块的长度，这个字通常被称为头字段或头
  - 对于每个已分配的块，需要额外的一个字

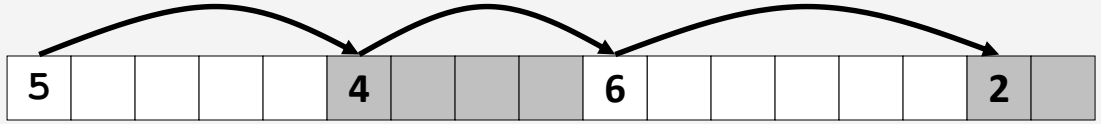


基本概念

Basic concepts

如何跟踪空闲块?

- 方法1：隐式空闲链表，使用包含了块长度的头部信息，以连接所有的块



- 方法2：显式空闲链表，在空闲块之间使用指针连接

- 方法3：分离的空闲链表：为不同的大小类别使用不同的空闲链表
- 方法4：按大小排序的块：可以使用平衡树（例如红黑树），在每个空闲块内部使用指针，并将长度用作键。

本章内容

Topic

- 基本概念
- 隐式空闲链表
- 显式空闲链表
- 分离的空闲链表
- 垃圾收集
- 与内存相关的错误

隐式空闲链表

Implicit Free Lists

- 每个块都需要提供块大小和当前分配状态的信息
  - 将如果这些信息存储在两个字中会造成空间的浪费!
- 一种标准的技巧
  - 如果块地址始终处于对齐状态，其低位地址的值总是0
  - 不存储这些始终为0的位，而是将其用作分配状态的标志
  - 在获取块大小信息时，必须屏蔽掉此位。

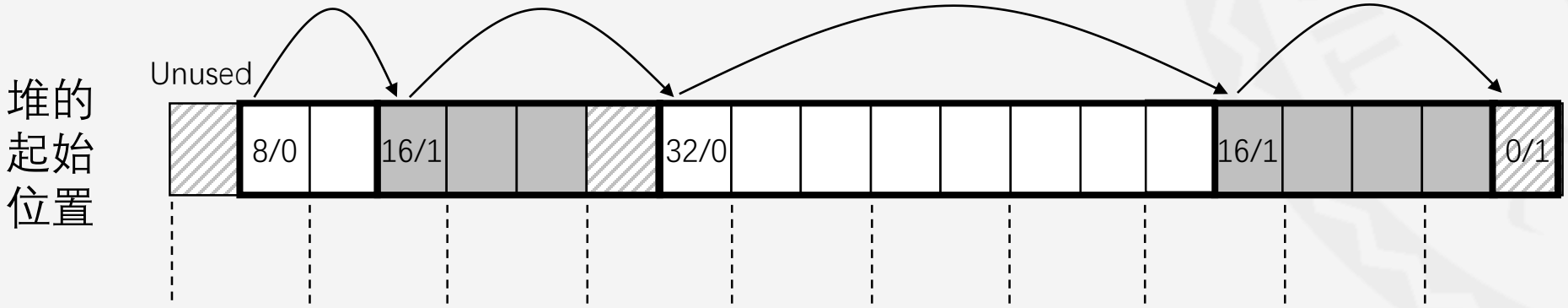
块的格式（已分配/空闲）

a = 1: 已分配块
a = 0: 空闲块

块大小包括头部、有效载荷和所有的填充

隐式空闲链表

Implicit Free Lists



双字对齐

阴影部分：已分配的块

无阴影部分：空闲块

头部：包含块字节大小和分配状态位

隐式空闲链表

Implicit Free Lists

搜索空闲块

- 首次适配算法（First Fit）
  - 从链表开头开始搜索，选择第一个适合的空闲块：

```
p = start;
while ((p < end) &&      \\ not passed end
       ((*p & 1) ||      \\ already allocated
       (*p  <= len)))    \\ too small
  p = p + (*p & -2);     \\ goto next block (word addressed)
```

  - 线性时间复杂度 O(n)
- 在实践中，这种方法可能会在列表开头的部分造成“碎片化”
  - 优点：尽量使用低地址空间，因而在高地址的空间可能会保留较大的空闲分区。所以，大进程申请的存储空间大都能在高地址端得到满足。
  - 缺点：由于每次只简单地使用找到的第一个分区，结果可能导致将较大的空闲分区不断地分割为较小的空闲分区。

搜索空闲块（2）

- 下次适配算法（Next Fit）
  - 类似于首次适配，但是从上次搜索结束的位置开始搜索列表
  - 通常比首次适配更快：避免重新扫描无法使用的块
- 一些研究表明，这种方法会产生更严重的碎片化问题
  - 该算法常常会导致内存中缺乏大分区，因为它会均衡地利用空闲分区，包括分割较大的空闲分区。从而使得大进程无法装入内存运行。
  - 下次适应算法可能会导致大量的外碎片，需要较频繁地实施紧凑操作，效率较低。

搜索空闲块（3）

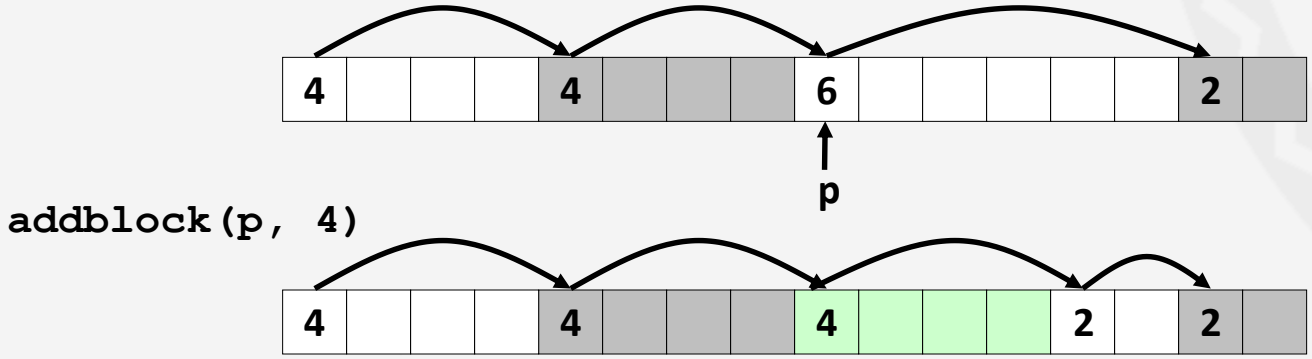
- 最佳适配（Best Fit）
  - 搜索列表，选择最佳的空闲块：适合，并且剩余字节数最少。
  - 保持碎片小——通常提高内存利用率。
  - 通常比首次适配运行得更慢。

隐式空闲链表

Implicit Free Lists

分割空闲块

- 由于分配的空间可能小于空闲空间，则需要将空闲块拆分




```
void addblock(ptr p, int len) {
  int newsize = ((len + 1) >> 1) << 1;  // round up to even
  int oldsize = *p & -2;                // mask out low bit
  *p = newsize | 1;                     // set new length
  if (newsize < oldsize)
    *(p+newsize) = oldsize - newsize;   // set length in remaining
}                                       // part of block
```

隐式空闲链表

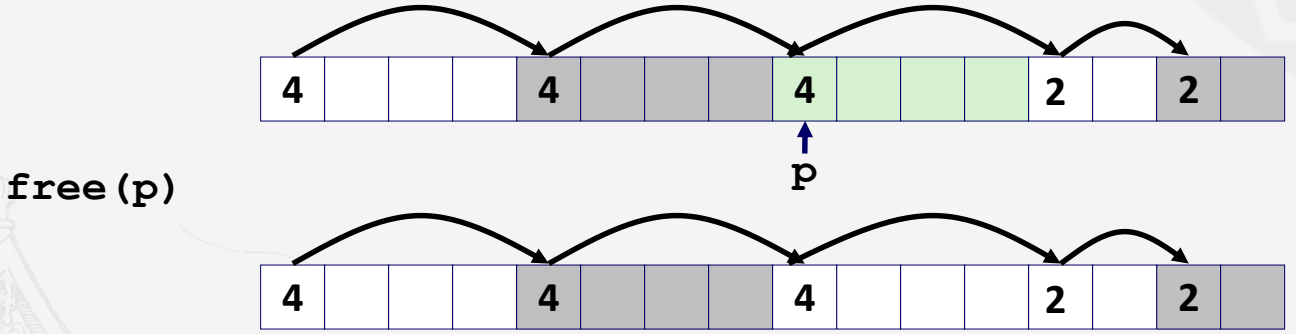
Implicit Free Lists

释放块空间

- 最简单的实现方式：清除“已分配”标志

```
void free_block(ptr p) {
    *p = *p & -2;
}
```

- 这可能导致“假碎片”



malloc(5) 无法分配！

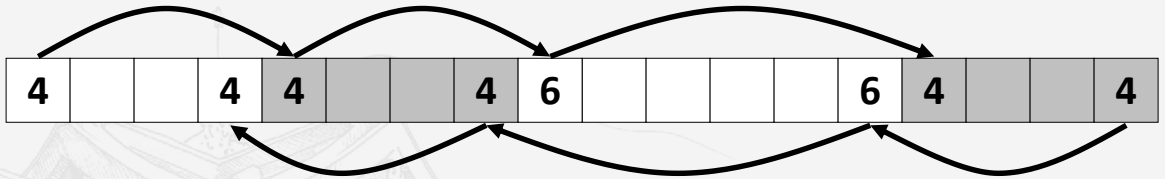
有足够的空闲空间，但分配器无法找到它

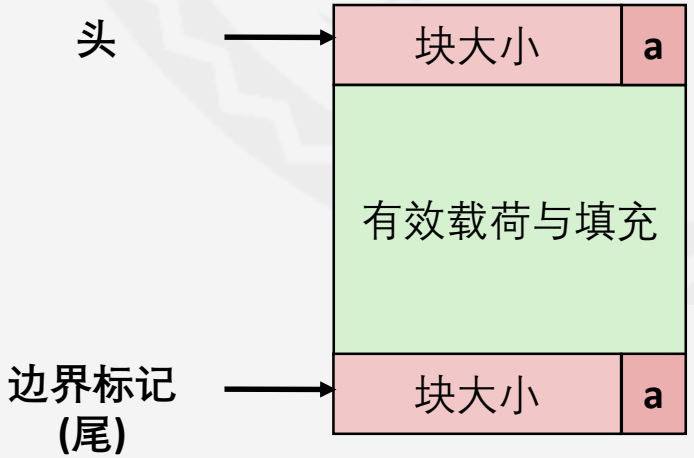
隐式空闲链表

Implicit Free Lists

双向合并

- 边界标记[Knuth73]
  - 在空闲块的“底部”（末尾）复制大小/分配字
  - 使我们能够向后遍历“列表”，但需要额外的空间
  - 这是一种重要且通用的技术!





隐式空闲链表

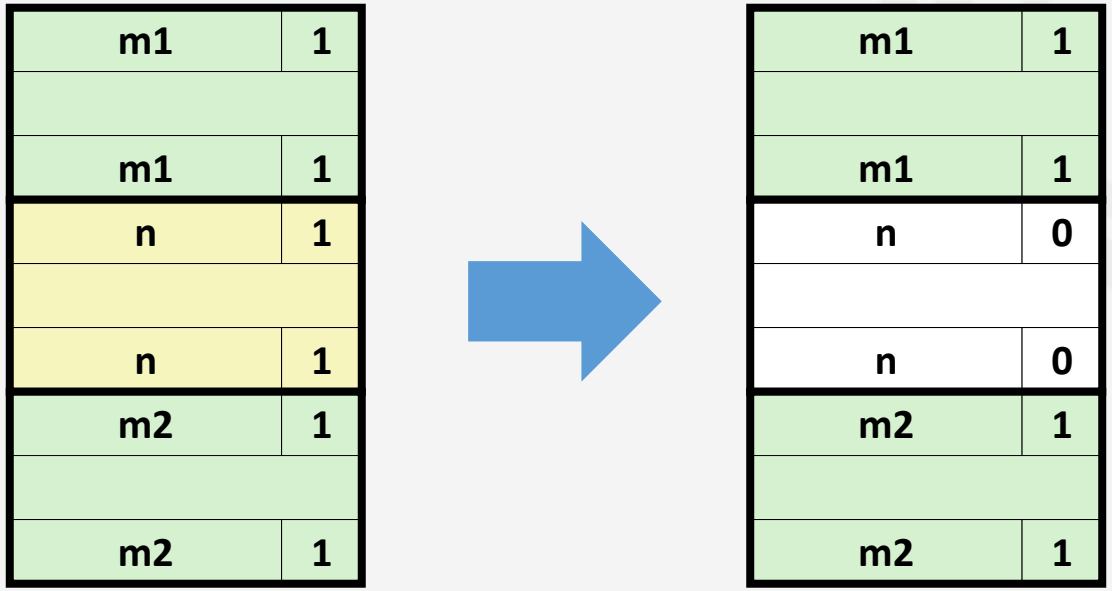
Implicit Free Lists

常数时间复杂度的合并：四种情况

隐式空闲链表

Implicit Free Lists

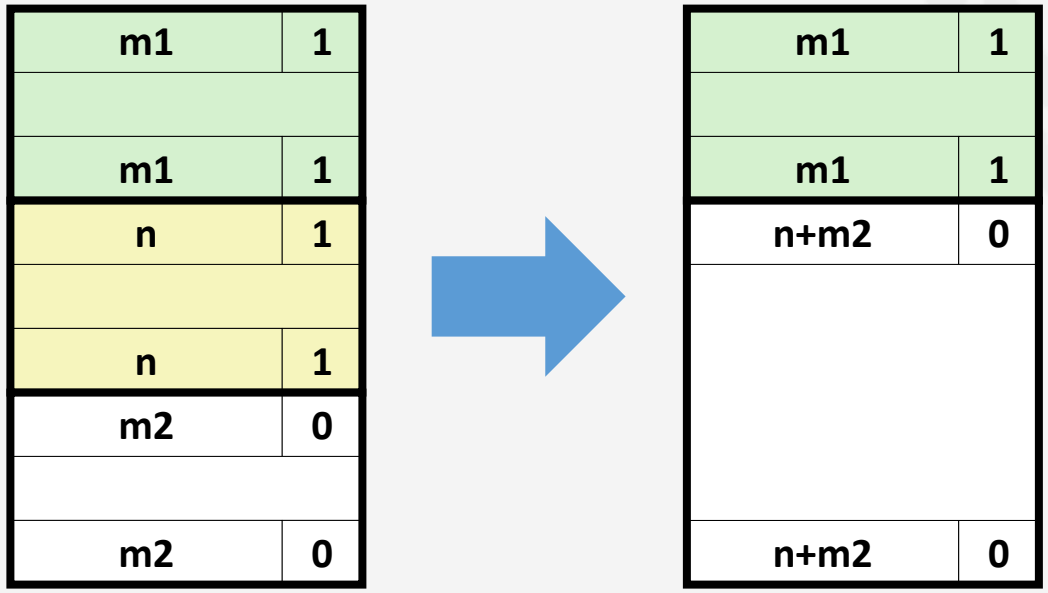
第一种情况：待释放块被前后都是已分配块



隐式空闲链表

Implicit Free Lists

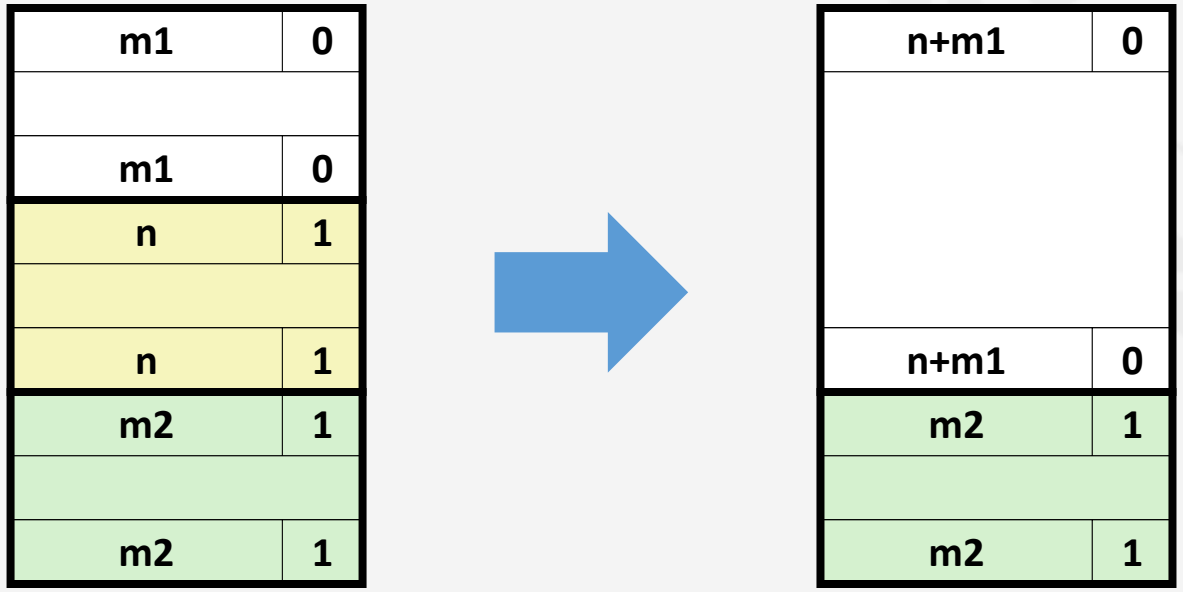
第二种情况：待释放块后面是空闲块



隐式空闲链表

Implicit Free Lists

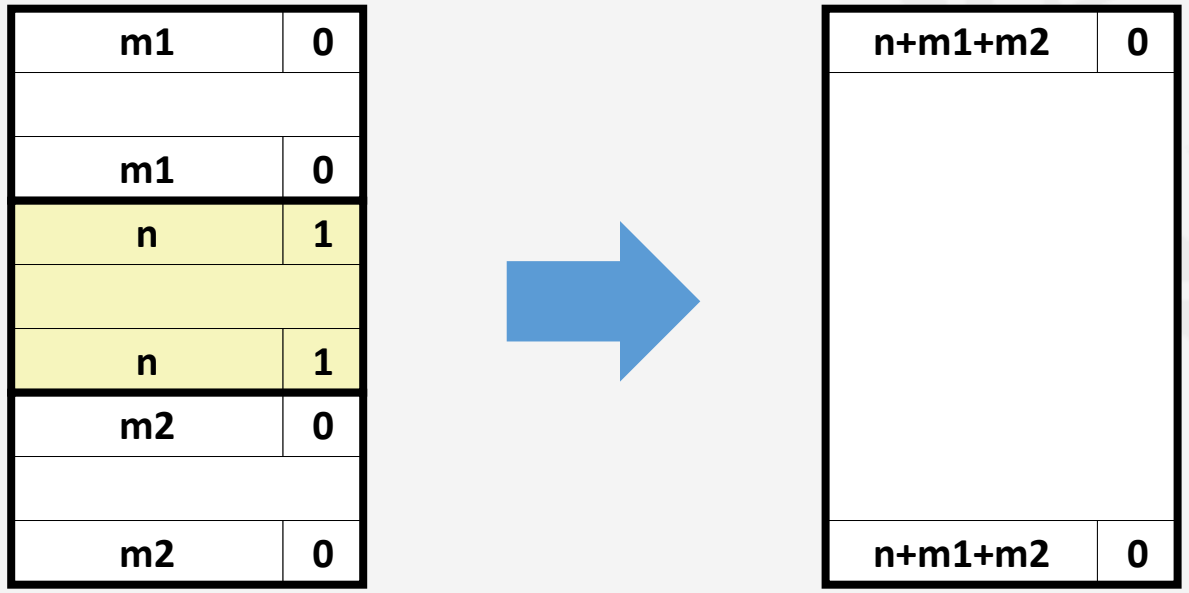
第三种情况：待释放块前面是空闲块



隐式空闲链表

Implicit Free Lists

第四种情况：待释放块前后都是空闲块



边界标记方法的缺点

- 边界标记占用的额外的存储
  - 当有效载荷较小时，边界标记的内存开销占比大
- 如何进行优化?
  - 哪些块需要底部“标签”
  - 这意味着什么?

隐式空闲链表

Implicit Free Lists

小结：分配器策略

- 放置策略：
  - 首次适配、下次适配、最佳适配等
  - 在低碎片化和较低吞吐量之间进行平衡
- 分割策略：
  - 何时进行空闲块拆分？
  - 愿意容忍多少内部碎片？
- 合并策略
  - 立即合并：每次调用空闲时都进行合并
  - 延迟合并：尝试通过延迟合并来改善free的性能。例如：
    - 当malloc扫描空闲列表时合并
    - 当外部碎片化达到某个阈值时合并

隐式空闲链表

Implicit Free Lists

小结：隐式空闲链表

- 实现：非常简单
- 分配成本：
  - 最坏情况下为线性时间
- 释放成本：
  - 最坏情况下为常数时间
  - 即使进行合并
- 内存使用：
  - 取决于放置策略
  - 首次适配、下次适配或最佳适配
- 由于线性时间分配复杂度，在malloc/free中并没有实际使用
  - 只在在许多特殊用途应用中使用
- 然而，分裂和边界标签合并的概念对所有分配器都是通用的

本章内容

Topic

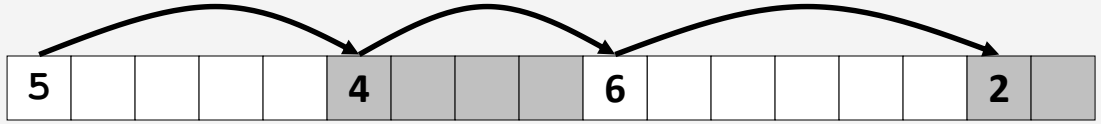
- 基本概念
- 隐式空闲链表
- 显式空闲链表
- 分离的空闲链表
- 垃圾收集
- 与内存相关的错误

显式空闲链表

Explicit Free Lists

回顾：跟踪空闲块

- 方法1：隐式空闲链表，使用包含了块长度的头部信息，以连接所有的块



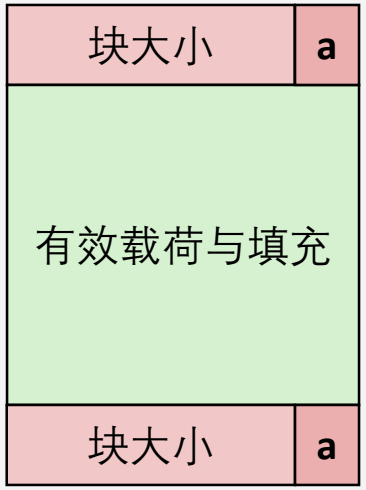
- 方法2：显式空闲链表，在空闲块之间使用指针连接

- 方法3：分离的空闲链表：为不同的大小类别使用不同的空闲链表
- 方法4：按大小排序的块：可以使用平衡树（例如红黑树），在每个空闲块内部使用指针，并将长度用作键。

显式空闲链表

Explicit Free Lists

分配后
(和隐式链表一样)



空闲时

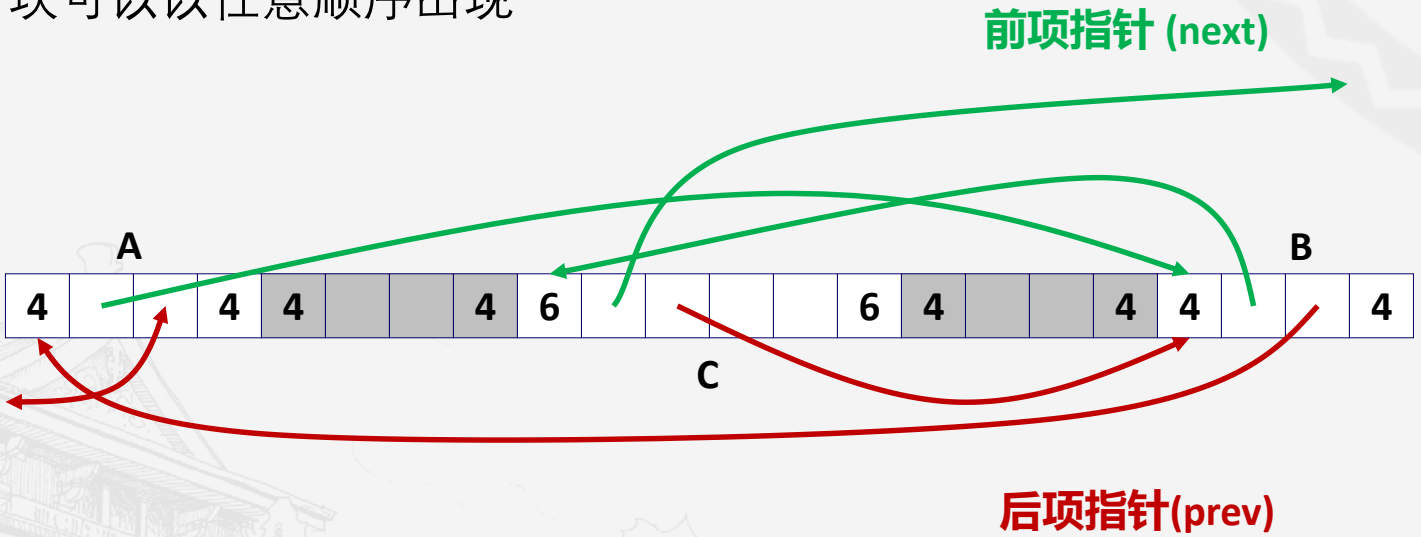
- 维护空闲块的列表，而不是所有块
  - “下一个”空闲块可能位于任何位置
    - 因此，需要存储前向/后向指针，而不仅仅是大小
  - 仍然需要边界标签进行合并
- 幸运的是，我们只跟踪空闲块，所以我们可以使用有效载荷区域

显式空闲链表

Explicit Free Lists

- 逻辑上：

- 物理上：块可以以任意顺序出现



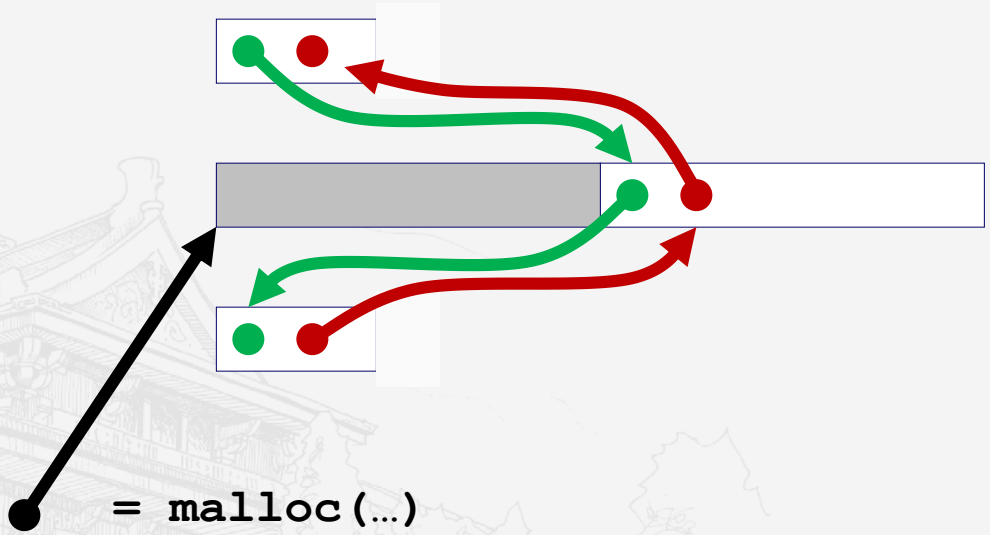
显式空闲链表

Explicit Free Lists

分配空间示意

分配前

分配后



(空闲块拆分)

显式空闲链表

Explicit Free Lists

释放空闲块

- 策略：如何在空闲列表中放置新释放的块的位置

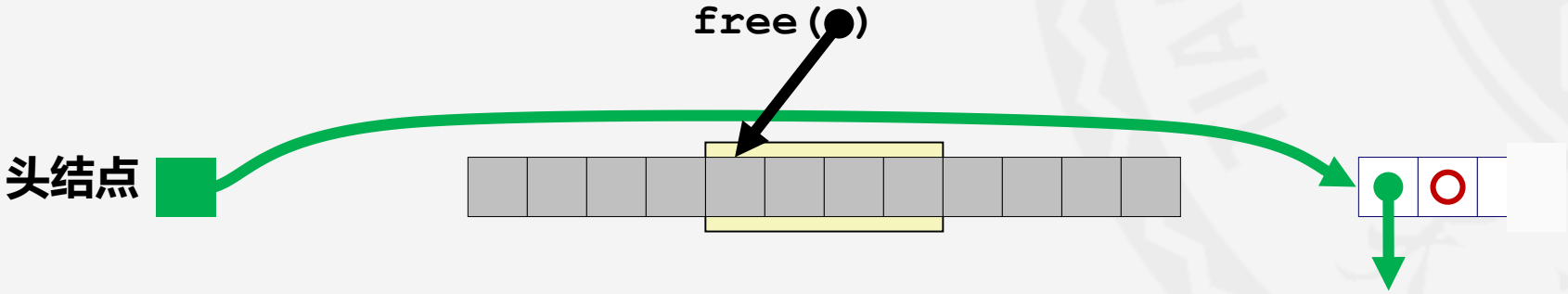
- 后进先出（LIFO）：
  - 将释放的块插入到空闲列表的开头
  - 优点：简单且常数时间
  - 缺点：研究表明碎片化比按地址顺序更严重
- 按地址顺序：
  - 插入释放的块，使得空闲列表块始终按地址顺序排列：addr(prev) < addr(curr) < addr(next)
  - 缺点：需要搜索
  - 优点：研究表明碎片化比LIFO策略更低

显式空闲链表

Explicit Free Lists

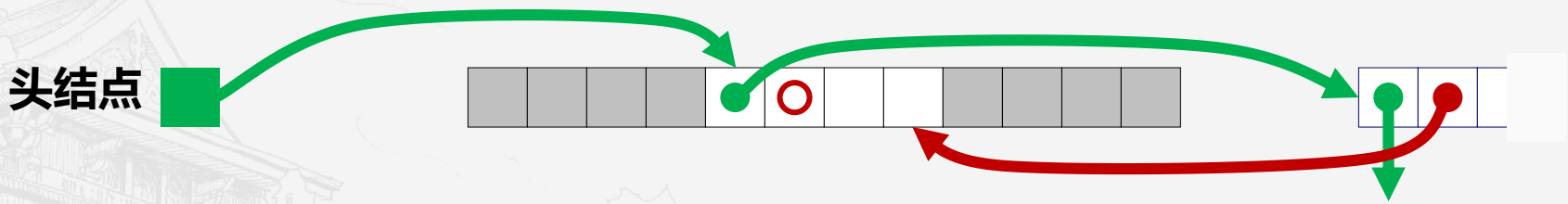
使用后进先出（LIFO）策略释放情况1

释放前



在链表的头结点后插入被释放的块

释放后

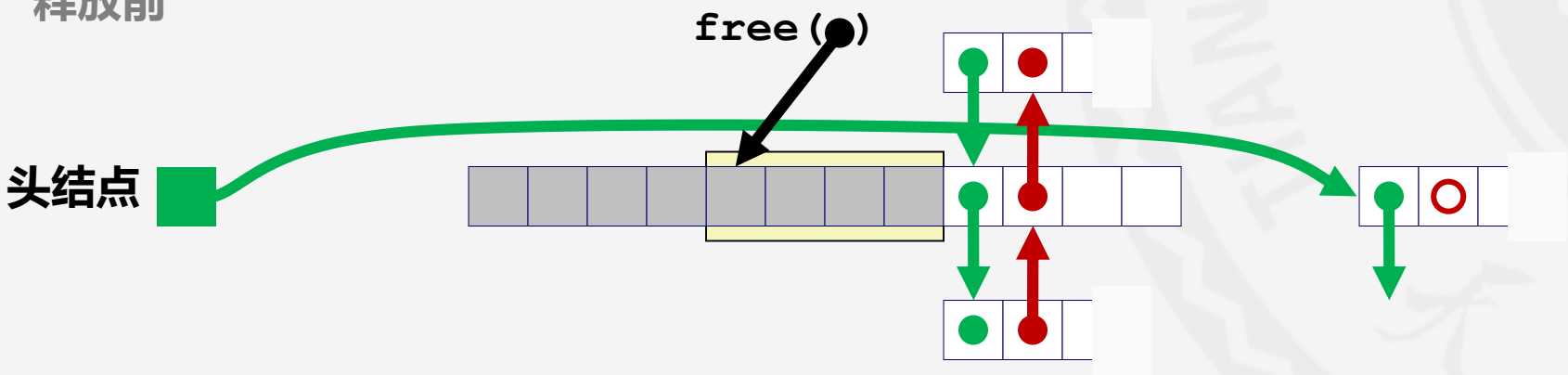


显式空闲链表

Explicit Free Lists

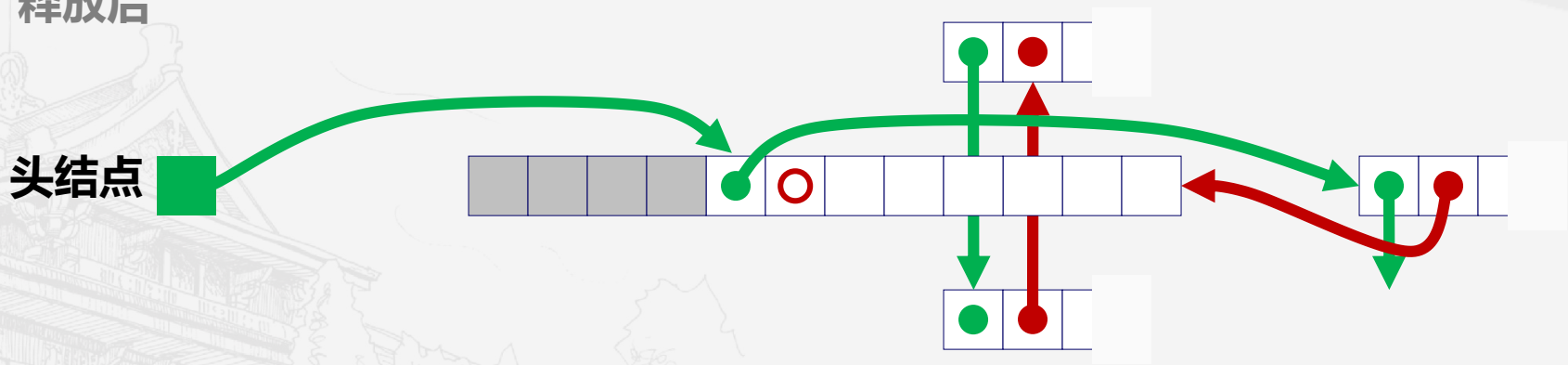
使用后进先出（LIFO）策略释放情况2

释放前



将后继块从链表删除，合并两个空闲块，并将新块插入到头结点后

释放后

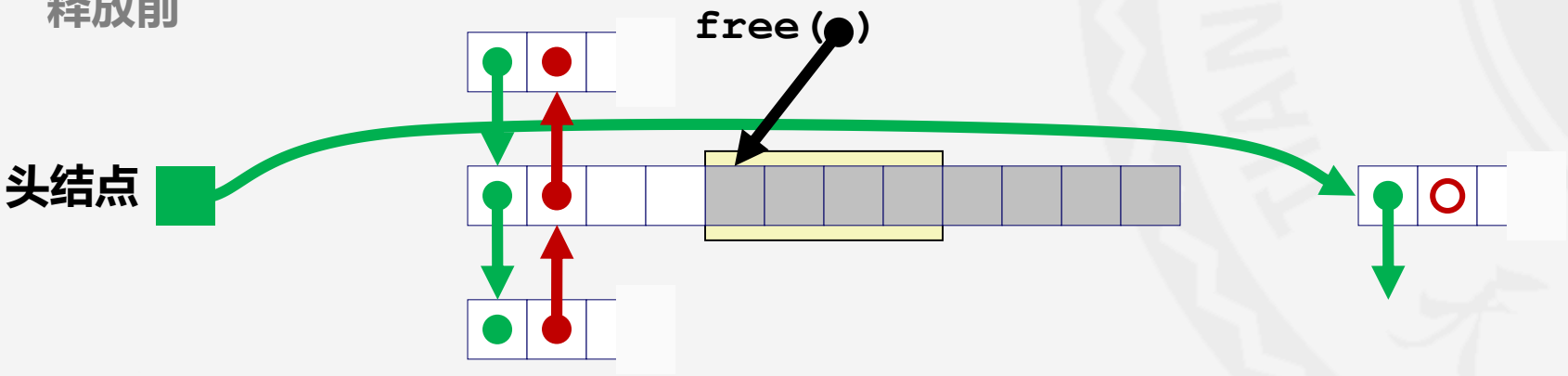


显式空闲链表

Explicit Free Lists

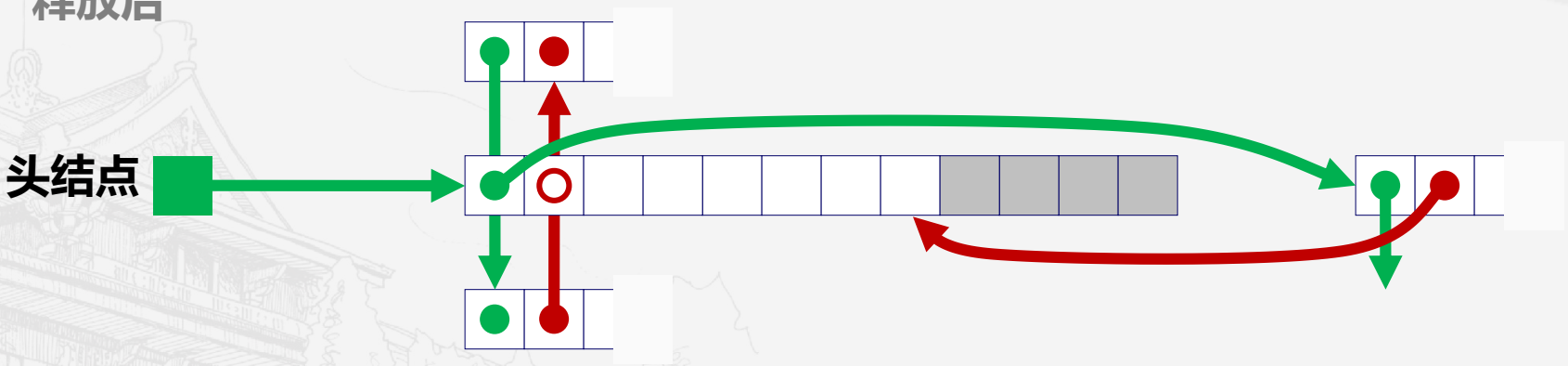
使用后进先出（LIFO）策略释放情况3

释放前



将前继块删除，合并两个空闲块，并将新块插入列表的根节点

释放后

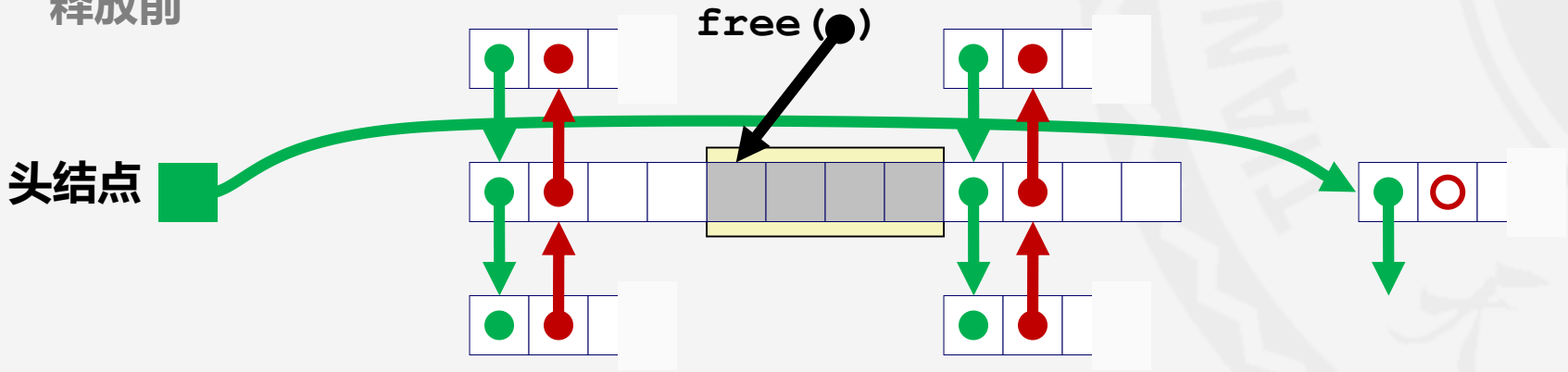


显式空闲链表

Explicit Free Lists

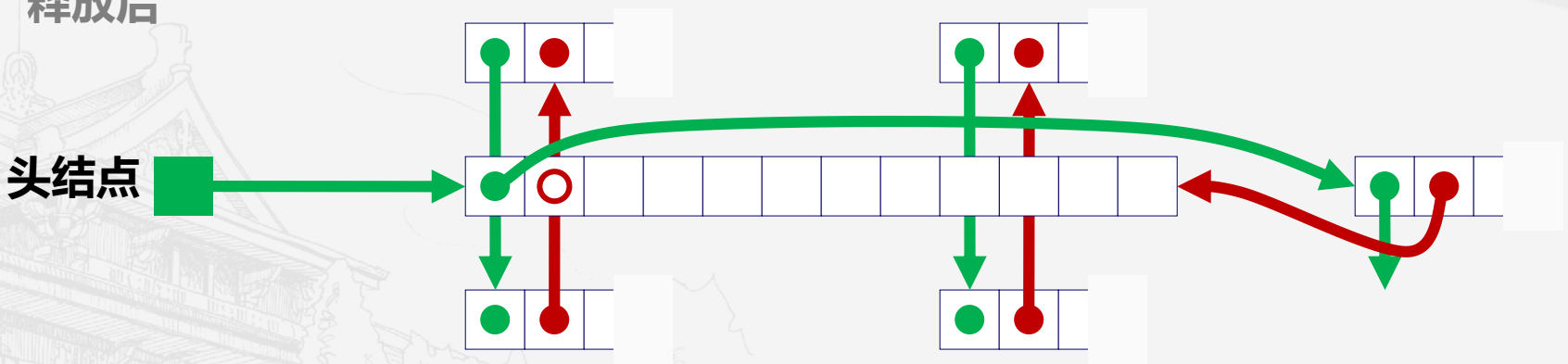
使用后进先出（LIFO）策略释放情况4

释放前



将前继块和后续块同时删除，合并三个空闲块，并将新块插入列表的根节点

释放后



显式空闲链表

Explicit Free Lists

小结

- 与隐式空闲链表相比：
  - 分配时间复杂度：在空闲块的数量的线性时间，而不是在所有块数量上的线性时间
    - 当大部分内存用满时速度更快
  - 分配和释放略复杂，因为需要在链表中删除块
  - 存放显式链接的指针需要额外的空间（每个块需要额外的2个字）
    - 这是否增加了内部碎片？
- 显式空闲链表最常见的用法是与分离的空闲链表结合使用
  - 保持多个不同大小类别的链表，或者可能是不同类型的对象

本章内容

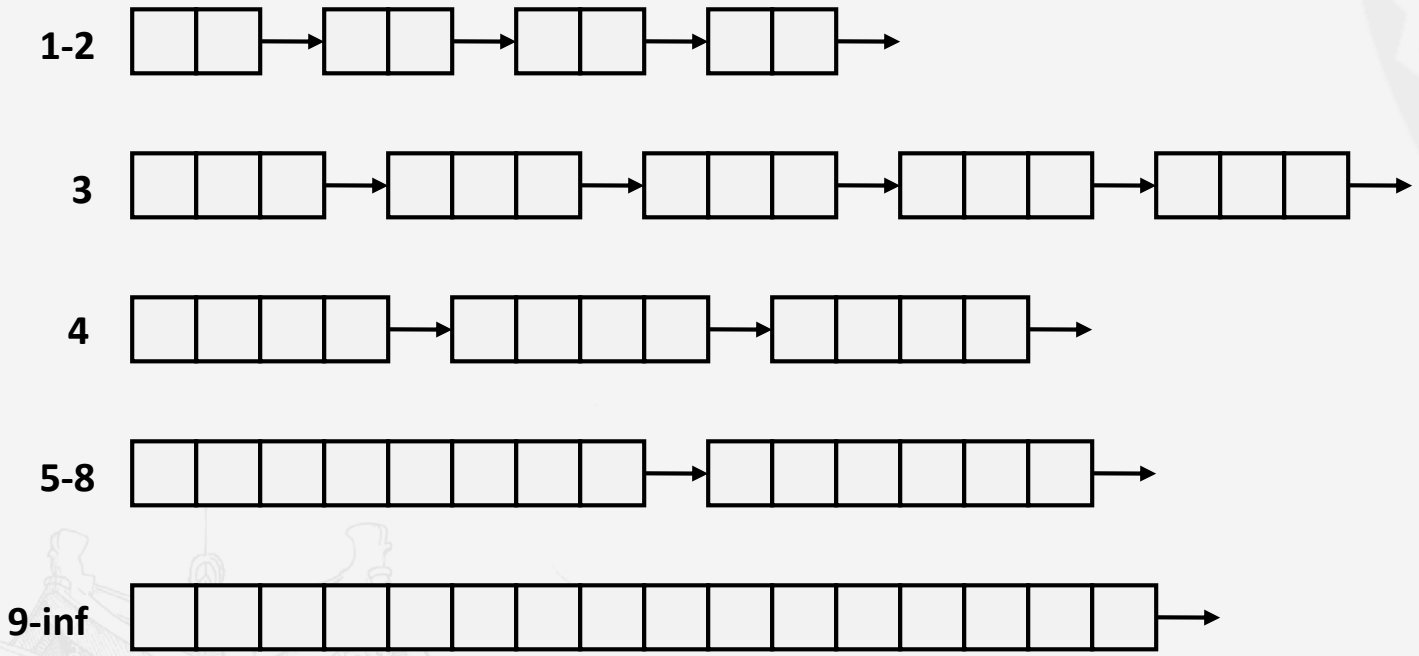
Topic

- 基本概念
- 隐式空闲链表
- 显式空闲链表
- 分离的空闲链表
- 垃圾收集
- 与内存相关的错误

分离的空闲链表

Segregated Free Lists

- 每个大小类别的块都有自己的空闲列表



- 通常为每个小尺寸单独设置类别
- 对于较大尺寸：按照2的幂进行分类

分离的空闲链表

Segregated Free Lists

- 给定一个空闲链表的数组，每个链表对应某个大小类别
- 为了分配大小为 n 的块:
  - 在适当的空闲链表中搜索大小为 m > n 的块
  - 如果找到合适的块:
    - 分配空间，将剩余的空闲部分，放置在适当的链表中（可选）
    - 如果未找到块，则尝试下一个更大的类别
  - 重复直到找到块
- 如果没找到块:
  - 从操作系统请求额外的堆内存（使用 `sbrk()`）
  - 从这个新内存中分配大小为 n 字节的块
  - 将剩余部分作为单个空闲块放置在最大的大小类别中。

分离的空闲链表

Segregated Free Lists

分配器（3）

- 释放一个块：
  - 尝试合并并放置在适当的空闲链表中
- 分离的空闲链表（seglist）分配器的优势
  - 更高的吞吐量：时间复杂度$O(\log_2 n)$
  - 更好的内存利用率
    - 分离的空闲列表的首次适配搜索近似于整个堆的最佳适配搜索
    - 极端情况：为每个块分配自己的大小类别相当于最佳适配。

参考文献

- D. Knuth, “The Art of Computer Programming”, 2nd edition, Addison Wesley, 1973
  - 动态存储分配
- Wilson et al, “Dynamic Storage Allocation: A Survey and Critical Review”, Proc. 1995 Int’l Workshop on Memory Management, Kinross, Scotland, Sept, 1995.
  - 综述

本章内容

Topic

- 基本概念
- 隐式空闲链表
- 显式空闲链表
- 分离的空闲链表
- 垃圾收集
- 与内存相关的错误

隐式的内存管理策略：垃圾收集

- 垃圾回收：自动回收堆分配的存储空间，应用程序无需手动释放

```
void foo() {
   int *p = malloc(128);
   return; /* p block is now garbage */
}
```

- 常见于许多动态语言：
  - Python、Ruby、Java、Perl、ML、Lisp、Mathematica
- 在C和C++语言中也存在着各种变体（“保守型”垃圾收集器）
  - 但是，并不一定能够收集所有的垃圾

垃圾收集

Garbage Collection

- 内存管理器如何知道何时可以释放内存?
  - 一般来说，我们无法知道将来会使用什么
  - 但我们可以确定：如果没有指针指向某些块，那么这些块是无法使用的

- 必须对指针做出一些假设
  - 内存管理器可以区分指针和非指针
  - 所有指针都指向块的起始位置
  - 不能隐藏指针（例如，通过将它们强制转换为int，然后再转换回来）

垃圾收集

Garbage Collection

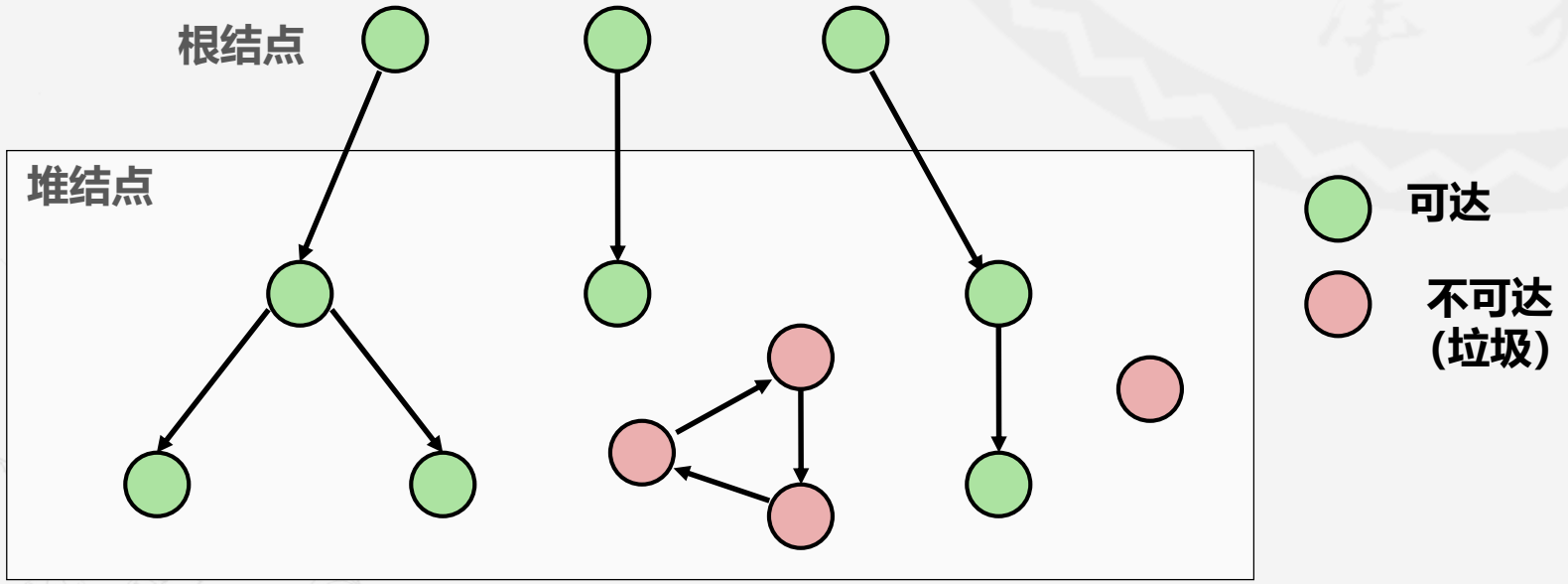
经典的垃圾回收算法

- Mark-and-sweep collection (McCarthy, 1960)
  - 不移动块（除非还需要进行“紧缩”）
- Reference counting (Collins, 1960)
  - 不移动快
- Copying collection (Minsky, 1963)
  - 移动块
- Generational Collectors (Lieberman and Hewitt, 1983)
  - 基于生命周期的收集
    - 大多数分配很快就变成垃圾，因此将回收工作聚焦在最近分配的内存区域上
- 更多信息：Jones and Lin, “Garbage Collection: Algorithms for Automatic Dynamic Memory”, John Wiley & Sons, 1996.

使用有向图表示内存

- 我们将内存视为一个有向图
  - 每个块都是图中的一个节点
  - 每个指针都是图中的一条边
  - 不在堆中但包含指向堆的指针的位置被称为根结点（例如，寄存器、堆栈上的位置、全局变量）

如果从任何根节点到达该节点，则该节点（块）是可达的。不可达的节点是垃圾（应用程序不需要）。

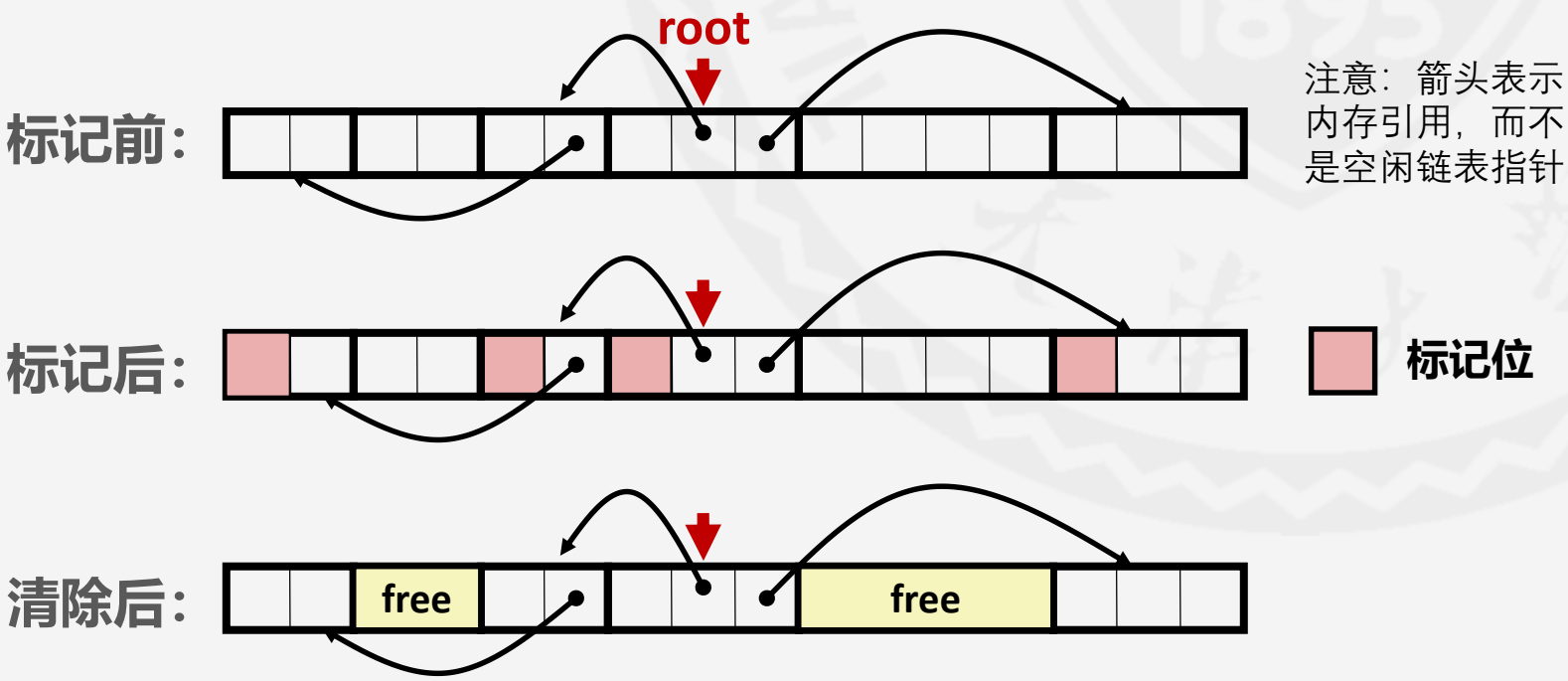


垃圾收集

Garbage Collection

Mark & Sweep垃圾收集器

- 可以在 malloc/free 的基础上实现
  - 使用malloc分配内存，直到“空间用完”为止。
- 当空间用完时：
  - 在每个块的头部使用额外的标记位
  - 标记：从根节点开始，在每个可达块上设置标记位
  - 清除：扫描所有块，并释放未标记的块



垃圾收集

Garbage Collection

以上实现的前提条件

- 应用程序：
  - new(n)：返回一个新块的指针，其中所有位置都已清除
  - read(b, i)：将块b的位置i读入寄存器
  - write(b, i, v)：将v写入块b的位置i
- 每个块都会有一个头部字
  - 地址在b[-1]的位置，其中b为一个块的起始地址
  - 在不同的收集器中用于不同的功能实现
- 垃圾收集器使用的指令：
  - is_ptr(p)：确定p是否为指针
  - length(b)：返回块b的长度，不包括头部
  - get_roots()：返回所有根节点

Mark & Sweep垃圾收集器（2）

使用深度优先遍历内存图进行标记

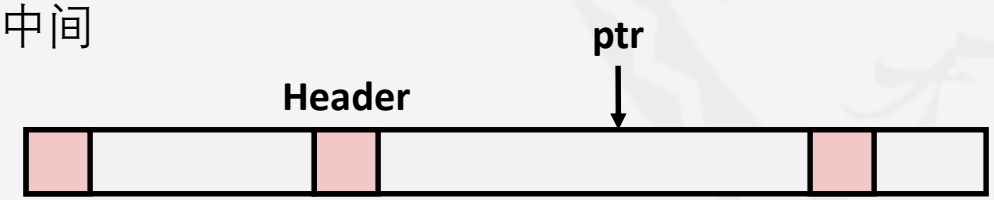
```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;          // do nothing if not pointer
   if (markBitSet(p)) return;       // check if already marked
   setMarkBit(p);                   // set the mark bit
   for (i=0; i < length(p); i++)    // call mark on all words
     mark(p[i]);                    // in the block
   return;
}
```

使用块的长度计算相邻块的位置以进行清除

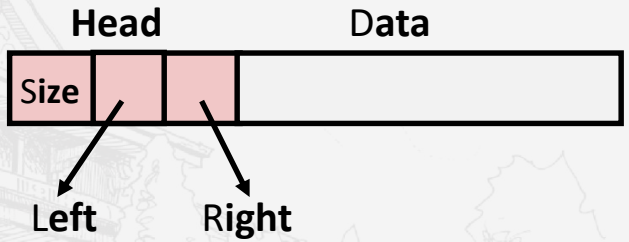
```
ptr sweep(ptr p, ptr end) {
   while (p < end) {
      if markBitSet(p)
         clearMarkBit();
      else if (allocateBitSet(p))
         free(p);
      p += length(p);
}
```

在C语言中实现保守的Mark & Sweep

- 一个用于C程序的“保守型垃圾收集器”
  - is_ptr()通过检查指针是否指向分配的内存块来确定一个字是否是指针
  - 但是，在C中指针可以指向块的中间



- 那么如何找到块的开始呢?
  - 可以使用平衡二叉树来跟踪所有分配的块（键是块的起始位置）
  - 平衡树指针可以存储在头部中（使用两个额外的字）



Left: 低地址
Right: 高地址

本章内容

Topic

- 基本概念
- 隐式空闲链表
- 显式空闲链表
- 分离的空闲链表
- 垃圾收集
- 与内存相关的错误

与内存相关的错误

Memory-related Perils and Pitfalls

- 解引用错误的指针
- 读取未初始化的内存
- 内存溢出
- 引用不存在的变量
- 重复释放堆内存
- 使用已释放的内存
- 内存泄漏

C语言的运算符

运算符	运算顺序
() [] -> .	从左到右
! ~ ++ -- + - * & (type) sizeof	从右到左
* / %	从左到右
+ -	从左到右
<< >>	从左到右
< <= > >=	从左到右
== !=	从左到右
&	从左到右
^	从左到右
|	从左到右
&&	从左到右
||	从左到右
?:	从右到左
= += -= *= /= %= &= ^= != <<= >>=	从右到左
,	从左到右

->、() 和 [] 具有最高的优先级，其次是 * 和 &
单目 +、- 和 *运算 比 双目形式具有更高的优先级

与内存相关的错误

Memory-related Perils and Pitfalls

C语言指针的声明

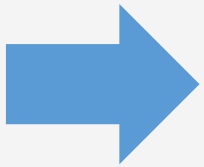
`int *p`	p是指向int类型的指针
`int *p[13]`	p是包含13个int型指针的数组
`int *(p[13])`	与`int *p[13]`含义相同
`int **p`	p是一个指向int型指针的指针
`int (*p)[13]`	p是一个指向包含13个int型元素数组的指针
`int *f()`	f是一个函数，返回至为一个int型指针
`int (*f)()`	f是一个指向函数的指针，函数返回类型为int
`int (*(*f())[13])()`	f是一个函数，返回一个指向包含13个函数（返回类型为int）指针的数组的指针
`int (*(*x[3])())[5]`	x是一个包含3个函数指针的数组，函数的返回值是指向包含5个int元素的数组指针

解引用错误的指针

- 经典的 scanf 函数的错误使用场景

```
int val;

...

scanf("%d", val);
```

X



```
int val;

...

scanf("%d", &val);
```

√

读取未初始化的内存

- 误以为堆中分配的数据已初始化为0

```
/* return y = Ax */
int *matvec(int **A, int *x) {
   int *y = malloc(N*sizeof(int));
   int i, j;

   for (i=0; i<N; i++)
      for (j=0; j<N; j++)
         y[i] += A[i][j]*x[j];
   return y;
}
```

与内存相关的错误

Memory-related Perils and Pitfalls

内存溢出

- 为指针分配了错误的大小

```
int **p;

p = malloc(N*sizeof(int));

for (i=0; i<N; i++) {
   p[i] = malloc(M*sizeof(int));
}
```

X

```
int **p;

p = malloc(N*sizeof(int *));

for (i=0; i<=N; i++) {
   p[i] = malloc(M*sizeof(int));
}
```

√

内存溢出（2）

- 没有检查缓冲区长度

```
char s[8];
int i;

gets(s);  /* reads “123456789” from stdin */
```

- 经典的缓冲区溢出攻击

内存溢出（3）

- 指针运算的错误使用

```
int *search(int *p, int val) {

   while (*p && *p != val)
      p += sizeof(int);

   return p;
}
```

内存溢出（4）

- 引用指针而不是它所指向的对象

```
int *BinheapDelete(int **binheap, int *size) {
   int *packet;
   packet = binheap[0];
   binheap[0] = binheap[*size - 1];
   *size--;
   Heapify(binheap, *size, 0);
   return(packet);
}
```

引用不能存在的变量

- 忘记局部变量在函数返回时消失

```
int *foo () {
    int val;

    return &val;
}
```

重复释放堆内存

```
x = malloc(N*sizeof(int));
        <manipulate x>
free(x);

y = malloc(M*sizeof(int));
        <manipulate y>
free(x);
```

使用已释放的内存

```
x = malloc(N*sizeof(int));
  <manipulate x>
free(x);
   ...
y = malloc(M*sizeof(int));
for (i=0; i<M; i++)
   y[i] = x[i]++;
```

内存泄漏

```
foo() {
    int *x = malloc(N*sizeof(int));
    ...
    return;
}
```

内存泄漏（2）

```
struct list {
    int val;
    struct list *next;
};

foo() {
    struct list *head = malloc(sizeof(struct list));
    head->val = 0;
    head->next = NULL;
    <create and manipulate the rest of the list>
     ...
    free(head);
    return;
}
```

修正内存访问错误的方法

- 使用GDB调试：容易发现坏指针，但是很难发现其他的内存错误
- 使用断言函数 assert
  - 在运行时静默运行，只在出现错误时打印消息
  - 作为定位错误的探测器使用
- 二进制翻译器：valgrind
  - 强大的调试和分析技术
  - 重写可执行目标文件的.text section
  - 在运行时检查每个单独的引用：检查坏指针、覆盖、超出分配块的引用
- 使用glibc提供的malloc检查代码：设置环境变量 **`setenv MALLOC_CHECK_ 3`**