

# **Virtual Memory (III)**

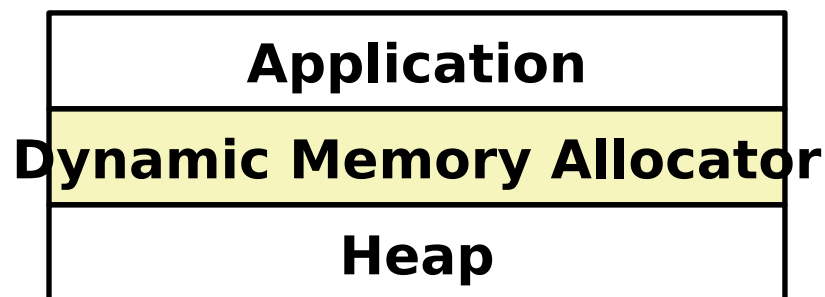
## **Dynamic Memory Allocation**

### **(CSAPP e3 9-9)**

# Dynamic Memory Allocation

---

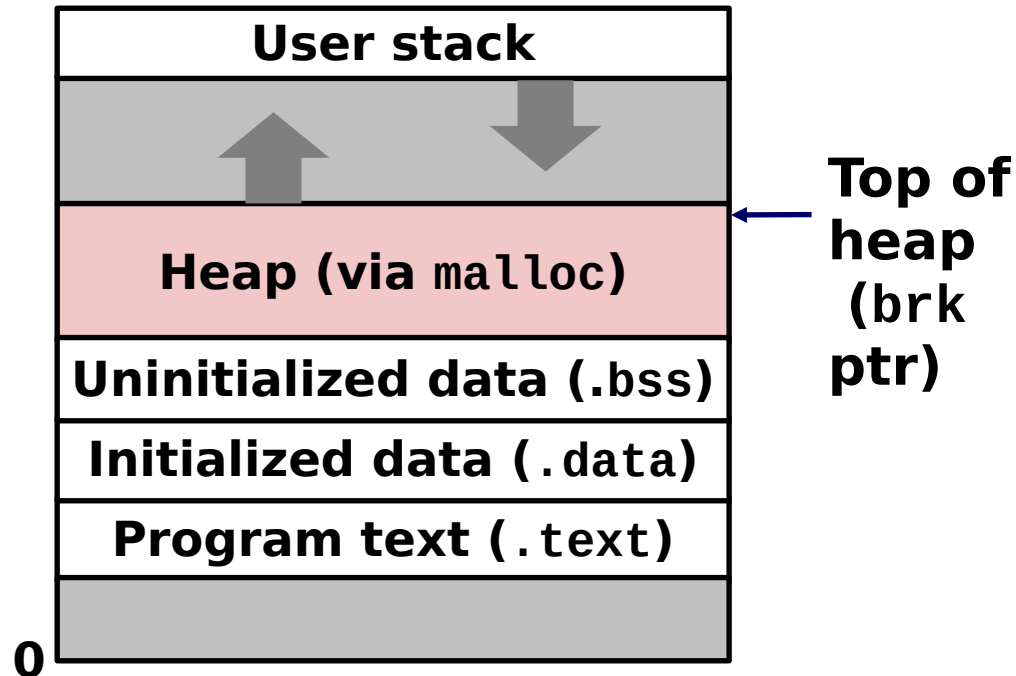
- Programmers use *dynamic memory allocators* (such as `malloc`) to acquire VM at run time.
  - 尤其是一些数据结构，只能在运行时才能知道其大小



# Dynamic Memory Allocation

---

- Dynamic memory allocators manage an area of process virtual memory known as the *heap*.



# Dynamic Memory Allocation

---

- Heap :
  - 进程中一段连续的虚拟地址空间
    - 有 page table 时，物理未必连续
  - Memory Allocator 把 heap 当作一组可变大小的 blocks ， blocks 或者是 allocated 状态，或者是 free 状态
  - 与数据结构“堆” (heap) 没有关系，这里是堆积、大量的意思

# Dynamic Memory Allocation

---

- Memory Allocator 类型
  - **Explicit allocator**: application allocates and frees space
    - E.g., malloc and free in C
  - **Implicit allocator**: application allocates, but does not free space
    - E.g. garbage collection in Java, ML, and Lisp

# The malloc Function

---

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

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

- Successful:
  - Returns a pointer to a memory block of at least **size** bytes **aligned to** an 8-byte (x86) or 16-byte (x86-64) boundary
  - If **size == 0**, returns NULL
  - ( 如果需要更严格的对其，例如 4K 对齐，需要用 memalign 函数 )
- Unsuccessful: returns NULL (0) and sets **errno**

# The free Function

---

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

- Returns the block pointed at by **p** to the pool of available memory
- **p** must come from a previous call to **malloc** or **realloc**

Other functions

- **calloc**: Version of **malloc** that initializes allocated block to zero.
- **realloc**: Changes the size of a previously allocated block.
- **sbrk**: Used internally by allocators to grow or shrink the heap

# malloc Example

```
#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);
}
```

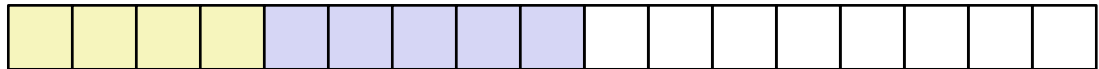
# Allocation Example

---

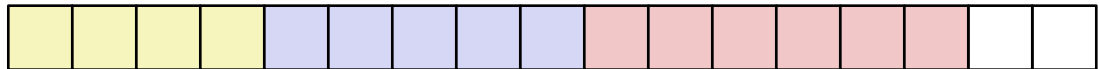
`p1 = malloc(4)`



`p2 = malloc(5)`



`p3 = malloc(6)`



`free(p2)`



`p4 = malloc(2)`



# Constraints

---

- Applications

- 可以任意调用 malloc 和 free 请求，形成任意顺序的 malloc 和 free 序列
- Free 请求必须针对一个已经 malloc 过的 block （也要避免 double free ）

- Allocators

- 不能控制分配 blocks 的数量或大小（由 application 决定）
- 必须立刻响应 malloc 请求
  - 不能通过 reorder 或 buffer 请求来优化性能
- 必须从 free 空间中分配 blocks
  - 不能覆盖已有内容
- 必须对齐分配的 blocks ，满足对齐要求
  - 8-byte (x86) or 16-byte (x86-64) alignment on Linux boxes
- 只能操作和修改空闲内存
- 不能移动已经分配的 blocks
  - 例如，不能做 compaction 操作（为了不影响性能）

# Performance Goal #1: Throughput

---

- Given some sequence of `malloc` and `free` requests:
  - $R_0, R_1, \dots, R_k, \dots, R_{n-1}$
- Goals: maximize **throughput** and peak **memory utilization**
  - These goals are often **conflicting**
- Throughput:
  - Number of completed requests per unit time
  - Example:
    - 5,000 `malloc` calls and 5,000 `free` calls in 10 seconds
    - Throughput is 1,000 ops

# Performance Goal #2: Memory Utilization

---

- Given some sequence of malloc and free requests:
  - $R_0, R_1, \dots, R_k, \dots, R_{n-1}$
- *aggregate payload*  $P_k$ 
  - `malloc(p)` results in a block with a **payload** of  $p$  bytes
  - After request  $R_k$  has completed, the **aggregate payload**  $P_k$  is the **sum** of currently **allocated** payloads
- *Current heap size*  $H_k$ 
  - Assume  $H_k$  is monotonically **nondecreasing** (不变或增长)
    - i.e., heap only grows when allocator uses `sbrk`
- *Peak memory utilization after  $k+1$  requests*
  - $U_k = (\max_{i \leq k} P_i) / H_k$

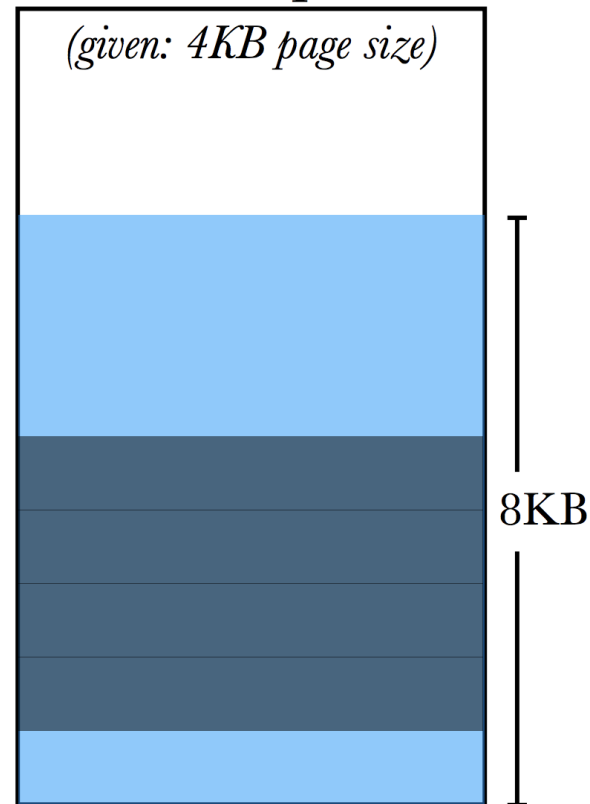
# Performance Goal: Peak Memory Utilization

---

- *Peak memory utilization* ) 例子

Heap

```
p1 = malloc(1024);  
// util = 1K/4K = 25%  
p2 = malloc(2048);  
// util = 3K/4K = 75%  
free(p1);  
// util = 2K/4K = 50%  
p3 = malloc(2048);  
// util = 4K/8K = 50%  
free(p3);  
// util = 2K/8K = 25%  
free(p2);  
// util = 0/8K = 0%  
  
// all non-leaking  
// programs end in 0%
```



# Fragmentation

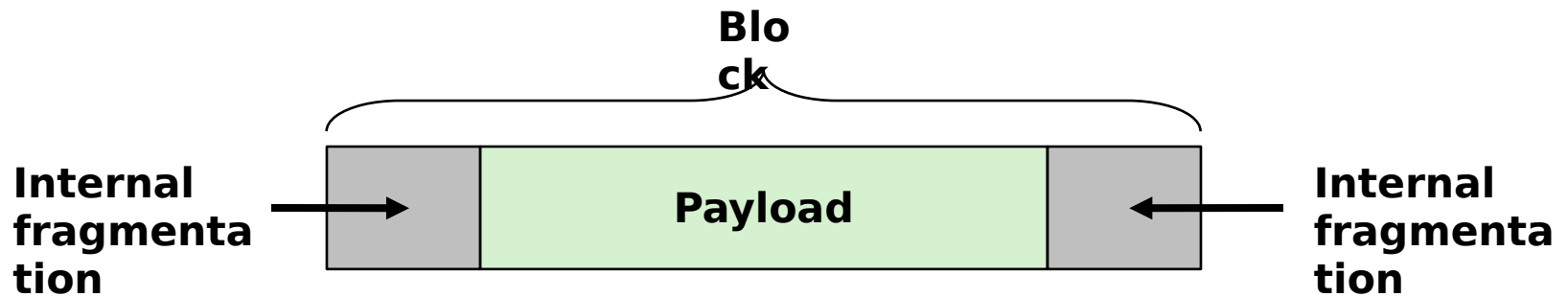
---

- Poor memory utilization caused by *fragmentation*
  - *internal* fragmentation
  - *external* fragmentation

# Internal Fragmentation

---

- For a given block, *internal fragmentation* occurs if payload is smaller than block size



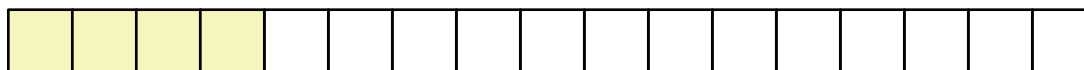
当 allocator 必须按 16B 对其分配内存时，`malloc(n)`，`n` 不能被 16 整除时就会产生 internal fragmentation

# External Fragmentation

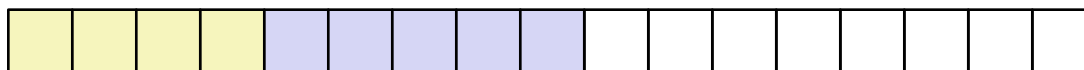
---

- Occurs when there is enough aggregate heap memory, but no single free block is large enough

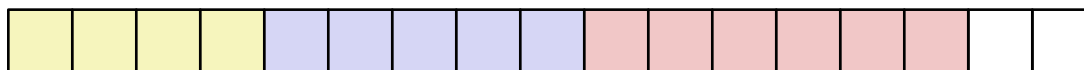
`p1 = malloc(4)`



`p2 = malloc(5)`



`p3 = malloc(6)`



`free(p2)`



`p4 = malloc(6)`

***Oops! (what would happen now?)***

- Depends on the pattern of future requests
  - Thus, difficult to measure ( 以上情况可能是碎片，也可能不是 )

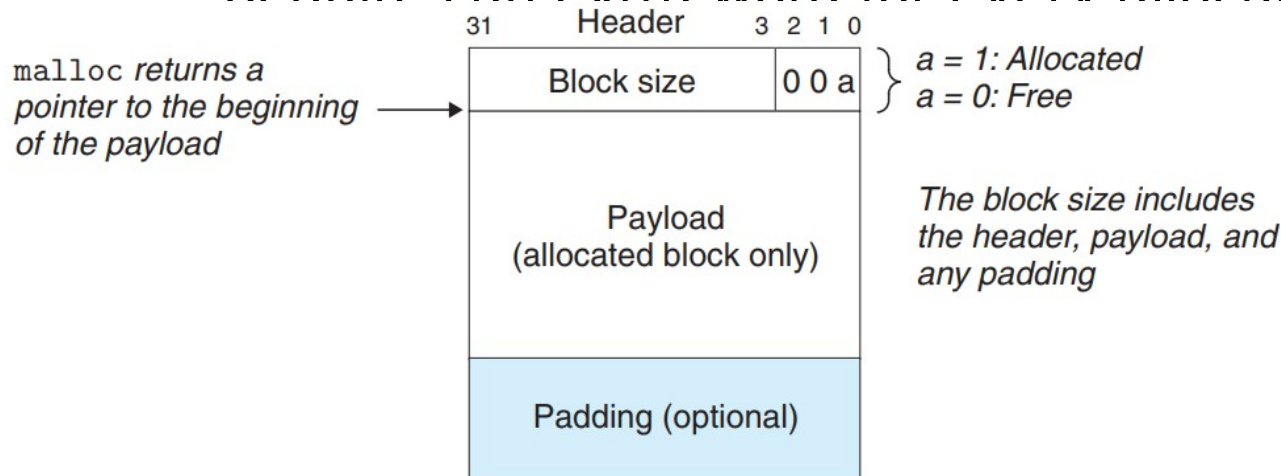
# Implementation Issues

---

- #1 Block 是变长的，如何记录各个 block 的 size?
- #2 free block 不一定连续，如何管理所有的 free blocks?

# Issue #1: Block metadata

- Standard method
  - Keep the length of a block in the word preceding the block.
    - This word is often called the **header field** or **header**
  - Requires an extra word for every allocated block

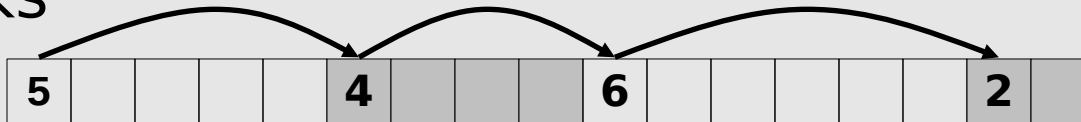


因为 **block size** 是 **16B aligned**，最低 **4bit** 总是 **0**，因此可以用来存一些标记位

每个 **block** 至少占用 **2 个 word**

## Issue #2: Keeping Track of Free Blocks

- Method 1: *Implicit list* using length—links all blocks



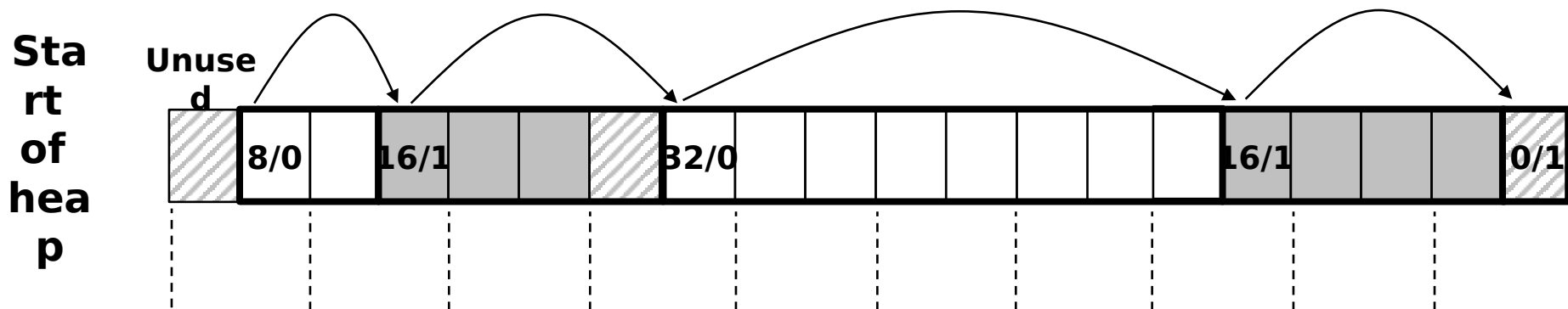
- Method 2: *Explicit list* among the free blocks using pointers



- Method 3: *Segregated free list*
  - Different free lists for different size classes
- Method 4: *Blocks sorted by size*
  - Can use a balanced tree (e.g. Red-Black tree) with

# Issue #2: Keeping Track of Free Blocks

## Method 1: *Implicit list*



- Double-word alignment
  - Allocated blocks: shaded
  - Free blocks: unshaded
  - Headers: labeled with size in bytes/allocated bit
- 32 位机器，一个 word 是 32 位 (4 字节)

# 课堂练习

---

确定下面的 malloc 请求序列得到的块大小和头部值。假设：1)分配器保持双字对齐，使用隐式空闲链表，以及图 9-35 中的块格式。2)块大小向上舍入为最接近的 8 字节的倍数。

请求	块大小 (十进制字节)	块头部 (十六进制)
malloc(3)		
malloc(11)		
malloc(20)		
malloc(21)		

# 练习答案

---

Request	Block size (decimal bytes)	Block header (hex)
<code>malloc(3)</code>	8	0x9
<code>malloc(11)</code>	16	0x11
<code>malloc(20)</code>	24	0x19
<code>malloc(21)</code>	32	0x21

# Implicit List: Finding a Free Block

---

- *First fit:*

- 从头搜索 list，选择第一个符合要求的 free block

```
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)
```

- 线性复杂度，与总块数（包括 allocated 和 free blocks）一个数量级
- 实际操作中，可能在 list 开始的部分导致很多碎片

- *Next fit:*

- 操作类似于 first fit，只是从上一次搜索结束的地方开始下一次搜索
- 均匀使用整个 list
- 可能比 first fit 更快（因为后面有一些较大的空闲块，而 first fit 开头部分很难满足）
- 有一些研究说，其碎片化现象更严重

# Implicit List: Finding a Free Block

---

- *Best fit:*
  - 搜索整个 list ，找到所有符合条件的 free blocks ，然后选择最小的一个
  - 尽可能保留大的 free blocks ，一般能够提高内存空间利用率
  - 搜索整个 list ，所以会比 first fit 更慢一些
- *Worst fit:*
  - 与 Best fit 操作类似，但策略正相反，返回最大的 free block
  - 能够避免 best fit 带来的小碎片
  - 但一般表现很差，碎片化严重
  - 性能与 best fit 类似

# Implicit List: Finding a Free Block

---

- 例子



- 分配 15 ， best fit ， 在 20 中分配（缺点是剩余的 5 有点小，可能不满足后续分配需求）



- 如果选择 worst fit ， 结果如下：



# Implicit List: Finding a Free Block

---

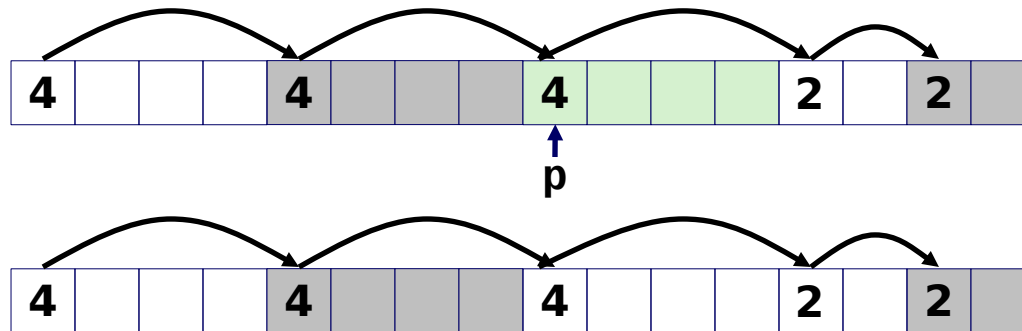
- 如果无法找到合适的 free block
  - 即使总的空闲空间够用
    - Fail , 返回 NULL
    - 调用 sbrk() 系统调用申请增大 heap 区, 增加新的物理内存, 映射到新扩大的 heap 区 VA



# Implicit List: Freeing a Block

- Simplest implementation:
  - Need only clear the "allocated" flag
  - But can lead to "false fragmentation"

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



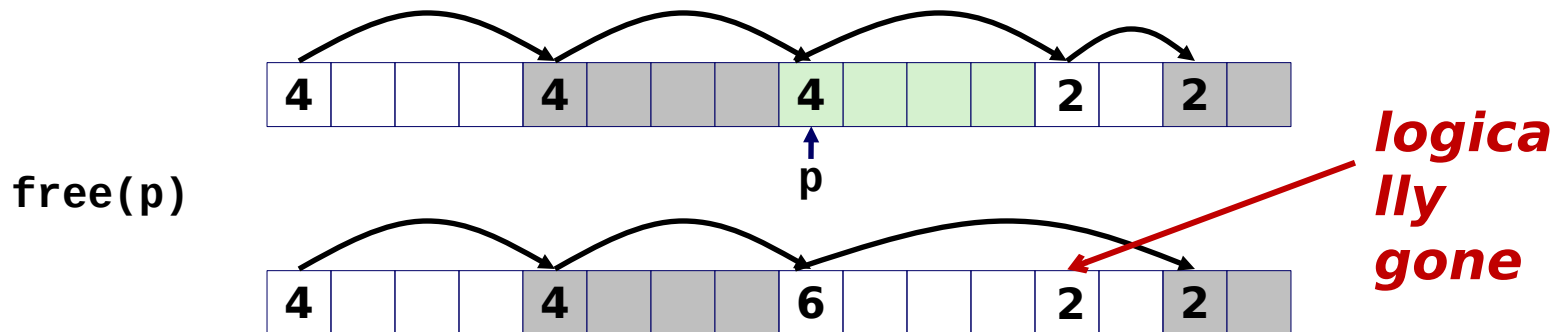
malloc(5) **Oops**

!

***There is enough free space, but the allocator won't be able to find it***

# Implicit List: Coalescing (合并)

- Merge (*coalesce*) with next/previous blocks, if they are free
  - Coalescing with next block immediately or later



Coalescing the next block is

easy:

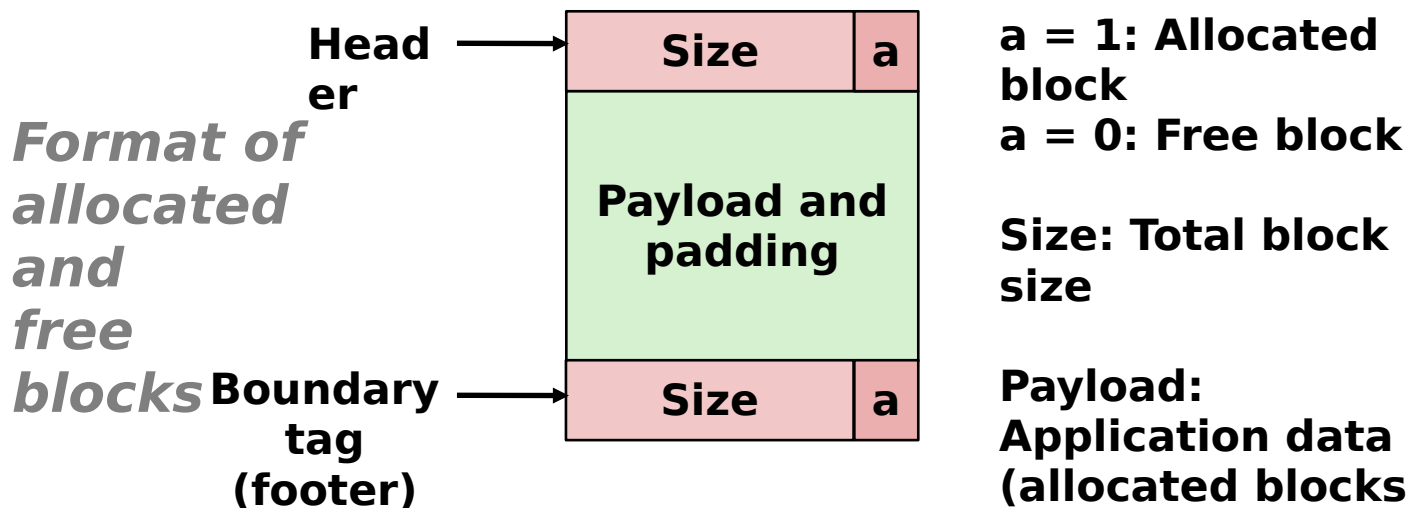
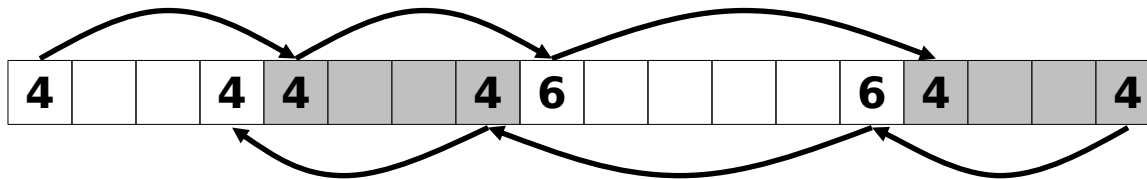
```
void free_block(ptr p) {  
    *p = *p & -2;           // clear allocated flag  
    next = p + *p;         // find next block  
    if ((*next & 1) == 0)  
        *p = *p + *next;   // add to this block if  
                            // not allocated  
}
```

But how do we coalesce with *previous* block?

# Implicit List: Bidirectional Coalescing

- *Boundary tags*

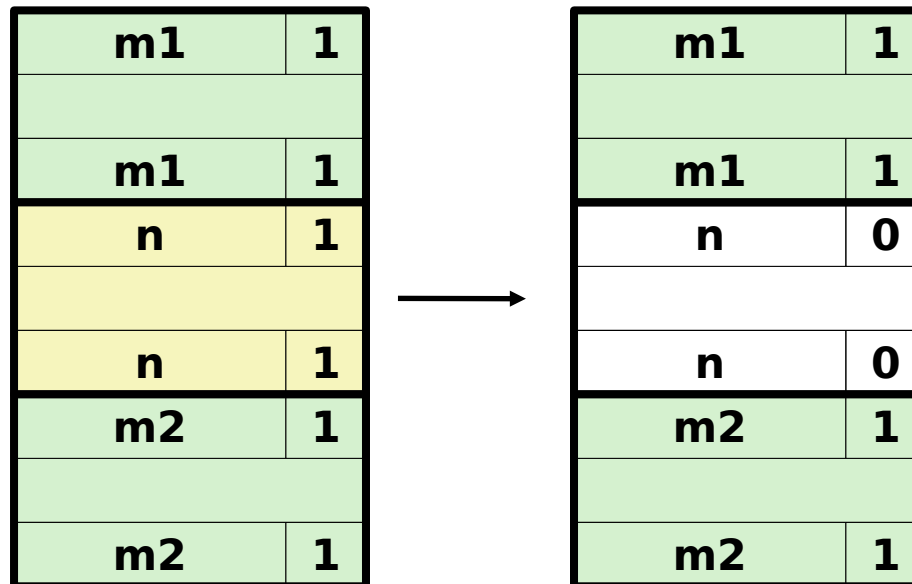
- Replicate size/allocated word at “bottom” (end) of free blocks
- Allows us to traverse the “list” backwards, but requires extra space
- Important and general technique!



# Constant Time Coalescing (Case 1)

---

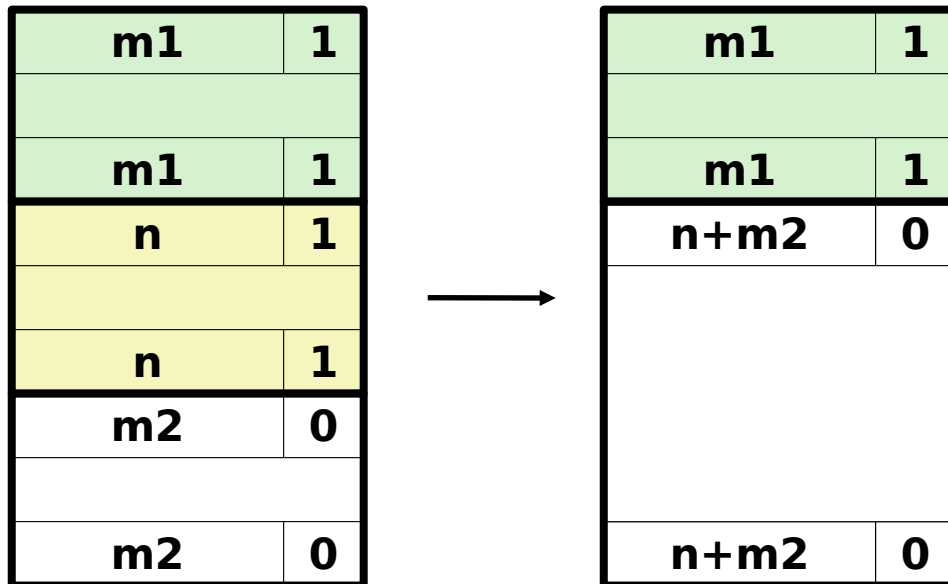
both previous and next blocks are allocated



# Constant Time Coalescing (Case 2)

---

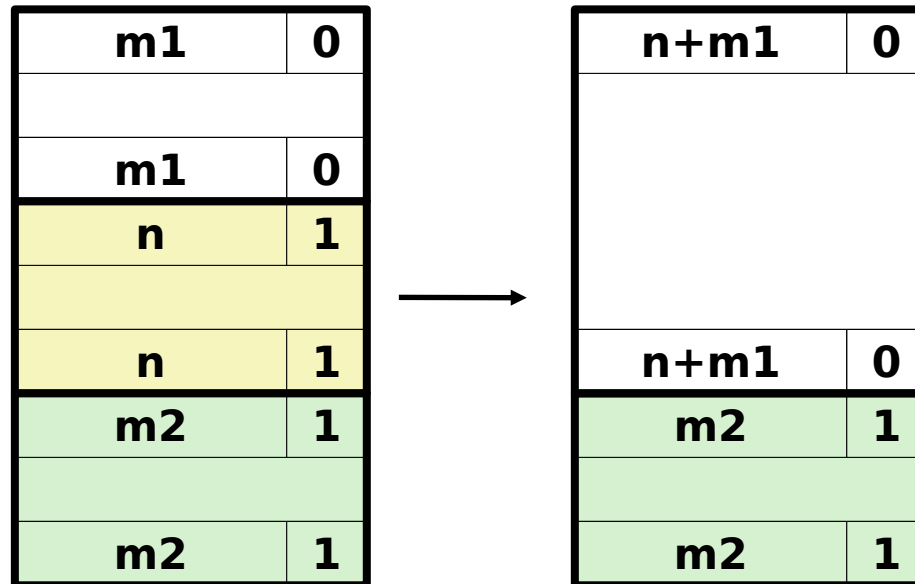
previous block is allocated, next block is free



# Constant Time Coalescing (Case 3)

---

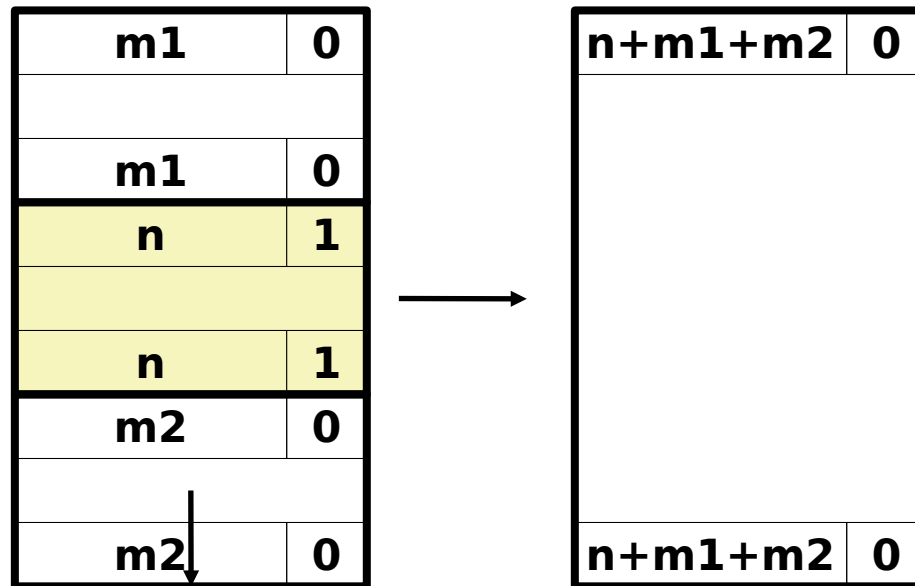
previous block is free, next block is allocated



# Constant Time Coalescing (Case 4)

---

both previous and next blocks  
are free



# Disadvantages of Boundary Tags

---

- 占用更多的内存空间
- Can it be optimized?
  - Which blocks need the footer tag?
    - 当前一块是空闲块时，它才需要有 footer
  - What does that mean?
    - 当前 block 的 Header 中增加一位记录前一块是否空闲
    - 空闲块还是有 footer ，不浪费
    - 已分配块不需要 footer
    - 已分配块释放时，除了看下一个块的 header ，可以往回看上一个块的 footer

# Summary of Key Allocator Policies

---

- Placement policy:
  - First-fit, next-fit, best-fit, etc.
  - Trades off lower throughput for less fragmentation
- Splitting policy:
  - When do we go ahead and split free blocks?
  - How much internal fragmentation are we willing to tolerate?
- Coalescing policy:
  - **Immediate coalescing:** coalesce each time free is called
  - **Deferred coalescing:** try to improve performance of free by deferring coalescing until needed. Examples:
    - Coalesce as you scan the free list for **malloc**
    - Coalesce when the amount of external fragmentation **reaches some threshold**

# Implicit Lists: Summary

---

- Implementation: very simple
- Allocate cost:
  - linear time worst case
- Free cost:
  - constant time worst case, even with coalescing
- Memory usage:
  - will depend on placement policy
  - First-fit, next-fit or best-fit
- Not used in practice for `malloc/free` because of linear-time allocation
  - used in many special purpose applications
- However, the concepts of **splitting** and **boundary tag coalescing** are general to *all* allocators

## 课堂练习

---

- 隐式空闲链表，不允许有效负载为 0，字长 4 字节，header 和 footer 都是 4 字节。
  - 对齐：指整个块大小，包括 head, footer 在内

对齐要求	已分配的块	空闲块	最小块大小 (字节)
单字	头部和脚部	头部和脚部	
单字	头部，但是无脚部	头部和脚部	
双字	头部和脚部	头部和脚部	
双字	头部，但是没有脚部	头部和脚部	

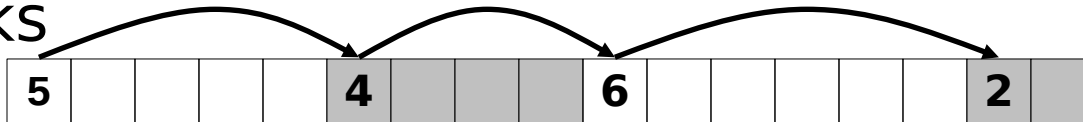
# 练习答案

---

对齐要求	已分配块	空闲块	最小块大小（字节）
单字	头部和脚部	头部和脚部	12
单字	头部, 但是没有脚部	头部和脚部	8
双字	头部和脚部	头部和脚部	16
双字	头部, 但是没有脚部	头部和脚部	8

# Keeping Track of Free Blocks

- Method 1: *Implicit free list* using length—links all blocks



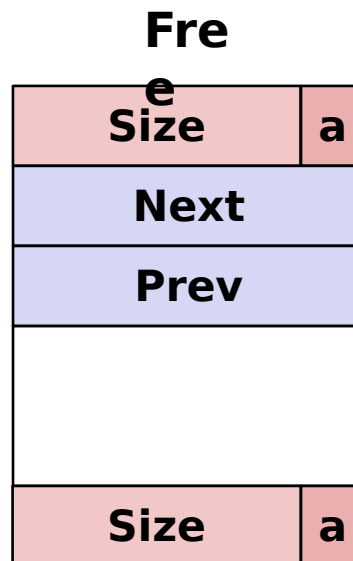
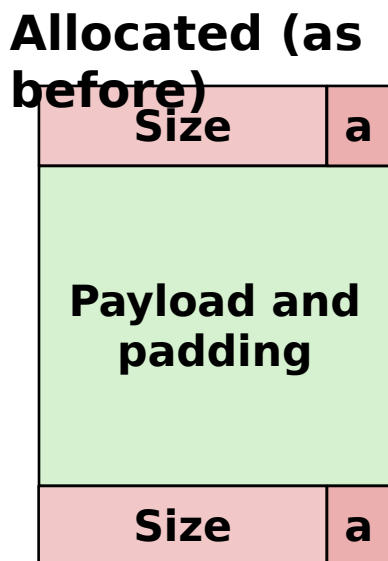
- Method 2: *Explicit free list* among the free blocks using pointers



- Method 3: *Segregated free list*
  - Different free lists for different size classes
- Method 4: *Blocks sorted by size*
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

# Explicit Free Lists

---



- Maintain list(s) of *free* blocks, not *all* blocks
  - "next" free block 可能 anywhere
    - 不像之前 implicit list 中 next 是在相邻的内存
    - 所以必须有 forward/backward pointers, 不能只是记录 size
  - 仍然需要 boundary tag, 用于合并相邻 block
  - 以上需求只针对 free block, 所以不需要占用有效内存空间

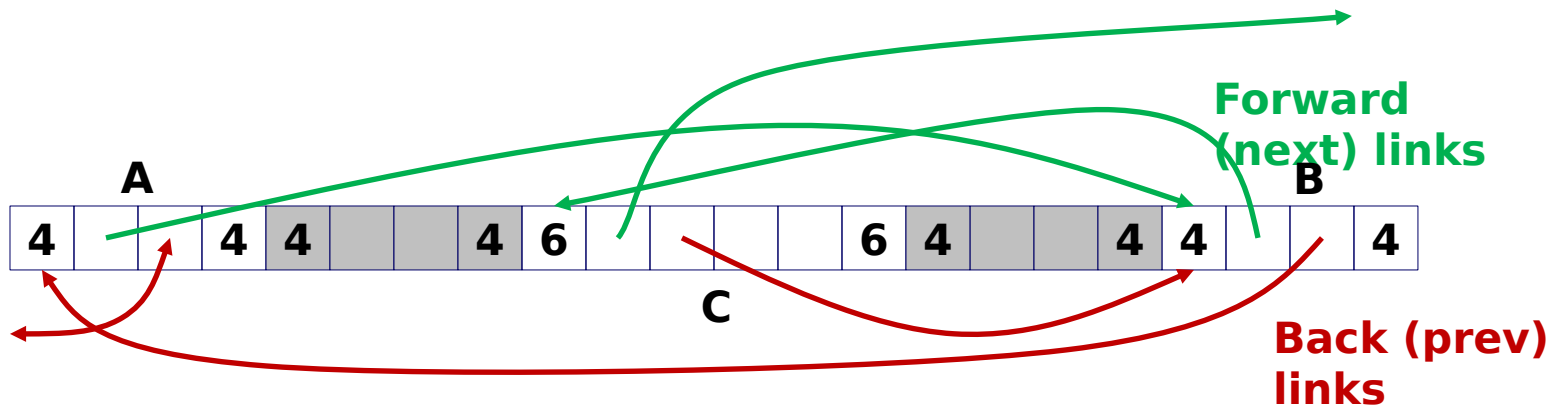
# Explicit Free Lists

---

- Logically:

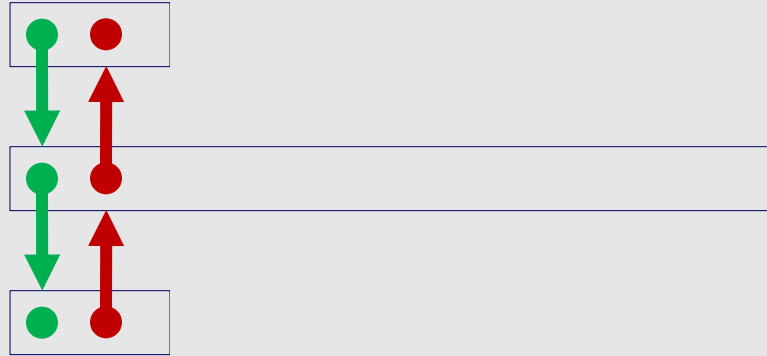


- Physically: blocks can be in any order

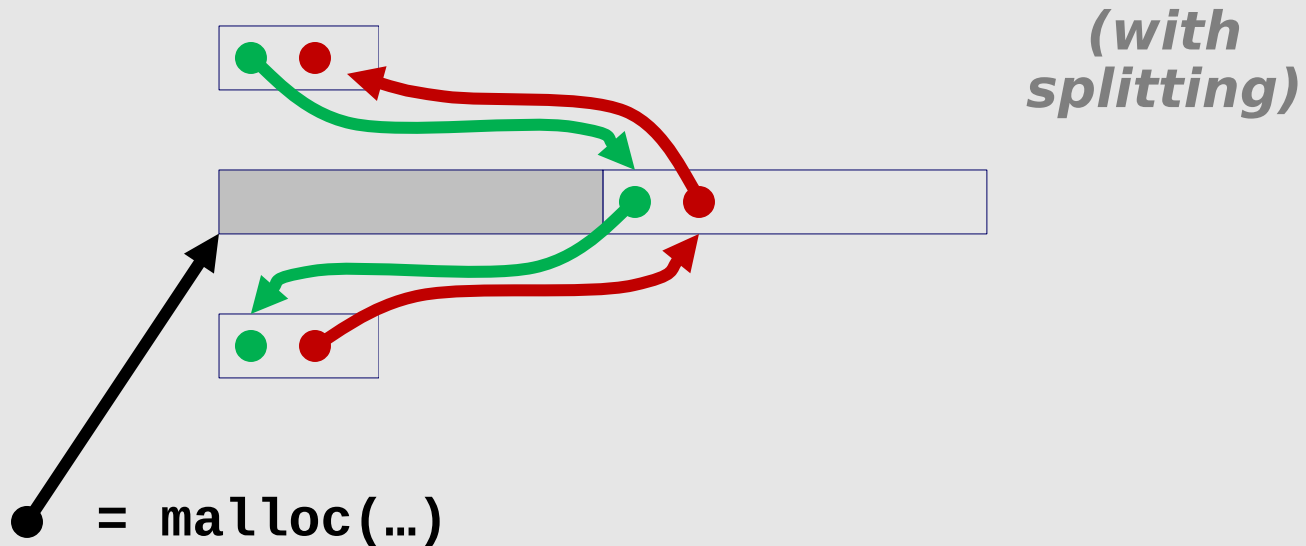


# Allocating From Explicit Free Lists

*Before*



*After*

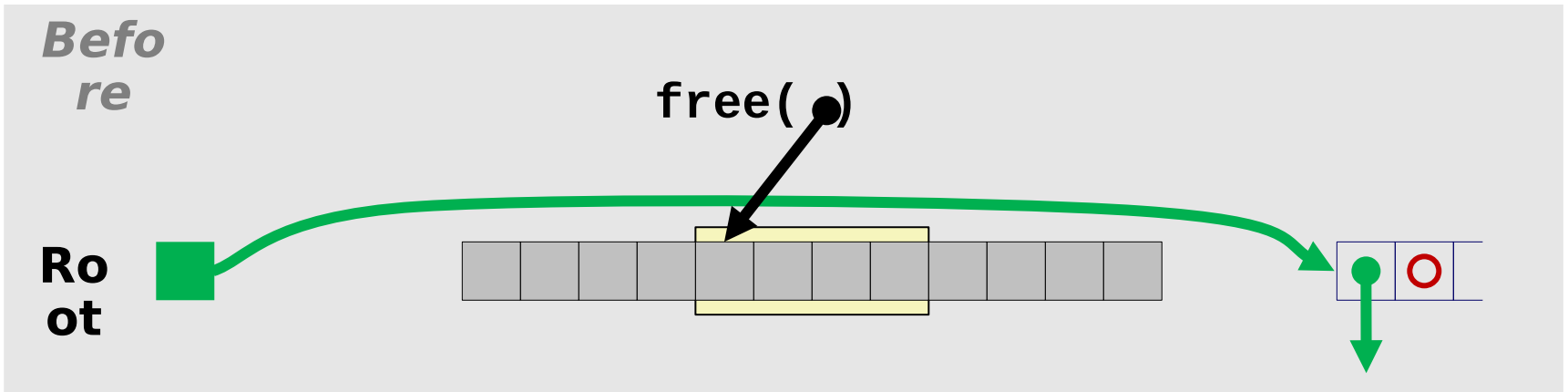


# Freeing With Explicit Free Lists

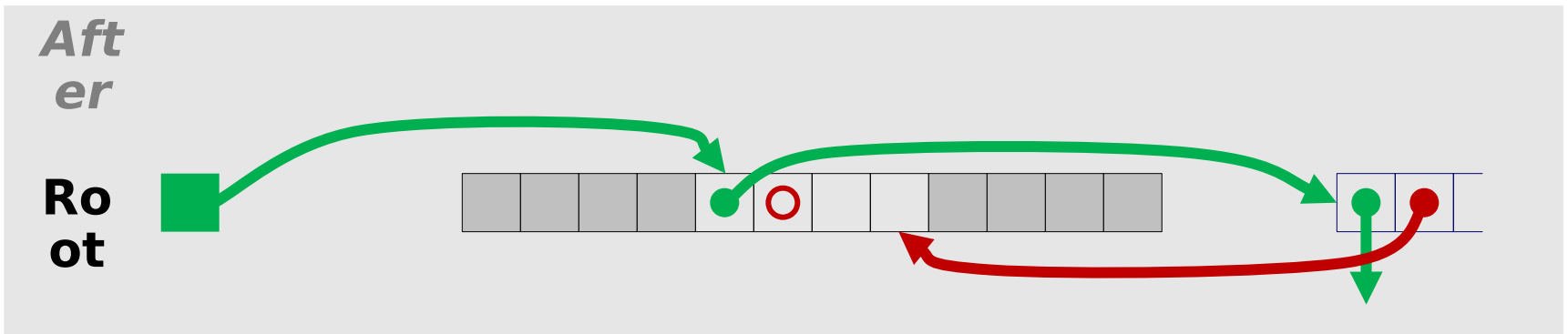
---

- *Insertion policy*: Where in the free list do you put a newly freed block?
- **LIFO (last-in-first-out) policy**
  - Insert freed block at the beginning of the free list
  - **Pro**: simple and constant time
  - **Con**: studies suggest fragmentation is worse than address ordered
- **Address-ordered policy**
  - Insert freed blocks so that free list blocks are always in address order:  
$$addr(prev) < addr(curr) < addr(next)$$
  - **Con**: requires linear search
  - **Pro**: studies suggest fragmentation is lower than LIFO

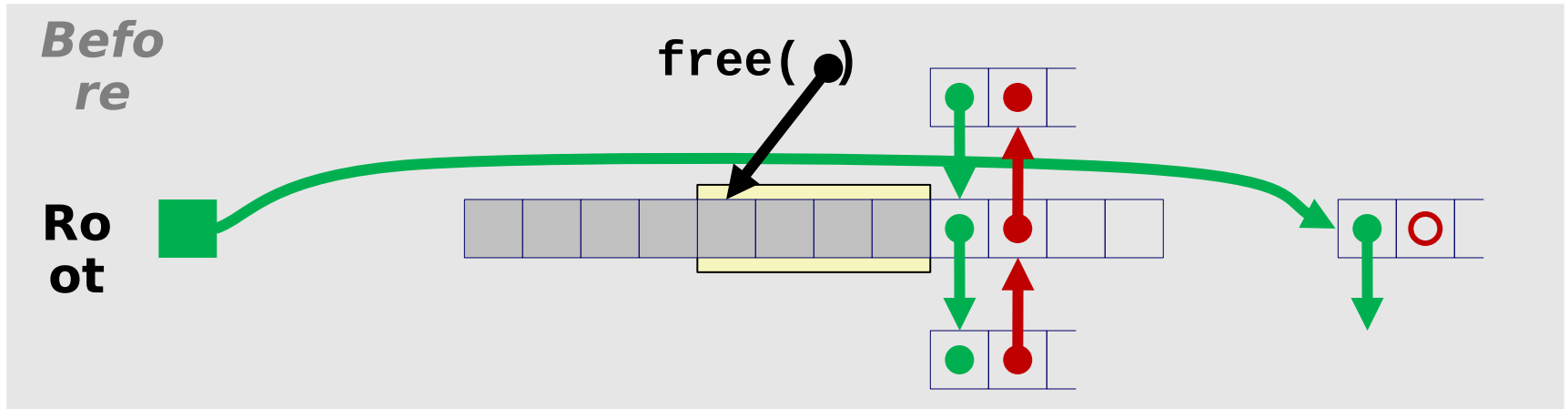
# Freeing With a LIFO Policy (Case 1)



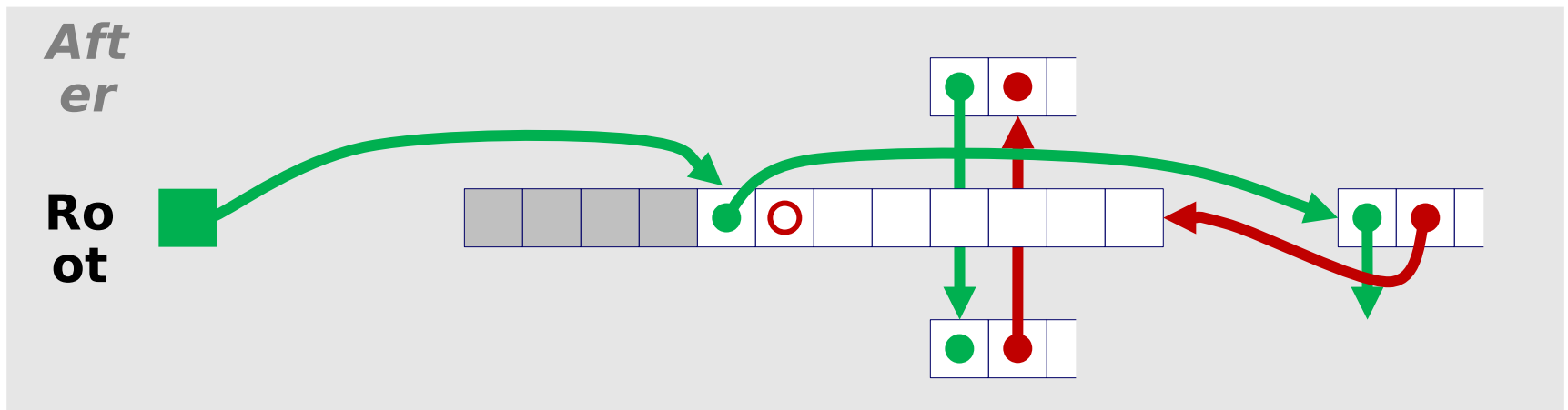
- Insert the freed block at the root of the list



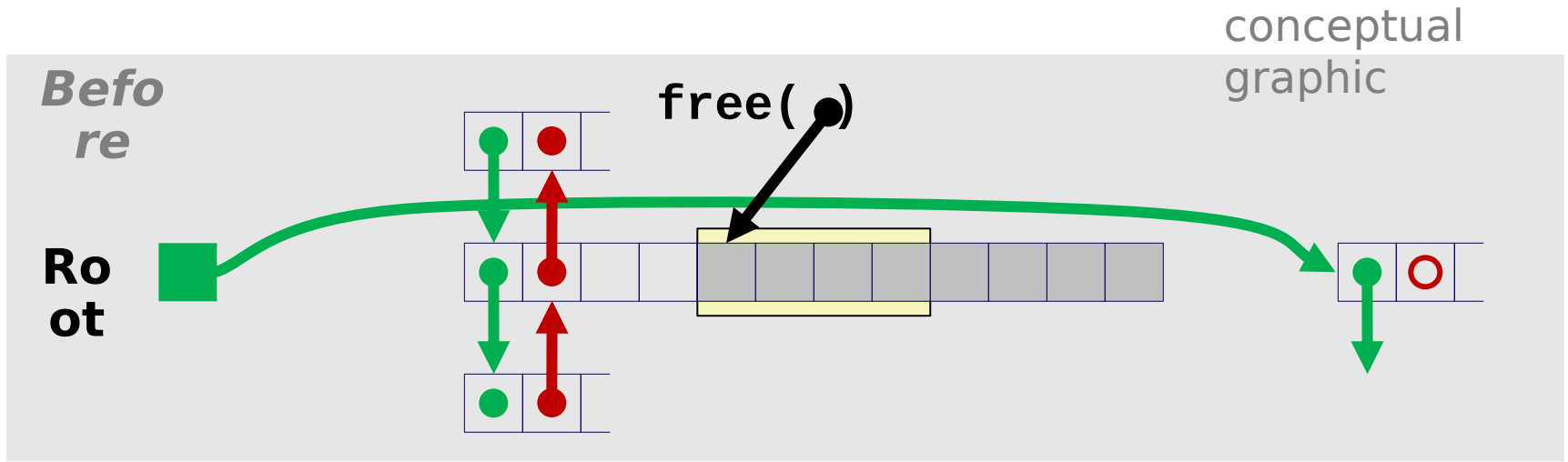
# Freeing With a LIFO Policy (Case 2)



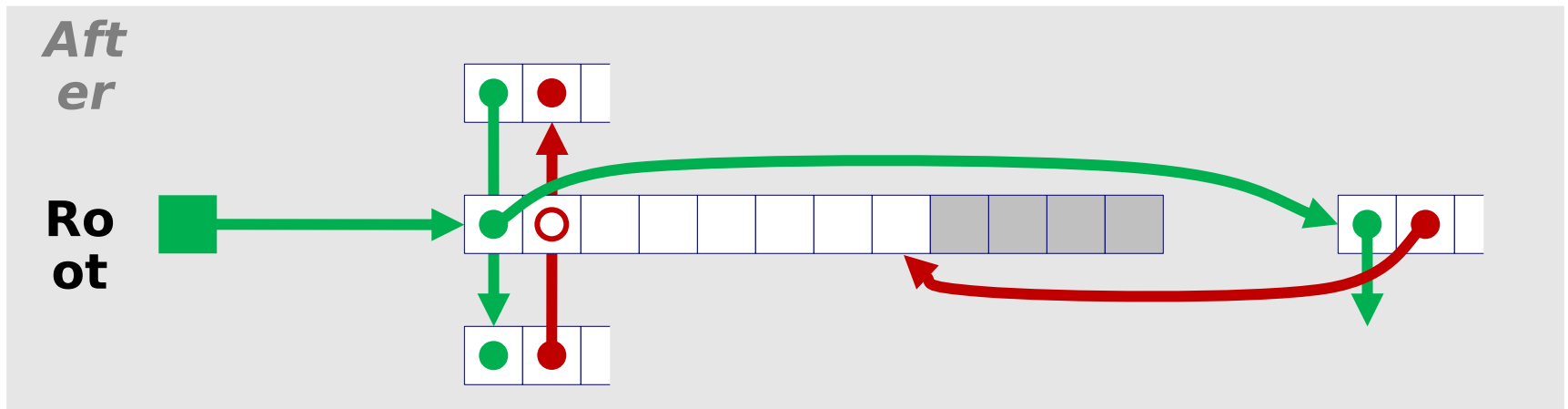
- 如果新插入的 free block 能够和相邻 free block 合并，那么合并后的新块会放在队首（root 指针指向的位置）



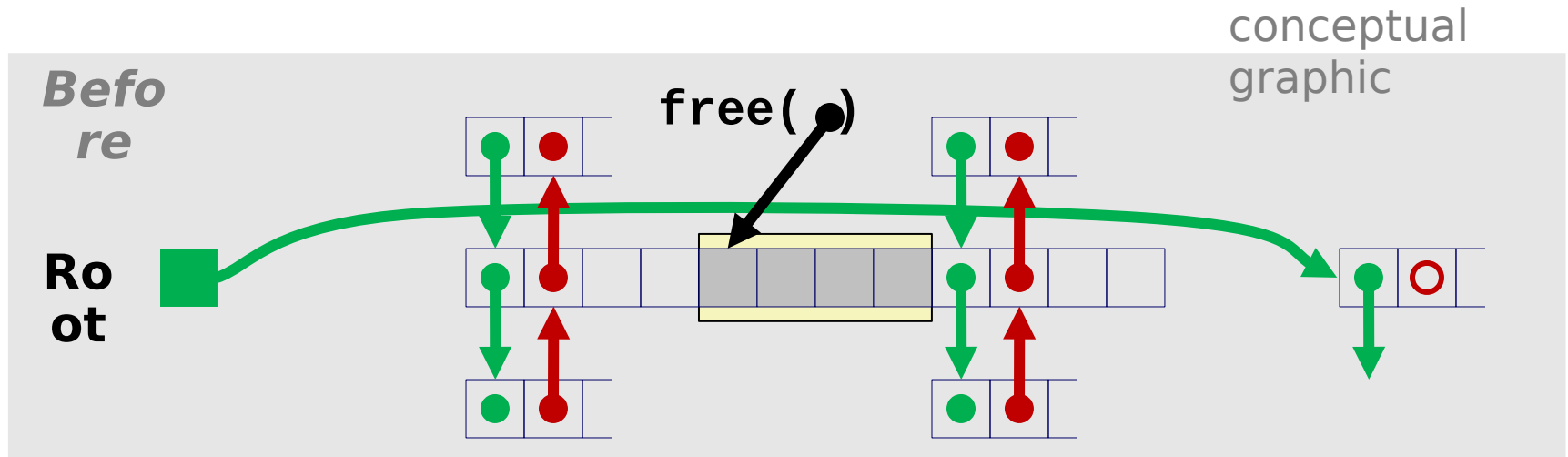
# Freeing With a LIFO Policy (Case 3)



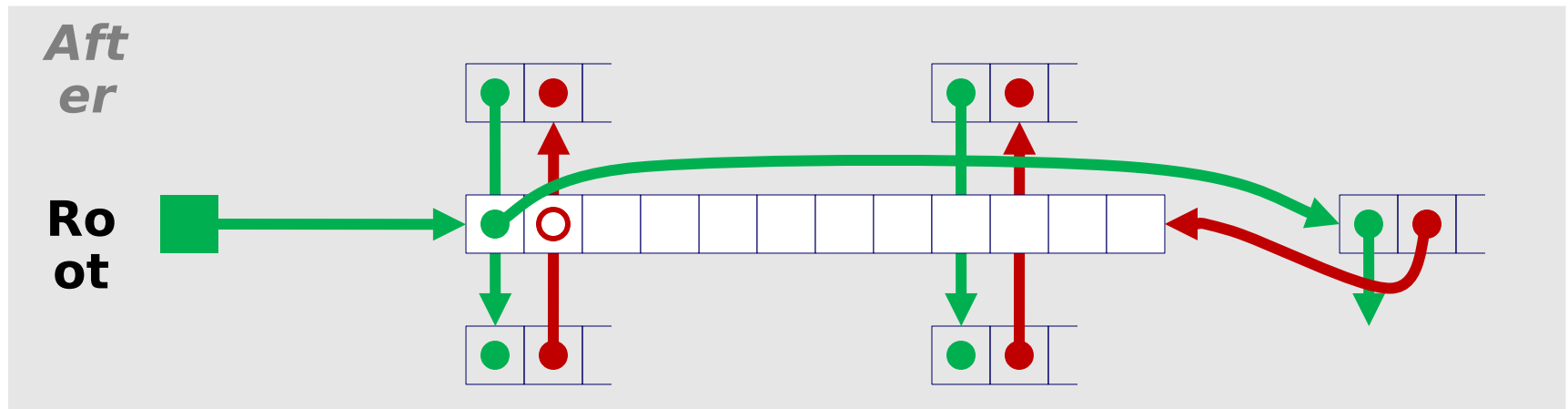
- 向后合并，同样的处理方式



# Freeing With a LIFO Policy (Case 4)



- 同时向前、向后 3 个 free blocks 合并，同样的处理方式



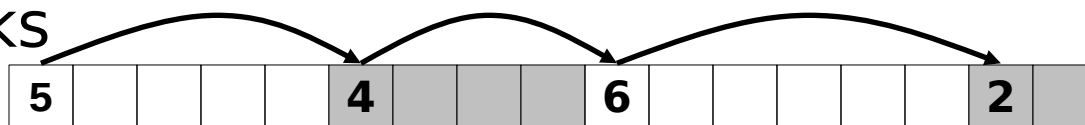
# Explicit List Summary

---

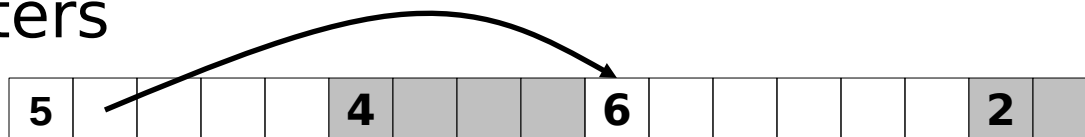
- 与 implicit list 比较：
  - 分配的时间开销与 free blocks 数量线性相关，而不是与所有 blocks
    - 当内存利用率比较高（比较满）时，时间复杂度低很多
  - 更多的链表操作，让分配和释放稍微更复杂一点
  - 维护链表需要额外的空间 (2 extra words needed for each block)
    - 会增加 internal fragmentation 吗？

# Keeping Track of Free Blocks

- Method 1: *Implicit list* using length—links all blocks



- Method 2: *Explicit list* among the free blocks using pointers

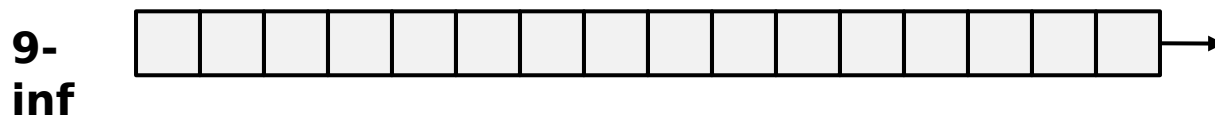
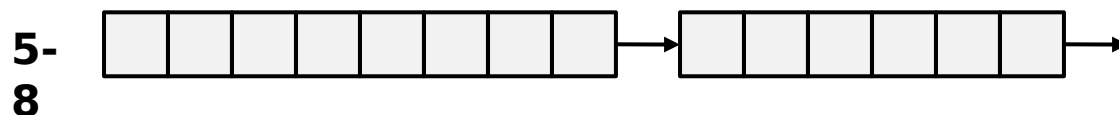
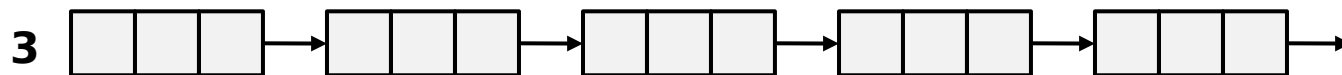
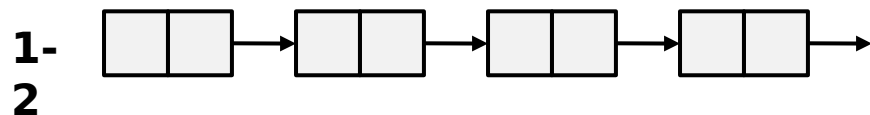


- Method 3: *Segregated free list*
  - Different free lists for different size classes
- Method 4: *Blocks sorted by size*
  - Use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

# Segregated List (Seglist) Allocators

---

- Each *size class* of blocks has its own free list



- Often have separate classes for each small size
- For larger sizes: One class for each *two-power* size
  - e.g., Memcached (slab)

# Segregated List (Seglist) Allocators

---

- #1. 简单分离存储 ( Simple Segregated Storage )
- #2. 分离适配 ( Segregated Fit )
- #3. 伙伴系统 ( Buddy System )

# Segregated List (Seglist) Allocators

---

- #1. 简单分离存储 (Simple segregated storage)
  - 每个大小类的空闲链表包含大小相等的块
    - 如 {17~32} 类，所有块的大小都是 32
  - 按照目标大小选择对应的类，从中分配一个块
    - 如目标大小是 20，则分配一个 32 的块
    - 如果该类中没有空间，就向 OS 申请一个这样大小的内存
  - 优点：
    - 分配和释放都很快速，常数时间
    - 操作简单，没有 splitting 和 coalescing
    - 已分配块不需要头尾（因为大小从类别中可知，也不合并）
    - 分配和释放都在 free list 头部进行，所以只需要单项链表
  - 缺点：内部碎片和外部碎片（因为不合并）

# Segregated List (Seglist) Allocators

---

- #2. 分离适配 ( Segregated Fit )
  - 每个 free list 中是一个空间范围的 free blocks
  - To allocate a block of size  $n$ :
    - 搜索 free list , 直到找到合适的 block ( $m \geq n$ )
    - 如果找到了一个合适的 block:
      - 按照实际大小  $n$  分割块
      - 剩下的部分放到合适的队列中 ( optional )
    - 如果找不到块, 就尝试下一级更大的队列
    - 不断重复, 直到找到合适的 free block
  - If no block is found:
    - 向 OS 申请额外的 heap memory (using `sbrk()`)
    - 从新内存中分配  $n$  bytes
    - 把剩下的部分放到合适的队列中

# Segregated List (Seglist) Allocators

---

- To free a block:
  - Coalesce and place on appropriate list
- Advantages of seglist allocators
  - Higher throughput
    - log time for power-of-two size classes
    - $2^n$ ，粗粒度，但简单高效（抓住主要矛盾）
  - Better memory utilization
    - **First-fit** search of segregated free list **approximates** a **best-fit** search of entire heap.
    - Extreme case: Giving each block its own size class is equivalent to best-fit.
- C 标准库中的 GNU malloc 采用 Segregated Fit

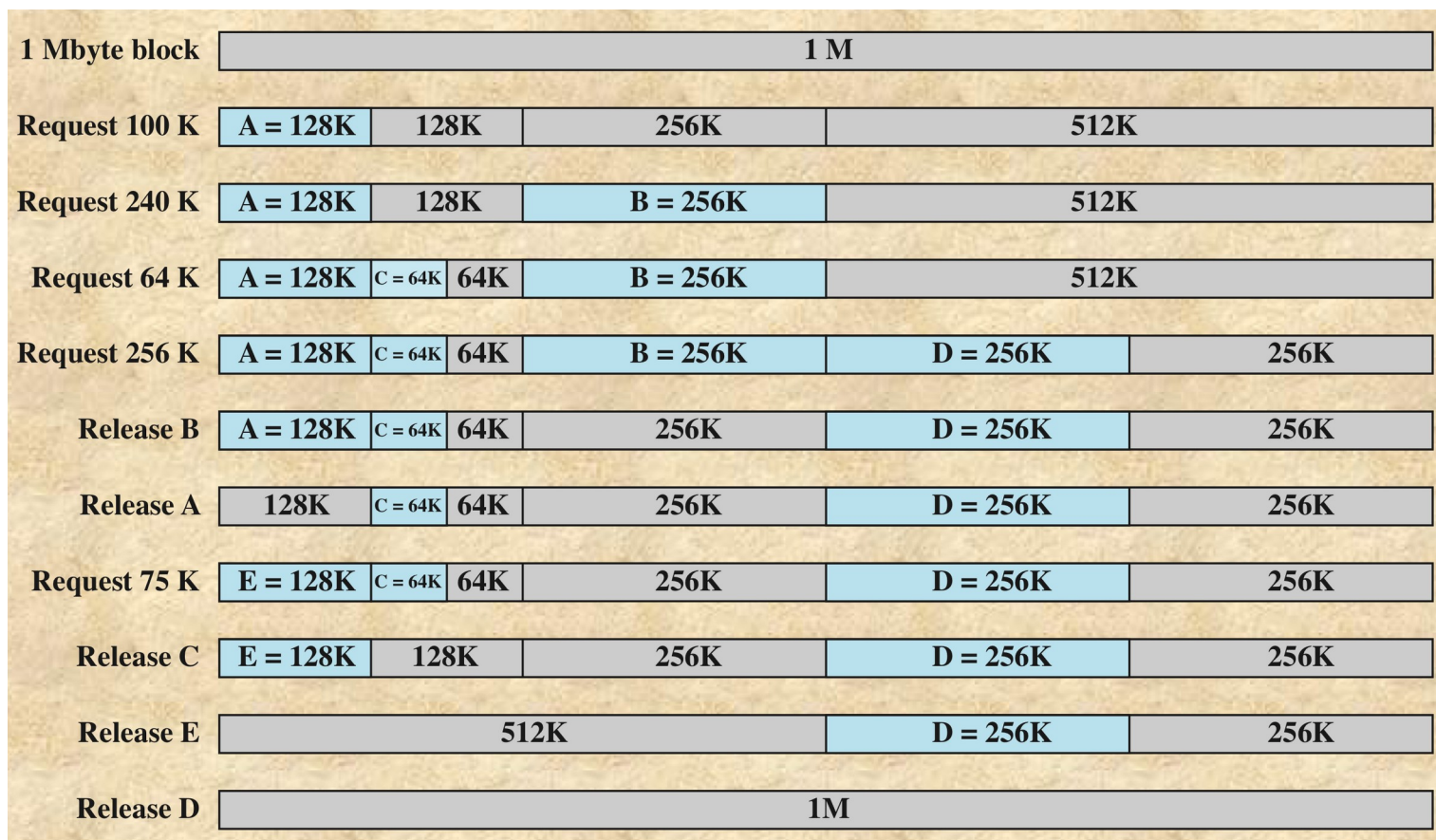
# Buddy Systems ( 伙伴系统 )

---

- 伙伴系统是 Segregated Fit 的一个特例
  - 每个大小类都是 2 的整数幂
- 堆空间  $2^m$ ，每个 class 的 block size 为  $2^k$  ( $0 \leq k \leq m$ )
  - 最开始只有一个  $2^m$  大小的块
- 为了分配  $2^k$  大小的块，我们找到一个大小为  $2^j$  的可用块 ( $k \leq j \leq m$ )
  - 如果  $j=k$ ，那么分配完成
  - 如果  $j>k$ ，就递归二分这个块，直到  $j=k$ ，并将剩下的半块（即伙伴）放到相应的 free list 中
- 当释放一个  $2^k$  的块，就找到其伙伴进行合并，递归进行，直到遇到已分配的伙伴为止

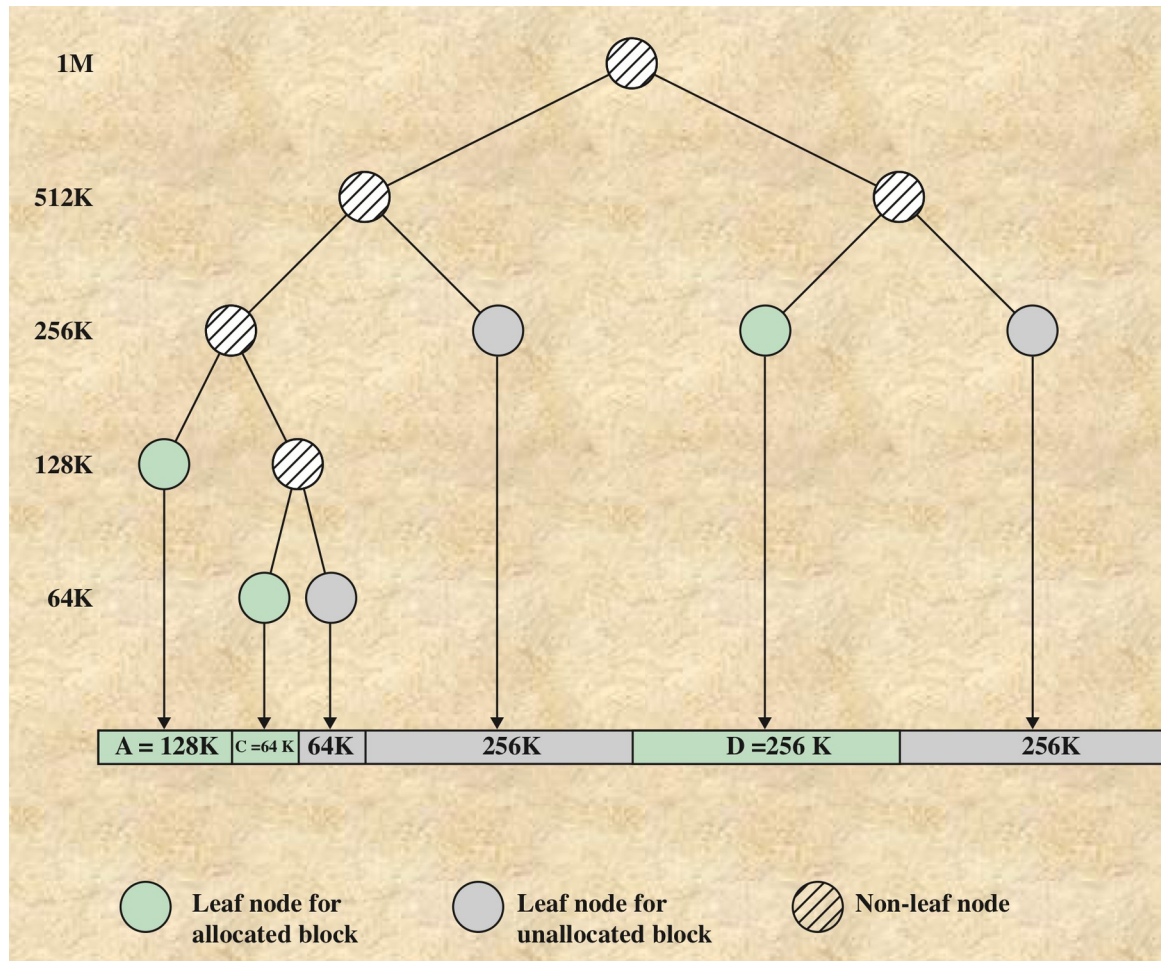
# Buddy System

- 例子，总空间是 1MB ( $2^{20}$ )



# Buddy System

- Tree Representation of Buddy System



# Buddy System

---

- 如何找到伙伴？
  - 给定一个块的地址和大小，很容易计算其伙伴的地址
  - 如一个 4B 的块，地址为
    - xxx...x000
  - 则它的伙伴地址为
    - xxx...x100
  - 只有一位不同

# Buddy System

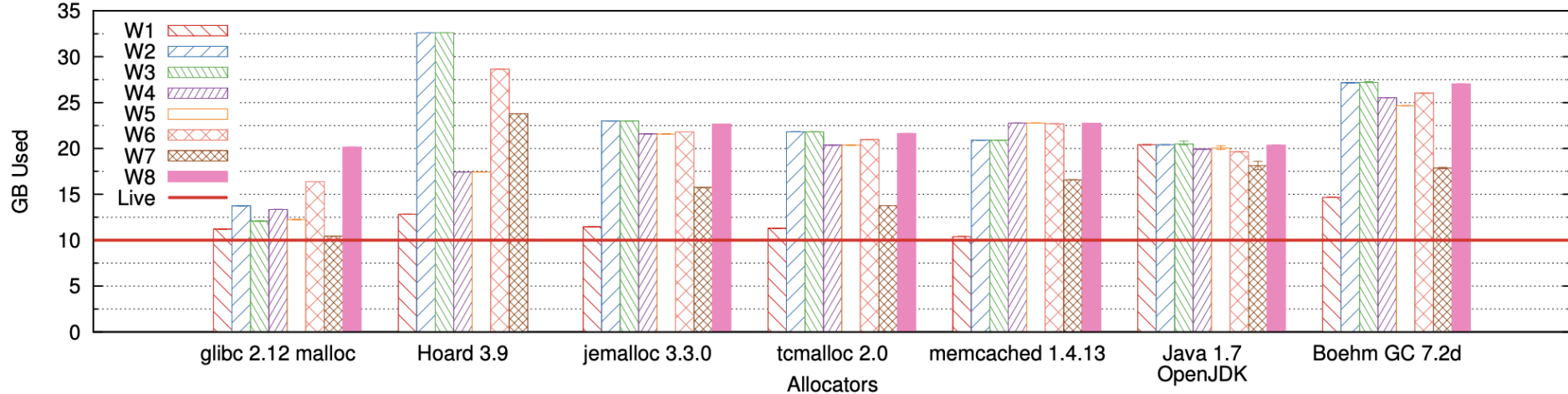
---

- 优点：
  - 整齐，快速搜索和快速合并
- 缺点：
  - 整齐，产生内部碎片
- 不适合通用目的的分配器，如果应用的分配单位以 2 的整数幂为主，则很适合

## Log-structured Allocator

[https://www.usenix.org/system/files/conference/fast14/fast14-paper\\_rumble.pdf](https://www.usenix.org/system/files/conference/fast14/fast14-paper_rumble.pdf)

- 当前各种 memory allocator 的内存利用率比较低



Workload	Before	Delete	After
W1	Fixed 100 Bytes	N/A	N/A
W2	Fixed 100 Bytes	0%	Fixed 130 Bytes
W3	Fixed 100 Bytes	90%	Fixed 130 Bytes
W4	Uniform 100 - 150 Bytes	0%	Uniform 200 - 250 Bytes
W5	Uniform 100 - 150 Bytes	90%	Uniform 200 - 250 Bytes
W6	Uniform 100 - 200 Bytes	50%	Uniform 1,000 - 2,000 Bytes
W7	Uniform 1,000 - 2,000 Bytes	90%	Uniform 1,500 - 2,500 Bytes
W8	Uniform 50 - 150 Bytes	90%	Uniform 5,000 - 15,000 Bytes

# Log-structured Allocator

---

- #1. Non-copying allocator
  - 目前 C 语言 malloc 等常见 allocator 都属于这一类
  - 不移动已分配的块，性能好，但外部碎片多
  - 尤其不适应 workload 特征变化的情况，例如释放 10GB small objects ，然后创建 10GB large objects
  - 这种变化 workload 对单个程序可能不容易发生，但对通用的存储系统来说却比较容易发生

# Log-structured Allocator

---

- #2. Copying allocator
  - 目前 Java 等很多语言的自动内存管理机制采用这类 allocator
  - 通过移动 allocated blocks 来减少碎片，但性能开销大
  - 所以一般都采用延迟 garbage collection (GC) 的方式
  - 因此，一般需要已分配数据 1.5-5 倍的空间来保障高性能
  - 另一个麻烦，当 allocated blocks 移动后，必须 halt threads 来更新指针的地址 (STW, stop-the-world)
    - 现有最好的方案，也可能导致上百 ms 的 pause time

# Log-structured File System

---

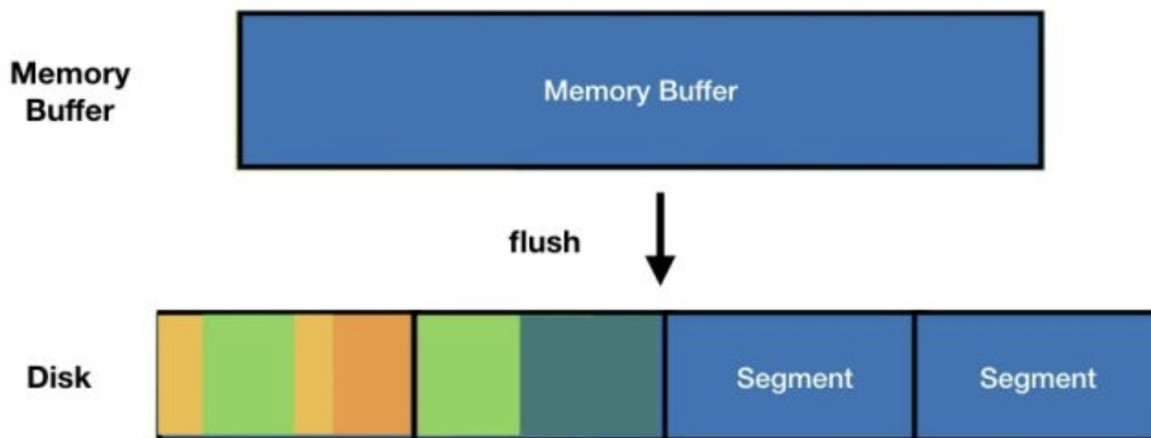
- Rosenblum M, Ousterhout J K. The design and implementation of a log-structured file system[J]. ACM Transactions on Computer Systems (TOCS), 1992, 10(1): 26-52
  - Rosenblum 是 Vmware 的联合创始人，Ousterhout 是 [Raft](#) 的作者之一
- 减少随机写，append-only 方式写入 insert/update 数据



**将随机写入变成顺序写入，提高写入性能**

# Log-structured File System

- 首先把磁盘空间分为固定大小的段（segment）
- 写操作首先会被写入到 memory buffer 中；当 buffer 中数据超过段的大小后，LFS 将数据一次性写入到空闲的段中



**以较大的粒度进行写入，进一步提高写入性能**

# Log-structured File System

---

## — 读操作

- LFS 采用 append only ，所以对同一个文件的同一个 data block 可能存在**多个版本**（在不同段中）
- 通过比较不同段的更新时间，LFS 能判断出哪个 segment 中的 data block 是最新版

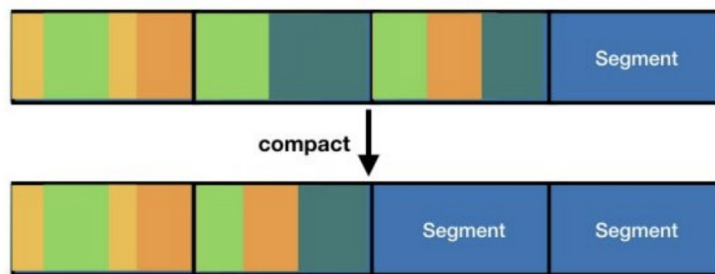
**多版本会造成一定的空间开销**

# Log-structured File System

## – 垃圾回收

- 多个包含过期数据 block 的段（文件内容被更新或是文件被删除）会被 compact 成新的数据段，同时其中的旧数据会被删除

### Garbage Collection



- Garbage Collector reads  $M$  consecutive segments with stale data blocks, compact their content into  $N$  new segments where  $N < M$ .

**垃圾回收会造成写放大（数据被多次拷贝）**

# Log-structured File System

---

- 新的文件系统例如 btrfs 也基于 LSM append only 思想实现了 **copy-on-write** 或是 **multi-version** 的特性
- 优点：写性能好（写入一时爽）
- 缺点：
  - 空间开销和写放大之间难以调和的矛盾
  - 可能造成文件中的 **block** 不连续，影响顺序读取性能
  - 适合 write-intensive 场景和基于 SSD 的存储系统（内部不能原位更新、随机读性能好）
- Log-structured 思想在很多计算机系统中得以应用
  - Log-structured merge tree (LSM-tree), 1996

# Log-structured Allocator

---

- Treating memory as a sequentially written log
- 80-90% memory utilizations while providing high performance
- 新申请的内存存在后面追加，避免了碎片
- 需要解决的问题：**如何高效的垃圾回收？**
  - 需要 copy ，移动有效数据，否则无法清除过期数据
  - 目标内存利用率越高，copy 越多。例如期望 90% 利用率，那可能需要搬移 9 bytes 有效数据，只为了释放 1 byte 垃圾数据

# Log-structured Allocator

- RAMCloud
  - 每个 master 独立进行垃圾回收
  - 1) cleaner 选择几个 segment 来清理，使用和 LFS 一样的 cost-benefit 方法（基于 free space 和 age of data 来选择 segment）
  - 2) 把 target segments 上的 live objects 拷贝到其他的 survivor segment 中；live objects 按照 age 来放置，相似 age 的 object 可能一起失效 减小碎片

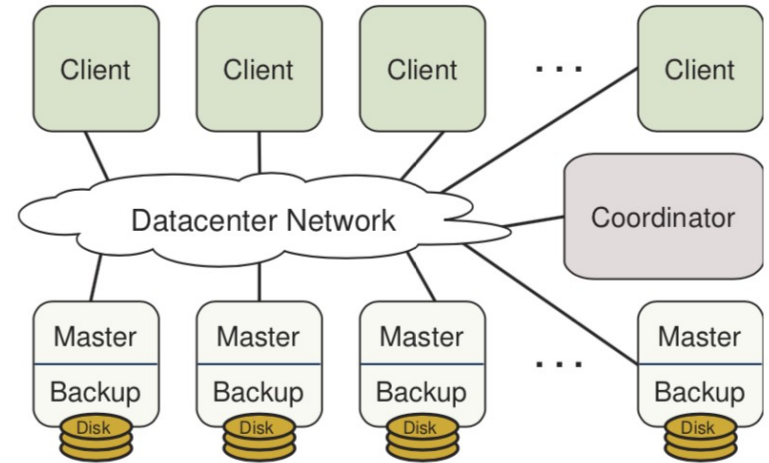
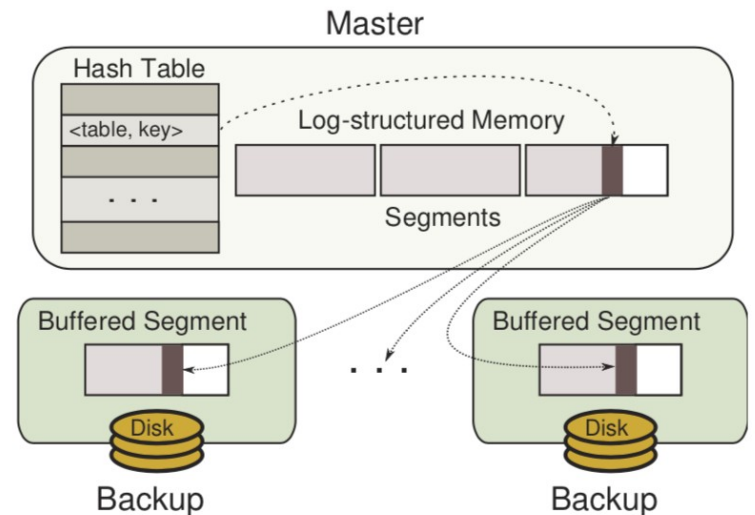


Figure 2: RAMCloud cluster architecture.



# Log-structured Allocator

- RAMCloud
  - 3) RAMCloud 内存 segment 是多备份的，一个 segment 回收了，会通知其他 replica 节点执行同样操作
  - 更多设计细节和性能优化技巧 详见论文

如何理解和做好一个系统  
研究工作?

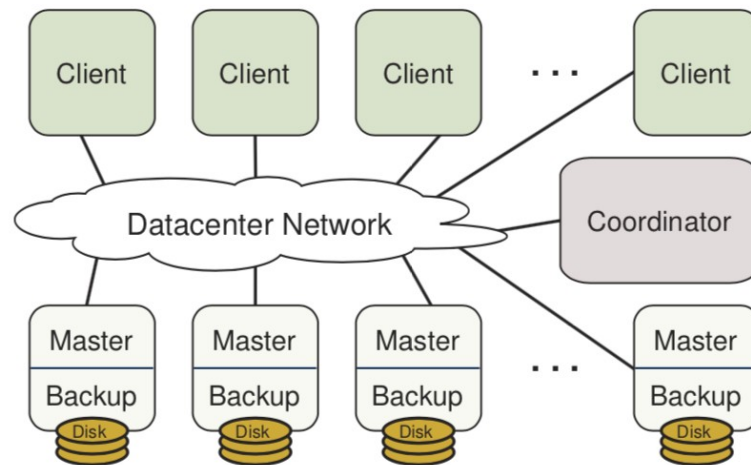
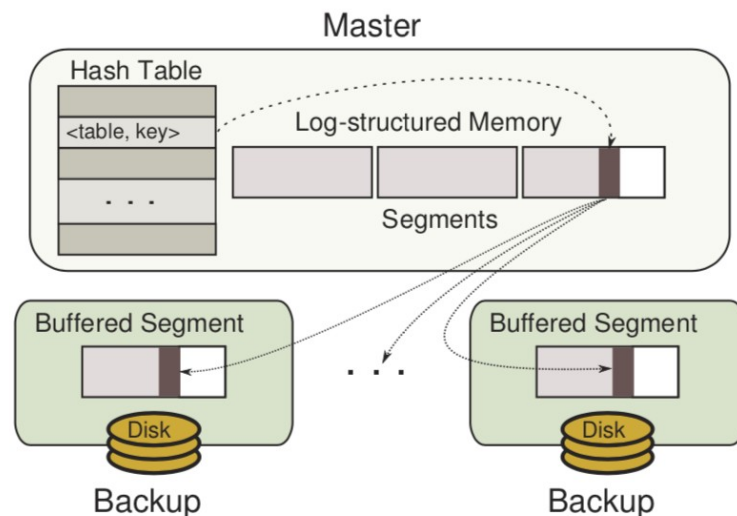


Figure 2: RAMCloud cluster architecture.



# Today

---

- Explicit free lists
- Segregated free lists
- **Garbage collection**
- Memory-related perils and pitfalls

# Explicit Memory Management

---

在 explicit memory management 中，用户程序要负责释放内存

否则会导致内存泄露

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

严重的内存泄露会导致 heap 中的空间无法有效回收  
进而影响后续内存分配，甚至导致 OOM (out-of-memory)

**正确的内存释放，对大多数程序员都是一个很大的挑战**

# Implicit Memory Management: Garbage Collection

---

- *Garbage collection*: 自动回收已分配的空间，应用程序不用负责 free（不要求程序员释放内存）
- 1960s 年代 Lisp 语言最早使用
- 如今应用在很多程序设计语言中：
  - Python, Ruby, Java, Perl, ML, Mathematica

# Classical GC Algorithms

---

- Mark-and-sweep collection (McCarthy, 1960)
  - Does not move blocks (unless you also “compact”)
- Reference counting (Collins, 1960)
  - Does not move blocks (not discussed)
- Copying collection (Minsky, 1963)
  - Moves blocks (not discussed)
- Generational Collectors (Lieberman and Hewitt, 1983)
  - Collection based on lifetimes
    - Most allocations become garbage very soon
    - So focus reclamation work on zones of memory recently allocated
    - For more information:
- Jones and Lin, *“Garbage Collection: Algorithms for Automatic Dynamic Memory”*, John Wiley & Sons, 1996.

# Reference counting ( 引用计数 )

---

引用计数比较简单:

- 为每个 block 维护一个 counter
- 每当有指针引用该 block 时, counter++
- 每当指针被销毁时, counter--
- 当 counter 减为 0 时, 释放 block

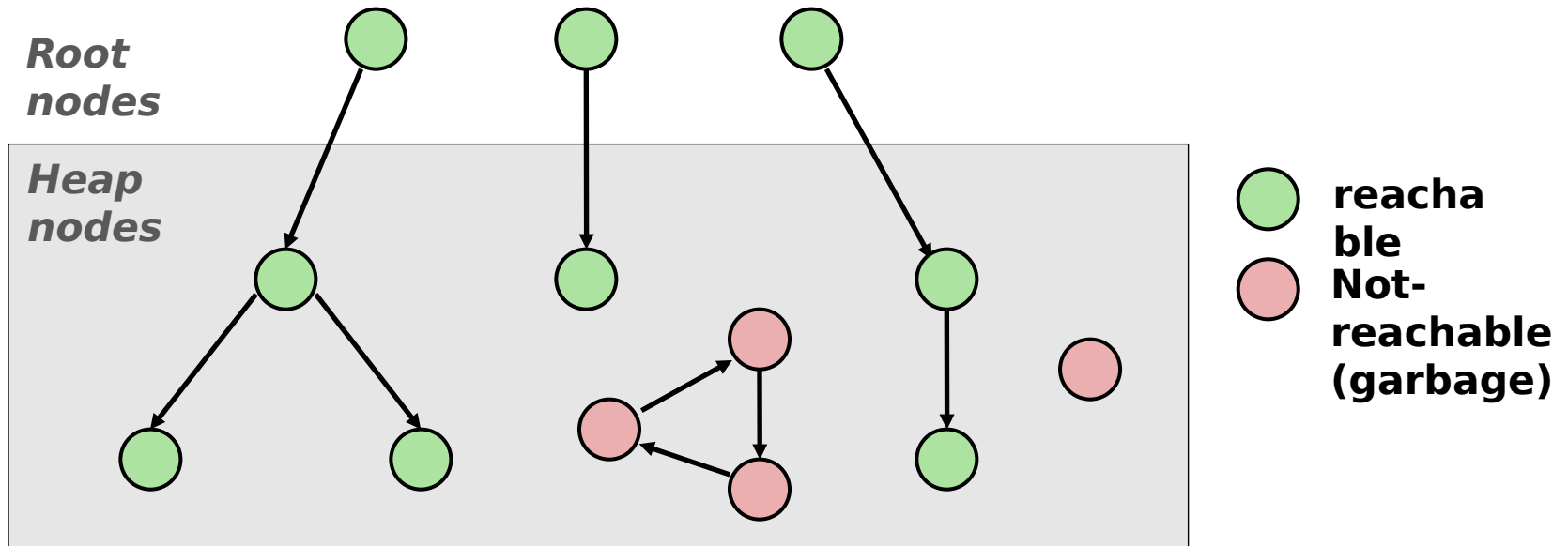
C++ 中的 shared\_ptr 采用的就是引用计数

```
std::shared_ptr<int> sp1(new int(10));  
std::shared_ptr<int> sp2 = sp1; // 引用计数  
+1  
sp1.reset(); // 引用计数 -1
```

引用计数无法回收两个互相引用的数据对象  
(block)

# Memory as a Graph

- 更完善的方案：将内存看做一个有向图
  - 每个 memory block 是一个 node
  - 每个指针是一个 edge
  - 指向 heap 但不存储在 heap 中的指针是 **root** nodes (例如 registers, locations on the stack, global variables)

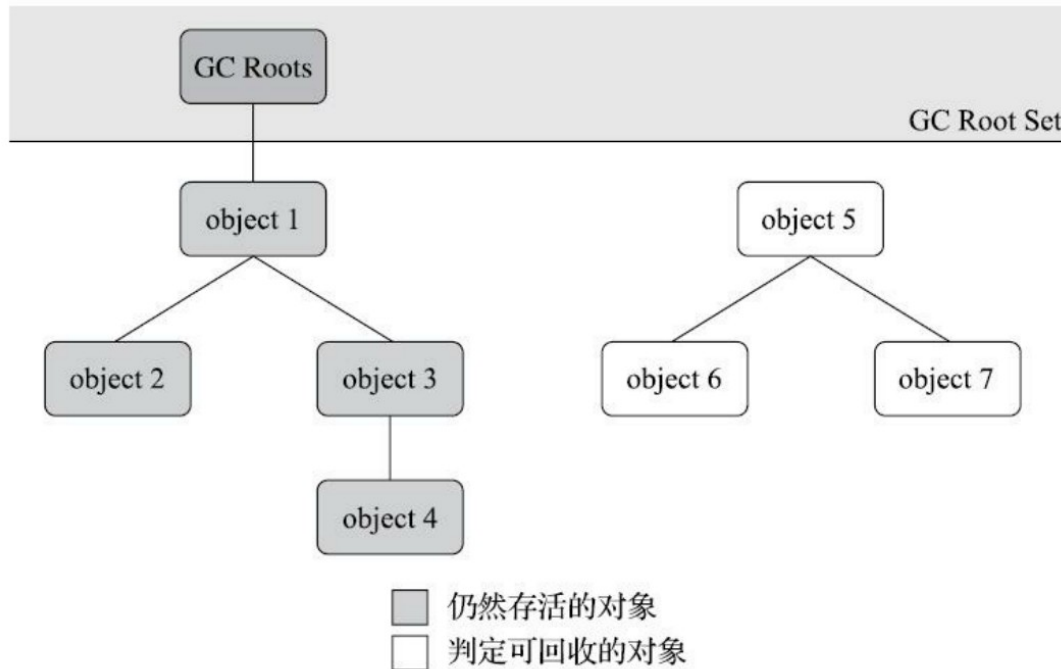


**reachable** : 存在从某个 root 到 node (block) 的可达路径

Non-reachable nodes are **garbage** (cannot be needed by the application)

# Memory as a Graph

root 节点的集合在 Java 等语言中称为 GC root set



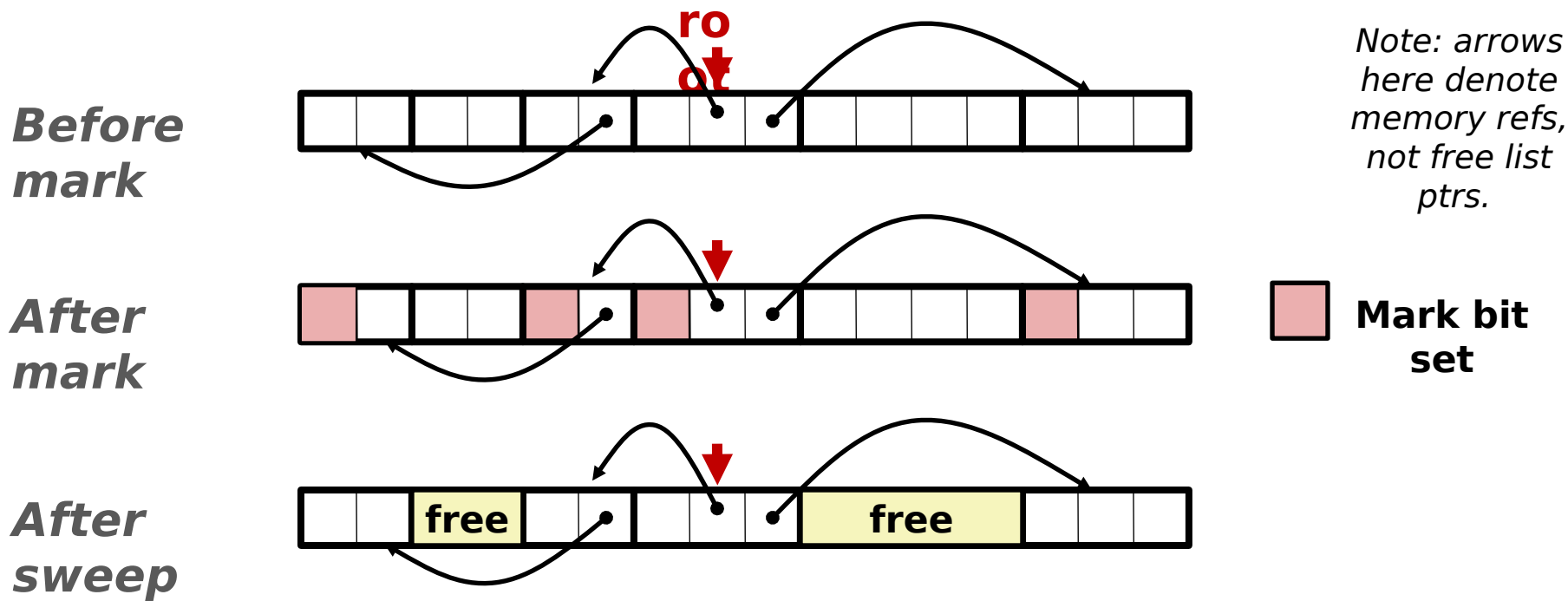
# Memory as a Graph

---

**Garbage collection 的关键就是：  
如何构建和维护 reachability graph  
如何通过 reachability graph 快速找出 heap 中的  
garbage**

# Mark and Sweep Collecting ( 标记清除 )

- 简单，可以在 c/c++ 的 malloc/free 基础上实现一个 collector
  - 程序员使用 malloc 分配内存，直到内存不足时开始 GC
- 当 heap 空间不足时：
  - Use extra **mark bit** in the head of each block
  - **Mark:** Start at roots and set mark bit on each reachable block
  - **Sweep:** Scan all blocks and free blocks that are not marked



# Assumptions For a Simple Implementation

---

- Application
  - **new(n)**: returns pointer to new block with all bytes cleared
  - **read(b, i)**: read location **i** of block **b** into register
  - **write(b, i, v)**: write **v** into location **i** of block **b**
- Each block will have a header word
  - addressed as **b[-1]**, for a block **b**
  - Used for different purposes in different garbage collectors
- Instructions used by the Garbage Collector
  - **isPtr(p)**: determines whether **p** is a pointer
  - **length(b)**: returns the length of block **b**, not including the header
  - **get\_roots()**: returns all the roots

# Mark and Sweep (cont.)

---

## Mark using depth-first traversal of the memory

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

## Sweep using lengths to find next block

```
ptr sweep(ptr p, ptr end) {
    while (p < end) {
        if (blockMarked(p))
            clearMark(p);
        else if (blockAllocated(p))
            free(p);
        p += length(p);
    }
}
```

# Mark and Sweep (cont.)

---

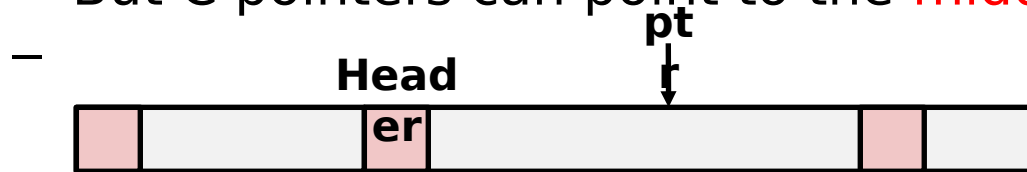
**isPtr() 非常关键**

**在一些语言中，会有专门的机制来识别某个 memory block 中的某个 word 是否是一个 pointer**

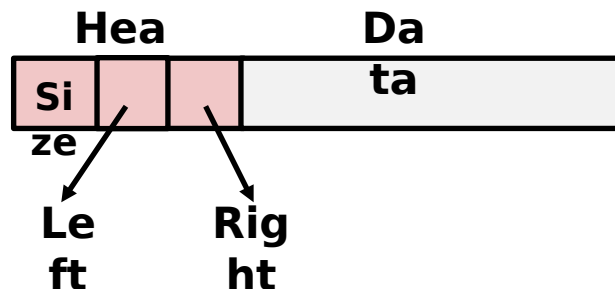
**然而 C 语言并不提供这种机制**

# Conservative Mark & Sweep in C

- A "conservative garbage collector" for C programs
  - `isPtr()` determines if a word is a pointer by checking if it points to an allocated block of memory
  - But C pointers can point to the **middle of a block**



- So how to find the beginning of the block?
  - Use a balanced binary tree to keep track of all allocated blocks (key is start-of-block)
  - Balanced-tree pointers can be stored in block header (use two additional words) : 利用 size 判定指针 p 是否在该段内存中



**Left:** smaller addresses  
**Right:** larger

# Conservative Mark & Sweep in C

---

- 存在 false positive 问题，即一个 unreachable pointer 可能被误认为 reachable 而被 mark
- 因此，可能导致一些 block 无法被正确释放
- 要解决这个问题，需要编程语言将 pointer 打上 type tag，以便和正常的 int/float 等基本数据类型区分

# Generational Hypothesis

---

1. 内存中的绝大多数数据对象都是朝生夕死的 (ephemeral) ， 可以很快被 GC
2. 熬过越多次垃圾回收的数据对象越难以消亡

Generational Garbage Collection :

将 heap 分为多个区域， 每个区域存放不同寿命的数据对象

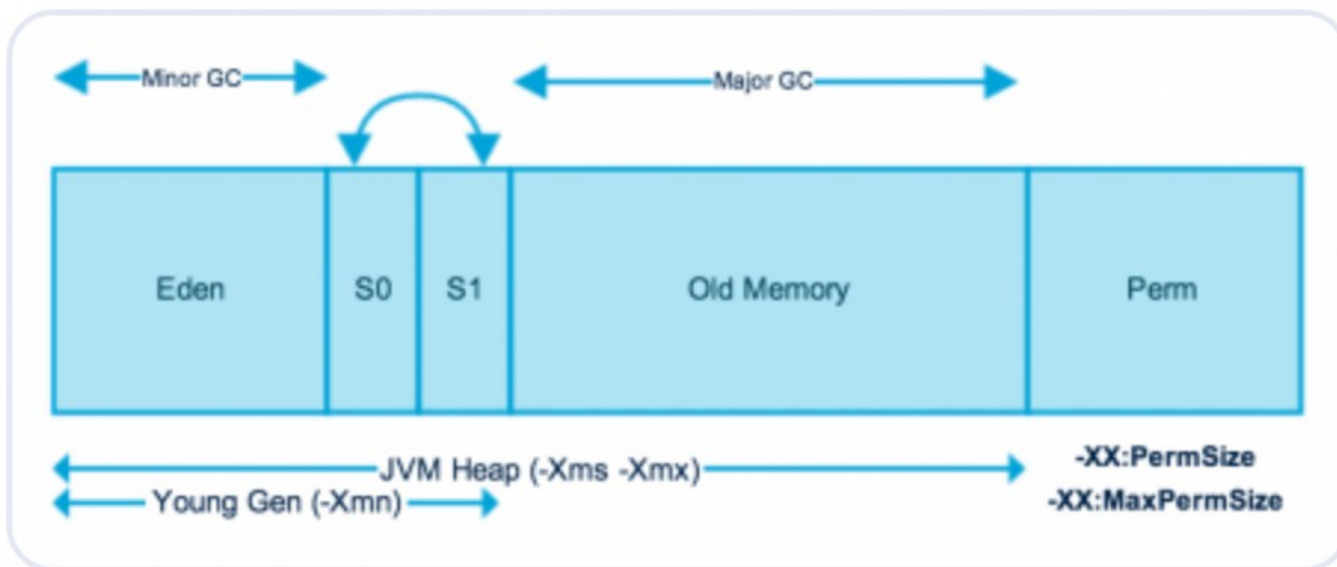
对不同区域采用不同的 GC 算法

- 例如分为新生代 (young generation) 和老生代 (old generation)

# Generational GC

在 Java 等语言中广泛使用

- 一些 Java GC 中将 young generation 称为 eden



# Today

---

- Explicit free lists
- Segregated free lists
- Garbage collection
- Memory-related perils and pitfalls (危険、  
陷阱)

# Memory-Related Perils and Pitfalls

---

- 间接引用坏指针
- 读未初始化的内存
- 允许栈缓冲区溢出
- 假设指针和它指向的对象大小相同
- 造成错位错误
- 误解指针运算
- 引用指针而不是它指向的对象
- 引用不存在的变量
- 多次 free
- 引用被释放的块
- 没有释放块

# C Pointer Declarations: Test Yourself!

---

<code>int *p</code>	p is a pointer to int
<code>int *p[13]</code>	p is an array[13] of pointer to int
<code>int *(p[13])</code>	p is an array[13] of pointer to int
<code>int **p</code>	p is a pointer to a pointer to an int
<code>int (*p)[13]</code>	p is a pointer to an array[13] of int
<code>int *f()</code>	f is a function returning a pointer to int
<code>int (*f)()</code>	f is a pointer to a function returning int
<code>int (*( *f())[13])()</code>	f is a function returning ptr to an array[13] of pointers to functions returning int

# 间接引用坏指针

---

- The classic scanf bug

```
int val; 正确  
  
...  
  
scanf("%d",  
&val);
```

```
int val; 错误  
  
...  
  
scanf("%d", val);
```

误将 **int** 变量值作为 **pointer** 使用

新的编译器可以对这种错误进行检查并给出警告

# 读未初始化的内存

---

- 误以为 heap 中的内容都初始化为 0
  - 需要自己初始化为 0，或用 calloc

```
/* 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;
}
```

## 假设指针和它指向的对象大小相同

---

- 第一次 malloc ，分配的应该是  $N * \text{sizeof}(\text{int} *)$
- 这里是按照 int 大小分配的，在 64 位机下出错

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

# 造成错位错误

---

- Off-by-one error

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

# 允许栈缓冲区溢出

---

- Not checking the max string size

```
char s[8];  
int i;  
  
gets(s); /* reads "123456789" from stdin */
```

- Basis for classic buffer overflow attacks
- 避免使用 gets 这种不限定输入长度的函数

# 误解指针运算

---

- 每次  $p+=4$  ，实际跳过了中间元素，应为  $p++$

```
int *search(int *p, int val) {  
    while (*p && *p != val)  
        p += sizeof(int);  
  
    return p;  
}
```

# 引用指针而不是它指向的对象

---

- `*size--` 应改为 `(*size)--`
  - `--` 与 `*` 同级，但从右向左结合，因此对 `size` 指针进行了操作，而不是它指向的对象

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

# C operators

## Operators

### - Associativity

<code>() [] -&gt; .</code>	left to right
<code>! ~ ++ -- + - * &amp; (type) sizeof</code>	right to left
<code>* / %</code>	left to right
<code>+ -</code>	left to right
<code>&lt;&lt; &gt;&gt;</code>	left to right
<code>&lt; &lt;= &gt; &gt;=</code>	left to right
<code>== !=</code>	left to right

`&` left to right

`>`, `()`, and `[]` have high precedence, with `*` and `&` just below

`^` left to right  
`Unary +, -, and *` have higher precedence than binary forms

`|` left to right **Source: K&R**

# 引用不存在的变量

---

- 不要将局部变量的指针返回给作用域之外

```
int *foo () {  
    int val;  
  
    return &val;  
}
```

# 多次 free

---

- Nasty!

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

# 引用被释放的块

---

- Evil!

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

# 没有释放块 (Memory Leaks)

---

- Slow, long-term killer!

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

# 没有释放块 (Memory Leaks)

---

- Freeing only part of a data structure

```
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;
}
```

# Dealing With Memory Bugs

---

- Debugger: `gdb`
  - Good for finding bad pointer dereferences
  - Hard to detect the other memory bugs
- Binary translator: `valgrind`
  - Powerful debugging and analysis technique
  - Rewrites text section of executable object file
  - Checks each individual reference at runtime
    - Bad pointers, overwrites, refs outside of allocated block
- `glibc malloc` contains checking code
  - `setenv MALLOC_CHECK_ 3`

# 课堂练习

---

下面给出了三组关于内存管理和垃圾收集的陈述。在每一组中，只有一句陈述是正确的。你的任务就是判断哪一句是正确的。

- 1) a) 在一个伙伴系统中，最高可达 50% 的空间可以因为内部碎片而被浪费了。  
b) 首次适配内存分配算法比最佳适配算法要慢一些(平均而言)。  
c) 只有当空闲链表按照内存地址递增排序时，使用边界标记来回收才会快速。  
d) 伙伴系统只会有内部碎片，而不会有外部碎片。

# 课堂练习

---

- 2) a) 在按照块大小递减顺序排序的空闲链表上，使用首次适配算法会导致分配性能很低，但是可以避免外部碎片。
- b) 对于最佳适配方法，空闲块链表应该按照内存地址的递增顺序排序。
- c) 最佳适配方法选择与请求段匹配的最大的空闲块。
- d) 在按照块大小递增的顺序排序的空闲链表上，使用首次适配算法与使用最佳适配算法等价。

## 课堂练习

---

- 3) Mark&Sweep 垃圾收集器在下列哪种情况下叫做保守的：
- a) 它们只有在内存请求不能被满足时才合并被释放的内存。
  - b) 它们把一切看起来像指针的东西都当做指针。
  - c) 它们只在内存用尽时，才执行垃圾收集。
  - d) 它们不释放形成循环链表的内存块。

# Homework 6

---