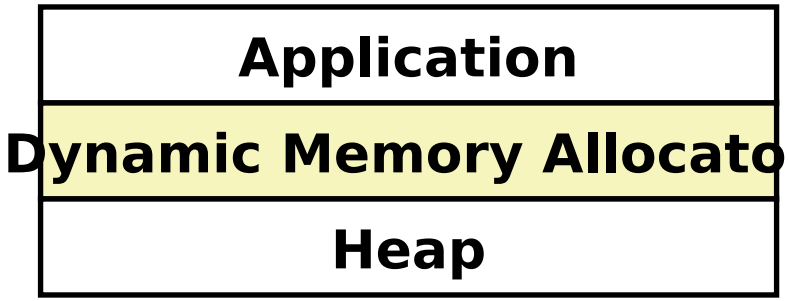
Virtual Memory (IV)
Dynamic Memory Allocation

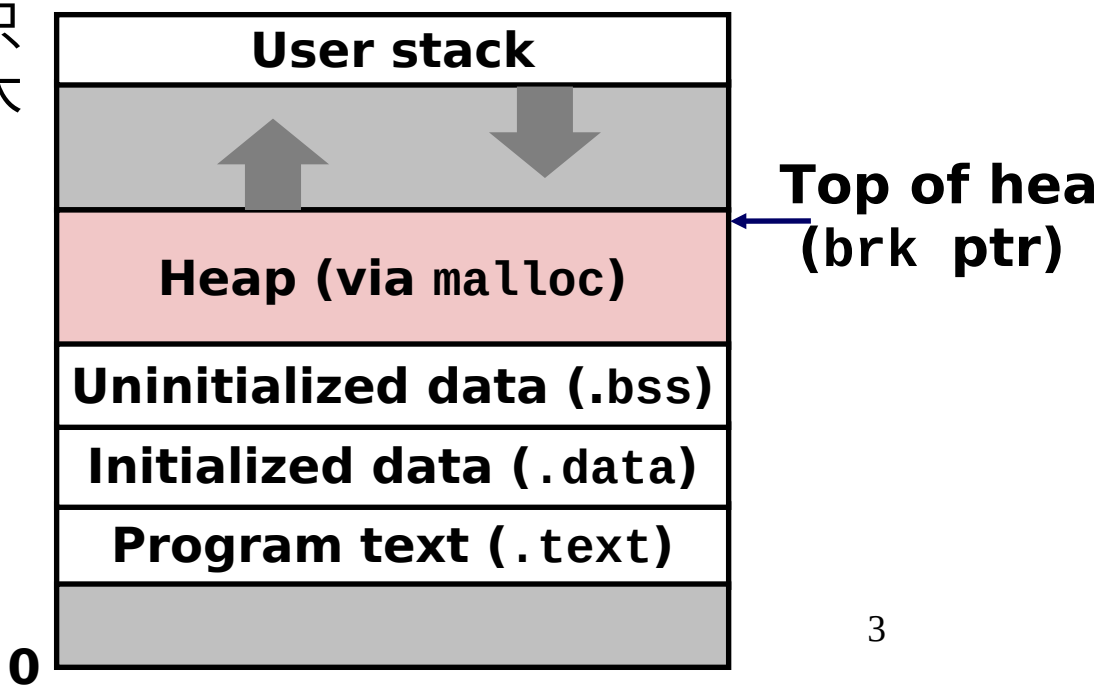
Outline

- Basic concepts
- Implicit free lists

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





Dynamic Memory Allocation

- Heap：
  - 一段连续空间（虚存空间，有 page table 时，物理未必连续）
  - 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
- Types of allocators
  - ***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
- Will discuss simple explicit memory allocation today

The `malloc` Package

```
#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 `malloc` Package

`void free(void *p)`

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

Other functions

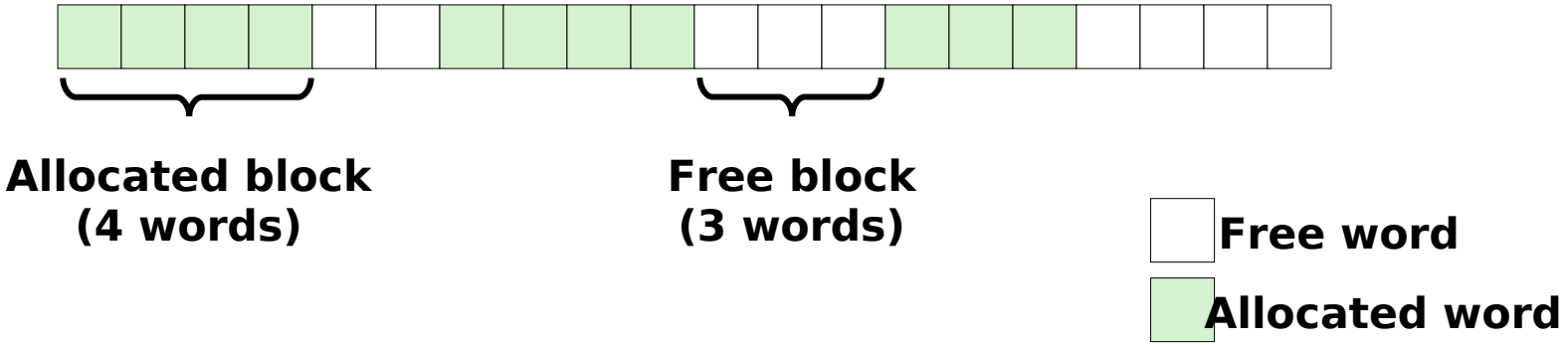
- **`calloc`**: Version of **`malloc`** that initializes allocated block to zero.
- **`realloc`**: Changes the size of a previously allocated block.
- **`sbrk`**: Used internally by allocators to grow or shrink the heap

`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);
}
```

Assumptions Made in This Lecture

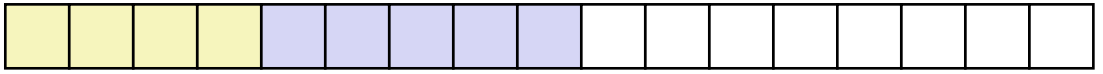
- Memory is word addressed (each word can hold a pointer)



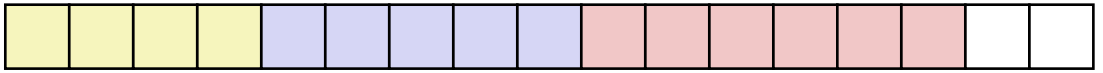
Allocation Example

```
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，不同做 compaction 操作（为了不影响性能）

Performance Goal: Throughput

- Given some sequence of `malloc` and `free` requests:
  - $R_0, R_1, ..., R_k, ... , R_{n-1}$

- 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 operations/second

Performance Goal: Peak Memory Utilization

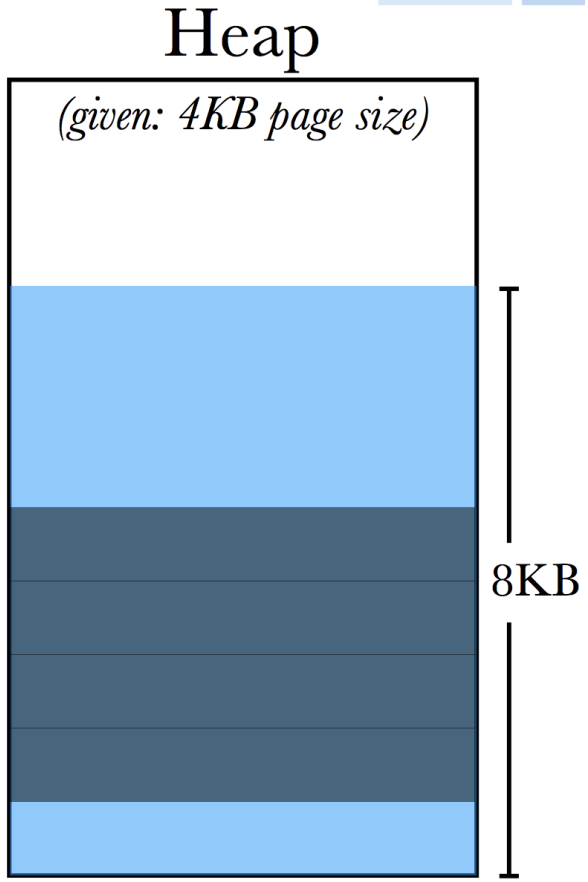
- Given some sequence of `malloc` and `free` requests:
  - $R_0, R_1, ..., R_k, ... , R_{n-1}$
- *Def: 聚集有效负载（aggregate payload） P_k*
  - **`malloc(p)`** results in a block with a ***payload*** of **`p`** bytes
  - After request R_k has completed, the ***aggregate payload*** P_k is the sum of currently allocated payloads
- *Def: 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`**
- *Def: 峰值利用率（Peak memory utilization） after k+1 requests*
  - $U_k = (max_{i<=k} P_i) / H_k$
  - 的最大值

Performance Goal: Peak Memory Utilization

- *峰值利用率（Peak memory utilization）例子*

实时利用率 -->
（非峰值利用率）

```
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%
```

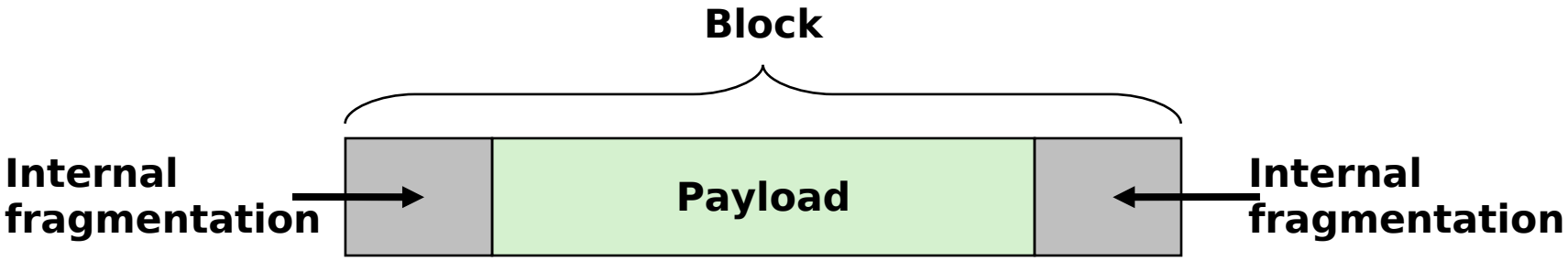


Fragmentation

- 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



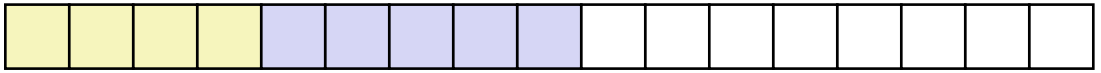
- Caused by
  1. Overhead of maintaining heap data structures (e.g., pointers)
  2. Padding for alignment purposes
  3. Explicit policy decisions (e.g., to return a big block to satisfy a small request)
- Depends only on the pattern of *previous* requests
  - Thus, easy to measure

External Fragmentation

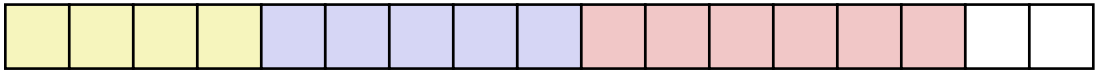
- Occurs when there is enough aggregate heap memory, but no single free block is large enough

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

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



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



```
free(p2)
```

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

Oops! (what would happen now?)

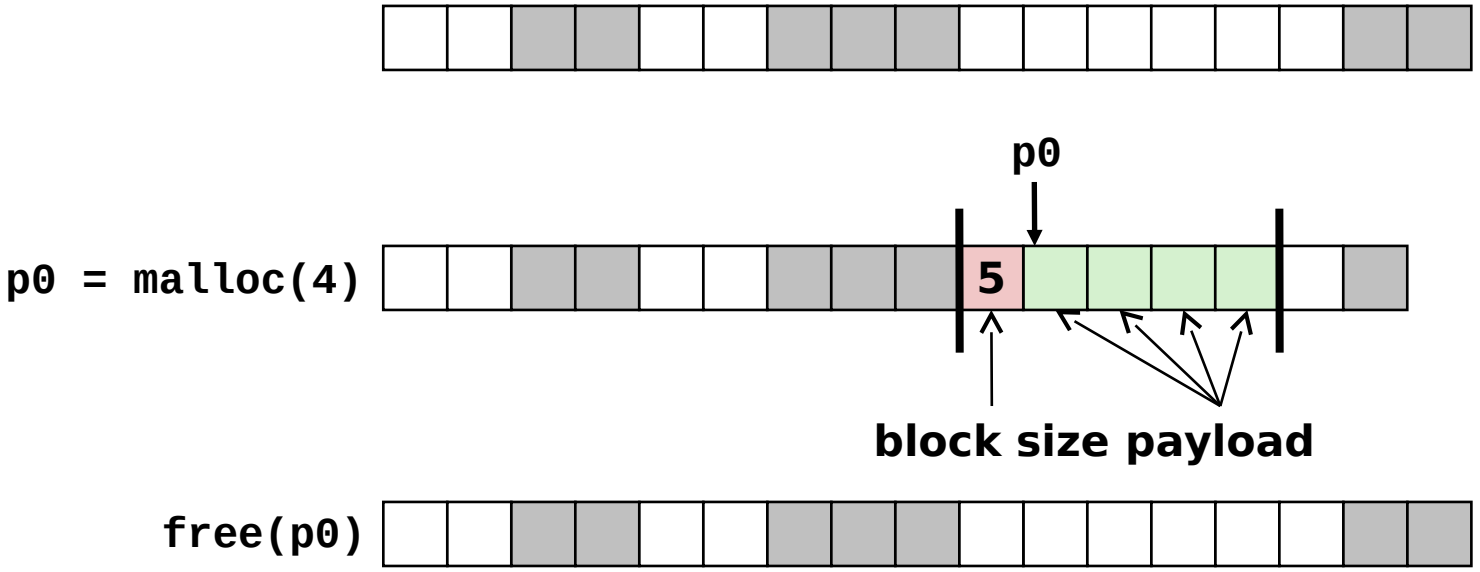
- Depends on the pattern of future requests
  - Thus, difficult to measure (以上情况可能是碎片，也可能不是)

Implementation Issues

- 如何知道 free 一个指针时，需要释放多少内存空间?
- 如何组织管理空闲块?
- 当我们分配的数据结构比一个 free block 小时，如何处理额外的空间?
- 我们如何从 free blocks 中选择一个用来分配? 可能有很多个符合条件的
- 如何插入被释放的块?

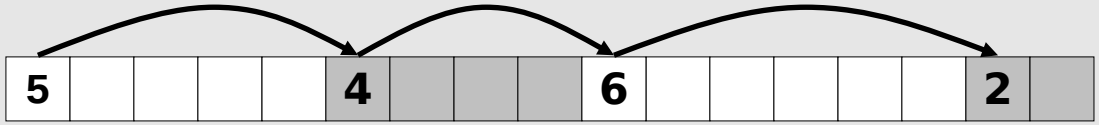
Knowing How Much to Free

- 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

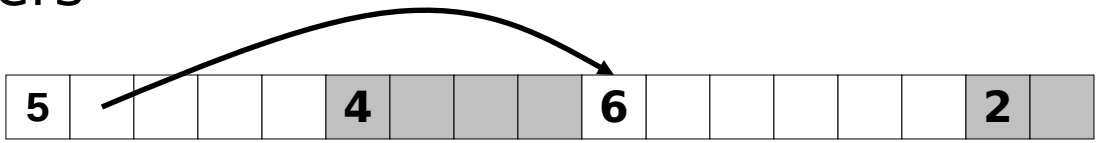


Keeping Track of Free Blocks

- 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

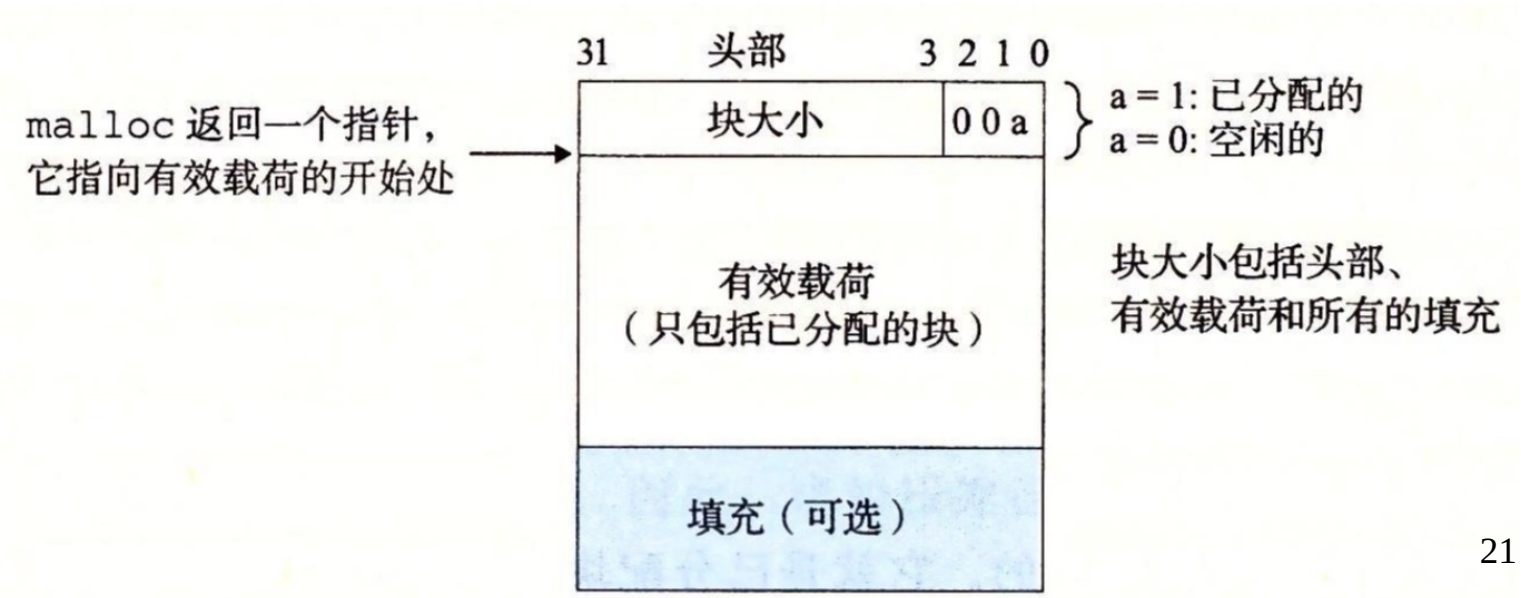
- Method 4: *Blocks sorted by size*
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

Outline

- Basic concepts
- Implicit free lists

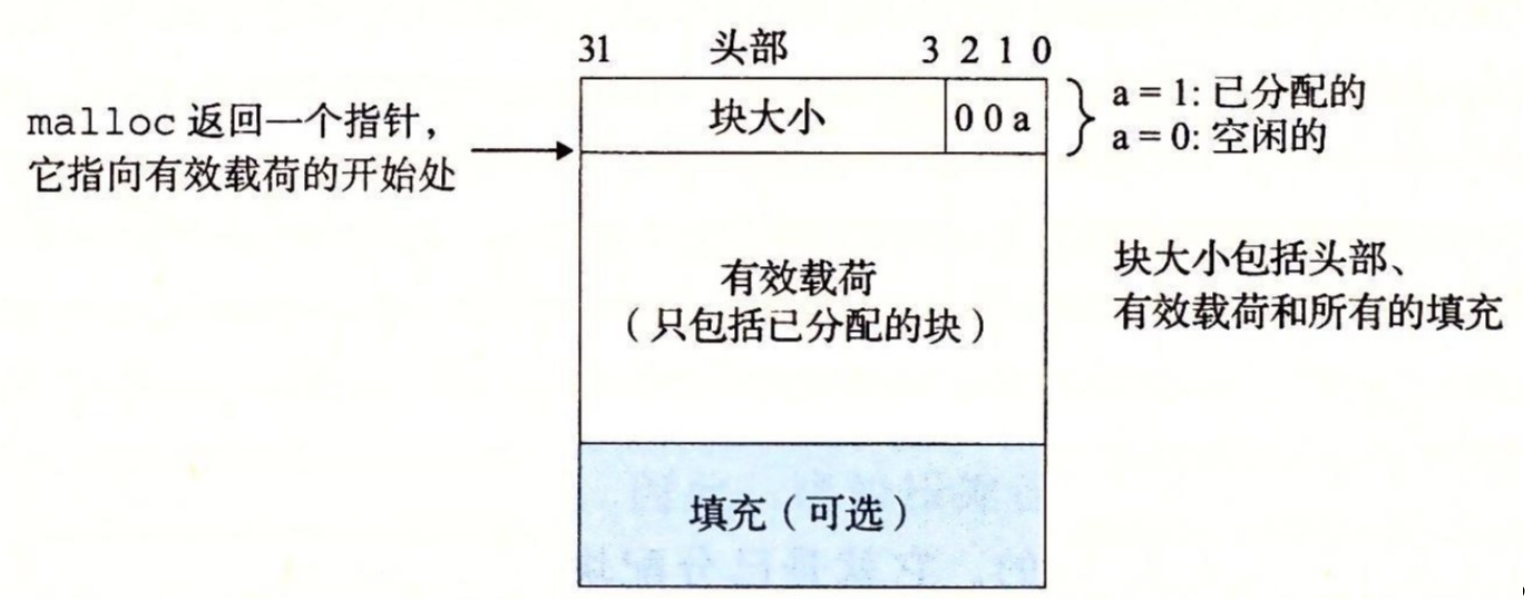
Method 1: Implicit List

- For each block we need both size and allocation status
  - Could store this information in two words: wasteful!
- Standard trick
  - If blocks are aligned, some low-order address bits are always 0 (例如，双字对齐，则块大小是 8 的倍数，块大小最后三位一点是 0)
  - Instead of storing an always-0 bit, use it as an allocated/free flag
  - When reading size word, must mask out this bit

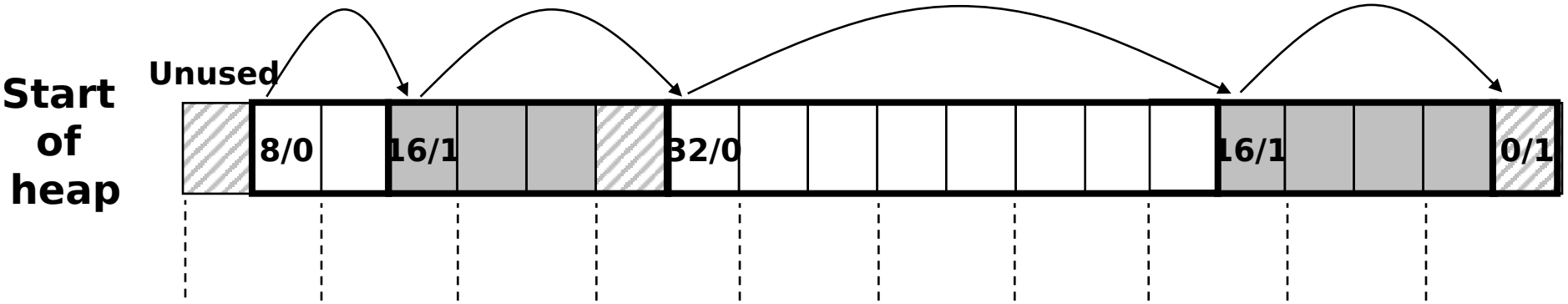


Method 1: Implicit List

- 块大小为 24 （0x18）字节，已分配块，头部将是
  - 0x00000018 | 0x1 = 0x00000019
- 块大小为 40 （0x28）字节，未分配块，头部将是
  - 0x00000028 | 0x0 = 0x00000028



Detailed Implicit Free List Example



Double-word aligned

Allocated blocks: shaded
Free blocks: unshaded
Headers: labeled with size in bytes/allocated

32 位机器，一个 word 是 32 位（4 字节）

Detailed Implicit Free List Example

- Implicit Free List
  - 优点：简单
  - 缺点：任何操作的开销都很大，线性复杂度

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)
`malloc(3)`	8	`0x9`
`malloc(11)`	16	`0x11`
`malloc(20)`	24	`0x19`
`malloc(21)`	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（因为后面有一些较的空闲块，而 first fit 开头部分很难满足，实际上总在搜索一些 unhelpful blocks）
  - 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（不会提前结束
- *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 5 有点小，可能不满足后续分配需求）

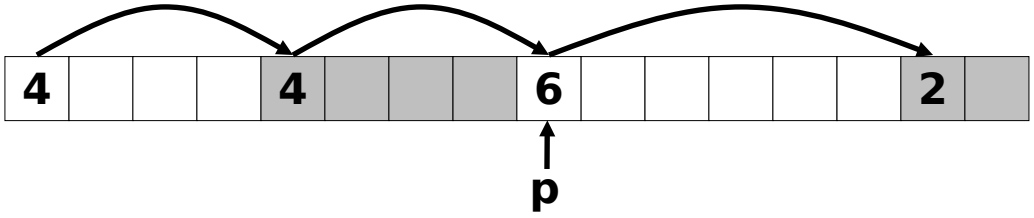
  - 如果选择 worst fit，结果如下：

Implicit List: Finding a Free Block

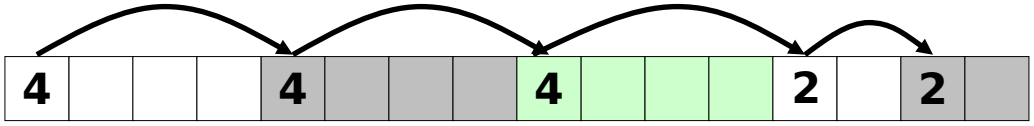
- 如果无法找到合适的 free block
  - 即使总的空闲空间够用（由于碎片
    - Fail，返回 NULL
    - 调用 sbrk() 系统调用申请增大 heap 区，增加新的物理内存，映射到新扩大的 heap 区 VA

Implicit List: Allocating in Free Block

- Allocating in a free block: *splitting*
  - Since allocated space might be smaller than free space, we might want to split the block



addblock(p, 4)



E.g., len=4;
Len=3;
Len=5

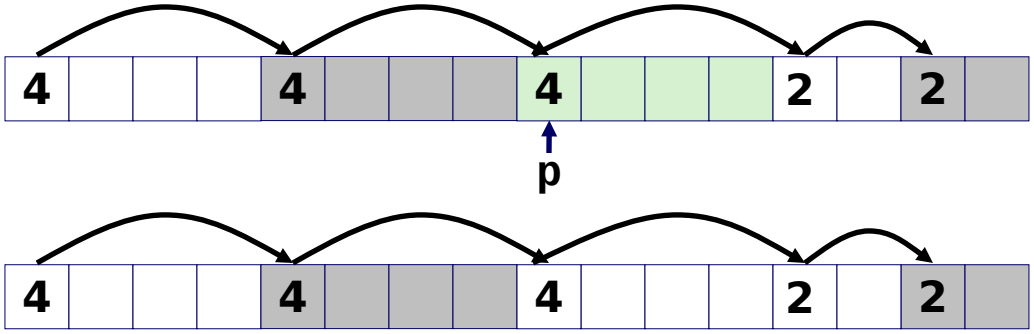
```
void addblock(ptr p, int len) {                          按 8 对齐怎么办?
  int newsize = ((len + 1) >> 1) << 1;  // round up to even
  int oldsize = *p & -2;                // mask out low bit
  *p = newsize | 1;                     // set new length
  if (newsize < oldsize)
    *(p+newsize) = oldsize - newsize;   // set length in remaining
}                                       //   part of block
```

Implicit List: Freeing a Block

- Simplest implementation:
  - Need only clear the “allocated” flag

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

  - But can lead to “false fragmentation”

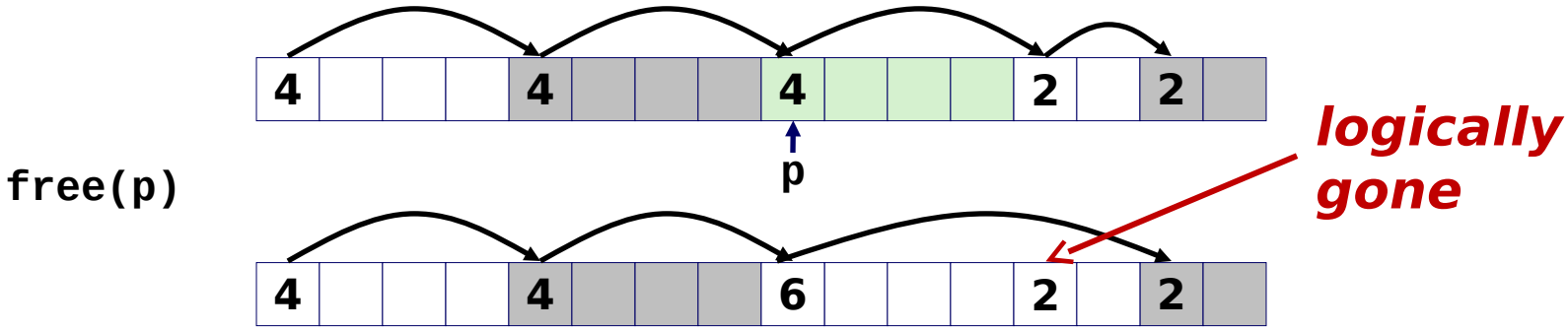


`malloc(5)` ***Oops!***

ere is enough free space, but the allocator won’t be able to find

Implicit List: Coalescing （合并）

- Join *(coalesce)* with next/previous blocks, if they are free
  - Coalescing with next block

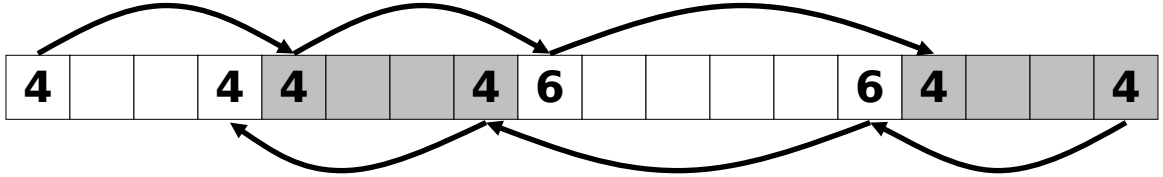



```
void free_block(ptr p) {
    *p = *p & -2;           // clear allocated flag
    next = p + *p;          // find next block
    if ((*next & 1) == 0)
      *p = *p + *next;      // add to this block if
}                           //    not allocated
```

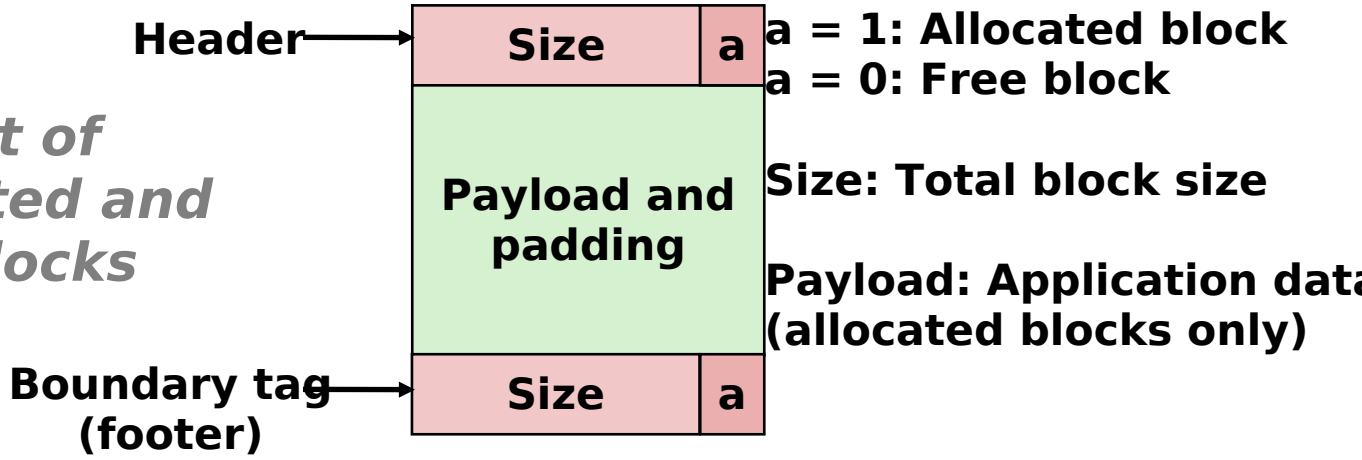
  - But how do we coalesce with *previous* block?

Implicit List: Bidirectional Coalescing

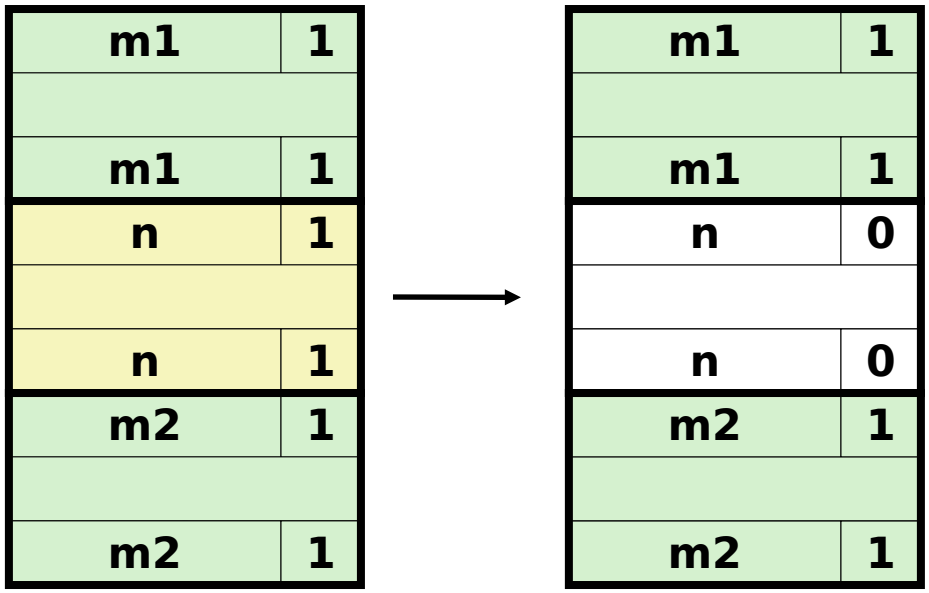
- *Boundary tags* [Knuth73]
  - Replicate size/allocated word at "bottom" (end) of free blocks
  - Allows us to traverse the "list" backwards, but requires extra space
  - Important and general technique!



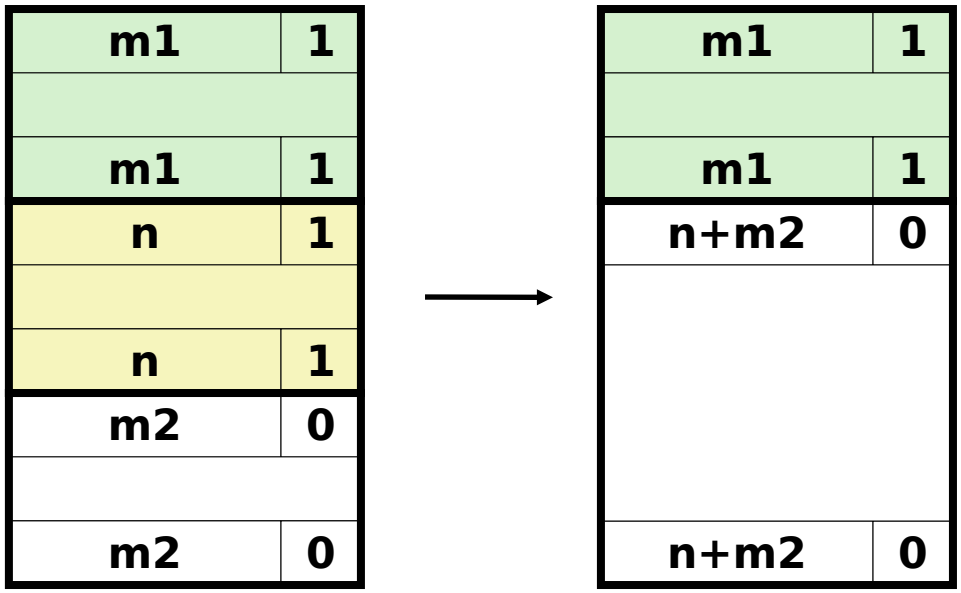
Format of allocated and free blocks



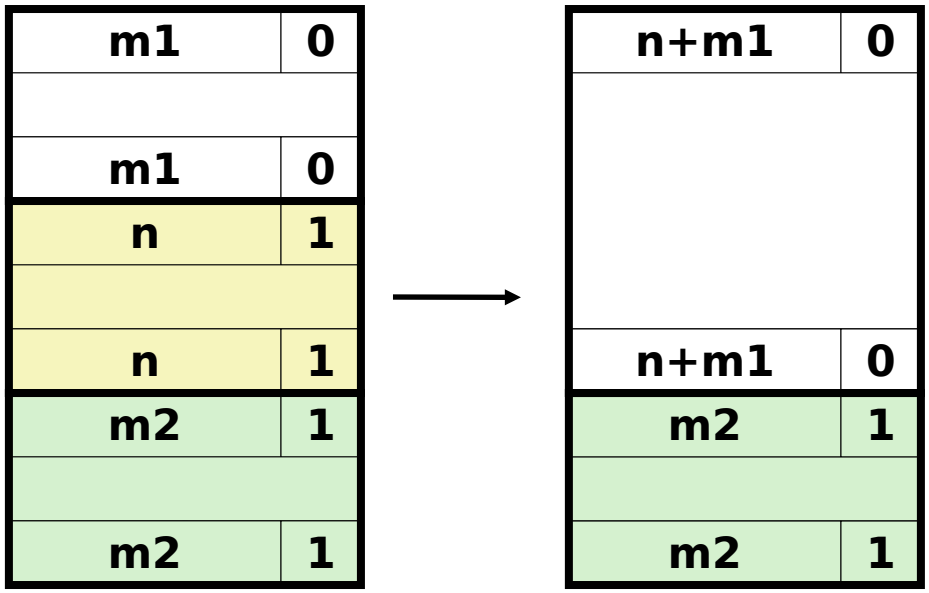
Constant Time Coalescing (Case 1)



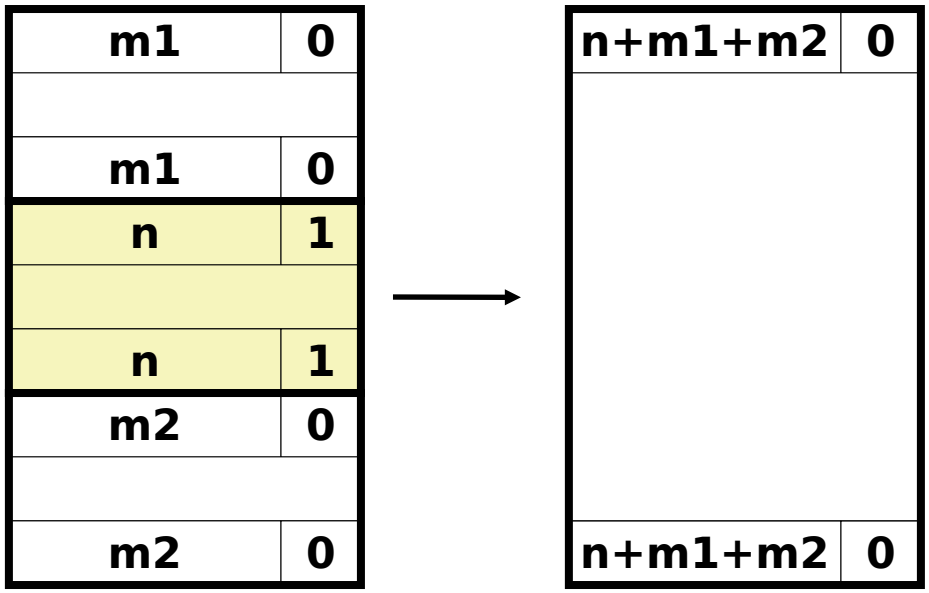
Constant Time Coalescing (Case 2)



Constant Time Coalescing (Case 3)



Constant Time Coalescing (Case 4)



Disadvantages of Boundary Tags

- Internal fragmentation

- Can it be optimized?
  - Which blocks need the footer tag?
    - 前面块是空闲块才需要 footer
  - What does that mean?
    - Header 空闲位增加记录前一块是空闲，还是占用
    - 空闲块还是有 footer，不浪费
    - 占用块不需要 footer
    - 占用块释放时，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 ，头部和脚部都是 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

实现一个简单的内存分配器

假设

- 基于隐式空闲链表
- 最大块 4GB （2^32）
- 代码是 64 位干净的，即可以在 32 位 /64 位下不加修改的正确运行

- 自学

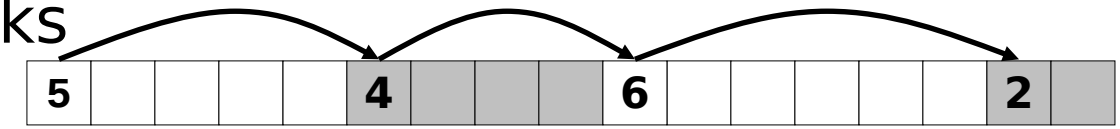
Dynamic Memory Allocation: Advanced Concepts

Outline

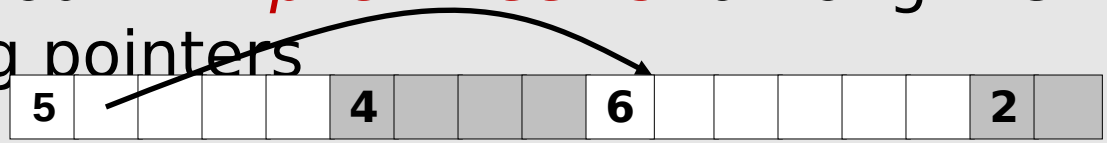
- Explicit free lists
- Segregated free lists
- Garbage collection
- Memory-related perils and pitfalls

Keeping Track of Free Blocks

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



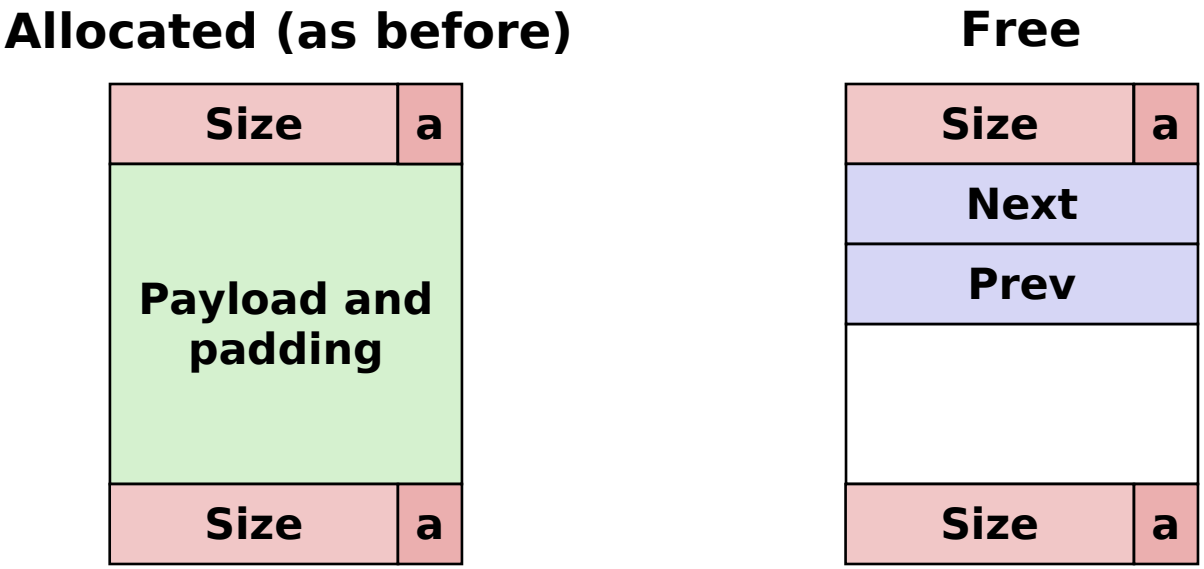
- Method 2: *Explicit free list* among the free blocks using pointers



- 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

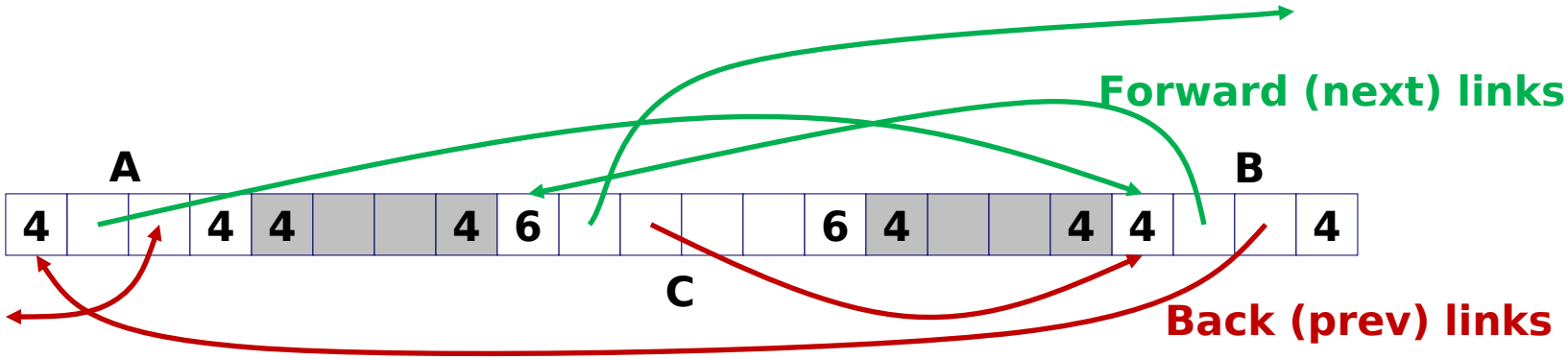


- Maintain list(s) of *free* blocks, not *all* blocks
  - “next” free block 可能在任何地方
    - 不像之前 implicit list 中， next 是在相邻的内存
    - 所以必须有 forward/backward pointers ，不能只是记录 size
  - 仍然需要 boundary tag ，用于合并
  - 幸运的是，以上需求只针对 free block ，所以不需要占用有效内存空间

Explicit Free Lists

- Logically:

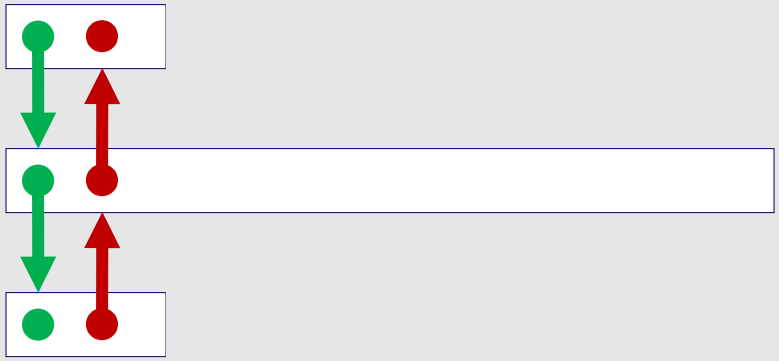
- Physically: blocks can be in any order



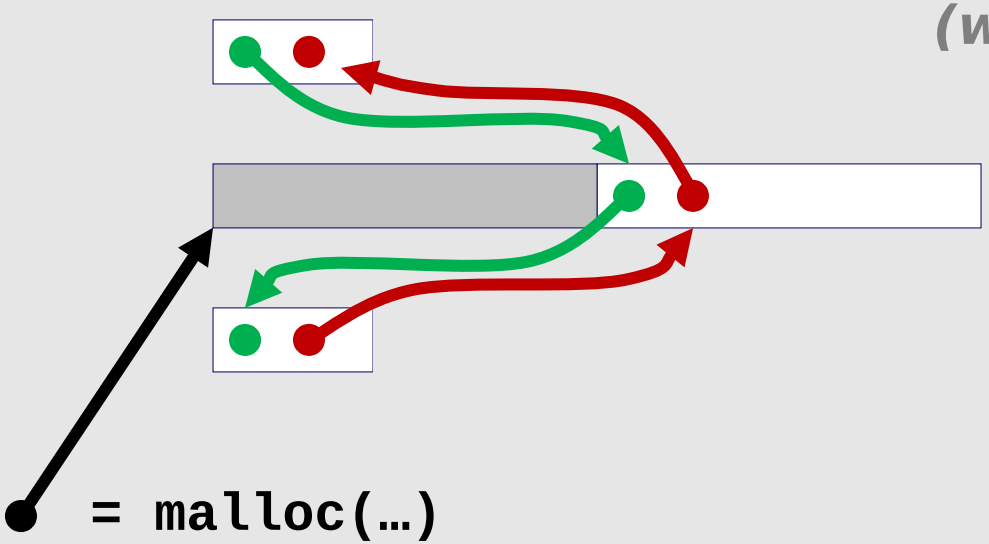
Allocating From Explicit Free Lists

conceptual graphic

Before



After *(with splitting)*



Freeing With Explicit Free Lists

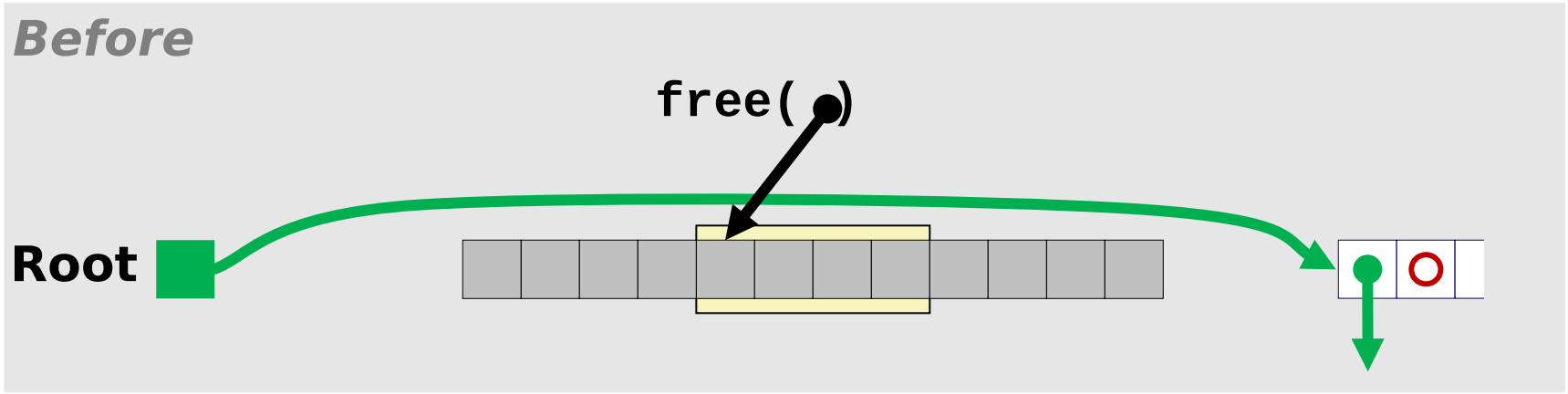
- *Insertion policy*: Where in the free list do you put a newly freed block?
- **LIFO (last-in-first-out) policy**
  - Insert freed block at the beginning of the free list
  - ***Pro:*** simple and constant time
  - ***Con:*** studies suggest fragmentation is worse than address ordered

- **Address-ordered policy**
  - Insert freed blocks so that free list blocks are always in address order:

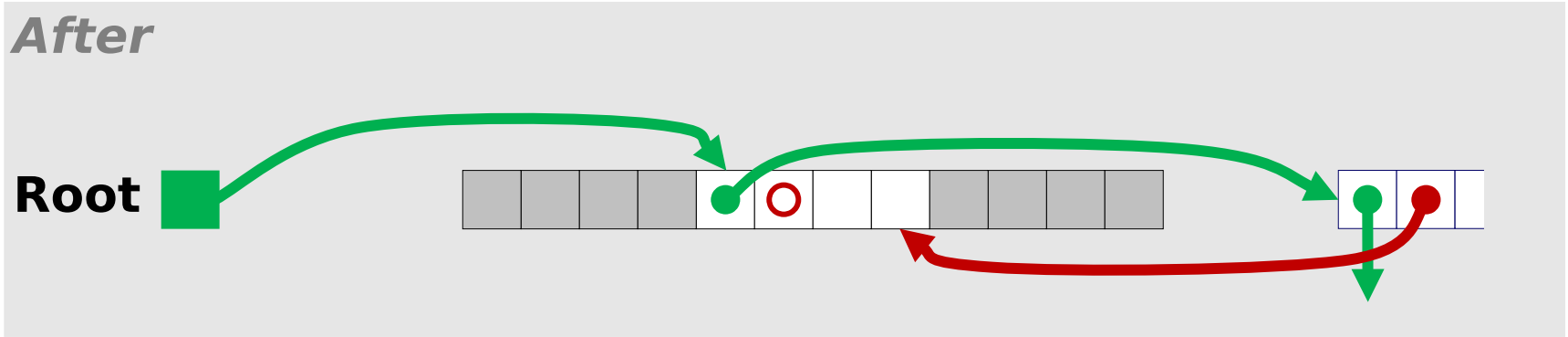
    addr(prev) < addr(curr) < addr(next)
  - ***Con:*** requires search
  - ***Pro:*** studies suggest fragmentation is lower than LIFO

Freeing With a LIFO Policy (Case 1)

conceptual graphic

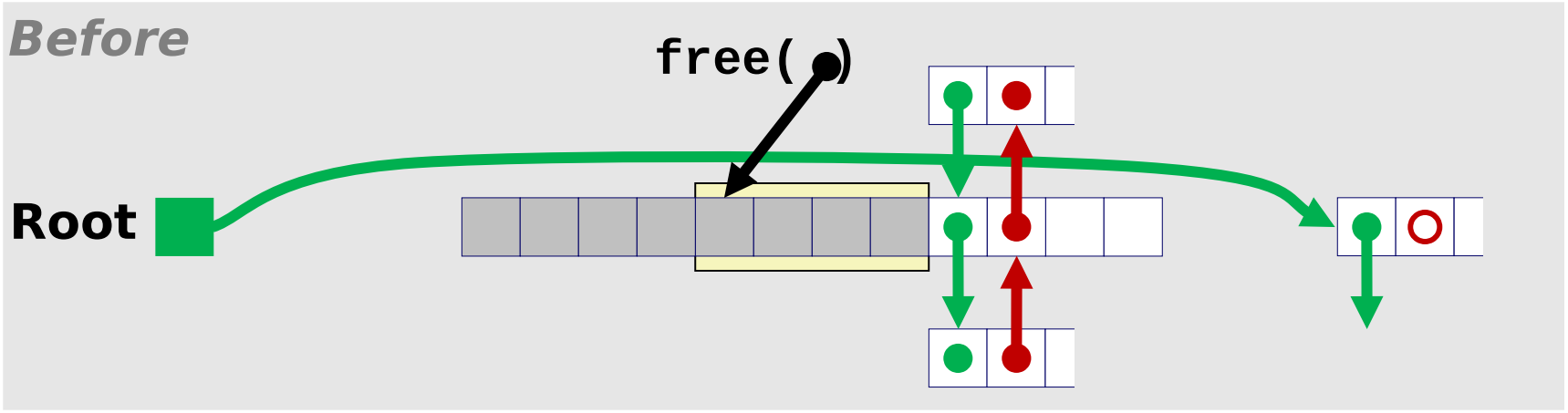


- Insert the freed block at the root of the list

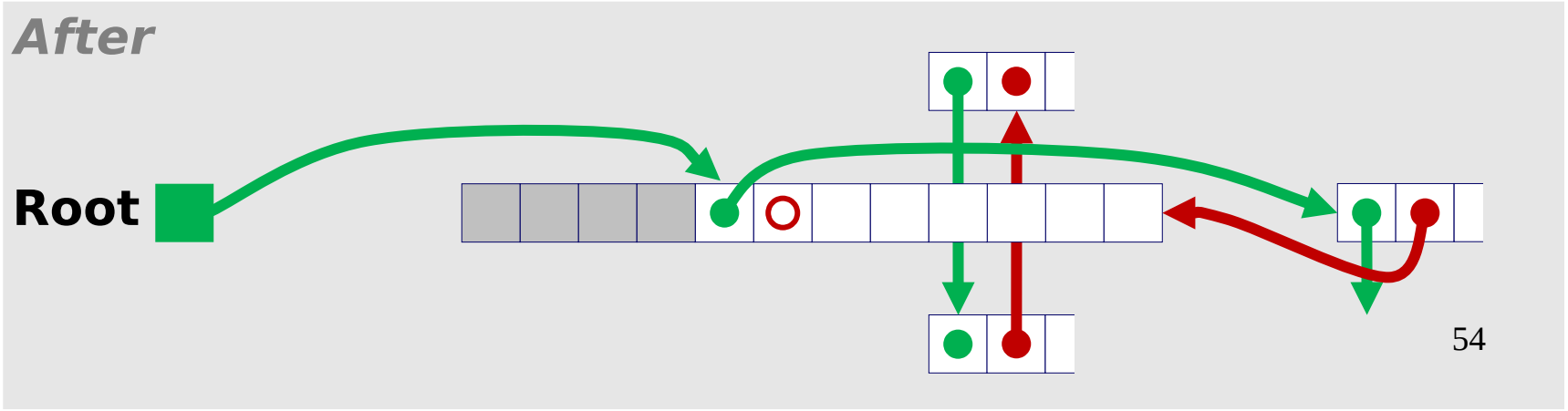


Freeing With a LIFO Policy (Case 2)

conceptual graphic

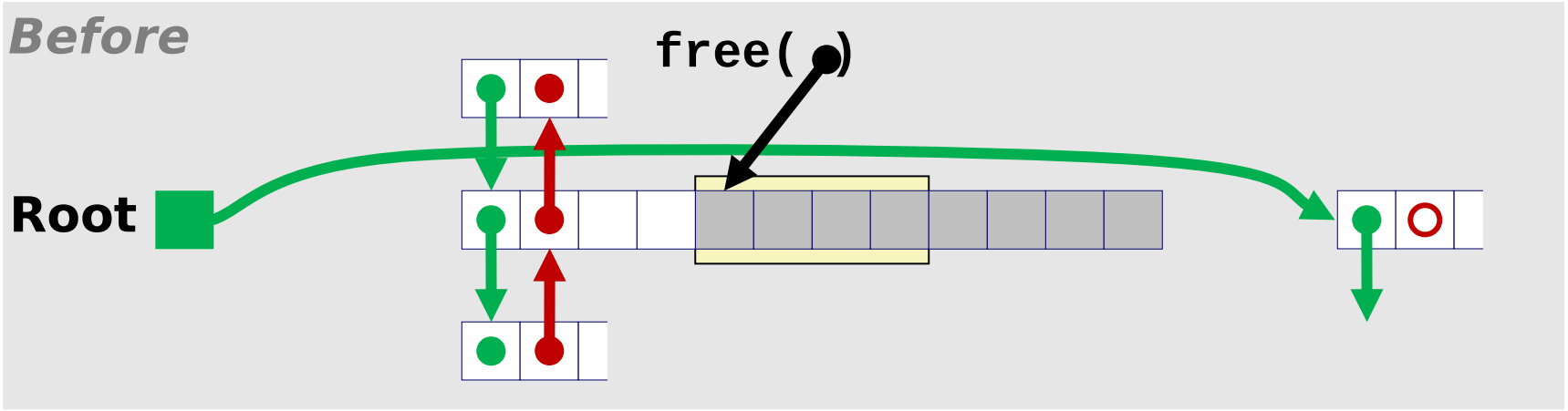


- 如果新插入的 free block 能够和其他 free block 合并，那么合并后的新块会放在队首（root 指针指向的位置）

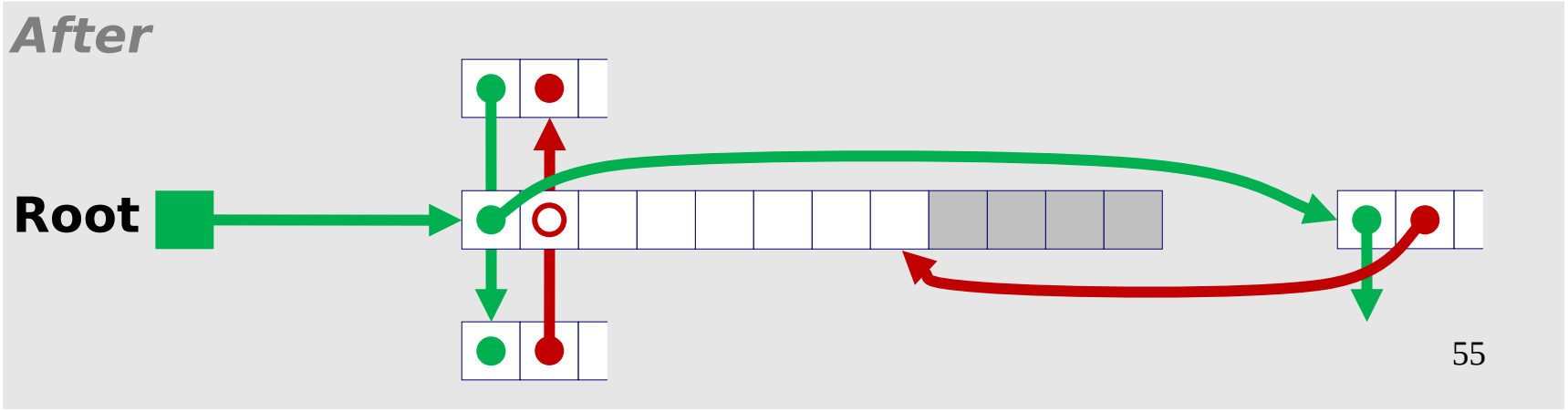


Freeing With a LIFO Policy (Case 3)

conceptual graphic

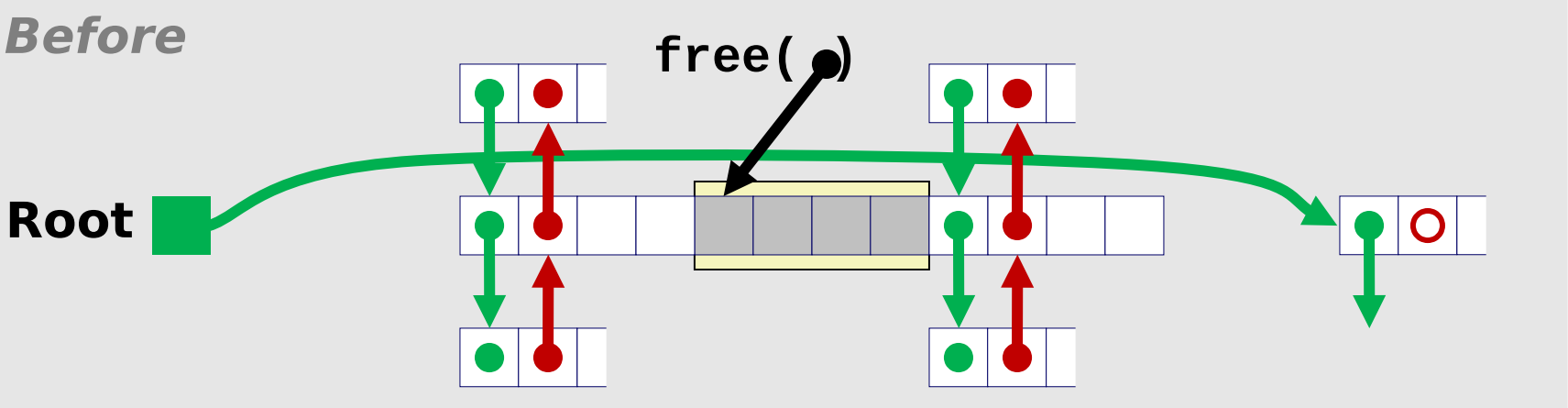


- 向后合并，同样的处理方式

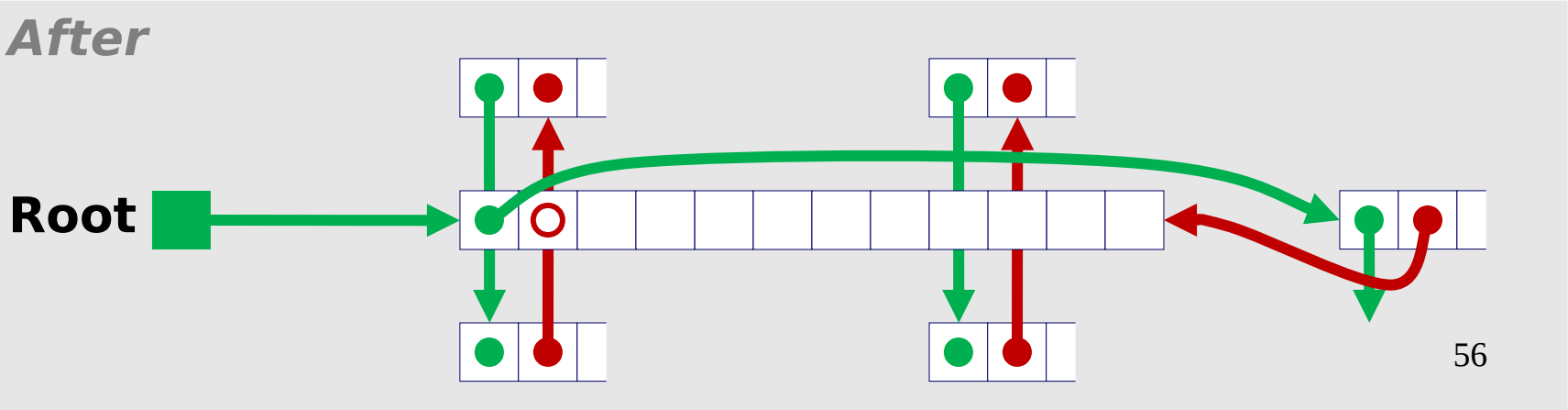


Freeing With a LIFO Policy (Case 4)

conceptual graphic



- 同时向前、向后 3 个 free blocks 合并，同样的处理方式



Explicit List Summary

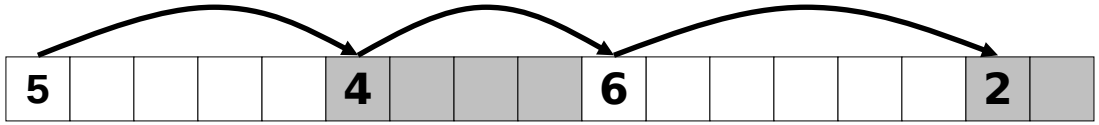
- 与 implicit list 比较：
  - 分配的时间开销与 free blocks 数量线性相关，而不是与所有 blocks
    - 当内存占用比较满的情况下，时间复杂度低很多
  - 释放块的复杂度略有增加，因为要遍历寻找合并目标
  - 链接需要一些额外的空间 (2 extra words needed for each block)
    - 会增加 internal fragmentation 吗？
- 这种方面一般是作为 segregated free lists 中每个独立队列的解决方案（很少独立使用）
  - 每个 size 范围的块用一个 explicit free list 进行管理

Today

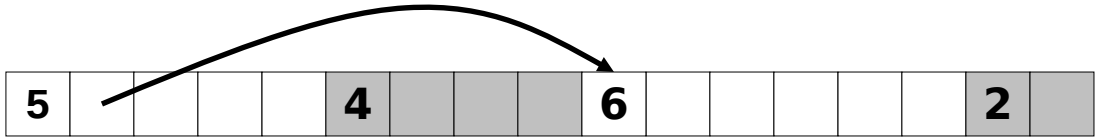
- Explicit free lists
- Segregated free lists
- Garbage collection
- Memory-related perils and pitfalls

Keeping Track of Free Blocks

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



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

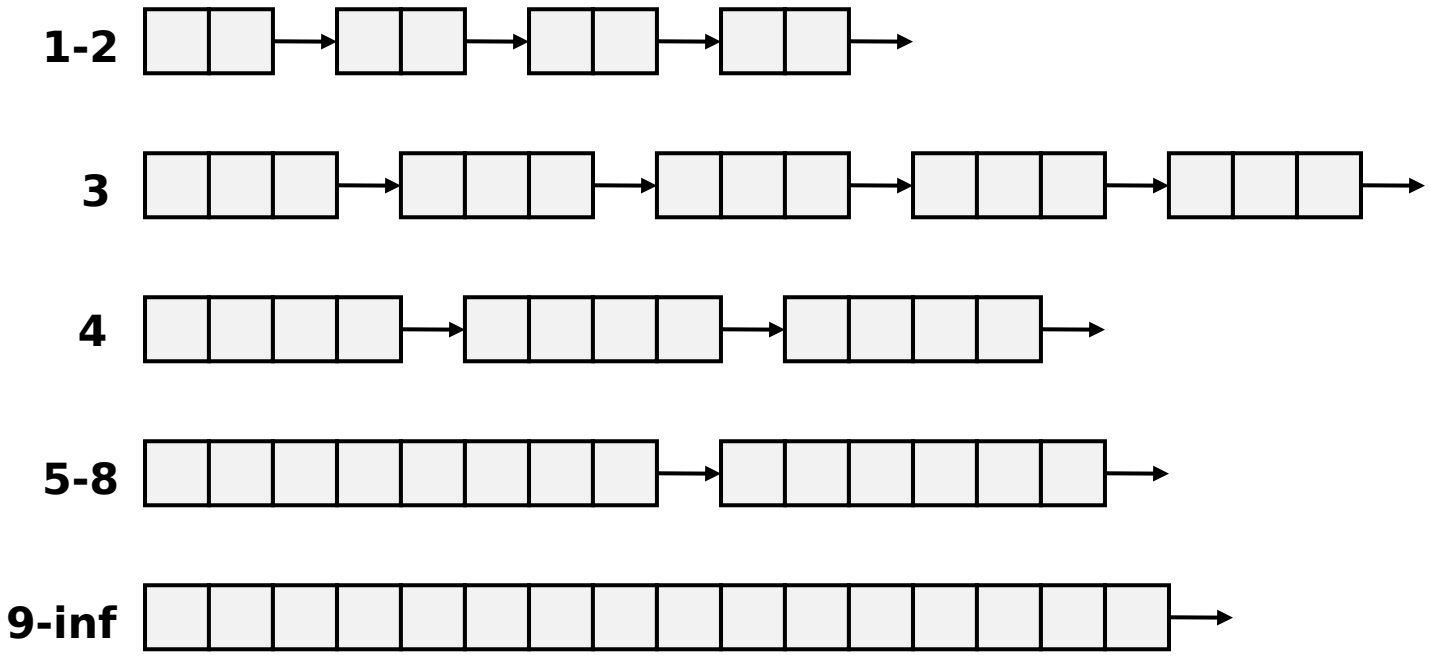


- Method 3: *Segregated free list*
  - Different free lists for different size classes

- Method 4: *Blocks sorted by size*
  - Can use a balanced tree (e.g. Red-Black tree) with 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，空闲空间就浪费（内部碎片）
    - 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，不分割，不合并
    - 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 > 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

Seglist Allocator (cont.)

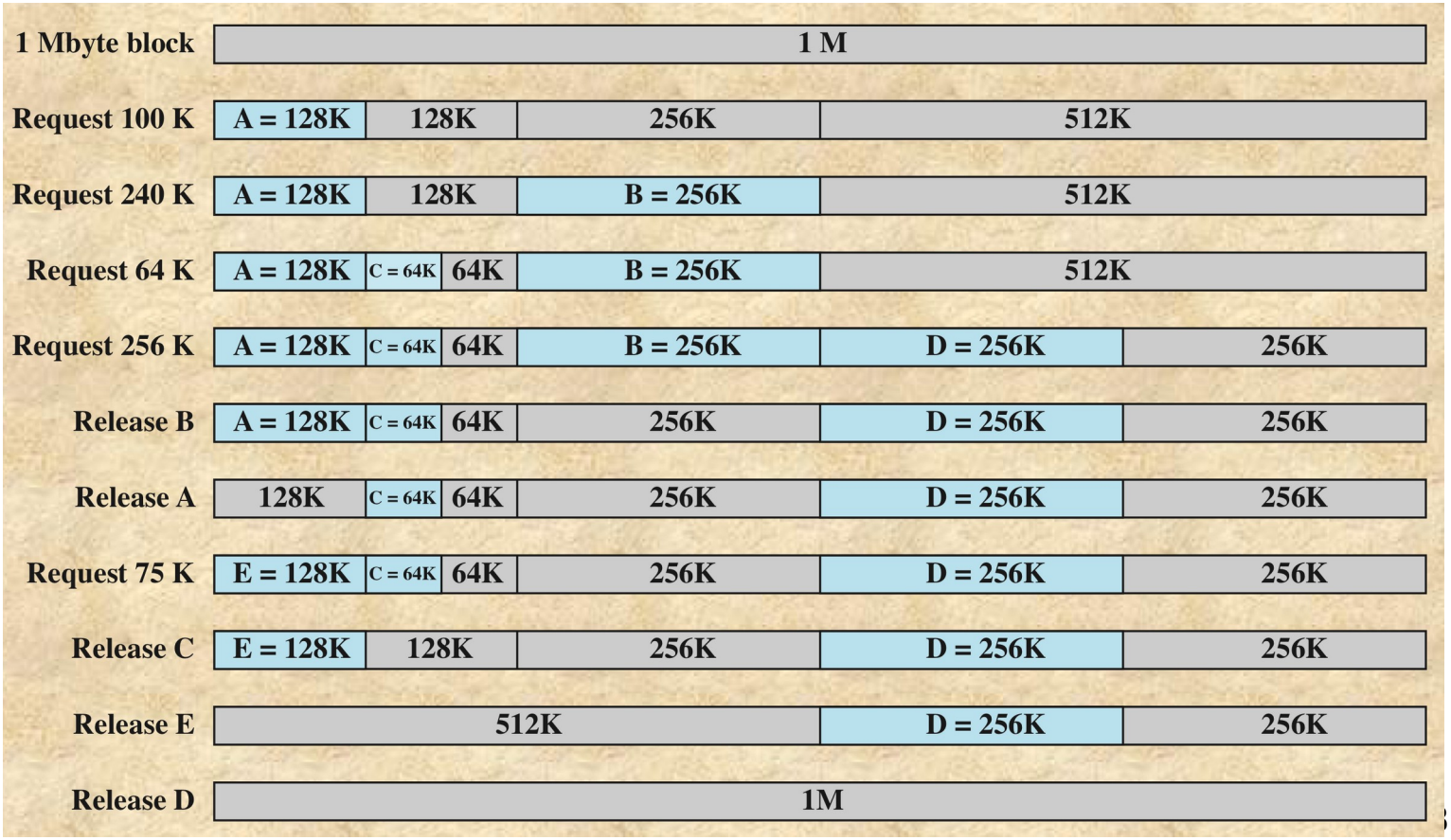
- To free a block:
  - Coalesce and place on appropriate list
- Advantages of seglist allocators
  - Higher throughput
    - log time for power-of-two size classes
    - 2^n，粗粒度，但简单高效（抓住主要矛盾
  - Better memory utilization
    - First-fit search of segregated free list approximates a best-fit search of 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

Buddy System

- 伙伴系统，是分离适配的一个特例
  - 每个大小类都是 2 的整数幂
- 堆空间共 2^m，组织为 2^k 大小的一组块（0<=k<=m）
  - 最开始只有一个 2^m 大小的块
- 为了分配 2^k 大小的块，分配我们找到的第一个 2^j 大小的可用块（k<=j<=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
  - 每次剩下的半块（即伙伴）被放到相应的 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

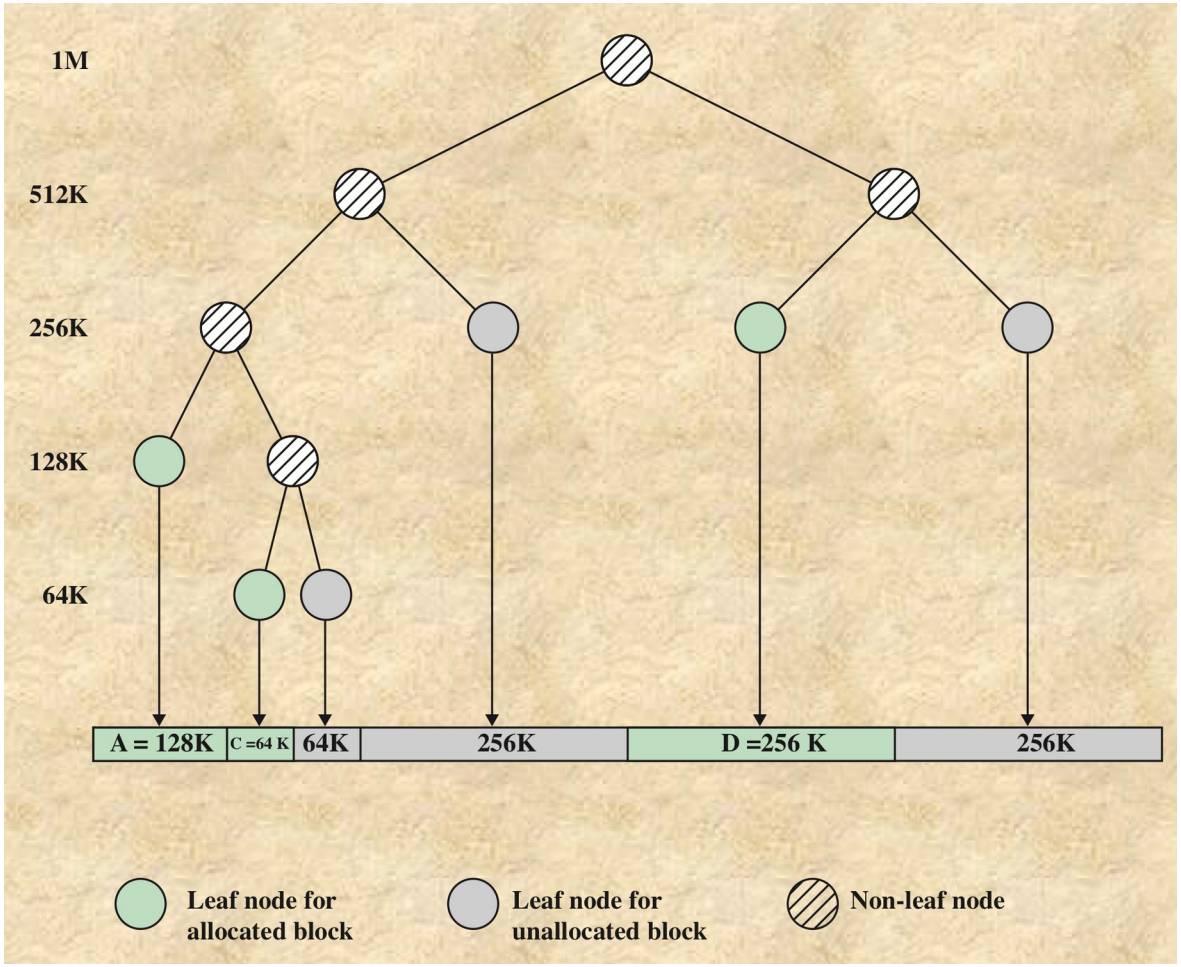
Buddy System

- 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（美感
- 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

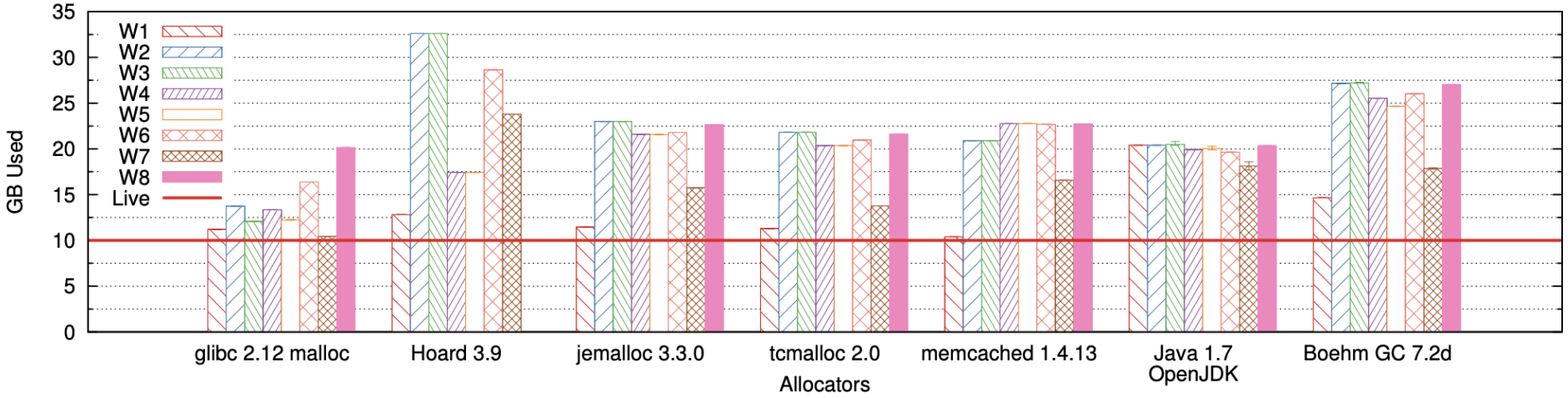
More Info on Allocators

- D. Knuth, "*The Art of Computer Programming*", 2nd edition, Addison Wesley, 1973
  - The classic reference on dynamic storage allocation

- Wilson et al, "*Dynamic Storage Allocation: A Survey and Critical Review*", Proc. 1995 Int'l Workshop on Memory Management, Kinross, Scotland, Sept, 1995.
  - Comprehensive survey

Log-structured malloc

- 目前的 malloc 版本（c lib）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 malloc

- #1. Non-copying allocator
  - 不移动已分配的块，性能好，但碎片多
  - 尤其不适应 workload 特性变化的情况，例如删除 10GB small objects，然后写入 10GB large objects
  - 这种特性变化的情况对单个程序可能不容易发生，但对通用的存储系统来说却很容易发生

Log-structured malloc

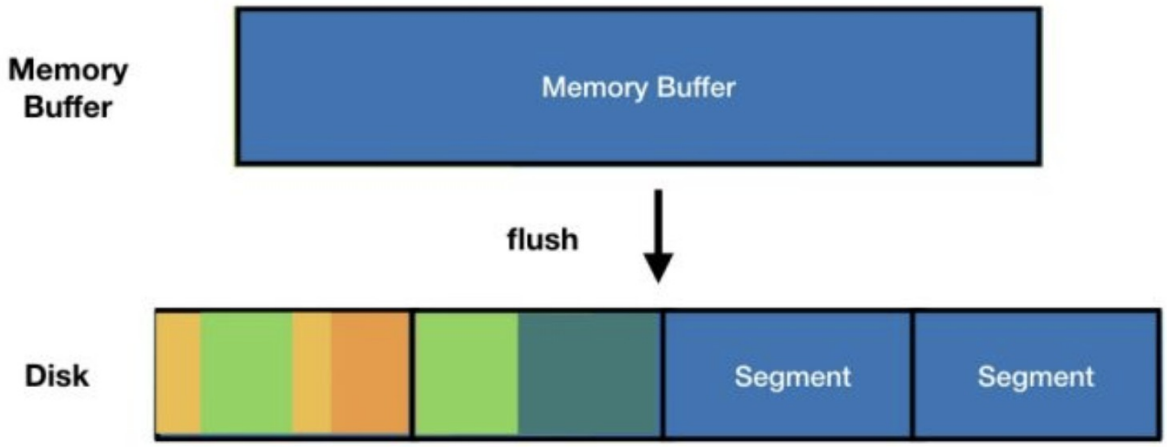
- #2. Copying allocator
  - 通过移动 allocated blocks 来减少碎片，但性能开销大
  - 所以一般都采用延迟 garbage collection (GC) 的方式
  - 因此，一般需要已分配数据 1.5-5 倍的空间来保障高性能
  - 另一个麻烦，当 allocated blocks 移动后，必须 halt threads 来更新指针的地址
    - 现有最好的方案，也可能导致上百 ms 的 pause time
    - Java collector 可能导致 3-4 秒的 pause time

Log-structured malloc

- 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. (被引用 2176 次 , google scholar)
    - 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

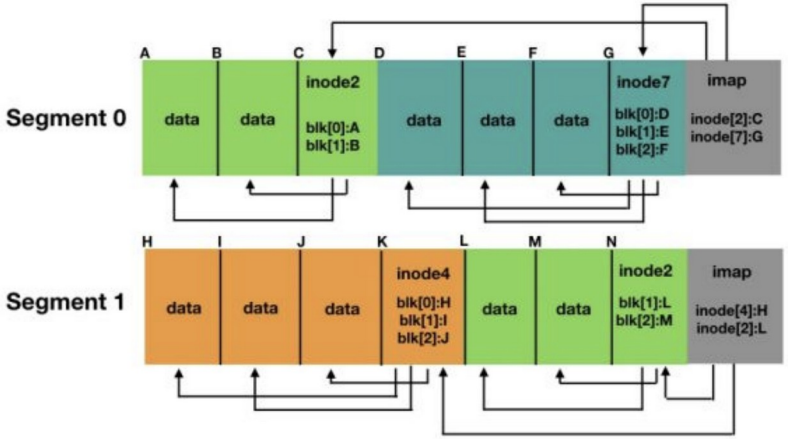
Log-structured malloc

- Log-structured file system
  - 磁盘空间有限，需要回收空间
  - 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内存中；当内存中缓存的数据超过段的大小后， LFS 将数据一次性写入到空闲的段中



Log-structured malloc

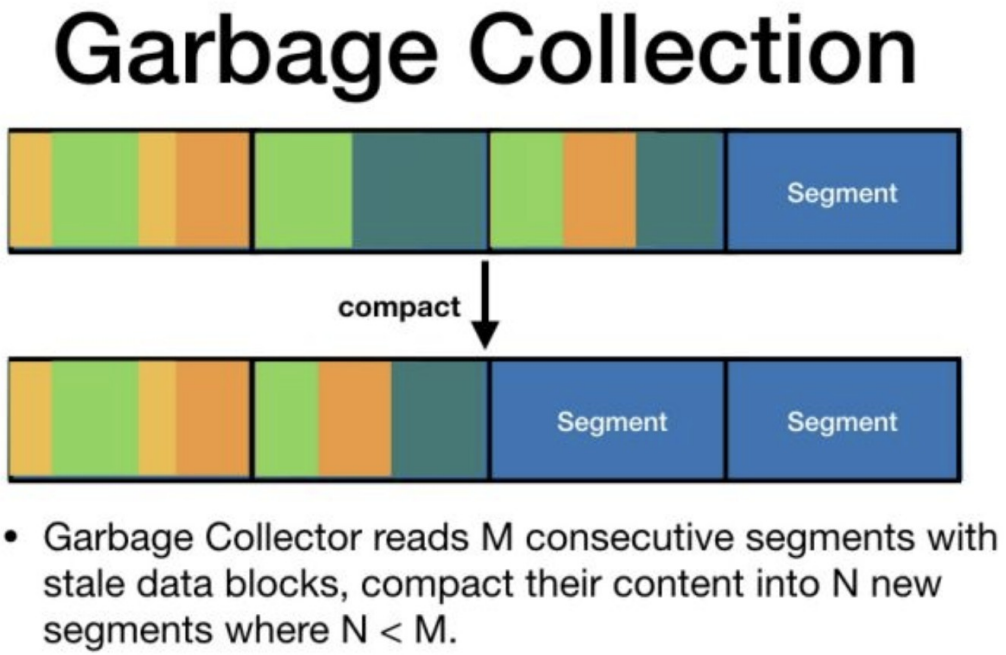
- Log-structured file system
  - 读操作
    - 每个 segment 中也包括了元数据（inode），通过一层一层的 inode，找到目标文件数据
    - LFS 是 append only，所以对同一个文件的同一个 data block 可能存在多个版本（在不同段中）。但是通过比较不同段的更新时间，LFS 就能判断出哪个 segment 中的 data block 是最新



- Each segment has an inode map
- inode map are further cached into main memory

Log-structured malloc

- 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



Log-structured malloc

- Log-structured file system
  - 新的文件系统例如 btrfs 也基于 LSM append only 的特点实现了 copy-on-write 或是 multi-version 的特性
  - LFS 性能比一般文件系统差（数据拷贝多），但是更适合 SSD 等新硬件（内部不能原位更新），应用开始增多
  - 优点是连续写（批处理写），充分利用硬件特性
  - Log-structured 的思想在其他领域也开始应用
    - Log-structured merge tree (LSM-tree)
    - Log-structured malloc

Log-structured malloc

- Rumble S M, Kejriwal A, Ousterhout J. Log-structured memory for DRAM-based storage[C]//Proceedings of the 12th {USENIX} Conference on File and Storage Technologies ({FAST} 14). 2014: 1-16.
  - Stanford, RAMCloud 团队
- Log-structured malloc
  - Treating memory as a sequentially written log
  - 80-90% memory utilizations while providing high performance
  - 新申请的内存在后面追加，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，否则无法有效清除数据
    - 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

顶级系统 Lab

- UC Berkeley AMPLab -> RiseLab
  - Spark, Mesos, Alluxio, GraphX, Ray
    - https://amplab.cs.berkeley.edu
- Standford, PlatformLab
  - John Ousterhout
  - RAMCloud, Raft
    - https://platformlab.stanford.edu
  - A Philosophy of Software Design
    - https://my.oschina.net/taogang/blog/1940597
- MIT CSAIL PDOS group
  - CSAIL: Computer Science & Artificial Intelligence Lab
  - PDOS: Parallel & Distributed Operating Systems Group
    - MIT 6.828 “Operating Systems Engineering”
    - https://pdos.csail.mit.edu/publications/

Ion Stoica
istoica@EECS.Berkeley.EDU

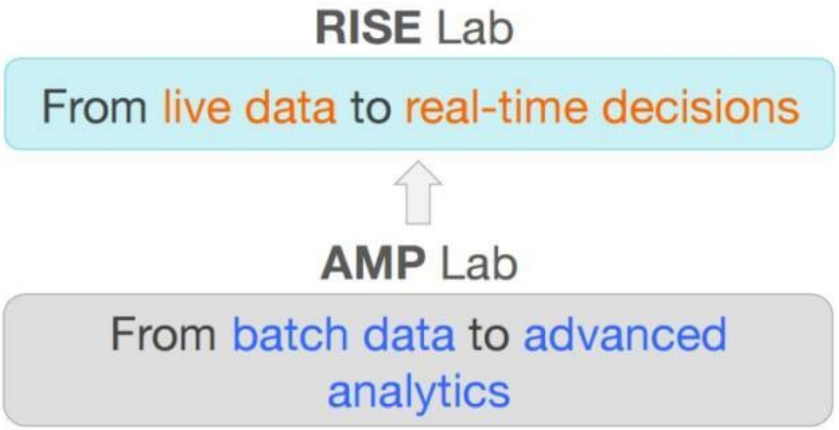
Michael Jordan
jordan@CS.Berkeley.EDU

Dave Patterson
pattrsn@CS.Berkeley.EDU

Robert Morris
LEAD

顶级系统 Lab

- RISELab
  - Spark 下一代：
    - 一款分析工具——较 Spark 延迟降低 100 倍，但吞吐量达到 Spark 的 1000 倍；
    - 机器学习算法会实时对输入不可见的噪声数据进行分析，得出可靠的结果；
    - 确保用户隐私与应用程序的安全性。

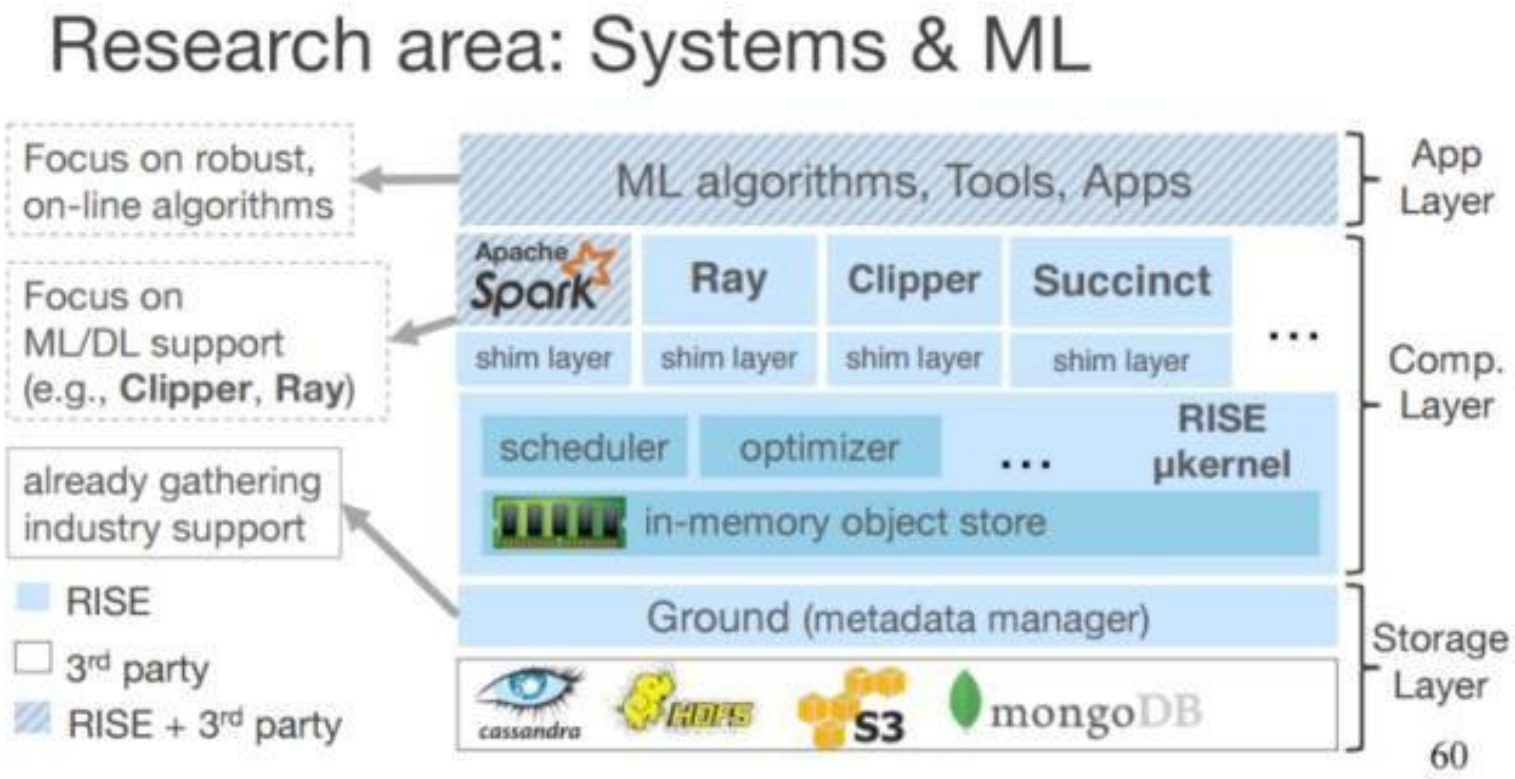


Challenges	RISE Lab
Automated decisions on live data are hard	Real-time, sophisticated decisions that guarantee worst-case behavior on noisy and unforseen live data
Poor security: exploits are daily occurrences	Ensure privacy and integrity without impacting functionality
One-off solutions, expensive and slow to build	General platform: **Secure Real-time Decision Stack**

31

顶级系统 Lab

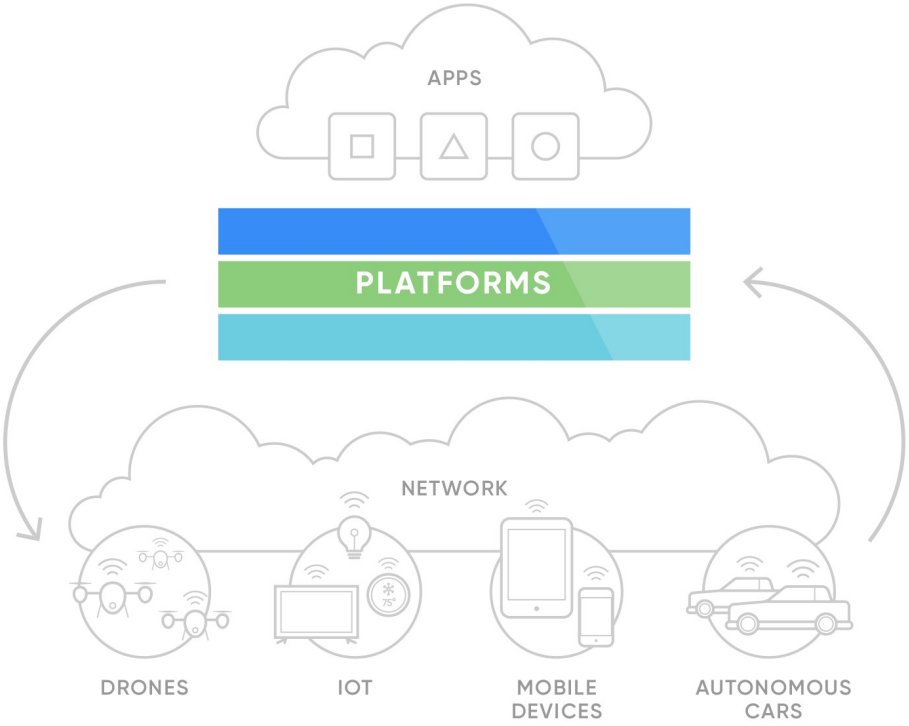
- RISELab
  - Focus on 实时分析 &ML，核心是内存存储



顶级系统 Lab

BIG CONTROL IS THE NEXT BIG THING.

We build platforms to enable control of big swarms of collaborative devices such as drones, self-driving cars, robots, and even datacenters.



Log-structured malloc

- 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 来防止，相似 age 的 object 可能一起失效，减少碎片

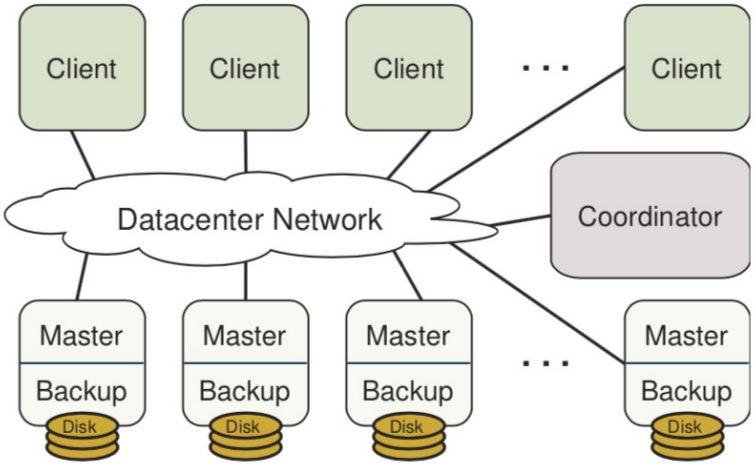
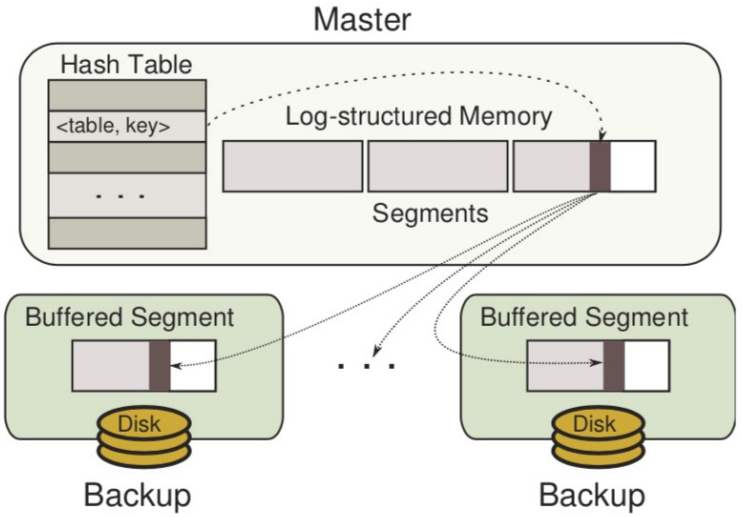


Figure 2: RAMCloud cluster architecture.



Log-structured malloc

- 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
    - System paper 三要素：
      - Abstraction
        - » Impose structure on unstructured problem
      - Trade-off
      - Performance

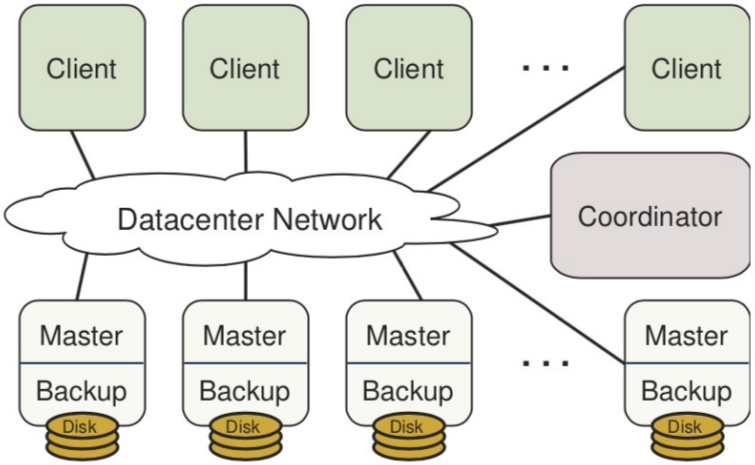
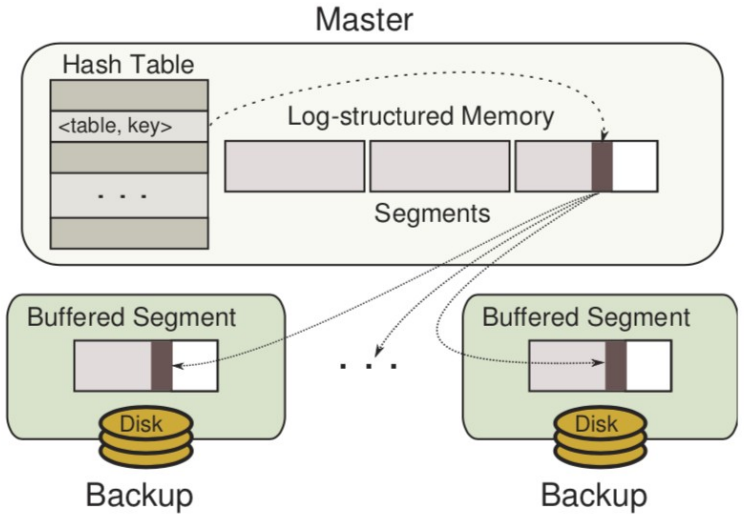


Figure 2: RAMCloud cluster architecture.



Today

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

Implicit Memory Management: Garbage Collection

- *Garbage collection:* 自动回收一分配的空间，应用不用负责 free （程序员可能忘了释放内存，内存泄漏）

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

- Common in many dynamic languages:
  - Python, Ruby, Java, Perl, ML, Lisp, Mathematica

- Variants (“conservative” garbage collectors) exist for C and C++
  - However, cannot necessarily collect all garbage

Garbage Collection

- 内存管理者如何指导那些内存能够被释放呢?
  - 一般我们不知道将来哪些内存会被使用
  - 但是如果没有指针指向的内存，可以确定将来没人使用（因为没办法使用）

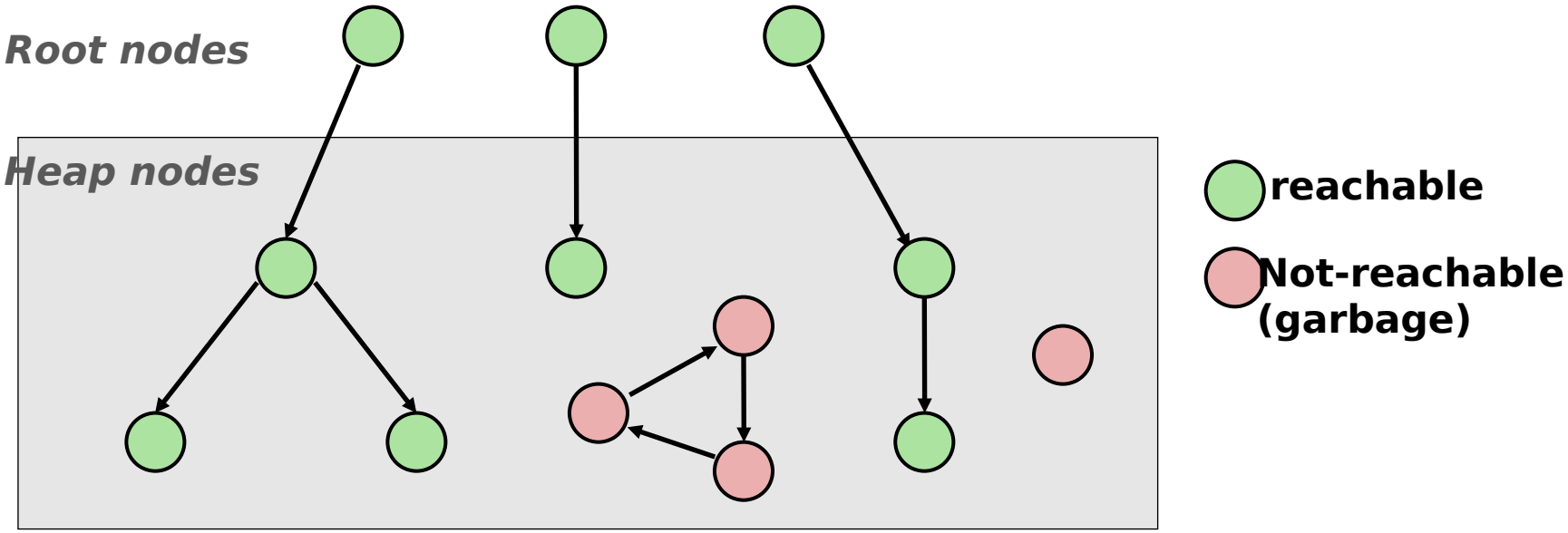
- 关于指针，必须首先确定的假设:
  - 内存管理者必须能够区分指针和非指针（防止意外碰上产生干扰）
  - 所有指针都指向数据块的起始地址
  - 不能隐藏指针（例如把指针强制转换为 int 存储，使用时再强制转回来）

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.

Memory as a Graph

- We view memory as a directed graph
  - Each block is a node in the graph
  - Each pointer is an edge in the graph
  - Locations not in the heap that contain pointers into the heap are called ***root*** nodes (e.g. registers, locations on the stack, global variables)

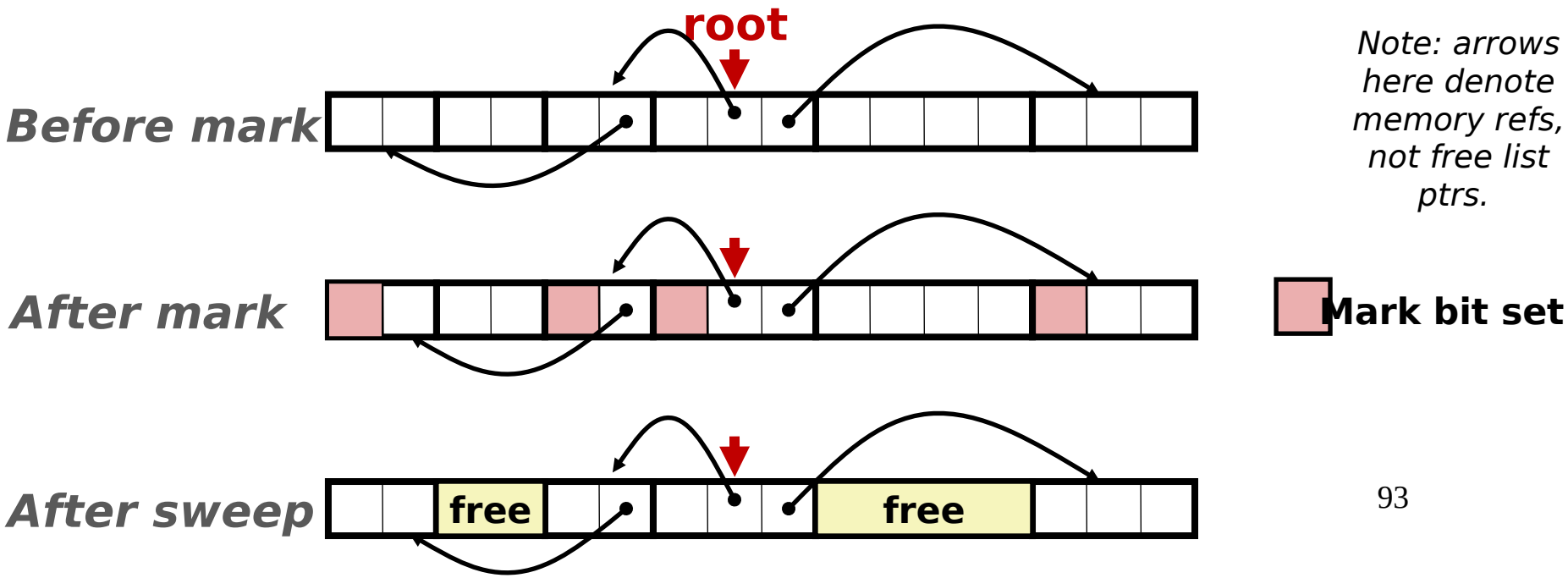


A node (block) is *reachable* if there is a path from any root to that node.

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

Mark and Sweep Collecting

- Can build on top of malloc/free package
  - Allocate using `malloc` until you "run out of space"
- When out of space:
  - 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



Note: arrows here denote memory refs, not free list ptrs.

Assumptions For a Simple Implementation

- Application
  - **`new(n):`** returns pointer to new block with all locations 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 collectors

- Instructions used by the Garbage Collector
  - **`is_ptr(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

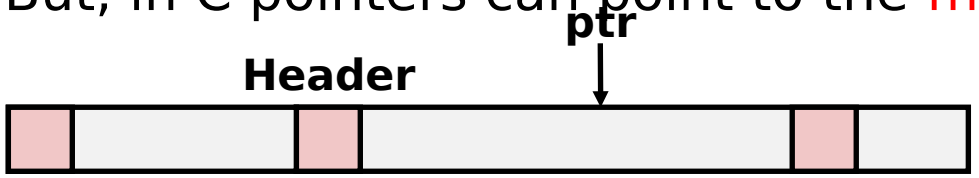
```
ptr mark(ptr p) {
    if (!is_ptr(p)) return;          // do nothing if not pointer
    if (markBitSet(p)) return;       // check if already marked
    setMarkBit(p);                   // set the mark bit
    for (i=0; i < length(p); i++)    // call mark on all words
      mark(p[i]);           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

Sweep using lengths to find next block

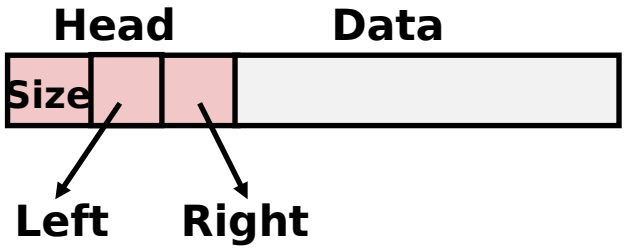
```
ptr sweep(ptr p, ptr end) {
   while (p < end) {
      if markBitSet(p)
         clearMarkBit();
      else if (allocateBitSet(p))
         free(p);
      p += length(p);
}
```

Conservative Mark & Sweep in C

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



- So how to find the beginning of the block?
  - Can use a balanced binary tree to keep track of all allocated blocks (key is start-of-block)
  - Balanced-tree pointers can be stored in 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 addresses

Today

- Explicit free lists
- Segregated free lists
- Garbage collection
- Memory-related perils and pitfalls （危险、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!

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

Source: K&R Sec 5.12

间接引用坏指针

- The classic scanf bug

```
int val;

...

scanf(“%d”, val);
```

读未初始化的内存

- Assuming that heap data is initialized to zero

  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*sizeof(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

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*
`() [] -> .`	left to right
`! ~ ++ -- + - * & (type) sizeof`	right to left
`* / %`	left to right
`+ -`	left to right
`<< >>`	left to right
`< <= > >=`	left to right
`== !=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right
`?:`	right to left
`= += -= *= /= %= &= ^= != <<= >>=`	right to left
`,`	left to right

- ->, ( ), and [ ] have high precedence, with * and & just below
- Unary +, -, and * have higher precedence than binary forms

Source: K&R page 53

引用不存在的变量

- Forgetting that local variables disappear when a function returns

```
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