



CSCI-UA.0201

Computer Systems Organization

Dynamic Memory Allocation

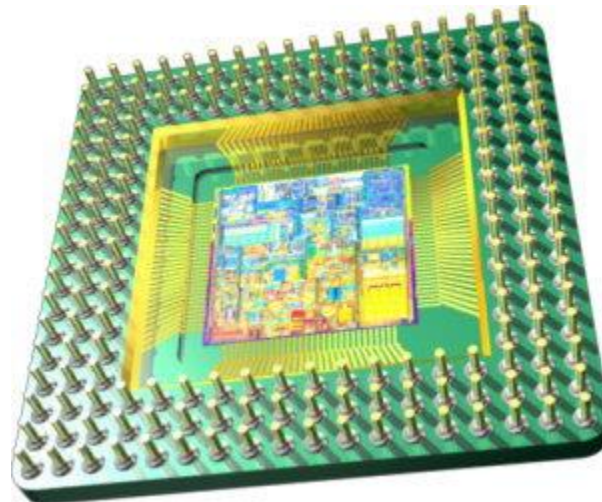
Mohamed Zahran (aka Z)

mzahran@cs.nyu.edu

<http://www.mzahran.com>

Slides adapted
(and slightly modified)
from:

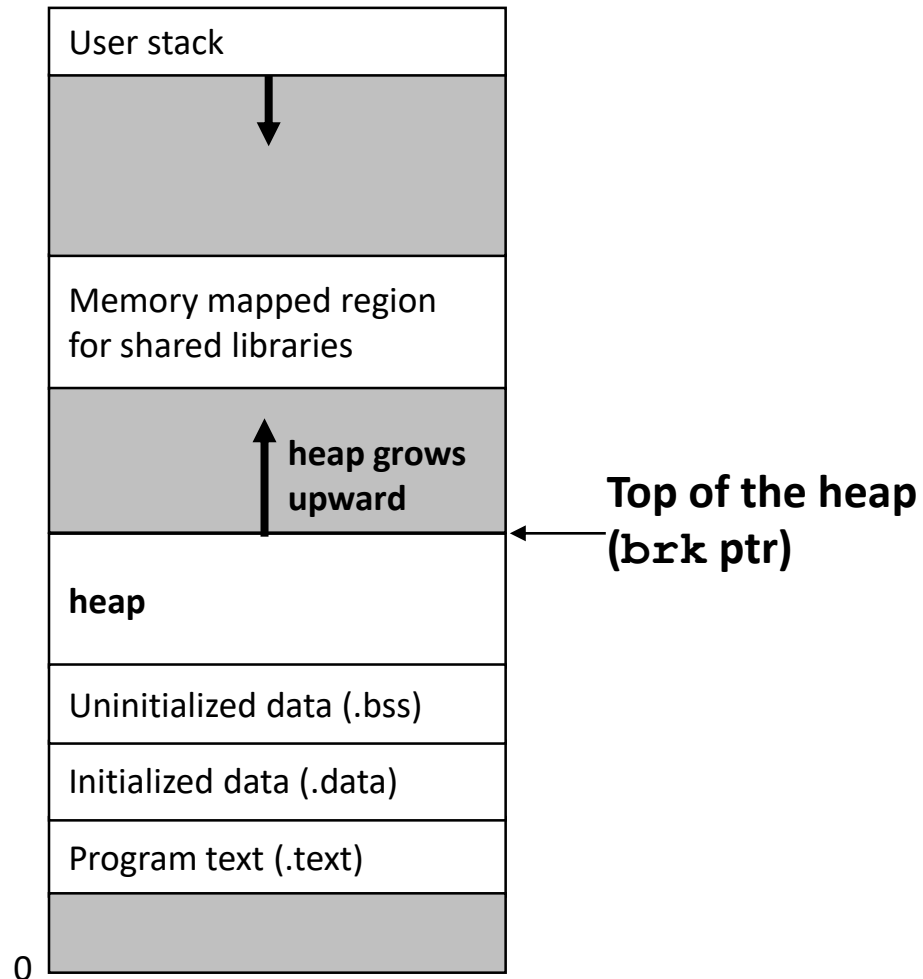
- Clark Barrett
- Jinyang Li
- Randy Bryant
- Dave O'Hallaron



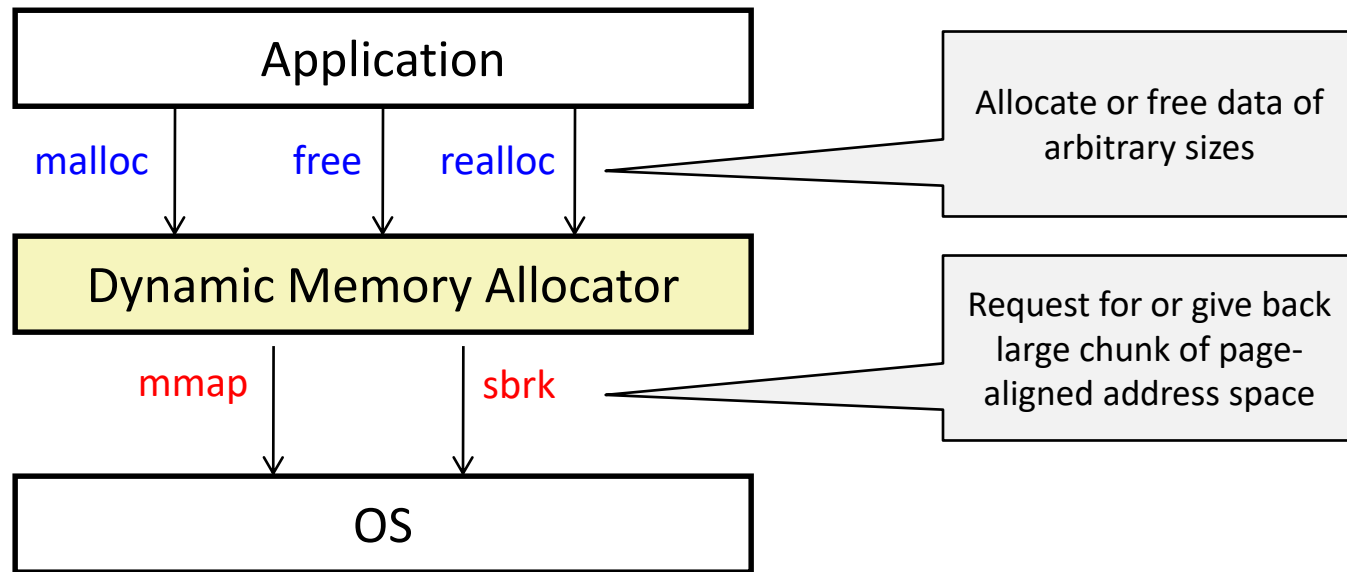
Why dynamic allocator?

- We've discussed two types of data allocation so far:
 - Global variables
 - Stack-allocated local variables
- Not sufficient!
 - How to allocate data whose size is only known at runtime?
 - E.g. when reading variable-sized input from network, file etc.
 - How to control lifetime of allocated data?
 - E.g. a linked list that grows and shrinks as items are inserted/deleted

The Heap



Dynamic Memory Allocation



Dynamic memory allocator is part of user-level library.
Why not implement its functionality in the kernel?

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int
prot, int flags, int fd, off_t offset);
```

- Read **length** bytes from file **fd** after bypassing **offset** bytes.
- Put those bytes read in memory, preferably at address **addr**.

flags:

- MAP_PRIVATE or MAP_SHARED

prot:

- PROT_EXEC, PROT_READ, PROT_WRITE, PROT_NONE

```
#include <unistd.h>
void *sbrk(int incr);
```

- Adds *incr* bytes to the break value and changes the allocated space accordingly.
- If *incr* is negative, the amount of allocated space is decreased by *incr* bytes.
- Returns the new value of *brk* ptr.

Types of Dynamic Memory Allocator

- **Explicit allocator** (used by C/C++): application allocates and frees space

- `malloc` and `free` in C
- `new` and `delete` in C++

Will
concentrate
on this

- **Implicit allocator** (used by Java,...): application allocates, but does not free space

- Garbage collection in Java, Python etc.

Challenges facing a memory allocator

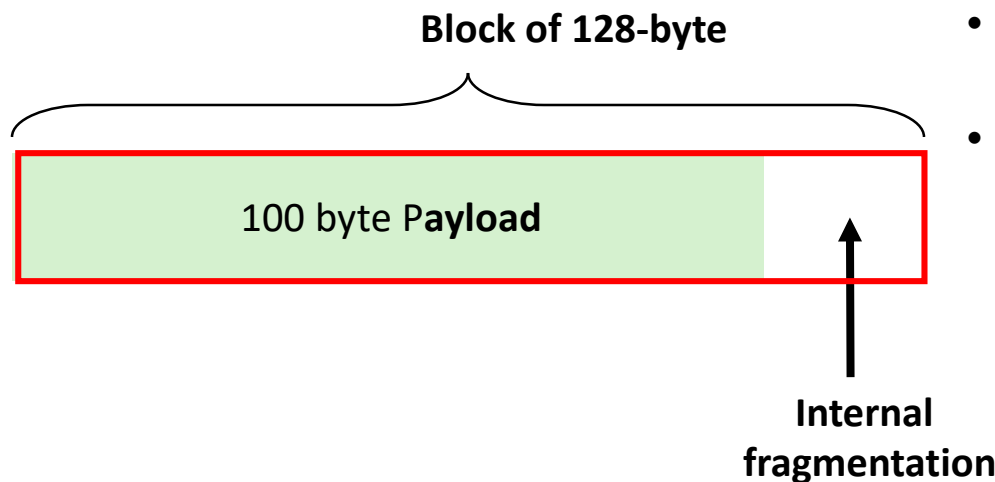
- Achieve good **memory utilization**
 - Apps issue arbitrary sequence of malloc/free requests of arbitrary sizes
 - Utilization = $\text{sum of malloc'd data} / \text{size of heap}$
- Achieve good **performance**
 - malloc/free calls should return quickly
 - Throughput = # ops/sec
- Constraints:
 - Can not touch/modify malloc'd memory
 - Can't move the allocated blocks once they are malloc'd
 - *i.e.*, compaction is not allowed

Fragmentation

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

Internal Fragmentation

- Malloc allocates data from **blocks of certain sizes**.
- **Internal fragmentation** occurs if **payload** is smaller than block size

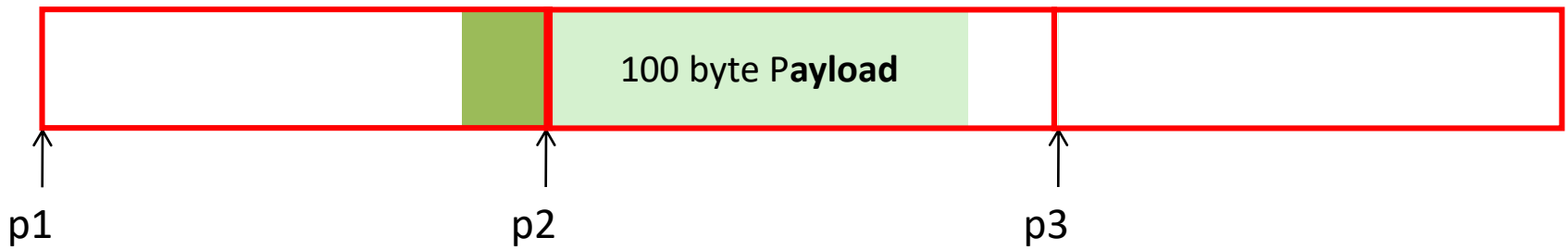


- Block size decided by allocator's designer.
- Payload is the number of bytes you want when you call malloc(),... .

- May be caused by
 - Limited choices of block sizes
 - Padding for alignment purposes
 - Other space overheads..

External Fragmentation

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



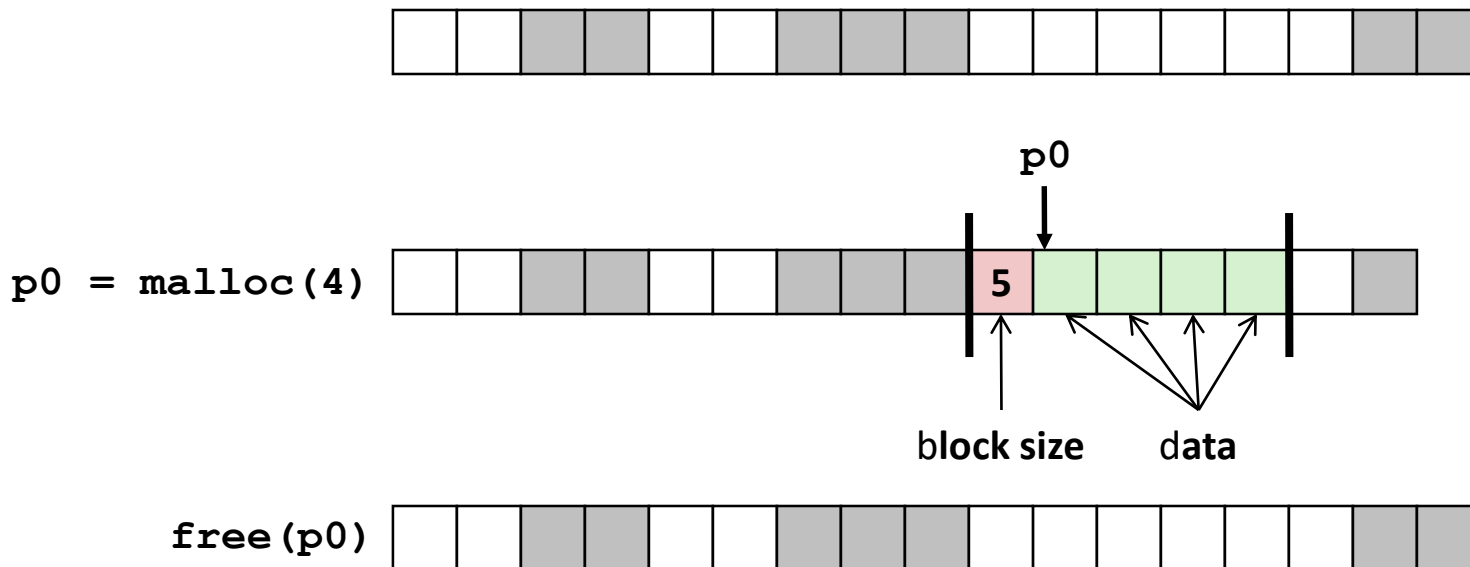
```
p1= malloc (100);  
p2 = malloc(100);  
p3 = malloc(100);  
  
free(p1);  
free(p3);  
malloc(200)?
```

Malloc design choices

- How do we know how much memory to free given just a pointer?
- How do we keep track of the free blocks?
- What do we do with the extra space when allocating a space that is smaller than the free block it is placed in?
- How do we pick a block to use for allocation -- many might fit?
- How do we reinsert freed block?

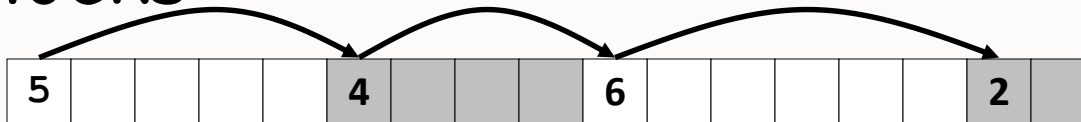
Knowing How Much to Free

- Standard method
 - Keep the length of a block in the **header field** preceding the block.
 - Requires header overhead for every allocated block

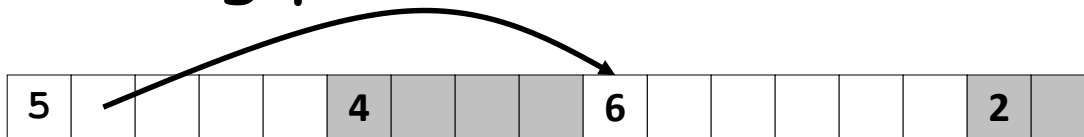


Keeping Track of Free Blocks

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



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

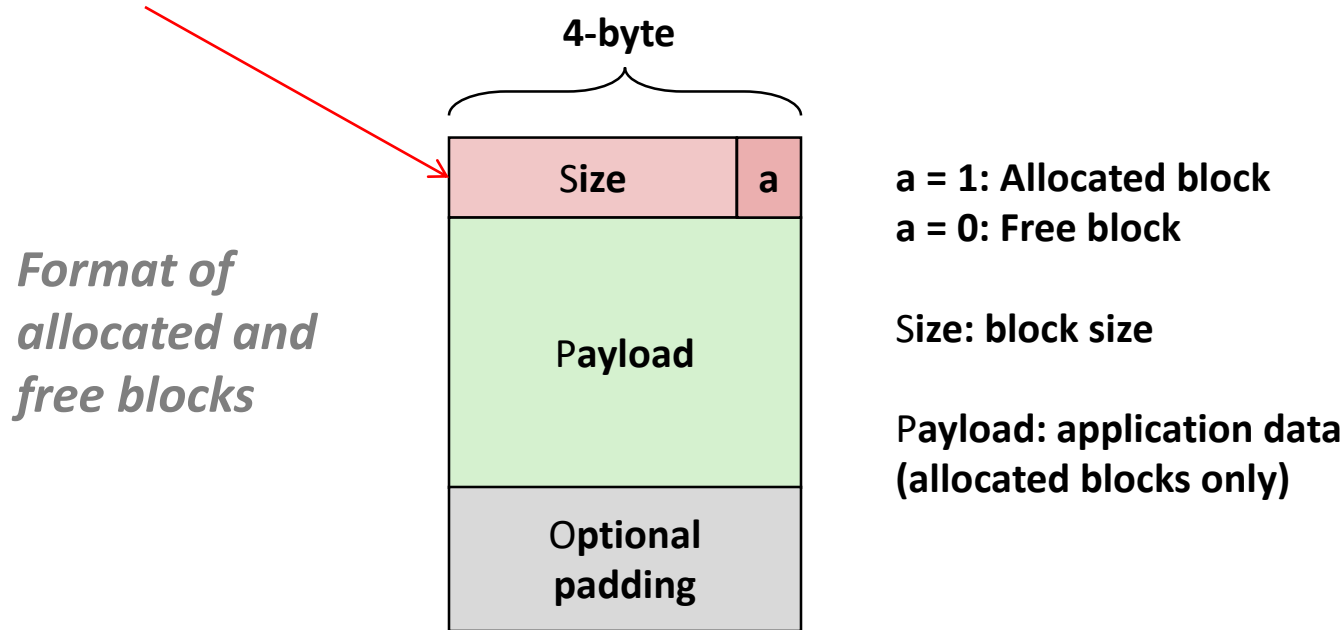


- Method 3: *Segregated free list*
 - Different free lists for different size classes

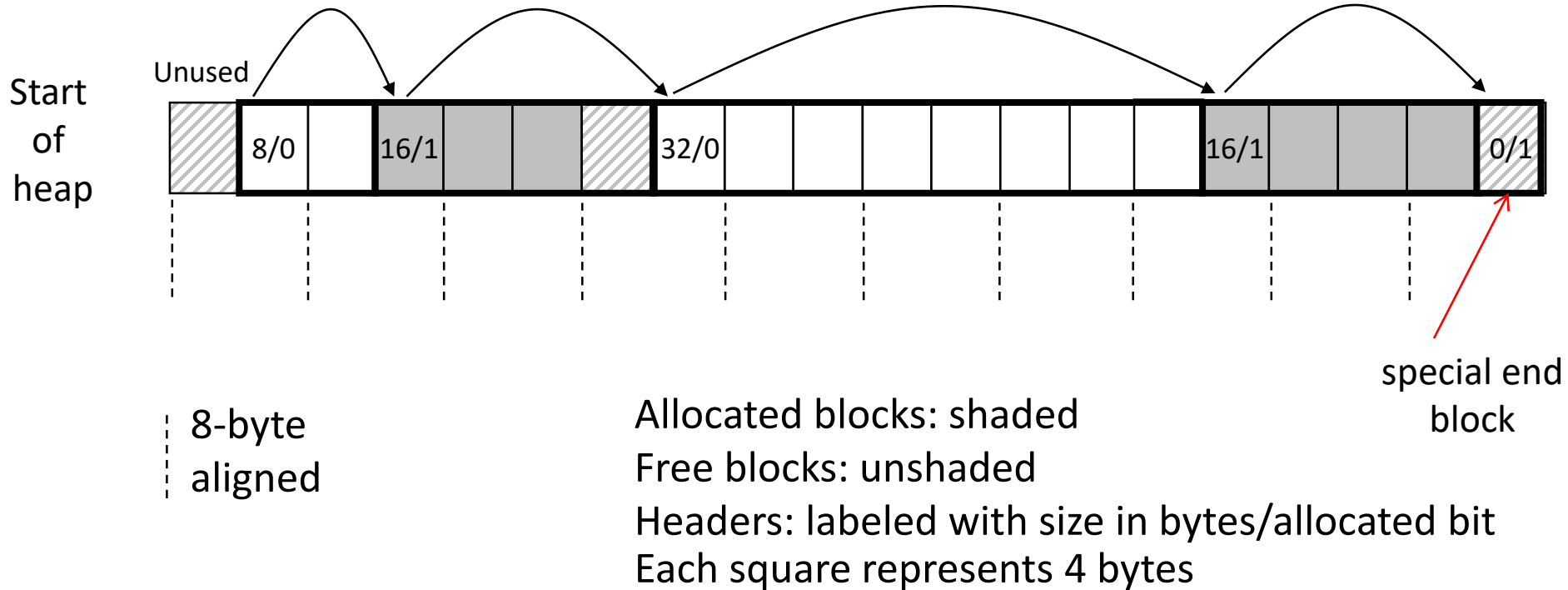
Method 1: Implicit List

- Malloc grows a contiguous region of heap by calling sbrk()
- Heap is divided into variable-sized blocks
- For each block, we need both size and allocation status

header + payload + padding



Detailed Implicit Free List Example

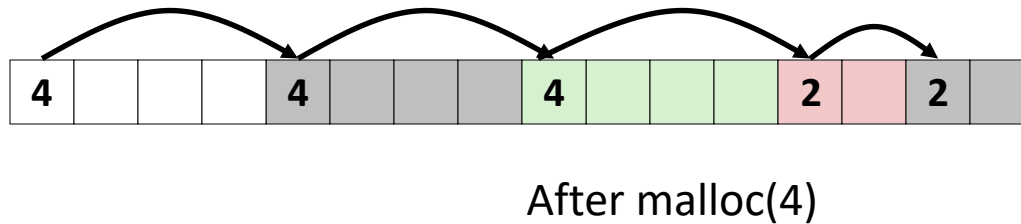
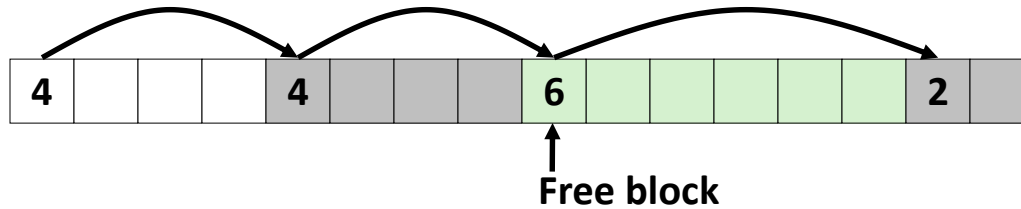


Implicit List: Finding a Free Block

- *First fit:*
 - Search from beginning, choose *first* free block that fits:
- *Next fit:*
 - Like first fit, except search starts where previous search finished
- *Best fit:*
 - Search the list, choose the *best* free block: fits, with fewest bytes left over (i.e. pick the smallest block that is big enough for the payload)
 - Keeps fragments small
 - Will typically run slower than first fit

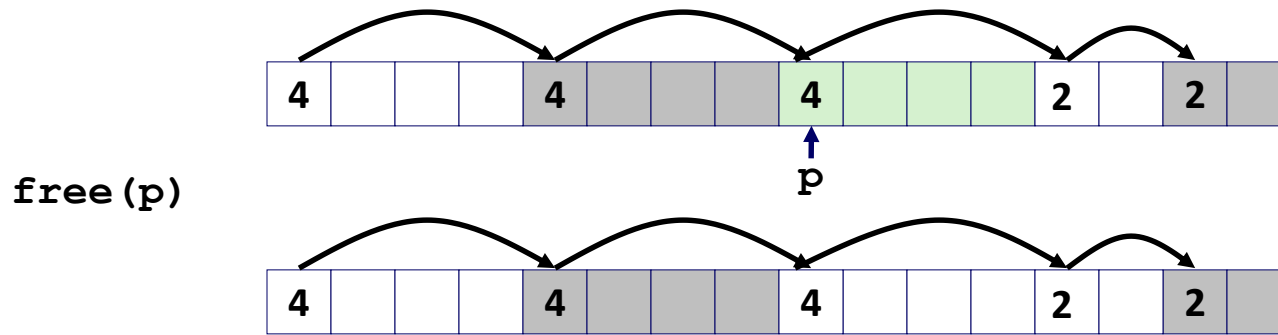
Implicit List: Allocating in Free Block

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



Implicit List: Freeing a Block

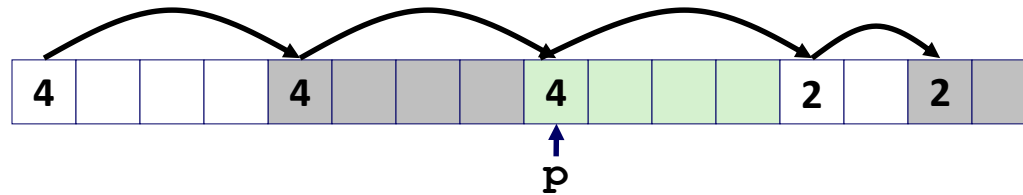
- Simplest implementation:
 - Need only clear the “allocated” flag
 - But can lead to “false fragmentation”



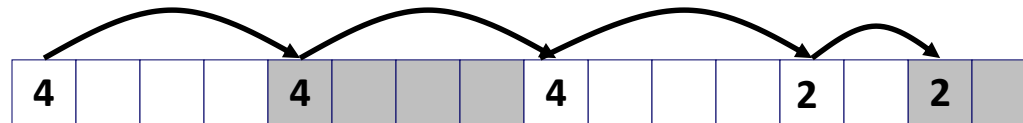
malloc (5) **Oops!**

Implicit List: Coalescing

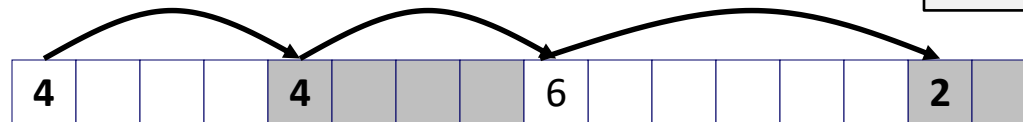
- Join (*coalesce*) with next/previous blocks, if they are free
 - Coalescing with next block



free (p)



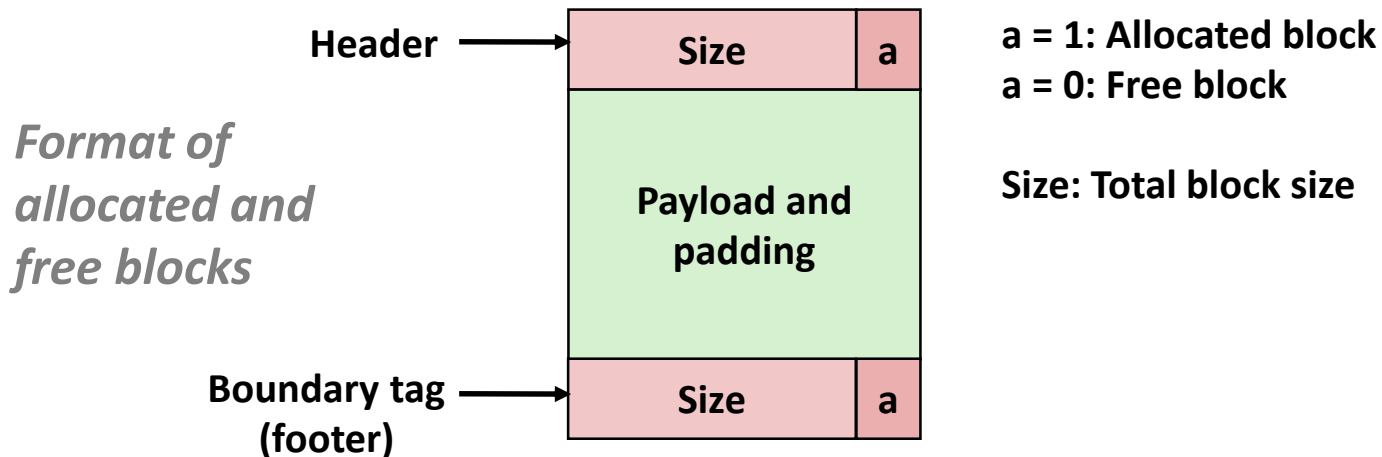
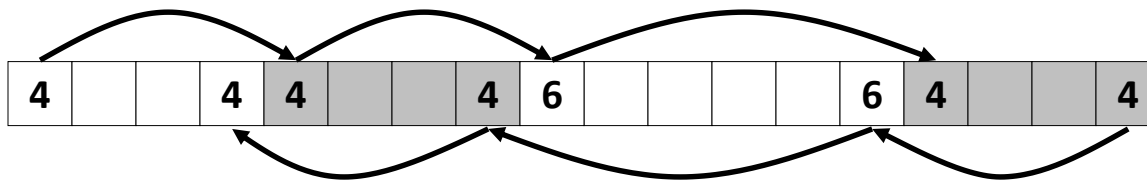
Check if next block is free



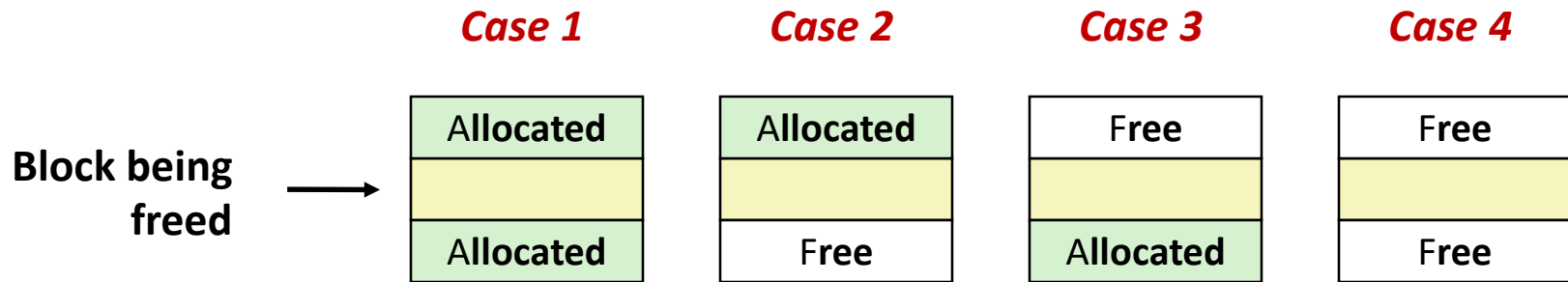
How to coalesce with a previous block?

Implicit List: Bidirectional Coalescing

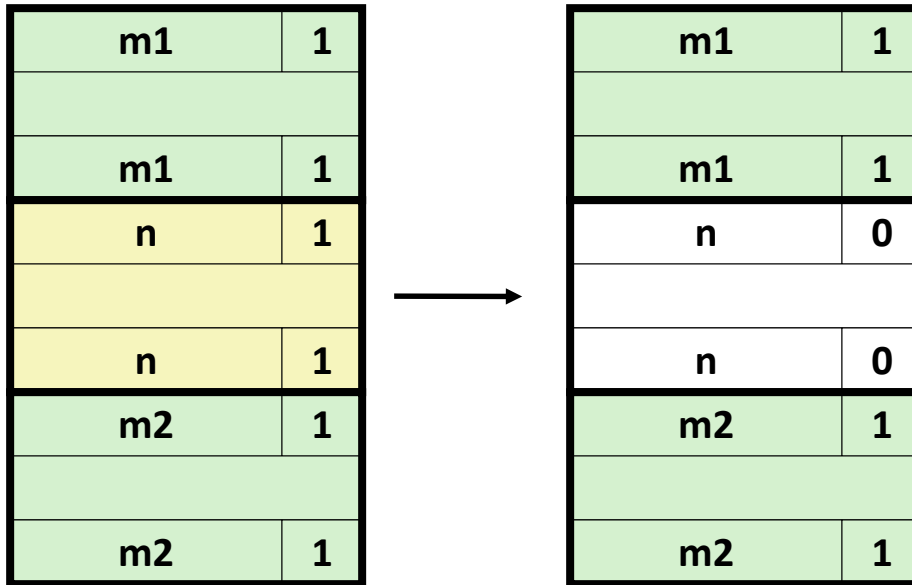
- **Boundary tags** [Knuth73]
 - Replicate size/allocated header at “bottom” (end) of blocks
 - Allows us to traverse the “list” backwards, but requires extra space



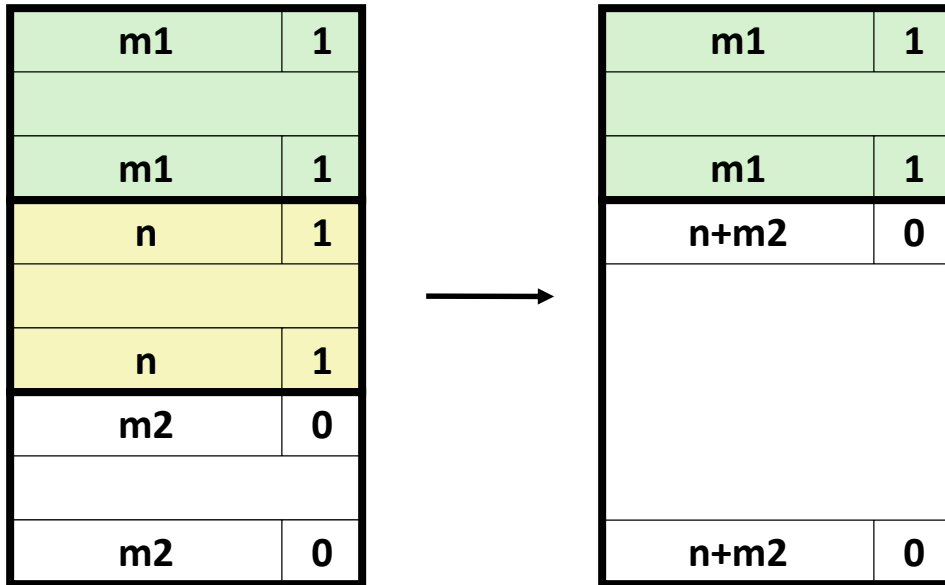
Coalescing



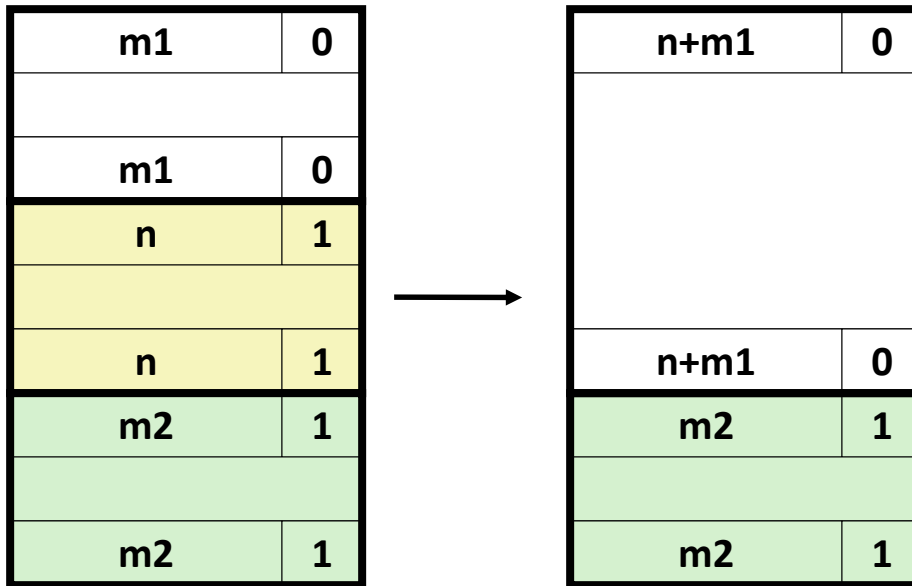
Coalescing (Case 1)



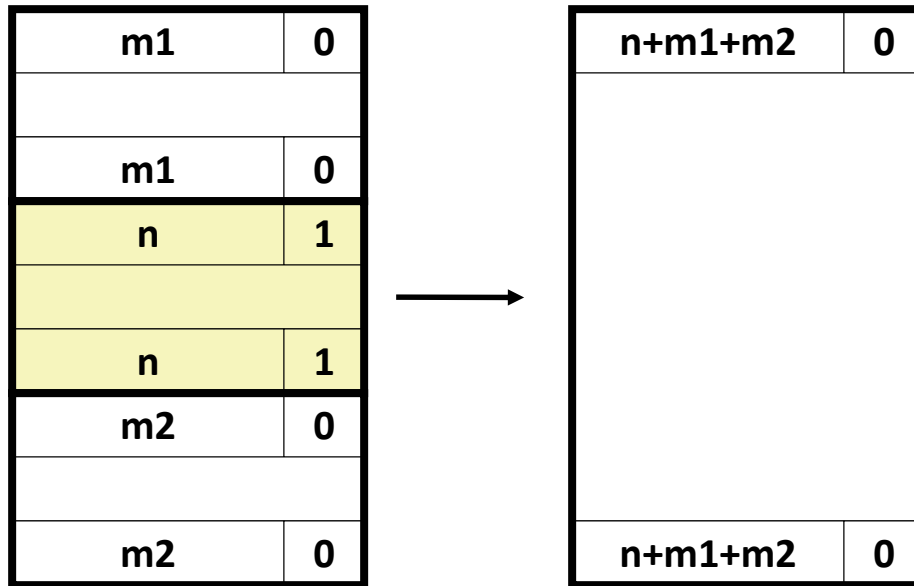
Coalescing (Case 2)



Coalescing (Case 3)



Coalescing (Case 4)



When to coalesce?

- **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

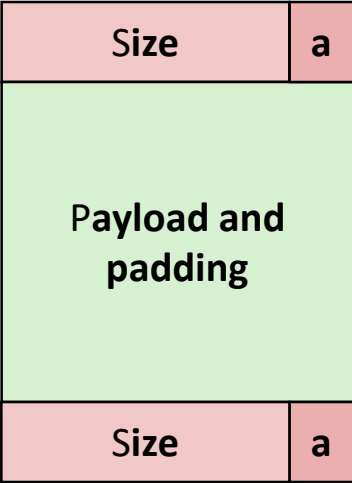
- Implementation: very simple
- Allocate cost:
 - linear time worst case
- Free cost:
 - constant time worst case, even with coalescing
- Memory usage:
 - will depend on first-fit, next-fit or best-fit
- Not used in practice for `malloc/free` because of linear-time allocation
 - used in many special purpose applications

Explicit Free list

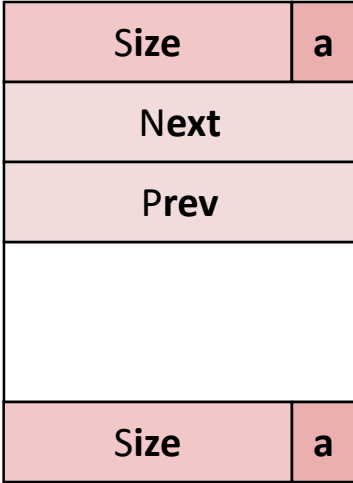
- Maintain list(s) of free blocks instead of all blocks
- Need to store forward/back pointers in each free block, not just sizes
 - because free blocks may not be contiguous in heap.

Explicit Free Lists

Allocated block



Free block



Store next/prev pointers in "payload" of free block.

Does this increase space overhead?

Freeing With Explicit Free Lists

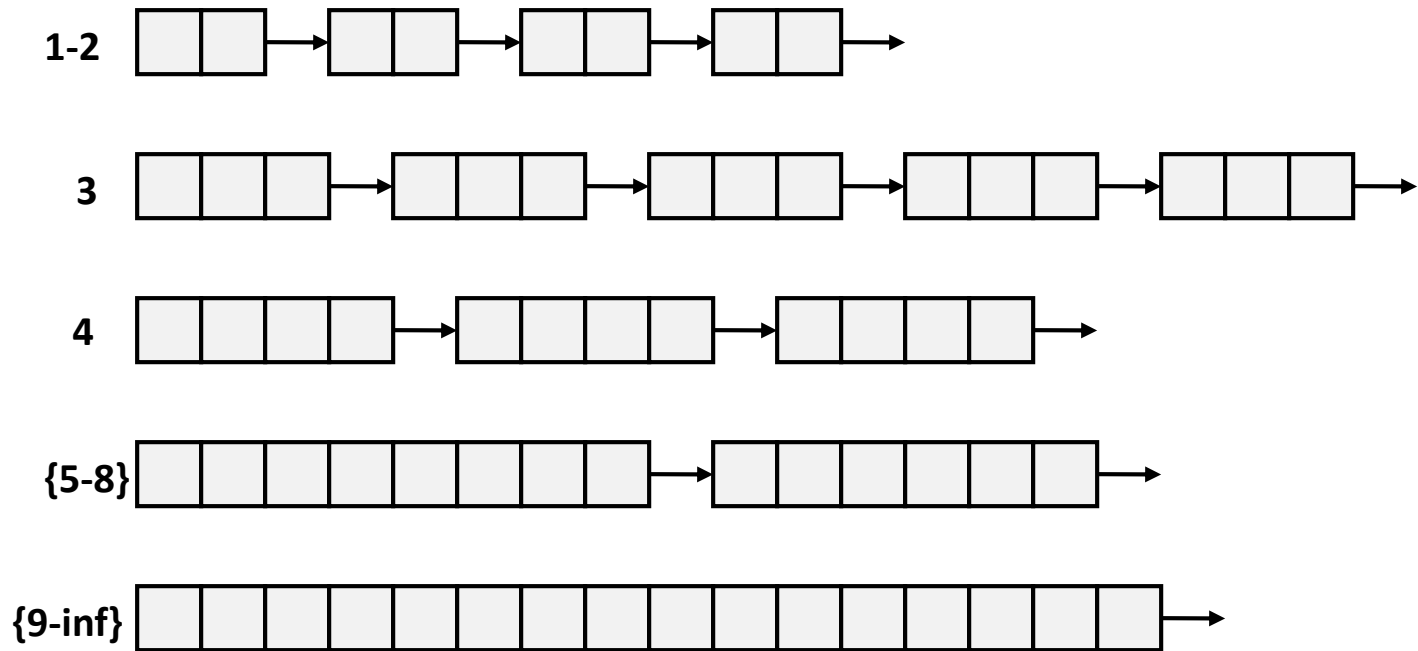
- Where in the free list to put a newly freed block?
 - Insert freed block at the beginning of the free list (LIFO)
 - **Pro:** simple and constant time
 - Insert freed blocks to maintain address order:
 - $addr(prev) < addr(curr) < addr(next)$
 - **Pro:** may lead to less fragmentation than LIFO

Explicit List

- ✓ Allocation is linear time in # of *free* blocks instead of *all* blocks
- Still expensive to find a free block that fits
 - How about keeping multiple linked lists of different size classes?

Segregated List (Seglist) Allocators

- Multiple free lists each linking free blocks of similar sizes



Seglist Allocator

- Given an array of free lists, each one for some size class
- To allocate a block of size n :
 - Search in appropriate free list containing size n
 - Split found block and place fragment on appropriate list
 - try next larger class if no blocks found
- If no block is found:
 - Request additional heap memory from OS
 - Allocate block of n bytes from this new memory
 - Place remainder as a single free block in largest size class.

Seglist Allocator (cont.)

- To free a block:
 - Coalesce and place on appropriate list
- Advantages of seglist allocators
 - Fast allocation
 - Better memory utilization
 - First-fit search of segregated free list approximates a best-fit search of entire heap

A Word About Garbage Collection

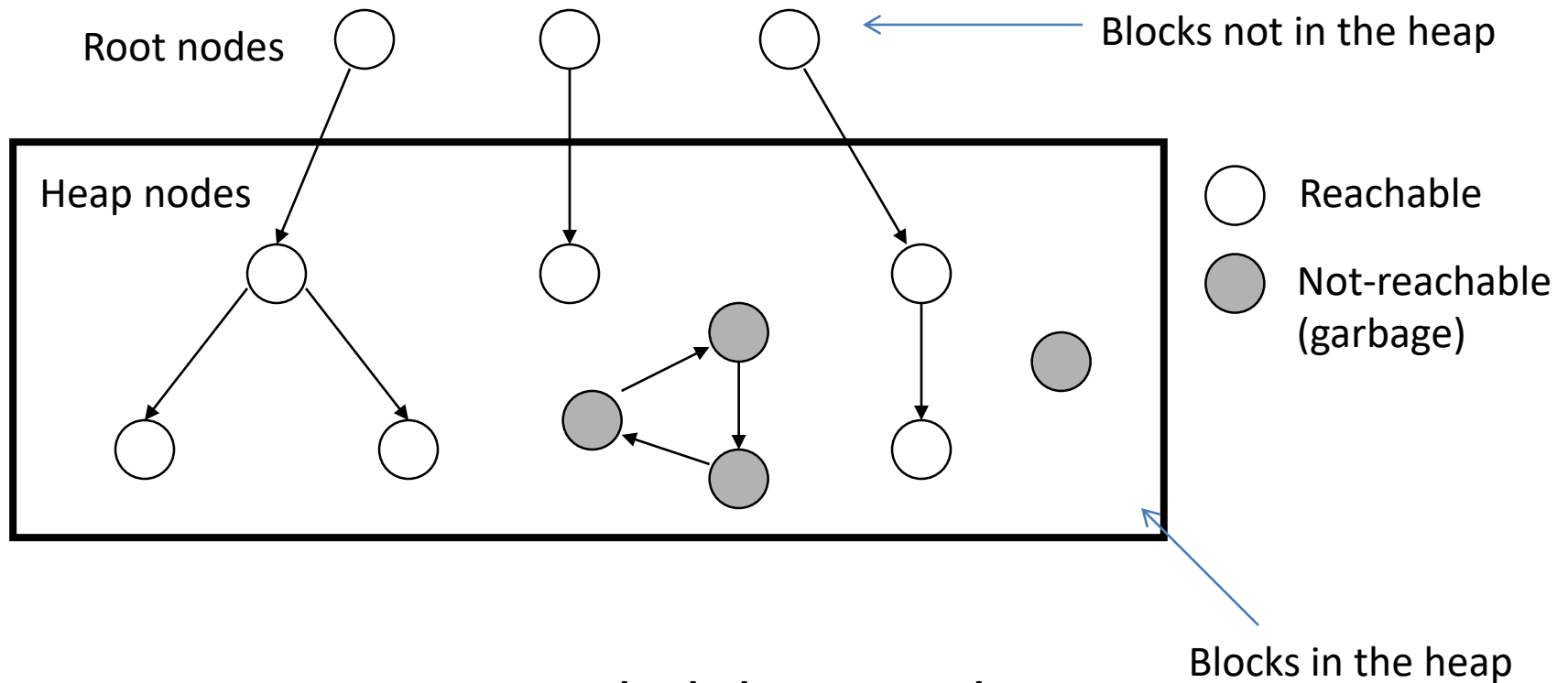
- In C, it is the programmer's responsibility to free any memory allocated by malloc/calloc/... .
- A garbage collection is a dynamic storage allocator that **automatically** frees allocated blocks that are no longer needed by the program.
- Allocated blocks that are no longer needed are called **garbage**.

A Word About Garbage Collection

- In systems that support garbage collection (e.g. Java, Perl, Mathematica, ...)
 - Applications explicitly allocate heap blocks
 - But never free them!
- The garbage collector **periodically** identifies garbage and make appropriate calls to free.

How does the garbage collector recognizes blocks that are no longer needed?

A Word About Garbage Collection



Reachability Graph

Conclusions

- Dynamic memory allocator manages the heap.
- Dynamic memory allocator is part of the user-space
- The allocator has two main goals:
 - reaching higher throughput (operations per second)
 - better memory utilization (i.e. reduces fragmentation).

Conclusions (cont'd)

- Explicit allocator
 - Works in terms of blocks
 - Keeping track of free blocks
 - Implicit list
 - Explicit list
 - segregated list
 - blocks sorted by size
- Implicit allocator