

# CS356 Unit 10

## Memory Allocation & Heap Management

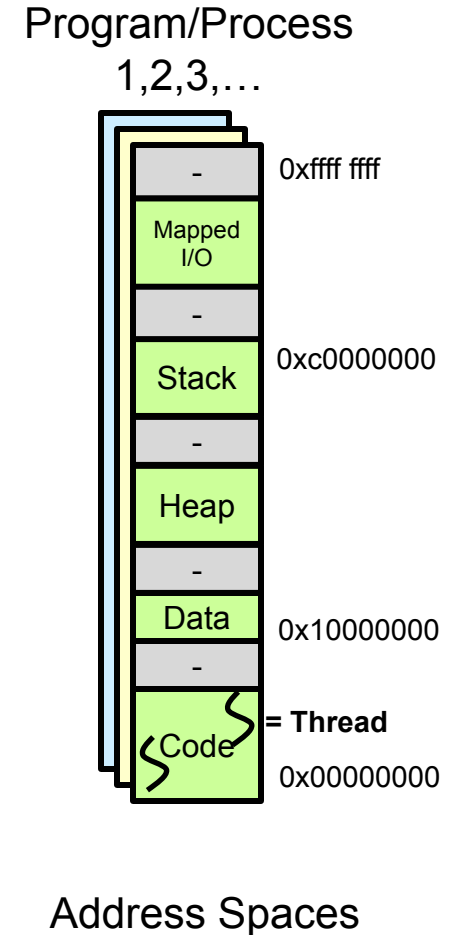
# **BASIC OS CONCEPTS & TERMINOLOGY**

# User vs. Kernel Mode

- **Kernel mode** is a special processor mode for executing trusted (OS) code
  - Certain features/privileges (such as I/O access) are only allowed to code running in kernel mode
  - OS and other system software should run in kernel mode
- **User mode** is where user applications are designed to run to limit what they can do on their own
  - Provides protection by forcing them to use the OS for many services
- User/kernel mode determined by bits in some processor control register
  - x86 Architecture uses lower 2-bits in the CS segment register (referred to as the Current Privilege Level bits [CPL])
  - 0=Most privileged (kernel mode) and 3=Least privileged (user mode)
    - Levels 1 and 2 may also be used but are not used by Linux

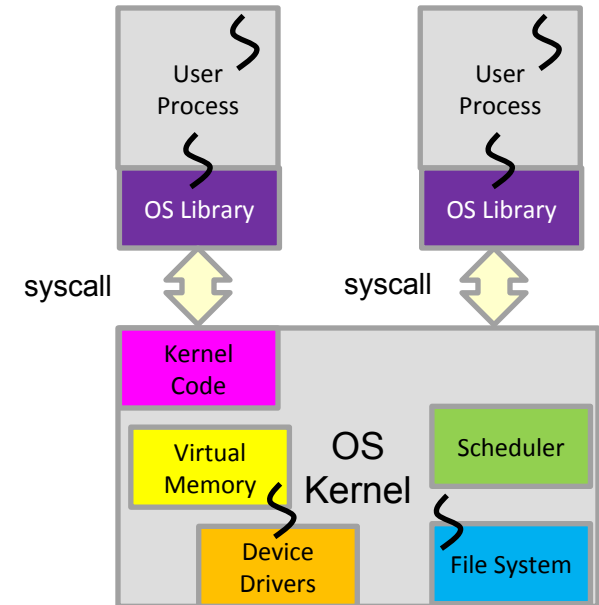
# Processes

- **Process**
  - (def 1.) **Address Space + Threads**
    - 1 or more threads
  - (def 2.) : **Running instance of a program that has limited rights**
    - Memory is protected: Address translation (VM) ensures no access to any other processes' memory
    - I/O is protected: Processes execute in user-mode (not kernel mode) which generally means direct I/O access is disallowed instead requiring **system calls** into the kernel
- Kernel is not considered a "process"
  - Has access to all resources and much of its code is invoked under the execution of a user process thread (i.e. during a system call)
- User process invokes the OS (kernel code) via **system calls (see next slide)**



# System Calls and Mode Switches

- What causes user to kernel mode switch?
  - An exception: interrupt, error, or **system call**
- **System Calls:** Provide a controlled method for user mode applications to call kernel mode (OS) code
  - OS will define all possible system calls available to user apps.



```
enum
{
    /* Projects 2 and later. */
    SYS_HALT,      /* 0 = Halt the operating system. */
    SYS_EXIT,     /* 1 = Terminate this process. */
    SYS_EXEC,     /* 2 = Start another process. */
    SYS_WAIT,     /* 3 = Wait for a child process */
    SYS_CREATE,   /* 4 = Create a file. */
    SYS_REMOVE,   /* 5 = Delete a file. */
    SYS_OPEN,     /* 6 = Open a file. */
    SYS_FILESIZE, /* 7 = Obtain a file's size. */
    SYS_READ,     /* 8 = Read from a file. */
    SYS_WRITE,    /* 9 = Write to a file. */
    ...
};
```

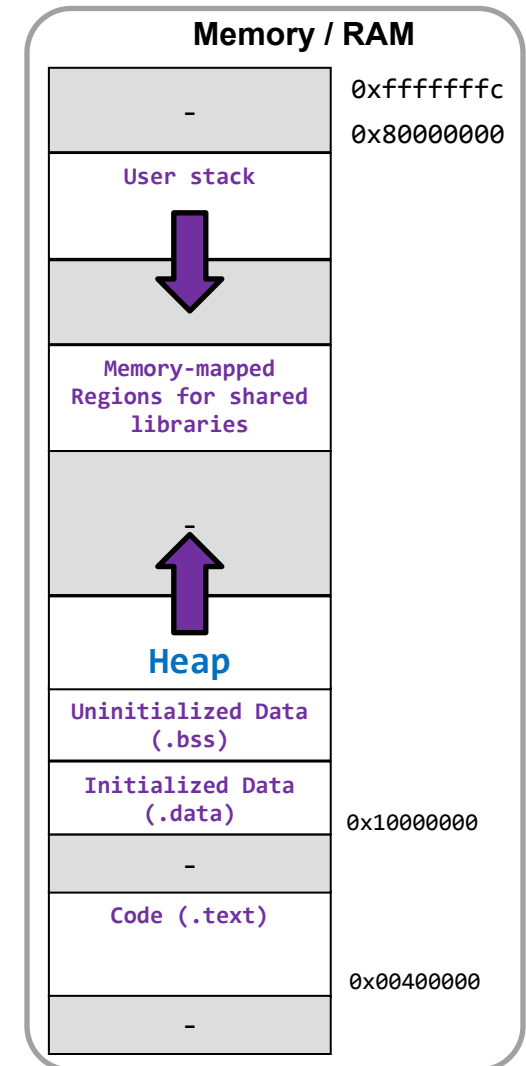
Syscalls from Pintos OS

# HEAP MANAGEMENT

# Overview

CS:APP 9.9.1

- Heap management is an important component that affects program performance
- Need to balance:
  - Speed & performance of allocation/deallocation
  - Memory utilization (reduce wasted areas)
  - Ease of usage by the programmer



# C Dynamic Memory Allocation

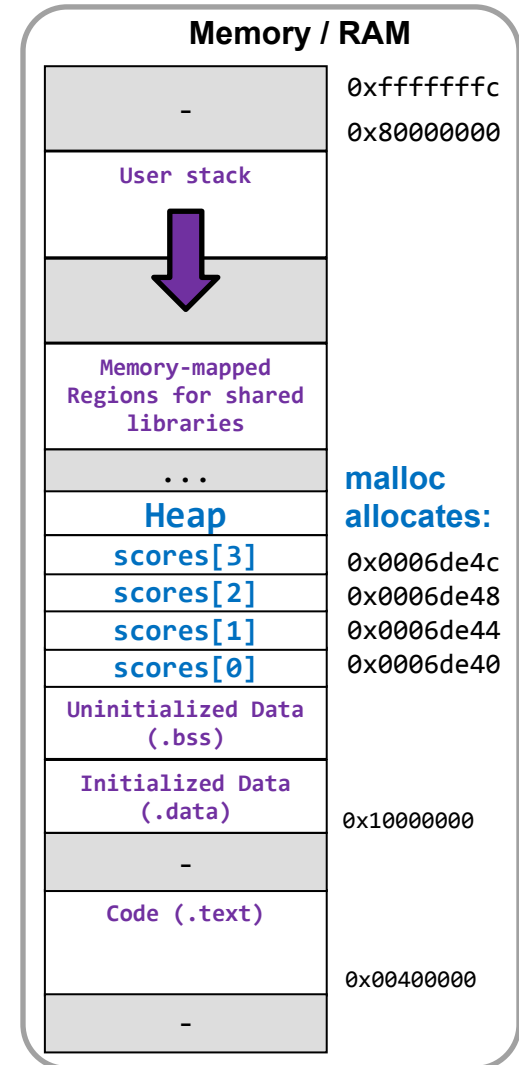
## Functions from `stdlib.h`

- `void *malloc(size_t size)`
  - Allocates **size** bytes and returns a pointer to the block
- `void *calloc(size_t nmemb, size_t size)`
  - Allocates **nmemb\*size** bytes, **sets the memory to 0**, returns a pointer to the block
- `void free(void *ptr)` function
  - Frees the block at address ptr (returned by malloc/calloc), returns it to the system for re-use by subsequent malloc calls

```
int main() {
    int num;
    printf("How many students?\n");
    scanf("%d", &num);

    // cast "(int*)" from void* not necessary in C
    int *scores = malloc(num * sizeof(int));

    // can now access scores[0] .. scores[num-1];
    free(scores); // deallocate
    return 0;
}
```

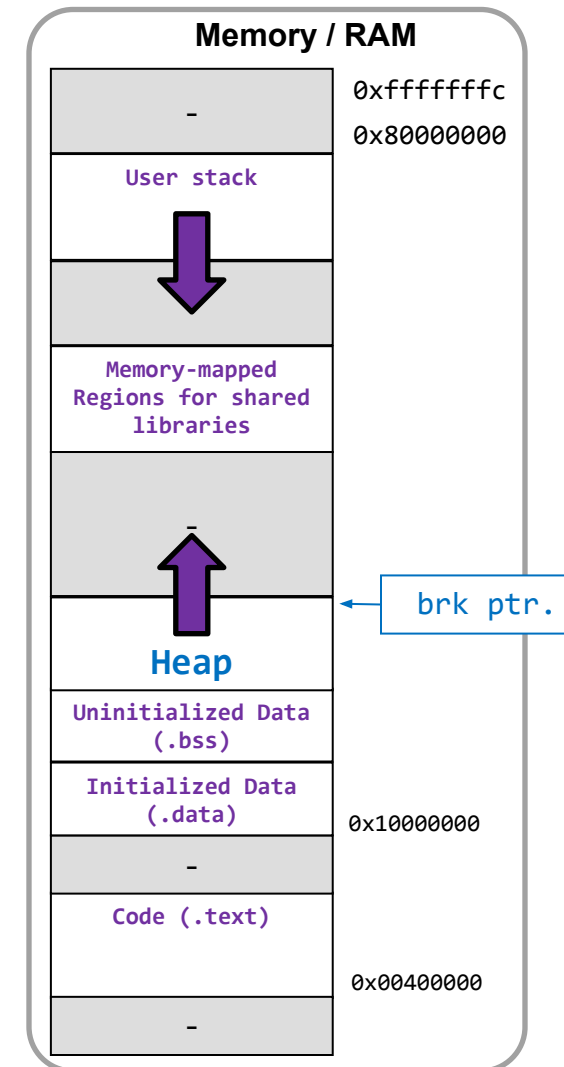




# OS & the Heap

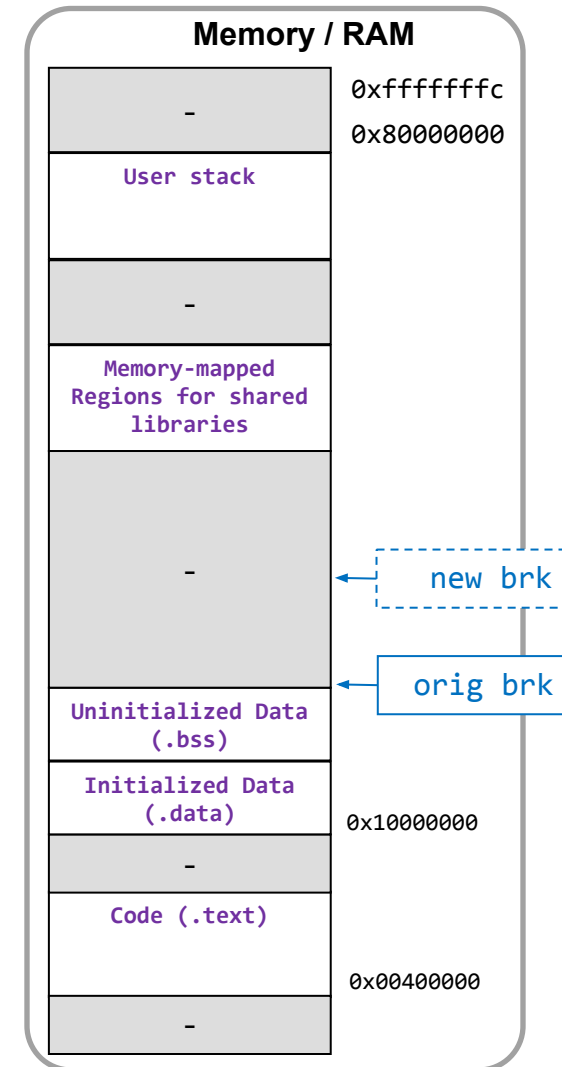
- The OS kernel maintains the **brk** pointer
  - Virtual address of the top of the heap
  - Per process (threads share the heap)
- **brk** pointer is updated via a system call (see Linux example below)
  - `#include <unistd.h>`
  - `void *sbrk(intptr_t increment);`
    - Increments the **brk** pointer (up or down) and returns the old **brk** pointer on success
  - Newly allocated memory is zero-initialized
- `malloc/free` allow the reuse of blocks allocated on the heap with `sbrk`

`intptr_t` is a signed integer type that will match the size of pointers (32- or 64-bits)



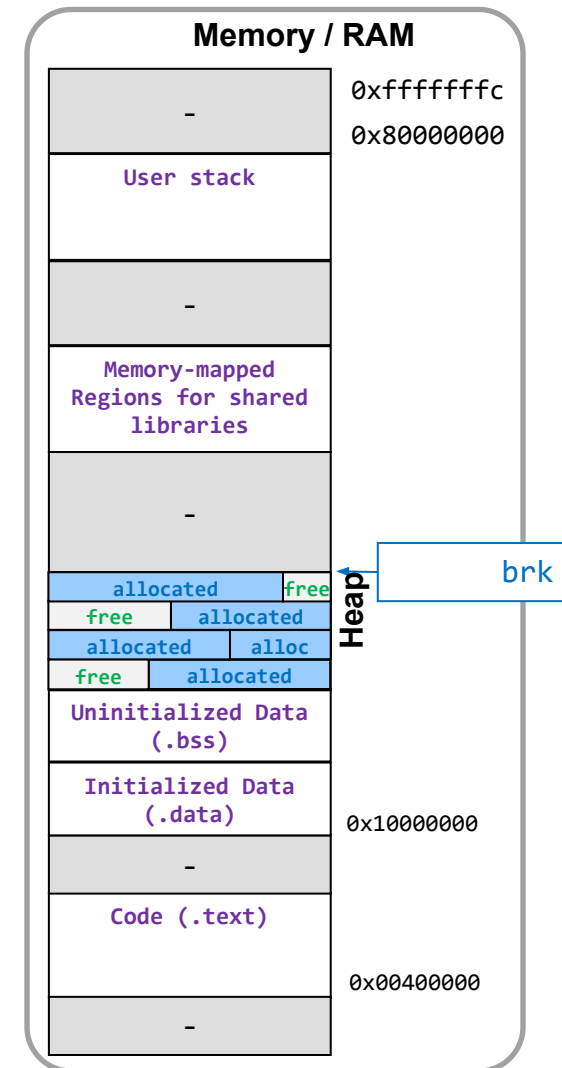
# A First Look at malloc (1)

- The C-library implementation will provide an implementation to manage the heap
- At startup, the C-Library will allocate an initialize size of the heap via sbrk
  - `void *heap_init;`
  - `heap_init = sbrk(1 << 20); // 1 MB`



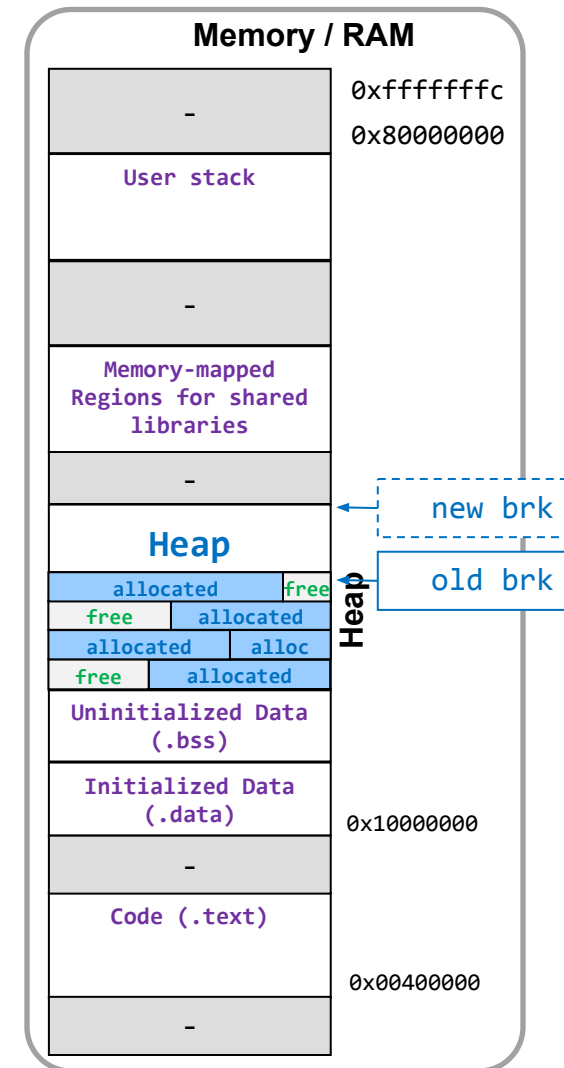
# A First Look at malloc (2)

- The C-library implementation will provide an implementation to manage the heap
- At startup, the C-Library will allocate an initialize size of the heap via sbrk
- Subsequent requests by malloc (or new) will give out portions of the heap
- Calls to free or delete will reclaim those memory areas



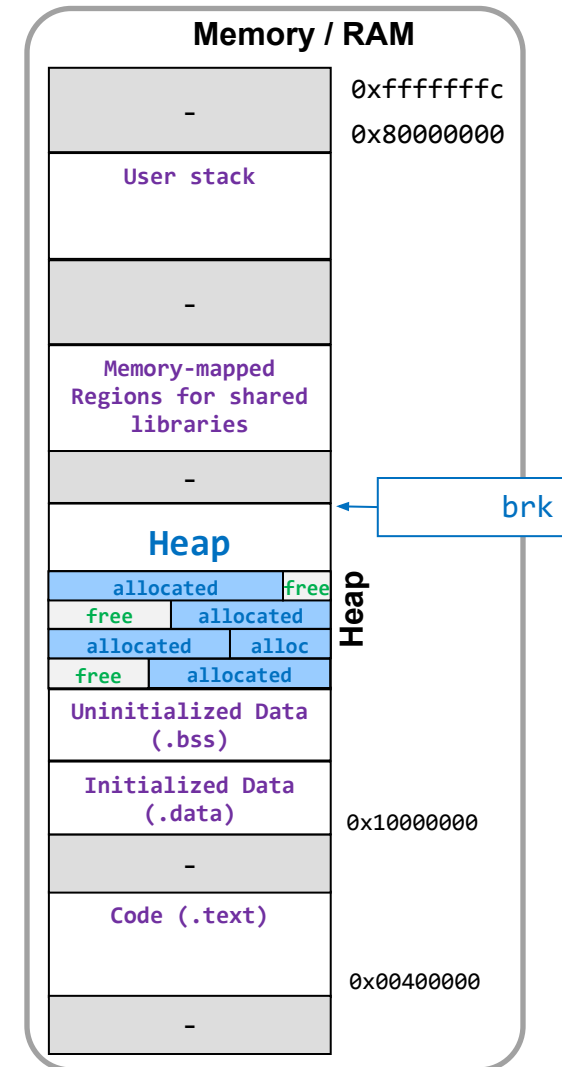
# A First Look at malloc (3)

- The C-library implementation will provide an implementation to manage the heap
- At startup, the C-Library will allocate an initialize size of the heap via sbrk
- Subsequent requests by malloc (or new) will give out portions of the heap
- Calls to free or delete will reclaim those memory areas
- If there is not enough **contiguous** free heap memory to satisfy a call to **malloc/new** then the library will use **sbrk** to increase the size of the heap
  - When no memory exists, an exception or NULL pointer will be returned and the program may fail



# Allocators and Garbage Collection

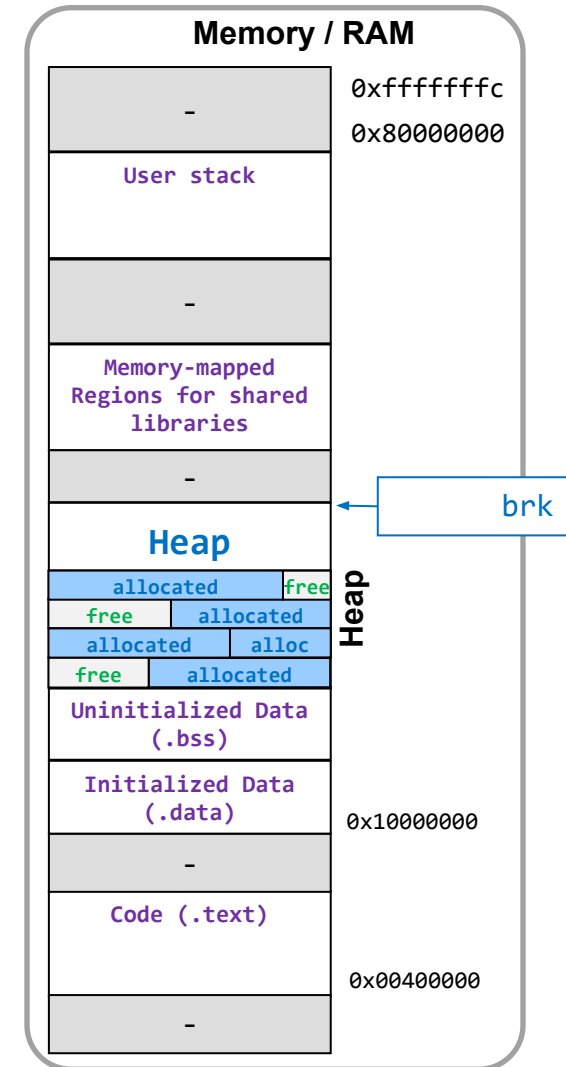
- An allocator will manage the free space of the heap
- Types:
  - **Explicit** Allocator: Requires the programmer to explicitly free memory when it is no longer used
    - Exemplified by malloc/new in C/C++
  - **Implicit** Allocator: Requires the allocator to determine when memory can be reclaimed and freed (i.e., known as garbage collection)
    - Used by Java, Python, etc.



# Allocator Requirements

CS:APP 9.9.3

- **Arbitrary request sequences**
  - No correlation to when allocation and free requests will be made
- **Immediate response**
  - Cannot delay a request to optimize allocation strategy
- **Use only the heap**
  - Any heap management data must exist on the heap or be scalar (single & not arrays) variables
- **Align blocks**
  - Allocated blocks must be aligned to any type of data
- **Previously allocated blocks may not be moved**
  - Once allocated the block cannot be altered by the allocator until it is freed

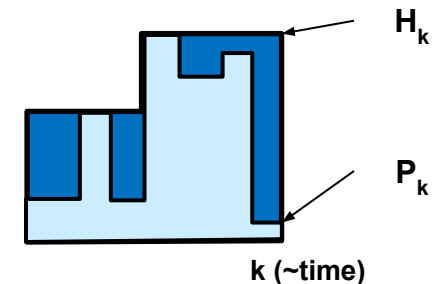


# Allocator Goals

- Maximize **throughput** (i.e., fast allocation / deallocation)
- Maximize **memory utilization**
  - Take as little memory as possible from the OS with `sbrk`
  - We need a formal definition of **peak memory utilization**
$$U(k) = \max \{ P(i) \text{ for } i = 1, \dots, k \} / H(k)$$
    - $P(i)$  = memory allocated after  $i$  malloc/free requests
    - $H(k)$  = total heap size (allocated/free) after  $k$  requests (Monotonically non-decreasing.)

These goals are at odds with each other!

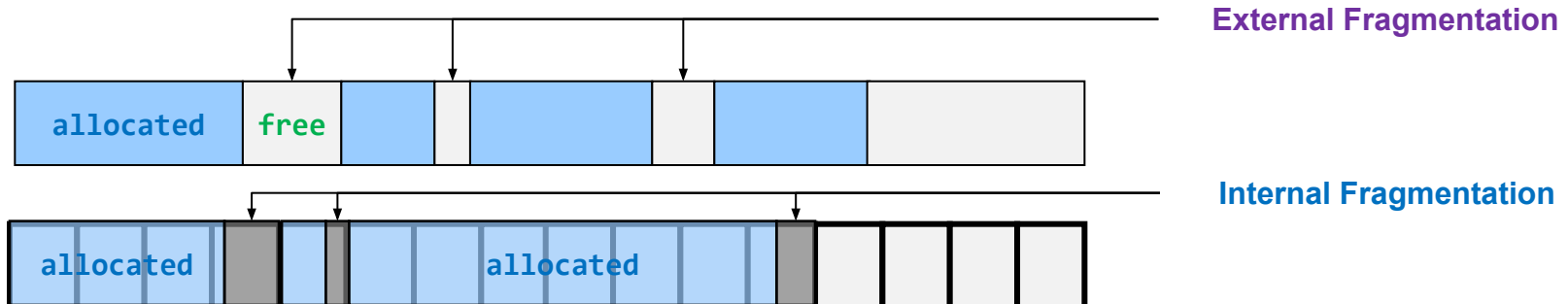
- E.g., no need to keep track of free blocks for reuse if we always allocate new blocks with `sbrk`.



# Fragmentation

CS:APP 9.9.4

- The enemy of high utilization is **fragmentation**
- Two kinds
  - **External**: Many small fragments of free space between allocated blocks
  - **Internal**: When payload of is smaller than the block size allocated
    - Often used when fixed size "chunks" are allocated
- Notice: There may be enough total free memory for a request but not **contiguous** free memory





# Implementation Issues

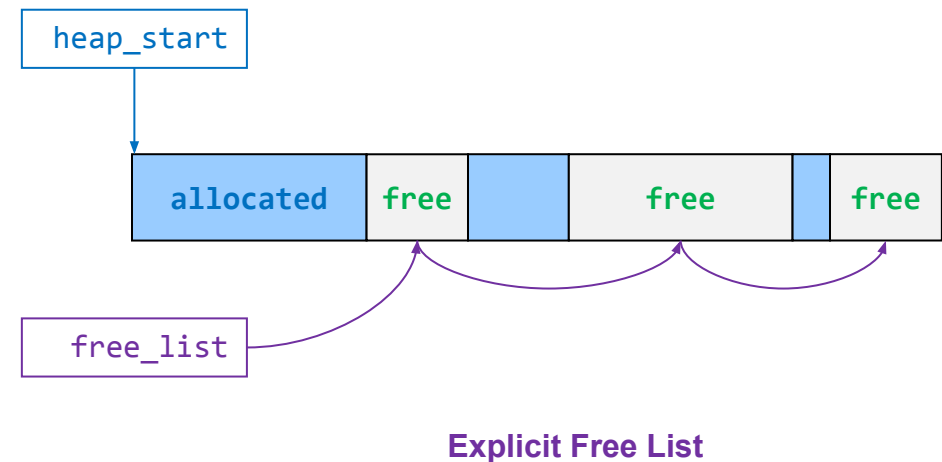
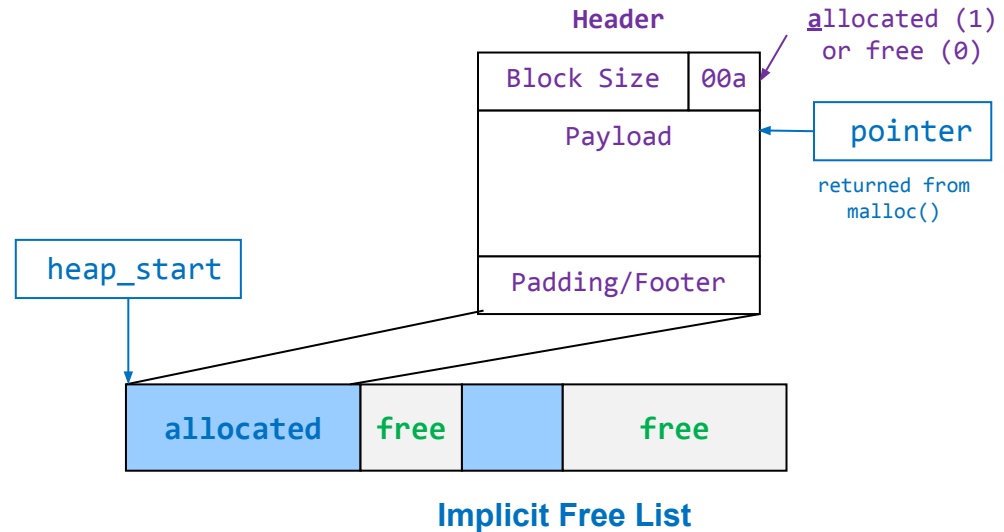
CS:APP 9.9.5

- Free block management
  - Tracking free areas on the heap
- Placement Algorithm
  - First-fit, next-fit, best-fit, buddy-system, ...
- Splitting/Coalescing
  - What overhead info do we keep when we split a block or need to coalesce (combine contiguous free) blocks



# Free Block Management

- Allocated blocks are the programmer's to manage and need not be tracked explicitly
- We must manage free lists to make new allocations
- Implicit free lists:
  - Scan through both allocated and free blocks to find an appropriate free block to allocate
- Explicit free lists:
  - Maintain explicit list of free blocks with each storing information to find the next (other) free block(s)

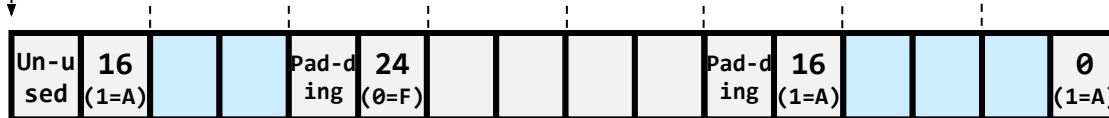


# Implicit Free List Implementation

CS:APP 9.9.6

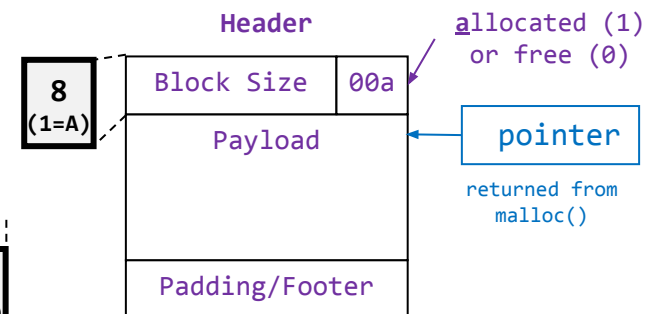
- A block must be aligned to largest type (double or pointer type) which is an 8-byte boundary for 64-bit systems
  - Book uses "word" to refer to an int size chunk (i.e. 4-bytes); thus "double word" refers to an 8-byte chunk
- Use headers so we can traverse the list to find free blocks

Heap start  
8-byte aligned



1  
word

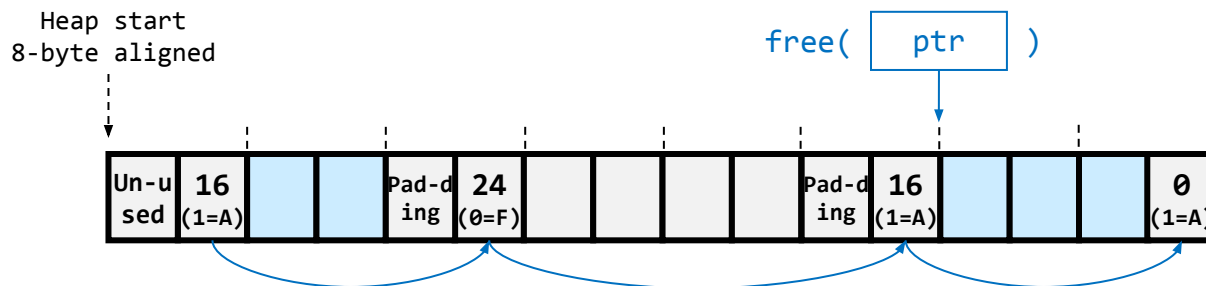
An Initial Implementation  
of an Implicit Free list



# Coalescing

CS:APP 9.9.10

- How would we coalesce the free blocks when the 12-byte chunk at the end is freed?
  - Nothing in the block being freed would help us find the previous block to see if we should coalesce the two?
  - Would need to scan from the beginning... $O(n)$
  - Could consider alternate organizations beyond just a linear list but there is still cost associated with finding the previous block
  - Instead, consider storing additional data to help find the previous block

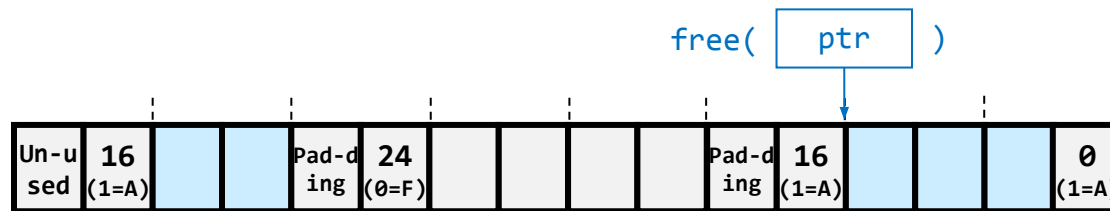


An Initial Implementation of an Implicit Free list

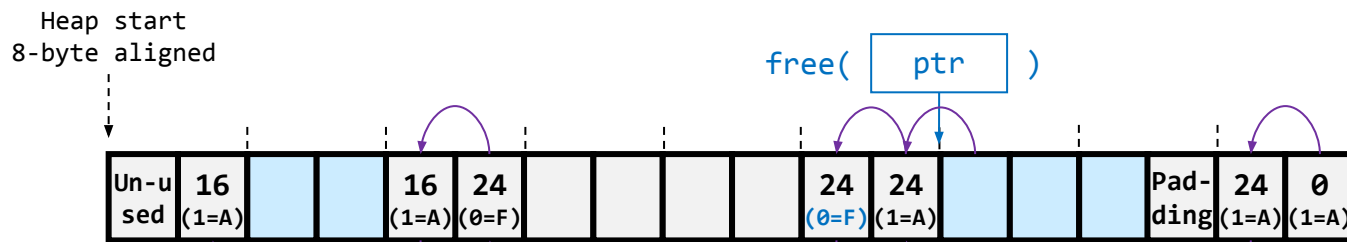


# Coalescing w/ Boundary Tags

- Store a footer (boundary tag) on each block that is really a copy of the header and indicates the size of the block
  - Each footer is always just before a header
  - When a block is freed, we can look at the footer before the header to determine *if* we should coalesce and *where* the previous header is
- Allows constant time  $O(1)$  coalescing (free) operation



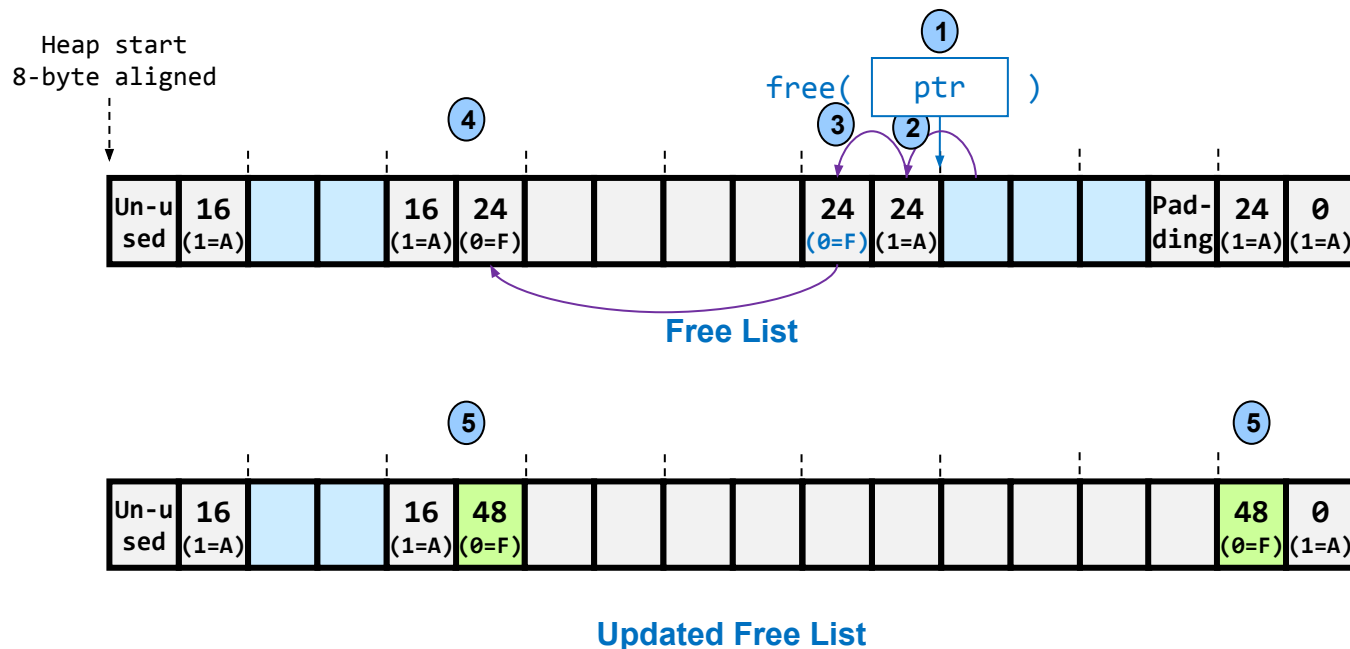
Original List



List with Boundary Tags

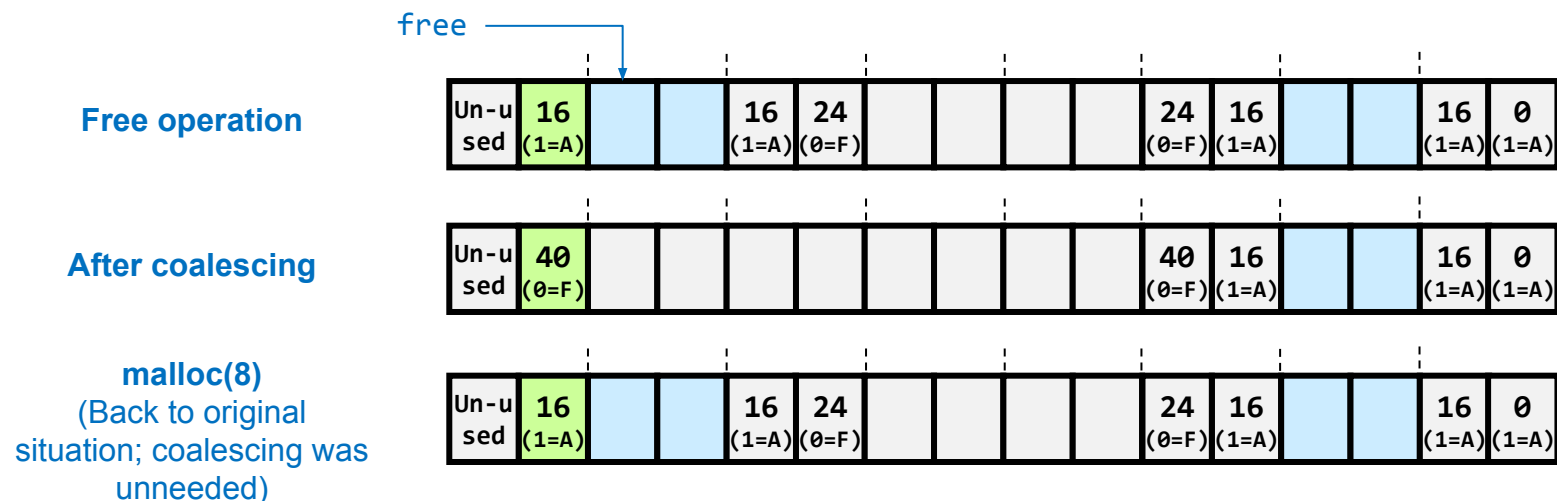
# Coalescing Example

- When we free the block given by ptr we would:
  - Start with the address provided by free
  - Walk one word back to find the header (and size) of this block
  - Walk another word back to find the footer (boundary tag) of the previous block from which we can determine if the block is free and needs to be coalesced
  - Walk to the header of the previous block ( $\&\text{footer\_block} - (\text{footer\_size} - 4)$ )
  - Update the size to be the sum of the two blocks and update the footer as well



# When To Coalesce

- We can coalesce:
  - Immediately when we free the block
    - Generally easier to implement
  - At some deferred time when we scan through and coalesce any contiguous free blocks
    - Likely when we can't find a large enough free block
    - May prevent wasted coalescing (thrashing)



# Coalescing Cases

- If we coalesce immediately then only 4 cases need be considered to ensure the list remains in an appropriate state





# Placement Algorithms

CS:APP 9.9.7

- **First Fit:** Scan from the start of the heap on each request and use the first free block that is large enough
- **Next Fit:** Scan starting from where the last allocation was made
- **Best Fit:** Find the smallest free block that is large enough for the request



First Fit



Next Fit



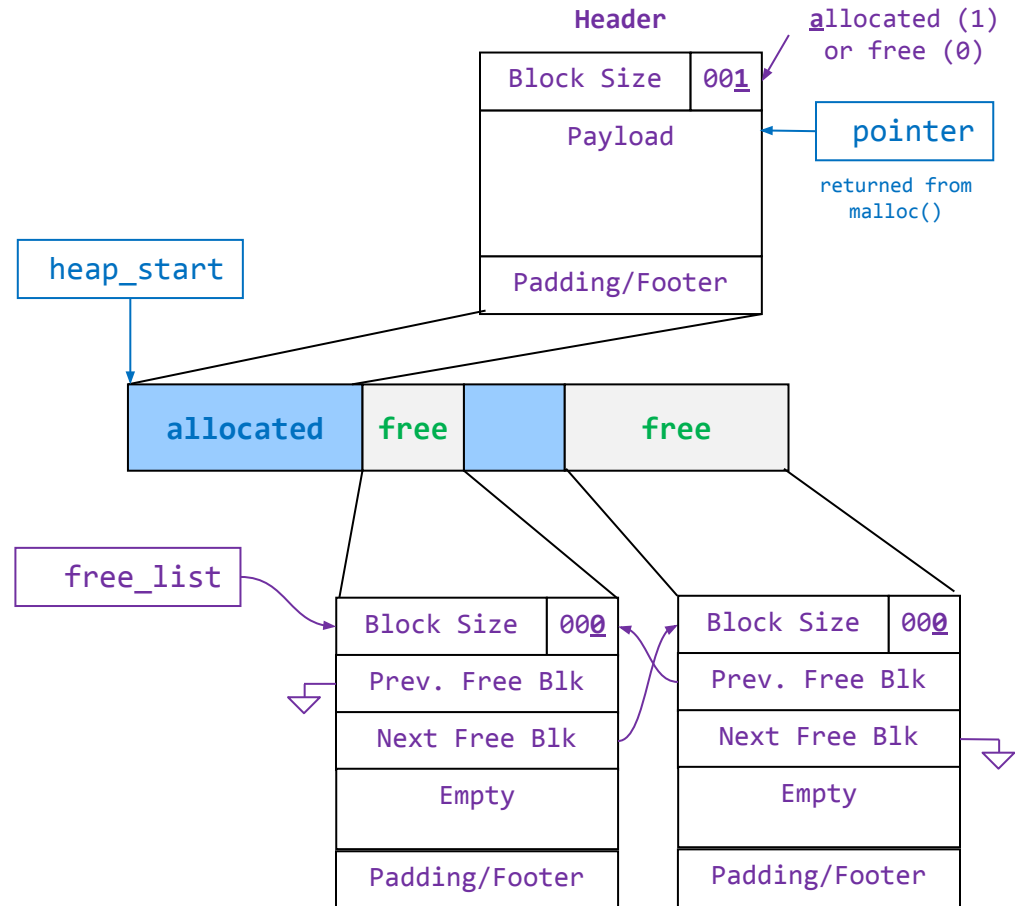
Best Fit

# EXPLICIT FREE LISTS

# Explicit Free Lists

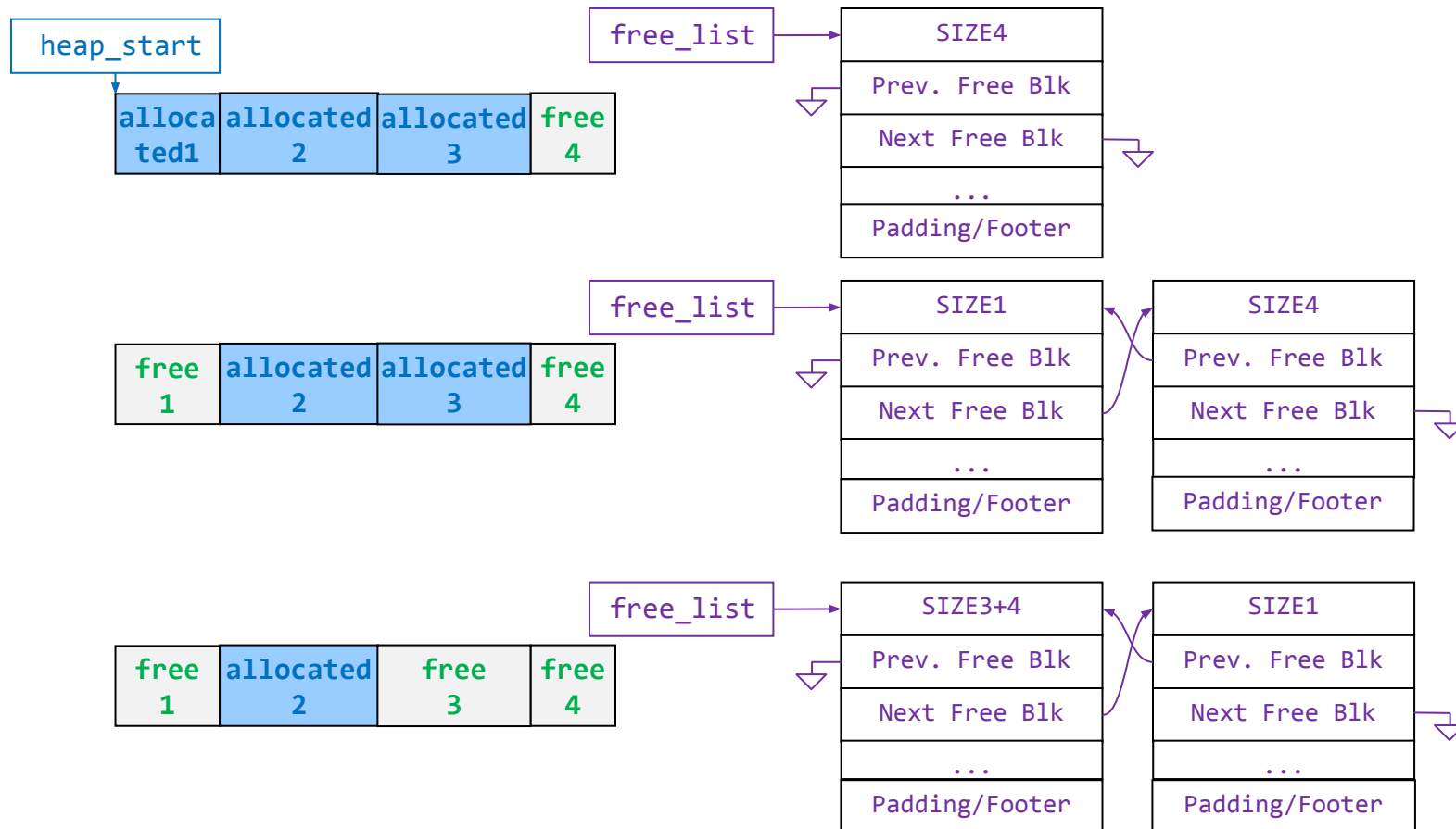
CS:APP 9.9.13

- When a block is free we can use some portion of the block to store explicit pointers to "other" free blocks
  - Could use a simple doubly-linked list or some other data structure
- Increases minimum size block (and potential *internal* fragmentation for small allocations)
- We can return the blocks in "any" order (more on the next slide)



# Explicit Free Lists

- Freed blocks can be placed at the front of the list (and coalescing can be immediate or deferred)



If coalescing is deferred, we can have 3 free blocks in the list in the order: 3, 1, 4

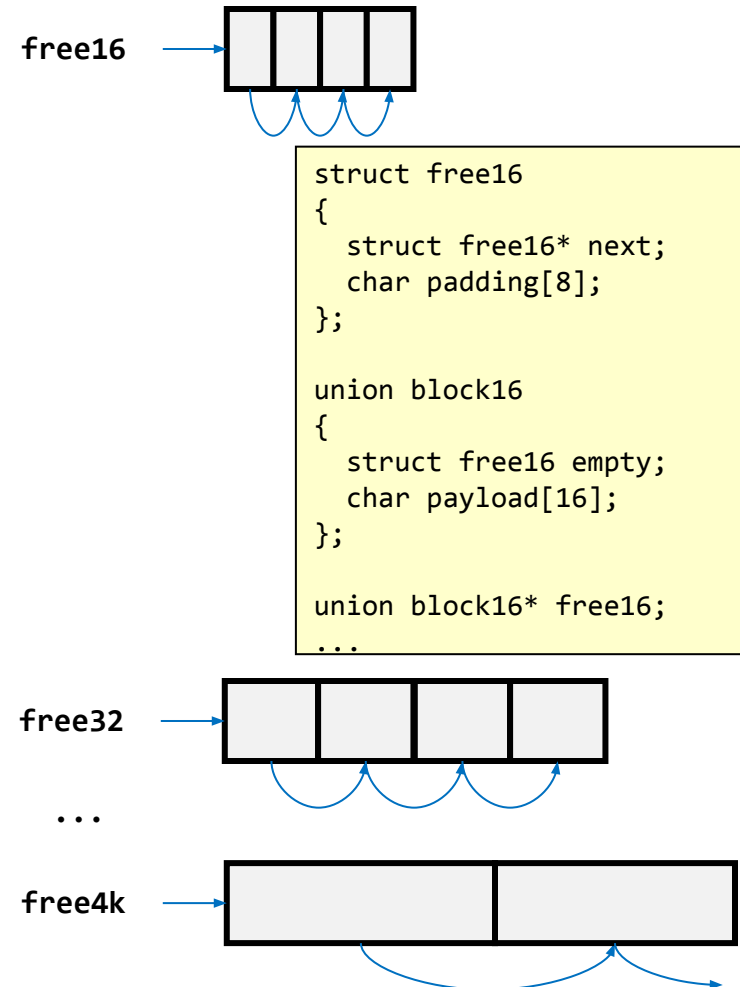
# Segregated Free Lists

CS:APP 9.9.14

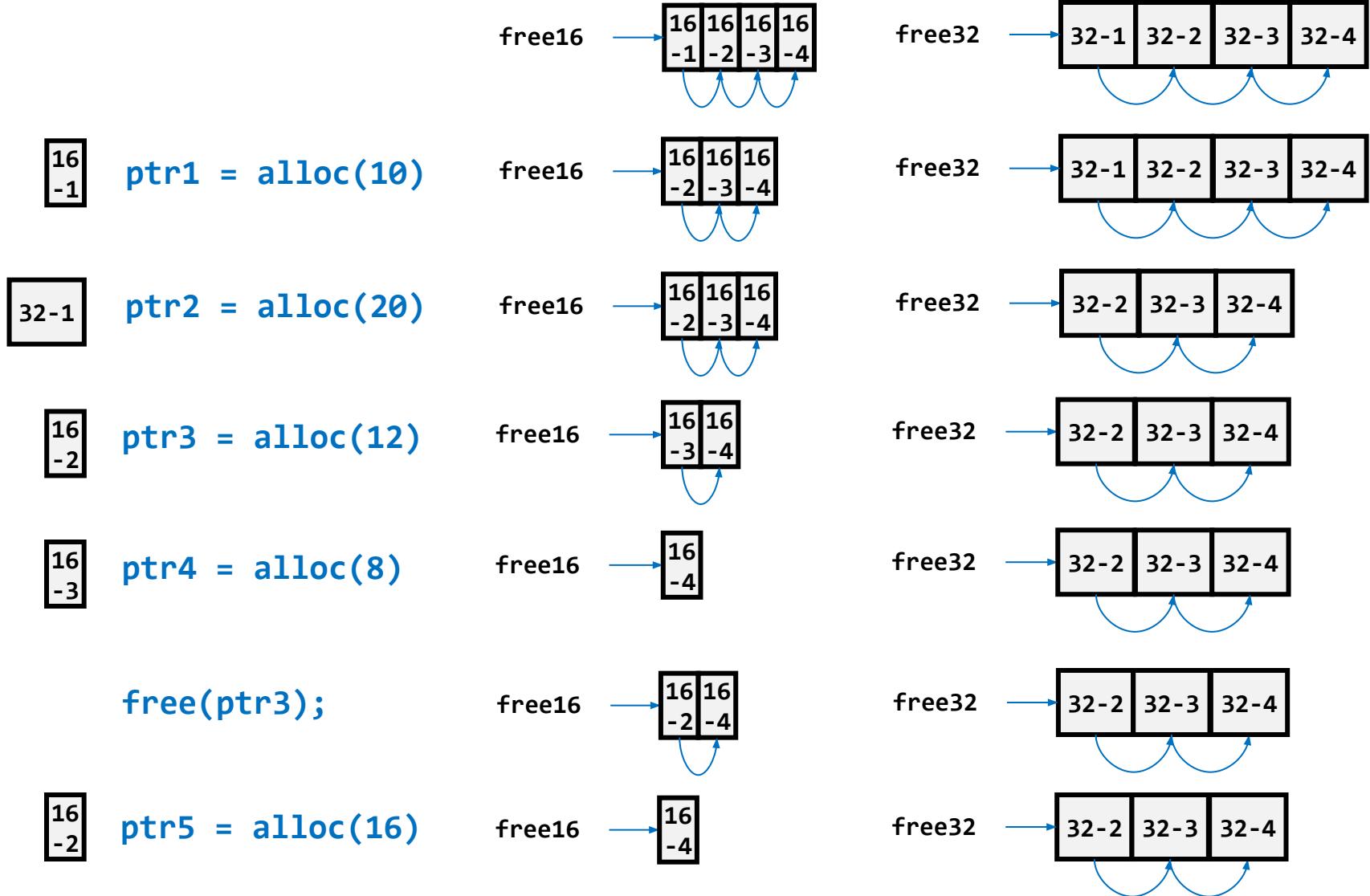
- Idea:
  - Keep separate free lists based on size of the free block
  - Based on the request, pick the appropriate list
- Variations:
  - Segregated Storage
  - Segregated Fit

# Segregated Storage

- One (common) implementation:
  - Maintain lists for fixed size chunks
  - Based on request, allocate smallest fixed size chunk that is free
- Fixed sized blocks allow:
  - No header size or allocated/free flag
  - No coalescing (thus no footer and only singly-linked list)
    - Allows small minimum block size
- If no free blocks in a specific list, allocate more heap space and break it into that size chunks
- Suffers from
  - Internal fragmentation (due to fixed size)
  - Can degenerate to pathological case in some circumstances (ascending order of requests)



# Segregated Storage Example



# Segregated Fit

- Separate lists for various size free chunks
  - Chunks in list size N are at least size N but no more than the lower limit of the next list size
- On allocation, split a chunk of appropriate size and put the fragment back in the appropriate list (based on its size)
- If no free chunk of desired size, keep moving up to larger sized lists
  - If largest list size has no free chunks allocated more heap spaces
- Can coalesce upon freeing a block

free16 →

free32 →

...

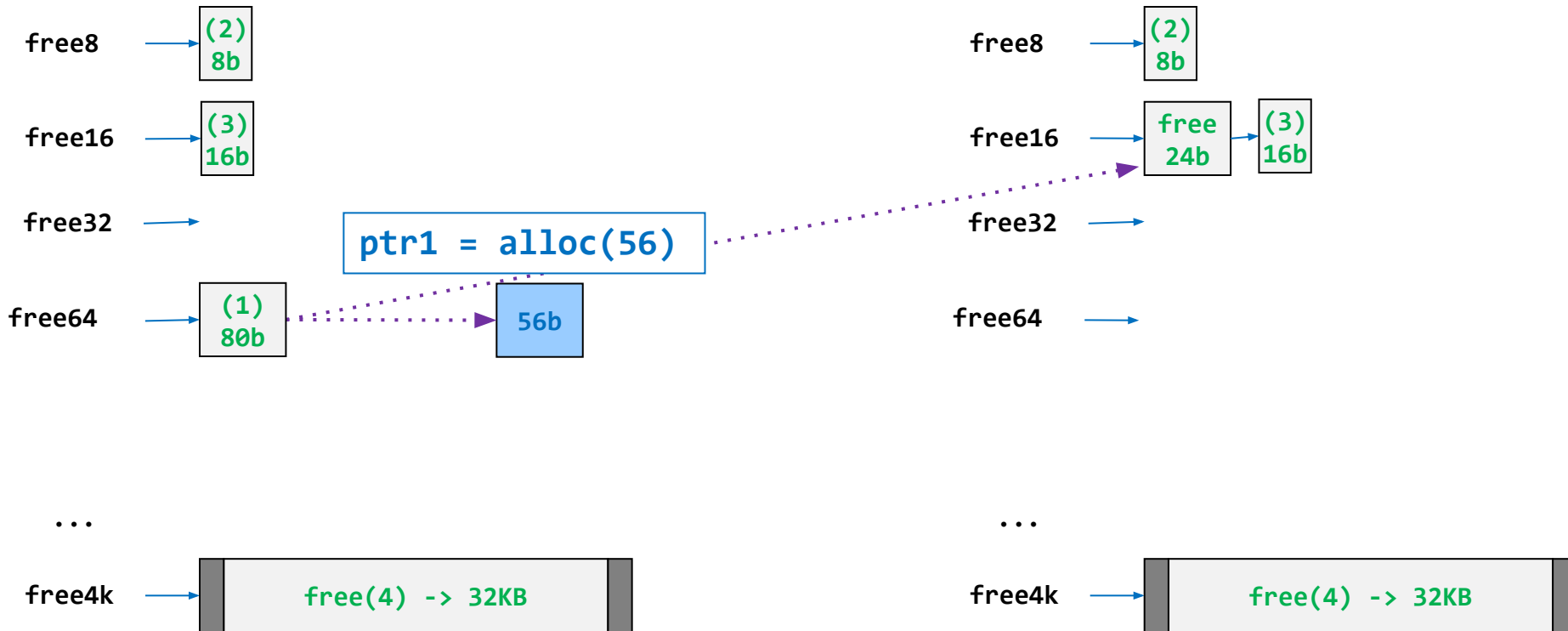
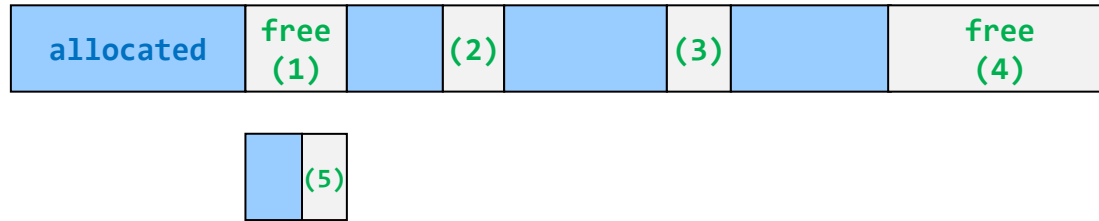
free4k →

free4k(1) -> 1 MB

At start only largest size may exist



# Segregated Fit

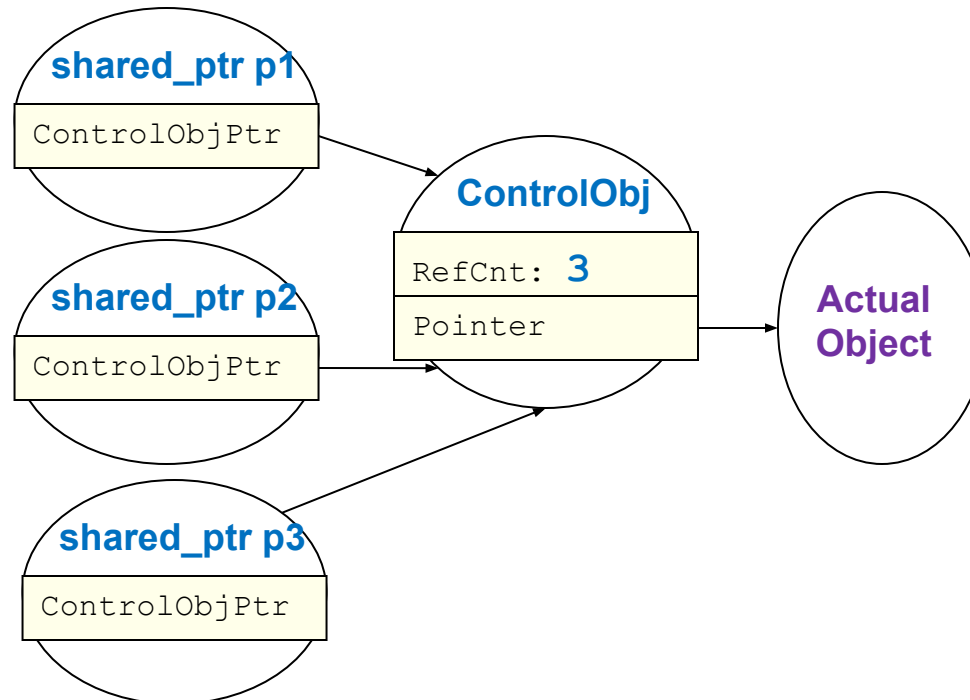


# GARBAGE COLLECTION

# Managed Pointers

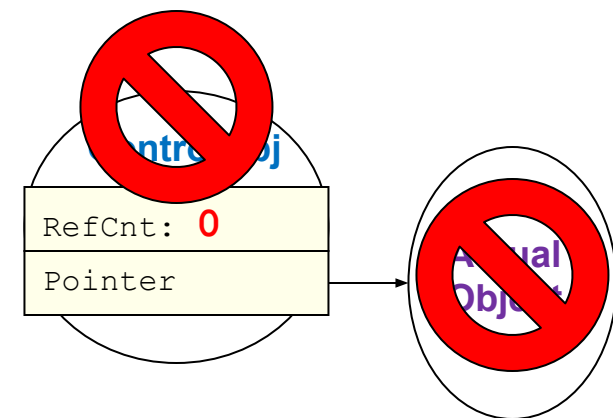
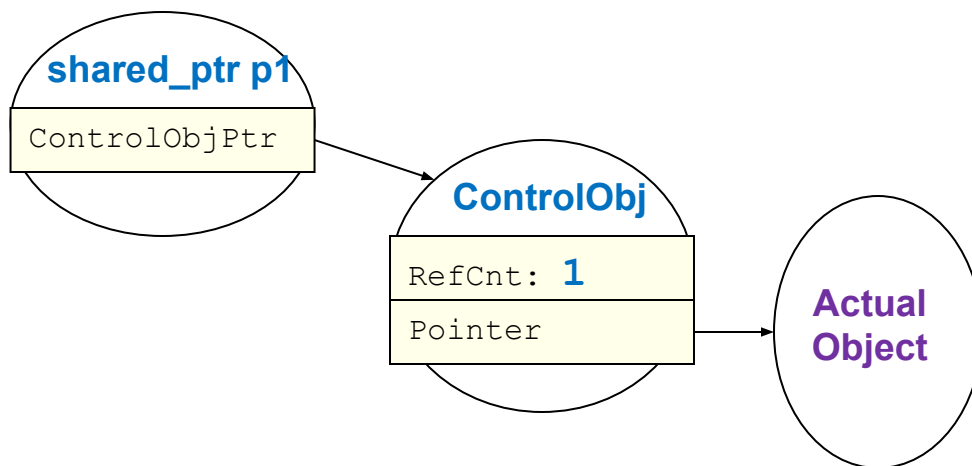
CS:APP 9.10

- Reference count how many items are pointing at the object and deallocate it when the count reaches 0
  - Some languages will perform this automatically, behind the scenes (i.e., Python)



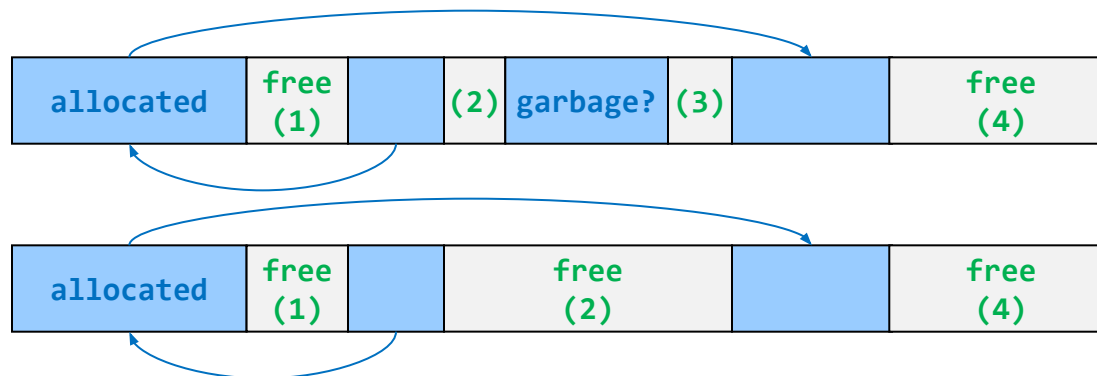
# Managed Pointers (2)

- When the last managed pointer dies or changes to point at another object, the reference count will be decremented to 0 and trigger deallocation



# Implicit Garbage Collection

- Can potentially perform an exhaustive search of allocated blocks (and the stack and globals) to see if any word (dword) is a pointer to another piece of memory in an allocated block
- Any allocated block that is not reachable through some pointer can be garbage collected and marked free
- Requires some intricate book keeping and can be expensive to compute



# Allocation Worksheet

- Consider an 80-byte heap starting at address 0 with the use of implicit free lists with boundary tags.
- Given the sequence of allocations and frees update the state of the heap.

Op	Return	0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	64	68	72	76	
Start (t=0)		4,1	72,0																	72,0	4,1	
1/A(8)	Ret. 8	4,1	16,1			16,1																4,1
2/A(18)	Ret. 24	4,1					32,1						Pad	32,1								4,1
3/A(12)	Ret. 56	4,1													24,1				Pad	24,1		4,1
F(8)		4,1	16,0			16,0																4,1
4/A(10)	Ret. 0	4,1																				4,1
F(24)		4,1	48,0												48,0							4,1
5/A(10)	Ret. 8	4,1	24,1				Pad	24,1	24,0					24,0								4,1
6/A(4)	Ret. 32	4,1							16,1	Pad	16,1	8,0	8,0									4,1
F(8)		4,1	24,0					24,0														4,1
F(56)		4,1											32,0								32,0	4,1
F(32)		4,1	72,0																		72,0	4,1