# Dynamic Memory Allocation

CSCI 237: Computer Organization
32nd Lecture, Friday, November 19, 2025

**Kelly Shaw**






## Administrative Details

- Read CSAPP 9.9
- Lab #6 due 12/5 at 5pm
- TA Feedback Form (posted on Slack)
  - https://forms.gle/mwaWEUy46iHT4MT37
- Colloquium talk today at 2:35pm in Wege
  - Alexis Korb
  - Fronters in Modern Cryptography





## Last Time

- Address translation (Ch 9.6)

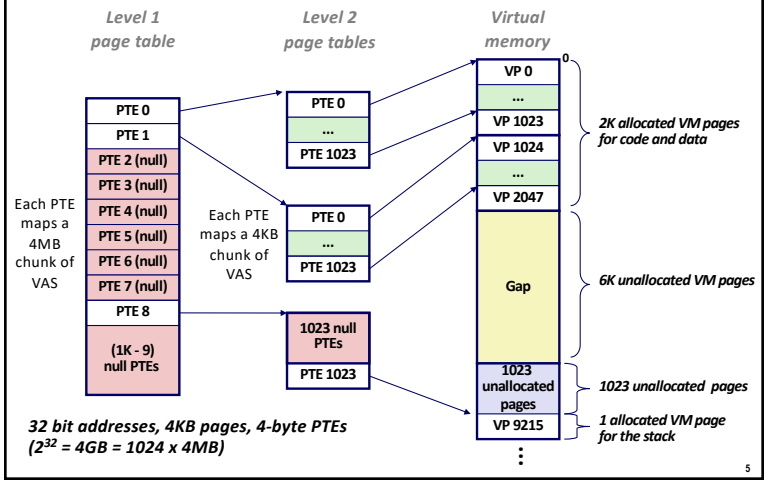




## Today: Dynamic Memory Allocation

- Address Translation
- Dynamic Memory Allocation (Ch 9.9)
  - Basic concepts
    - Fragmentation
      - Internal (fragmentation)
      - External (free space fragmentation)
  - Performance
    - How to Measures "allocator" performance?
      - Throughput
      - (Peak) Utilization

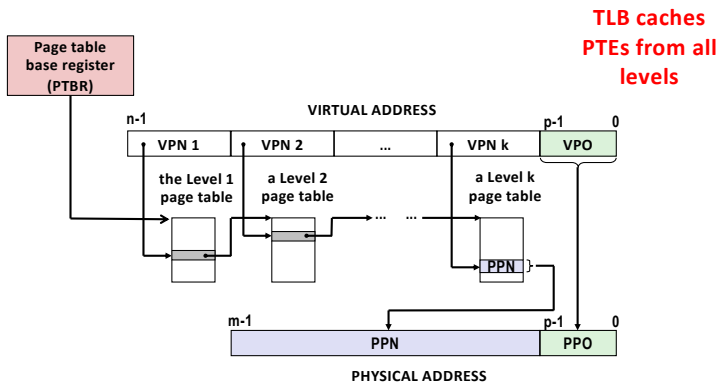
## A Two-Level Page Table Hierarchy




5

5

## Translating with a k-level Page Table




6

6

## Address Translation Summary (Ch 9.6)

- Programmer's view of virtual memory:
  - Each process has its own private linear address space
  - Cannot be corrupted by other processes
- System's view of virtual memory:
  - Uses memory efficiently by caching virtual memory pages
    - Efficient only because of locality
  - Simplifies memory management and programming
  - Simplifies protection by providing a convenient inter-positioning point to check permissions

7

7

## Today: Dynamic Memory Allocation

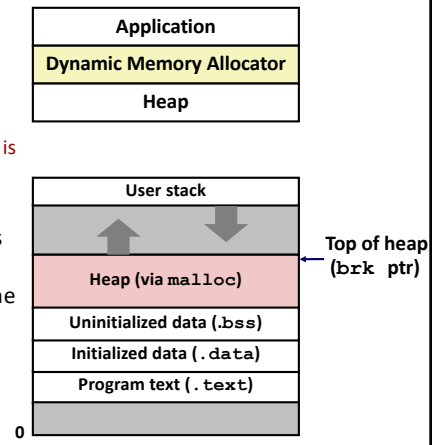
- Address Translation
- Dynamic Memory Allocation (Ch 9.9)
  - Basic concepts
    - Fragmentation
      - Internal (fragmentation)
      - External (free space fragmentation)
  - Performance
    - How to Measures "allocator" performance?
      - Throughput
      - (Peak) Utilization

8

8

## Dynamic Memory Allocation

- Programmers use *dynamic memory allocators* (such as `malloc`) to acquire VM at run time.
  - For data structures whose size is only known at runtime.
- Dynamic memory allocators manage an area of process virtual memory known as the *heap*.




9

9

## Dynamic Memory Allocation

- Allocator maintains heap as collection of variable sized *blocks*, which are either *allocated* or *free.*
- 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

10

10

## 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 16-byte boundary (on x86-64)
  - If **`size == 0`**, returns NULL
- Unsuccessful: returns NULL (0) and sets **`errno`**

```
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

11

11

## `malloc` Example

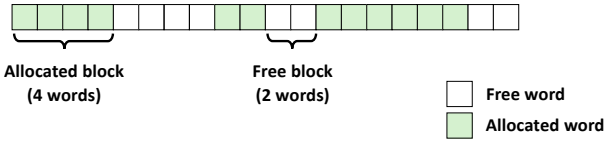
```
#include <stdio.h>
#include <stdlib.h>

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);
}
```

12

12

## Simplifying Assumptions Made in This Lecture

- Memory is word addressed.
- Words are int-sized (4 bytes).
  - Each box is a 4 byte word.
- Allocations are double-word (8 byte) aligned.



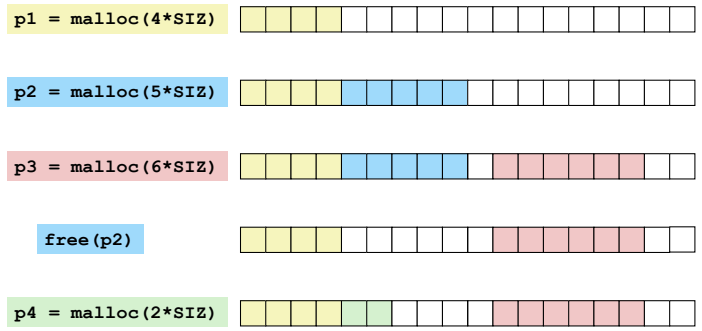

13

13

## Allocation Example

```
#define SIZ sizeof(int)
```

```
p1 = malloc(4*SIZ)
p2 = malloc(5*SIZ)
p3 = malloc(6*SIZ)
free(p2)
p4 = malloc(2*SIZ)
```



14

14

## Constraints

- Applications
  - Can issue arbitrary sequence of **`malloc`** and **`free`** requests
  - **`free`** request must be to a **`malloc`**'d block
- Allocators
  - Can't control number or size of allocated blocks
  - Must respond immediately to **`malloc`** requests
    - *i.e.*, can't reorder or buffer requests
  - Must allocate blocks from free memory
    - *i.e.*, can only place allocated blocks in free memory
  - Must align blocks so they satisfy all alignment requirements
    - 16-byte (x86-64) alignment on Linux machines
  - Can manipulate and modify only free memory
  - Can't move the allocated blocks once they are **`malloc`**'d
    - *i.e.*, compaction is not allowed

15

15

## Evaluating a Memory Allocator

- What does it mean for an allocator to be good?
  - How do we measure the "performance" of an allocator?
  - How do we measure "quality" of the allocations?
- As we talk about designs, think about the best and worst cases

16

16

## Performance Goals: Throughput

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

17

17

## Performance Goals: Peak Memory Utilization

- Given some sequence of `malloc` and `free` requests:
  - $R_0, R_1, ..., R_k, ..., R_{n-1}$
- *Def: 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
    - i.e., heap only grows when allocator uses **`sbrk`**
- *Def: Peak memory utilization after k+1 requests*
  - $U_k = ( max_{i \le k} P_i ) \ / \ H_k$

18

18

## Performance Goals: Peak Memory Utilization

- Given some sequence of `malloc` and `free` requests:
  - $R_0, R_1, ..., R_k, ..., R_{n-1}$
- *Def: Aggregate payload* $P_k$
- **Performance Goals:**
  - **1) Maximize throughput**
  - **2) Maximize peak memory utilization**
- **These goals are often conflicting!**
    - i.e., heap only grows when allocator uses **`sbrk`**
- *Def: Peak memory utilization after k+1 requests*
  - $U_k = ( max_{i \le k} P_i ) \ / \ H_k$

19

19

## Fragmentation

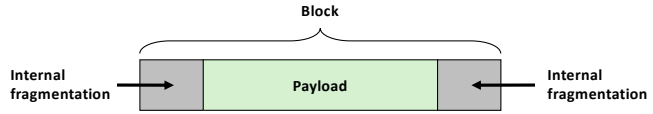
- Poor memory utilization often caused by *fragmentation*
- Two classes:
  - ***internal*** fragmentation
  - ***external*** fragmentation

20

20

## Internal Fragmentation

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




- Caused by
  - Overhead of maintaining heap data structures
  - Padding for alignment purposes
  - 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

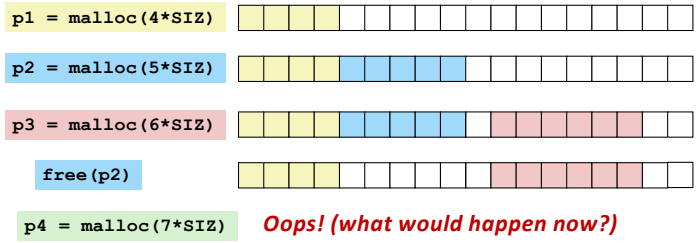
21

21

## External Fragmentation

```
#define SIZ sizeof(int)
```

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




- Depends on the pattern of future requests
  - Thus, difficult to measure

22

22

## Implementation Issues

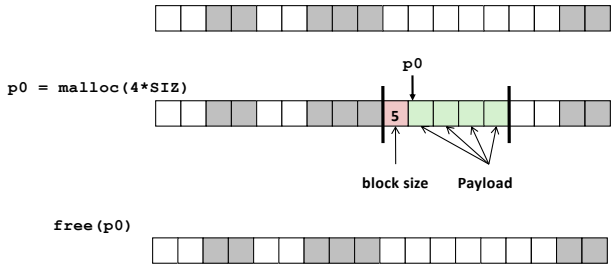
- How do we know how much memory to **free(void *)** given just a pointer?
- How do we keep track of the free blocks?
- What do we do with the extra space when allocating a structure that is smaller than the free block it is placed in?
- How do we pick a block to use for allocation -- many might fit?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

23

23

## 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***
  - Downside: this method requires an extra word for every allocated ***block***

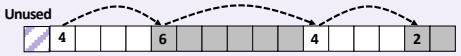



24

24

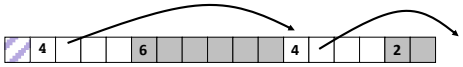
## Keeping Track of Free Blocks

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




Need to tag each block as allocated/free

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




Need space for 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

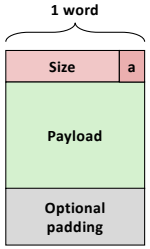
25

25

## Method 1: Implicit Free List (Ch 9.9.6)

- For each block we need both size and allocation status
  - We could store this information in two words, but that is wasteful!
- Standard trick
  - When blocks are aligned, some (3) low-order address bits are always 0
  - Instead of storing an always-0 bit, repurpose it as an allocated/free flag
  - When reading the `Size` word, we just mask out this bit

*Format of allocated and free blocks*




**a = 1: Allocated block**
**a = 0: Free block**

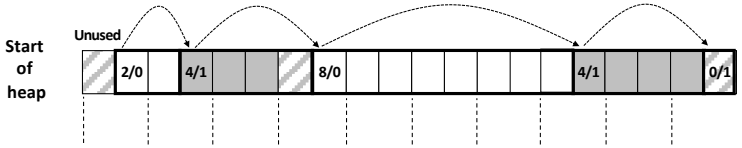
**Size: block size**

**Payload: application data (allocated blocks only)**

26

26

## Detailed Implicit Free List Example




Double-word aligned

**Allocated blocks:** shaded
**Free blocks:** unshaded
**Headers:** labeled with "size in words/allocated bit"

**Note:** both allocated and free blocks are all in the same implicit list

27

27

## Implicit List: Finding a Free Block (9.9.7)

- *First fit:*
  - Search list from beginning, choose ***first*** free block that fits:

```
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)
```

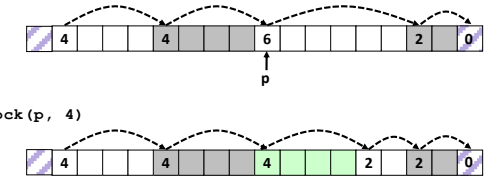
  - Can take linear time in total number of blocks (allocated *and* free)
  - In practice it can cause "splinters" at beginning of list – fragmentation!
- *Next fit:*
  - Like first fit, but search list starting where previous search finished
  - Should often be faster than first fit: avoids re-scanning unhelpful blocks
  - Some research suggests that fragmentation is worse
- *Best fit:*
  - Search the list, choose the ***best*** free block (i.e., fits with the fewest bytes left over)
  - Keeps fragments small—usually improves memory utilization
  - Will typically run slower than first fit

28

28

## 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




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

29

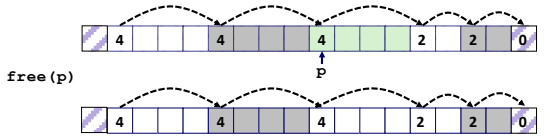
29

## 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*SIZ)
```

***Oops!***

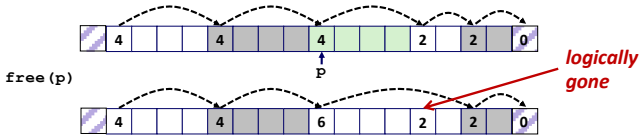
***There is enough contiguous free space, but the allocator won't be able to find it***

30

30

## Implicit List: Coalescing

- 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
```

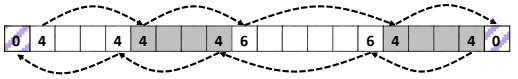
  - But how do we coalesce with *previous* block?

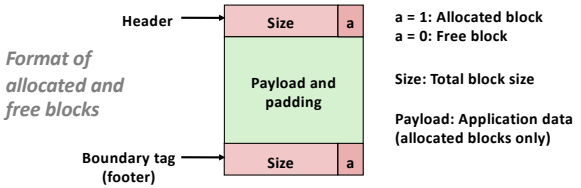
31

31

## 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!

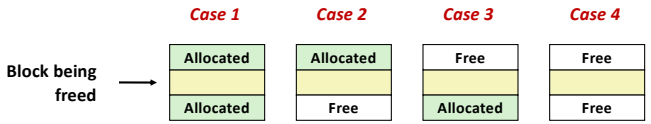



32

32

## Constant Time Coalescing with Boundary Tags



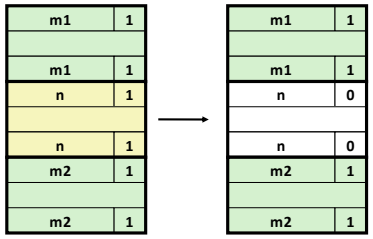

Given a block to free and its two neighbors, there are 4 unique combinations of free/allocated to consider. Let's look at each case individually

33

33

## Constant Time Coalescing (Case 1)

The block's immediate neighbors are both allocated.

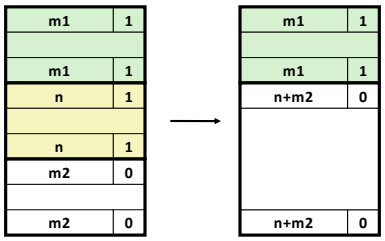



34

34

## Constant Time Coalescing (Case 2)

The block's predecessor is allocated, but its successor is free.

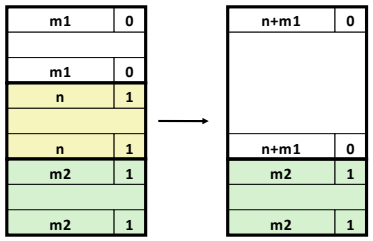



35

35

## Constant Time Coalescing (Case 3)

The block's predecessor is free, but it's successor is allocated.

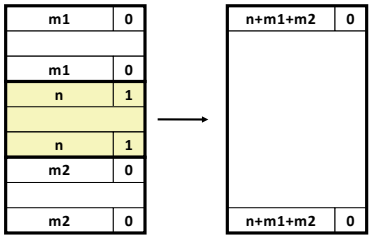



36

36

## Constant Time Coalescing (Case 4)

The block's immediate neighbors are both free.




37

37

## Disadvantages of Boundary Tags

- Internal fragmentation
  - Extra non-payload bytes needed for boundary tag/footer
- Can it be optimized?
  - Which blocks need the footer tag?
  - What does that mean?

38

38

## Disadvantages of Boundary Tags

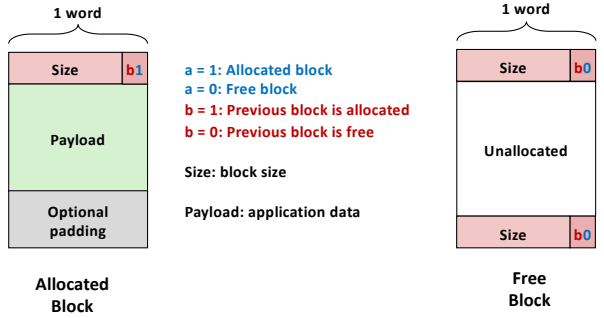
- Internal fragmentation
  - Extra non-payload bytes needed for boundary tag/footer
- Can it be optimized?
  - Which blocks need the footer tag? Only free blocks!
  - What does that mean? Can save space!

39

39

## Optimization: No Boundary Tag for Allocated Blocks

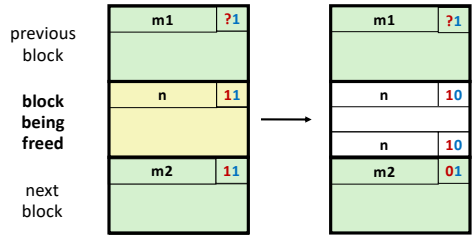
- Boundary tag is only needed for free blocks
- Insight: when sizes are multiples of 4 or more, have 2+ spare bits




40

40

## No Boundary Tag for Allocated Blocks (Case 1)

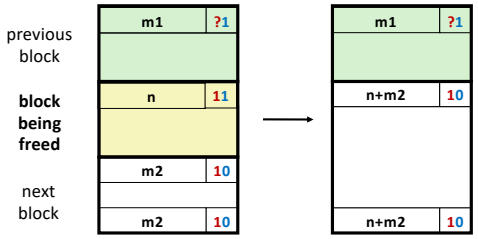



**Header:** Use 2 bits (always zero due to alignment):
(previous block allocated)<<1 | (current block allocated)

41

41

## No Boundary Tag for Allocated Blocks (Case 2)

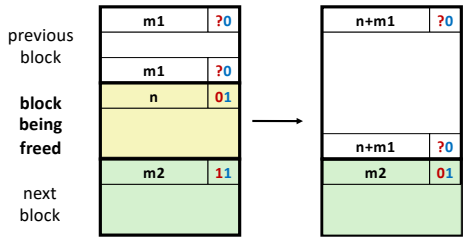



**Header:** Use 2 bits (always zero due to alignment):
(previous block allocated)<<1 | (current block allocated)

42

42

## No Boundary Tag for Allocated Blocks (Case 3)

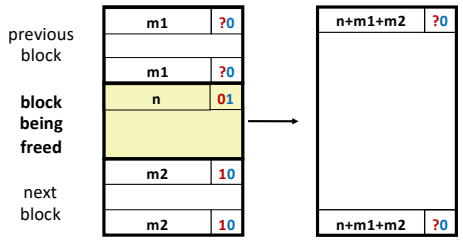



**Header:** Use 2 bits (always zero due to alignment):
(previous block allocated)<<1 | (current block allocated)

43

43

## No Boundary Tag for Allocated Blocks (Case 4)




**Header:** Use 2 bits (always zero due to alignment):
(previous block allocated)<<1 | (current block allocated)

44

44

## Summary of Key Allocator Policies

- Placement policy:
  - First-fit, next-fit, best-fit, etc.
  - Trades off lower throughput for less fragmentation
  - ***Interesting observation***: segregated free lists (next lecture) approximate a best fit placement policy without having to search entire free list
- 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

45

45

## 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

46

46

## Practice

- Assuming double-word alignment is used and the implicit free list format discussed in slides is used (i.e., 4 byte header). Block sizes are rounded up to nearest multiple of 8 bytes to maintain alignment.
- What would the block size be in bytes (including payload, header, and padding)? What would be stored in the block header (in hex)?
  - malloc(2)
  - malloc(9)
  - malloc(16)

47

47