Dynamic Memory Allocation

CSCI 237: Computer Organization
33rd Lecture, Monday, November 25, 2024

Kelly Shaw

Administrative Details

- Read CSAPP 9.9
- Lab #6 due 12/6 at 5pm
- No quiz this week
- Meet in Ward lab (TBL 301) on Monday 12/2 at 10am

Last Time

- Address translation (Ch 9.6)
- End-to-end example of a simple memory system
- Dynamic Memory Allocation (Ch 9.9)

Today: Dynamic Memory Allocation

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

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*
  - $U_k = (max_{i \le k} P_i) \ / \ H_k$

5

5

Fragmentation

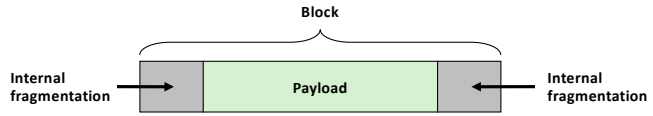
- Poor memory utilization often caused by *fragmentation*
- Two classes:
  - ***internal*** fragmentation
  - ***external*** fragmentation

6

6

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

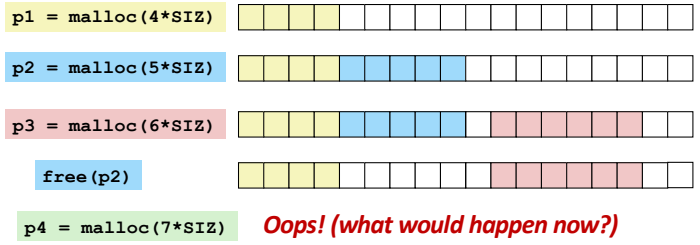
7

7

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

8

8

Implementation Issues

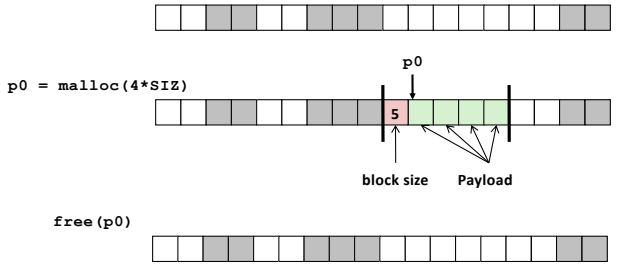
- How do we know how much memory to **free(void *)** given just a pointer?
- How do we keep track of the free blocks?
- What do we do with the extra space when allocating a structure that is smaller than the free block it is placed in?
- How do we pick a block to use for allocation -- many might fit?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

9

9

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***

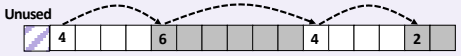


10

10

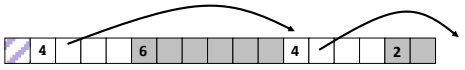
Keeping Track of Free Blocks

- 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

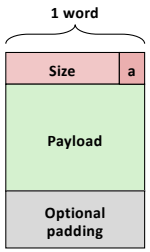
11

11

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

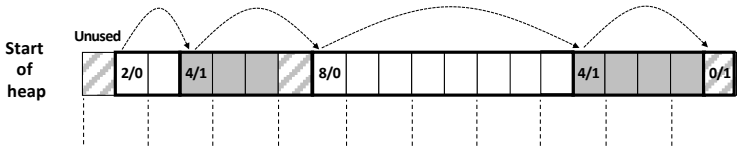
Size: block size

Payload: application data (allocated blocks only)

12

12

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

13

13

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)
```

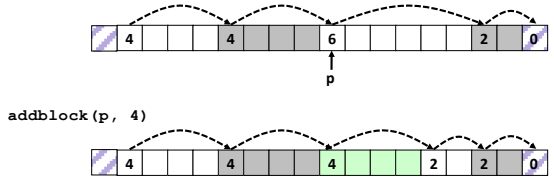
  - Can take linear time in total number of blocks (allocated *and* free)
  - In practice it can cause "splinters" at beginning of list – fragmentation!
- *Next fit:*
  - Like first fit, but search list starting where previous search finished
  - Should often be faster than first fit: avoids re-scanning 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

14

14

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
```

15

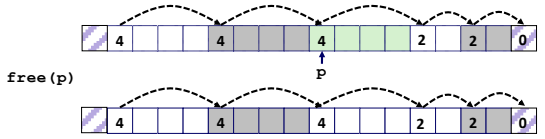
15

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!

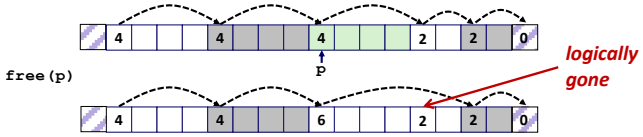
There is enough contiguous free space, but the allocator won't be able to find it

16

16

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
```

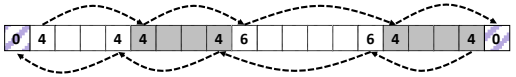
  - But how do we coalesce with *previous* block?

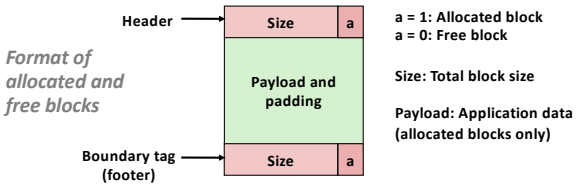
17

17

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!

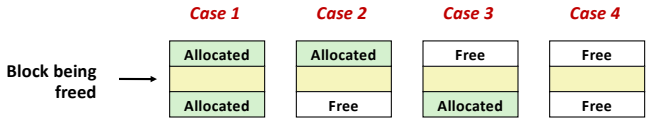




18

18

Constant Time Coalescing with Boundary Tags



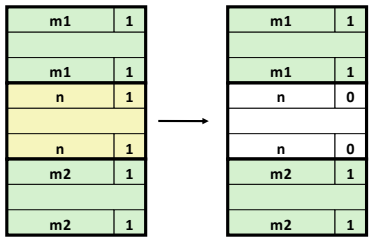
Given a block to free and its two neighbors, there are 4 unique combinations of free/allocated to consider. Let's look at each case individually

19

19

Constant Time Coalescing (Case 1)

The block's immediate neighbors are both allocated.

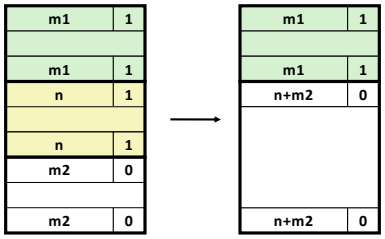


20

20

Constant Time Coalescing (Case 2)

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

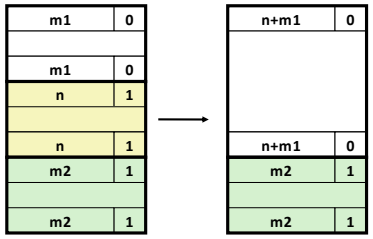


21

21

Constant Time Coalescing (Case 3)

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

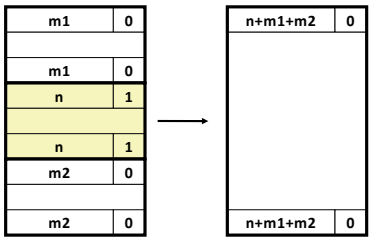


22

22

Constant Time Coalescing (Case 4)

The block's immediate neighbors are both free.



23

23

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?

24

24

Disadvantages of Boundary Tags

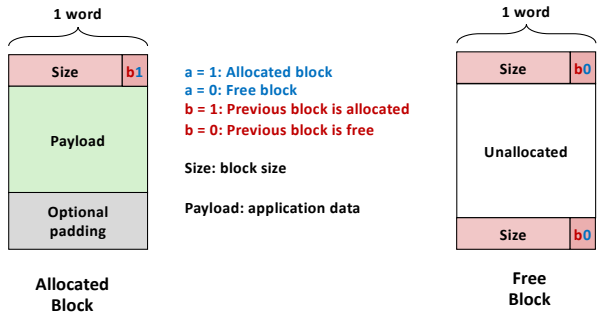
- Internal fragmentation
  - Extra non-payload bytes needed for boundary tag/footer
- Can it be optimized?
  - Which blocks need the footer tag? Only free blocks!
  - What does that mean? Can save space!

25

25

Optimization: No Boundary Tag for Allocated Blocks

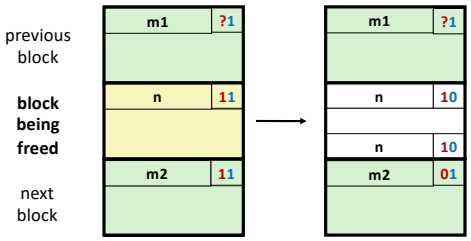
- Boundary tag is only needed for free blocks
- Insight: when sizes are multiples of 4 or more, have 2+ spare bits



26

26

No Boundary Tag for Allocated Blocks (Case 1)



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

27

27