Section 10: Memory Allocation Topics

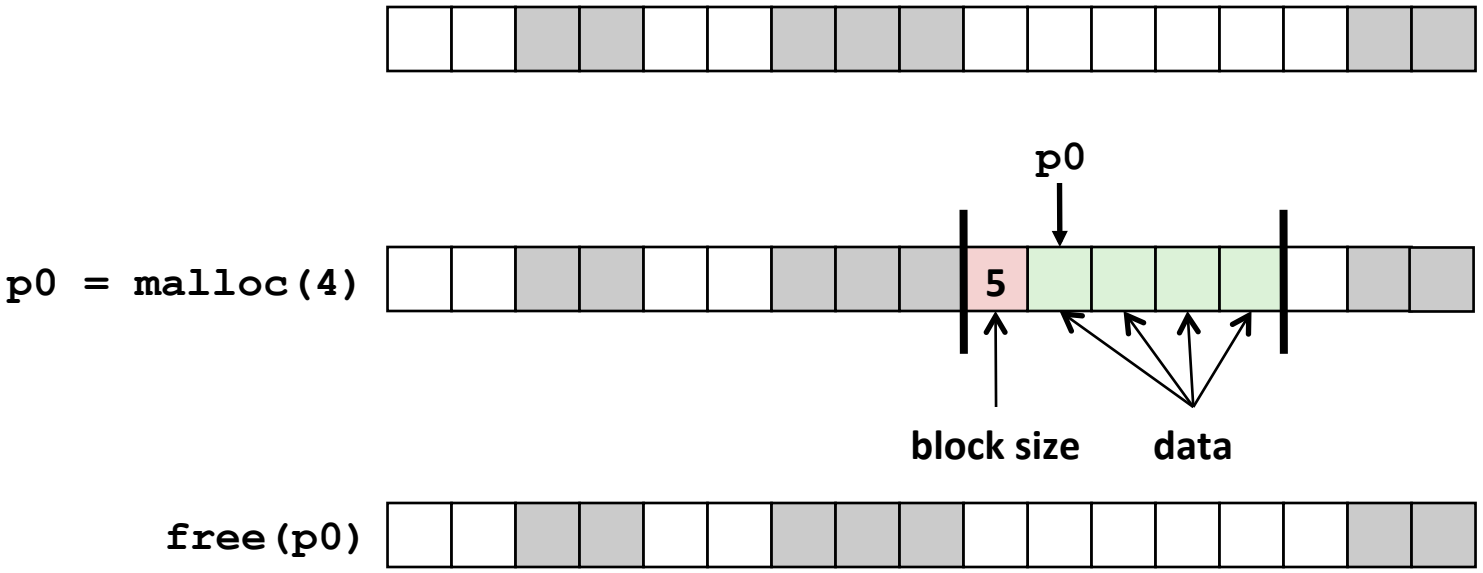
- ~~**Dynamic memory allocation**~~
  - ~~Size/number of data structures may only be known at run time~~
  - ~~Need to allocate space on the heap~~
  - ~~Need to de-allocate (free) unused memory so it can be re-allocated~~
- **Implementation**
  - Implicit free lists
  - Explicit free lists – subject of next programming assignment
  - Segregated free lists
- **Garbage collection**
- **Common memory-related bugs in C programs**

Implementation Issues

- How do we know how much memory to free given just a pointer?
- How do we keep track of the free blocks?
- How do we pick a block to use for allocation (when many might fit)?
- 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 reinsert freed block into the heap?

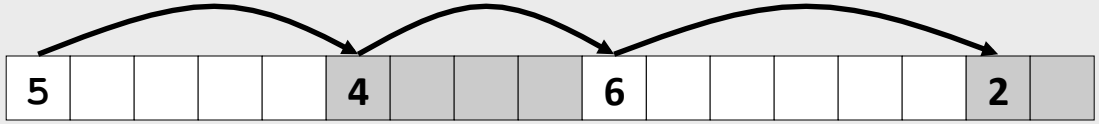
Knowing How Much to Free

- **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

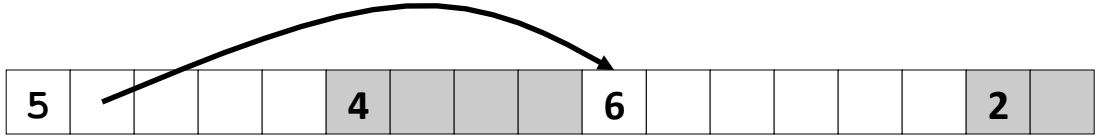


Keeping Track of Free Blocks

- **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 binary tree (e.g. red-black tree) with pointers within each free block, and the length used as a key

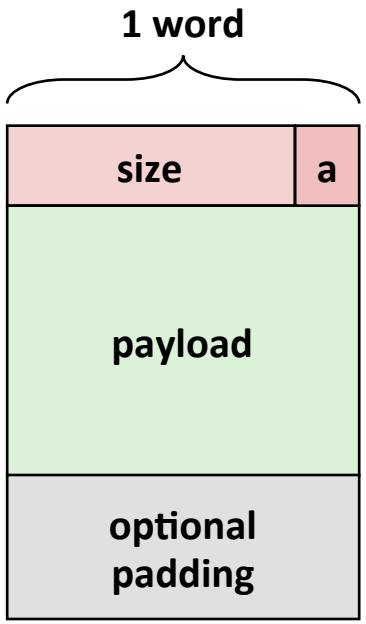
Implicit Free Lists

e.g. with 8-byte alignment, sizes look like:

00000000
00001000
00010000
00011000
…

- **For each block we need: size, is-allocated?**
  - Could store this information in two words: wasteful!
- **Standard trick**
  - If blocks are aligned, some low-order size bits are always 0
  - Instead of storing an always-0 bit, use it as a allocated/free flag
  - When reading size, must remember to mask out this bit

Format of allocated and free blocks



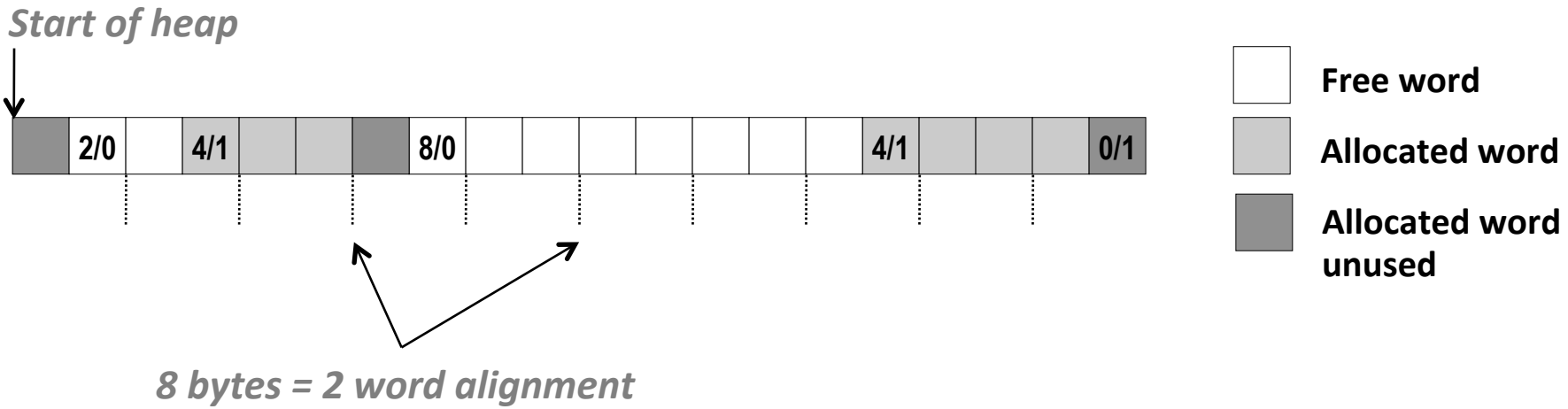
a = 1: allocated block
a = 0: free block

size: block size

payload: application data (allocated blocks only)

Implicit Free List Example

Sequence of blocks in heap: 2/0, 4/1, 8/0, 4/1 (size/allocated)



- **8-byte alignment**
  - May require initial unused word
  - Causes some internal fragmentation
- **One word (0/1) to mark end of list**

Implicit List: Finding a Free Block

*p gets the block *header*
*p & 1 extracts the allocated bit
*p & -2 masks the allocated bit, gets just the size

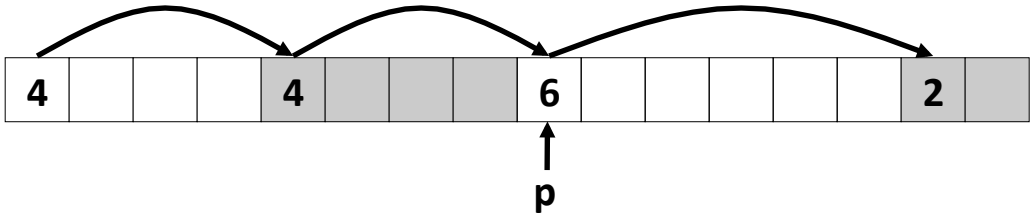
- ***First fit:***
  - Search list from beginning, choose ***first*** free block that fits:

```
p = heap_start;
while ((p < end) &&      \\ not passed end
       ((*p & 1) ||      \\ already allocated
       (*p <= len)))     \\ too small
  p = p + (*p & -2);     \\ goto next block
```

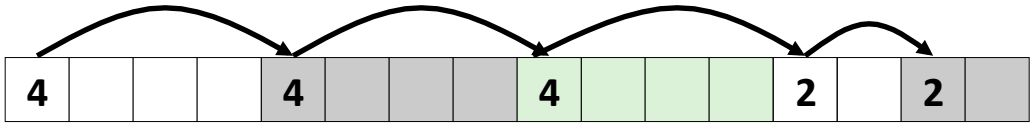
  - Can take time linear in total number of blocks (allocated and free)
  - In practice it can cause "splinters" at beginning of list
- ***Next fit:***
  - Like first-fit, but search list starting where previous search finished
  - Should often be faster than first-fit: avoids re-scanning unhelpful blocks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



`addblock(p, 4)`



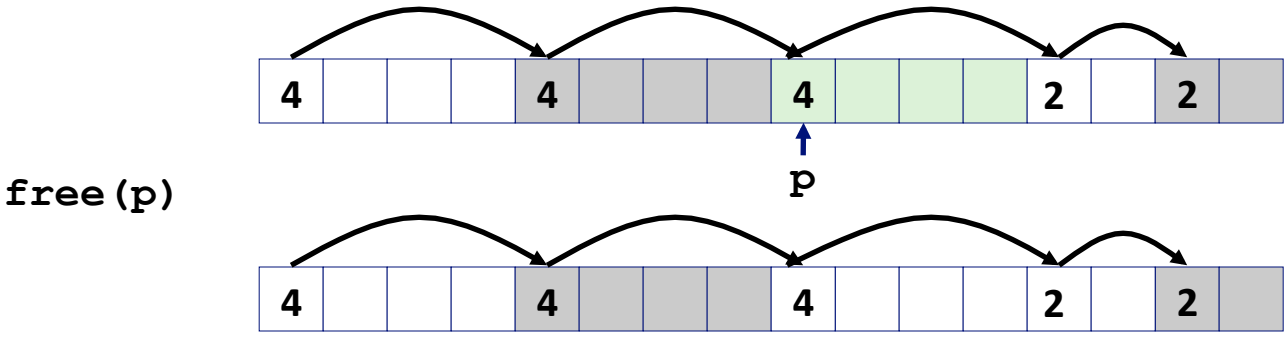

```
void addblock(ptr p, int len) {
  int newsize = ((len + 1) >> 1) << 1;  // round up to even
  int oldsize = *p & -2;                // mask out low bit
  *p = newsize | 1;                     // set new length + allocated
  if (newsize < oldsize)
    *(p+newsize) = oldsize - newsize;   // set length 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"

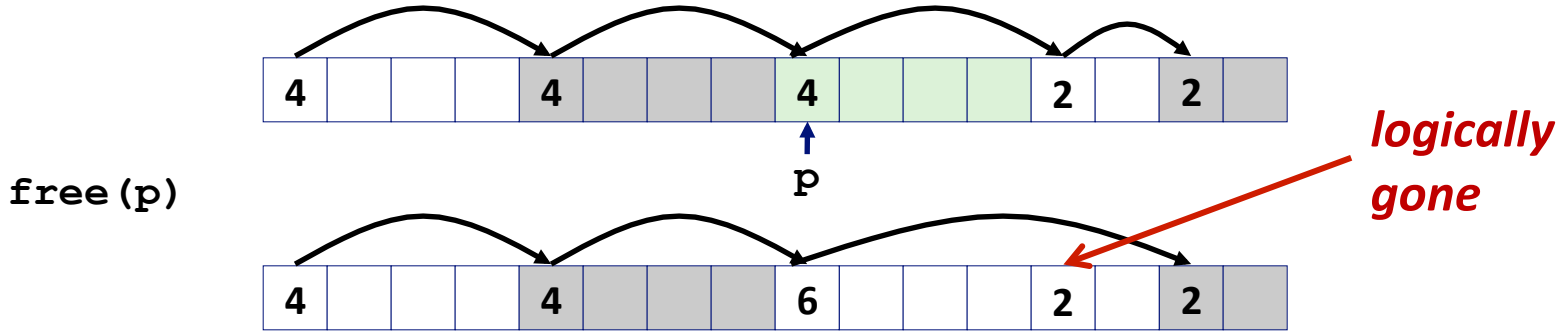


`malloc(5)` ***Oops!***

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

Implicit List: Coalescing

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

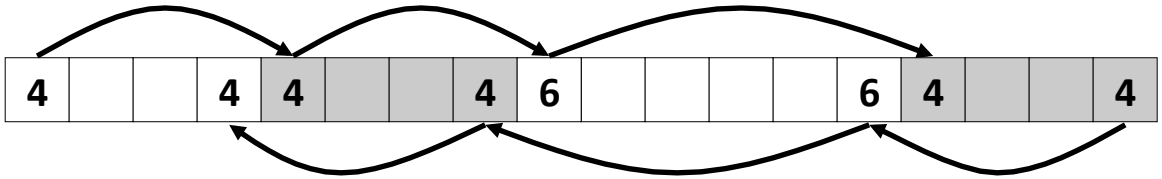



```
void free_block(ptr p) {
    *p = *p & -2;           // clear allocated bit
    next = p + *p;          // find next block
    if ((*next & 1) == 0)
      *p = *p + *next;      // add to this block if
}                           //    not allocated
```

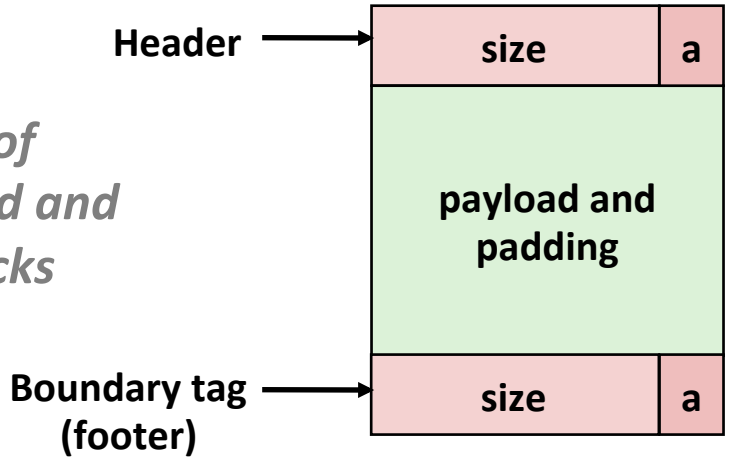
  - But how do we coalesce with the *previous* block?

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



a = 1: allocated block
a = 0: free block

size: total block size

payload: application data (allocated blocks only)

Implicit Free Lists: Summary

- **Implementation: very simple**
- **Allocate cost:**
  - linear time (in total number of heap blocks) worst case
- **Free cost:**
  - constant time worst case
  - even with coalescing
- **Memory utilization:**
  - 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 some special purpose applications

- **The concepts of splitting and boundary tag coalescing are general to *all* allocators**