Dynamic Memory Allocation: Implicit Free Lists

COMP 321

Dave Johnson

1

Keeping Track of the Available Free Blocks

Many design choices are possible, but mainly two general alternatives

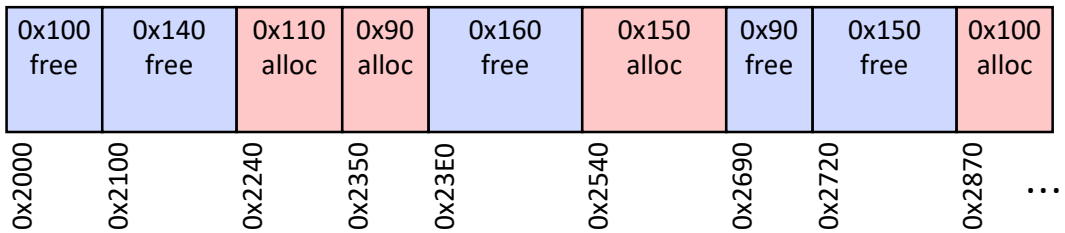
- ***Explicit free list***
  - Basically, the same as any other linked list, with the usual kind of pointer manipulations
  - Entries on the list are the free blocks themselves
  - Each free block points to (i.e., contains the address of) the next free block
- ***Implicit free list***
  - No actual pointers used to make the list
  - Each block implicitly "points to" next block, both free and allocated blocks
  - Much easier to describe and to implement than an explicit free list
  - ***We'll start with the idea of an implicit free list***

2

Implicit Free List: The Basic Idea

The heap itself acts as the free list, with no pointers needed within the list

- Each block contains the size of the block and its status (allocated vs. free)



- To traverse the free list, start at the beginning address of the heap
- Walk forward block by block by adding the size of the current block
  - Skip over allocated blocks and just go on to the next block

Where in the Block to Put the Size and Status?

When allocating a new block

- Increase the requested size by the extra space needed to hold the block's size and its allocation status
  - Actually allocate this new, larger size, not just the requested size
- Record the block size in the block as the size ***including*** this extra space
- And mark the block status there as allocated

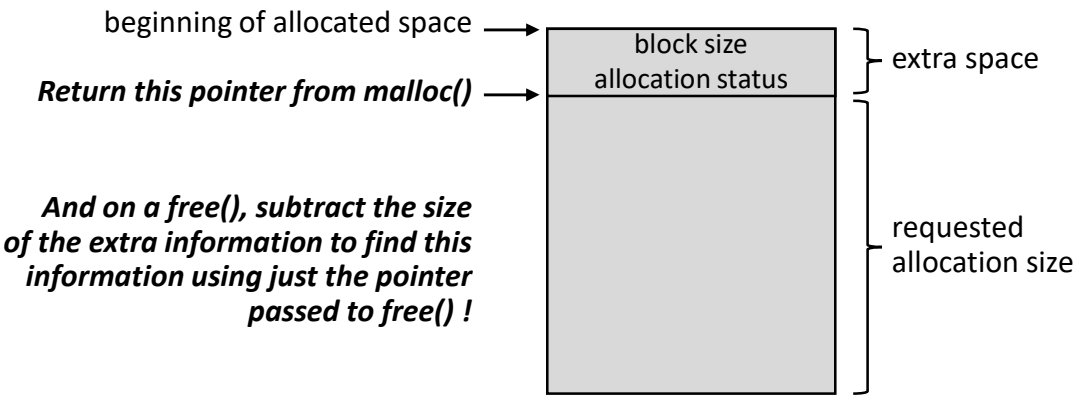
When freeing a block

- Simplest: Just change the status in the block to show that the block is free
- The block size is already recorded there from when the block was allocated
- And there is enough space in the block to hold this information, since space was made for it when the block was allocated (the increased size)

Where in the Block to Put the Size and Status?

Where within the block should this extra information be placed?

- Best place is at the beginning of the block



And on a free(), subtract the size of the extra information to find this information using just the pointer passed to free() !

How to Represent the Block Size and Status?

The malloc() alignment requirement creates a convenient solution

- The status only needs 1 bit to represent it (e.g., 0 = free, 1 = allocated)
- And if the alignment requirement is for at least at a multiple of 2
  - The aligned (rounded up) size will always be an even number
  - Meaning the least significant bit will always be a 0
- So we can "borrow" this least significant bit to store the allocation status bit
  - Store the value (aligned_size | status) // merges in the status bit
  - To get the status (stored_value & 1) // extracts just the status bit
  - To get the size (stored_value & ~1) // takes away the status bit
- So storing the block status takes effectively zero space!
- Can use, e.g., status = 0 for "free", status = 1 for "allocated"

Examples with an Alignment Requirement of 8

Meaning must round the real size up to a multiple of 8 to get aligned

- Example: malloc(65) = malloc(0x41)
  - size after rounding up to multiple of 8 = 72 = 0x48 = 0100 1000
- Example: malloc(999) = malloc(0x3E7)
  - size after rounding up to multiple of 8 = 1000 = 0x3E8 = 0011 1110 1000
- Example: malloc(1024) = malloc(0x400)
  - size after rounding up to multiple of 8 = 1024 = 0x400 = 0100 0000 0000

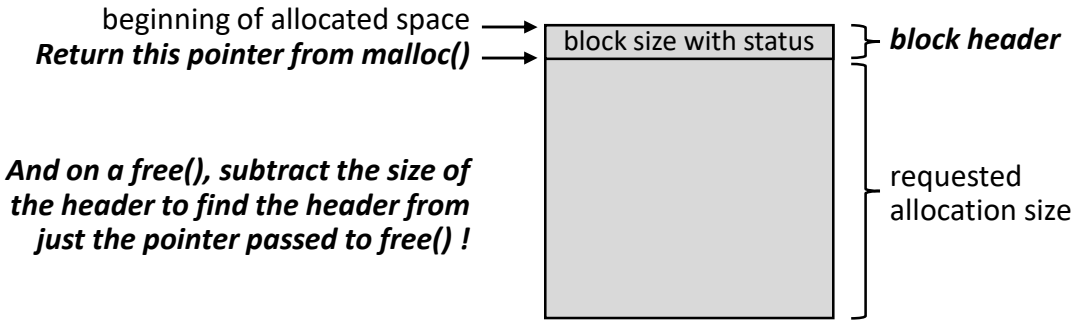
In <u>all</u> cases, after rounding up, the low 3 bits of the size will <u>always</u> be 000

- So we can safely use one of those bits to store the status
  - And we can still set or extract the size and status separately
  - Again, can use, e.g., status = 0 for "free", status = 1 for "allocated"

The Format of a Block

The block size and status together is often referred to as the block's "header"

- One word – the size needed to represent the largest supported block size
  - e.g., unsigned int, or unsigned long, or size_t
- With the least significant bit of this word "borrowed" to store the status



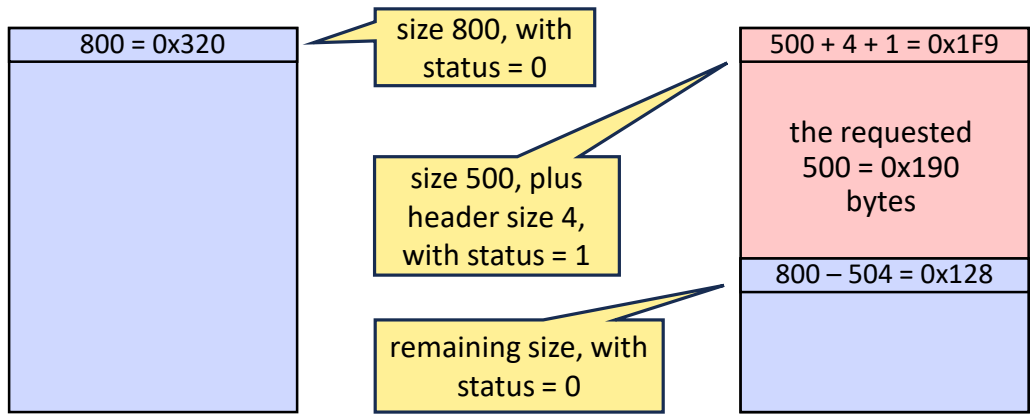
When Allocating a Block

Find the free block to use (e.g., first fit, next fit, or best fit), but then what?

- Change the allocated status to show the block is allocated
- Then, depending on the size of the block being used and the size requested
  - use the whole block, or
  - split the block and use part of it, and leave the rest as a (smaller) free block, ***with its own new header*** to specify its new size and status
- Example: If the "leftover" size would be smaller than a header
  - Then ***can't*** leave the rest as a free block, so ***use the whole block***
- Example: If the "leftover" size is big enough but is still fairly small
  - Then can ***optionally*** choose to ***use the whole block*** and not leave anything
- ***Otherwise, split the block and leave the rest as a free block***

Example: Splitting a Block (header size = 4)

- Requested size = decimal 500 = 0x190
- Existing size (including header) = decimal 800 = 0x320



When Freeing a Block

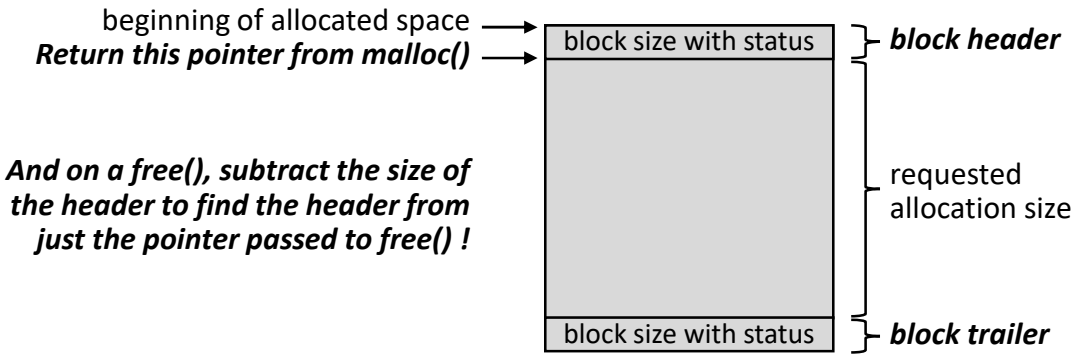
Change its status to free (e.g., change bit from 1 to 0), but then what?

- With an implicit free list, just changing its status to free adds it to the free list
- Then, if the ***preceding*** block and/or ***following*** block are also free, can ***"coalesce"*** them together into a single, larger free block
  - Checking the status of the ***following*** block is easy, since it's header is easy to find – the next word following the end of this block
  - Checking the status of the ***preceding*** block is much harder – traverse from the beginning of the list to find the preceding block?
  - ***A better solution***: duplicate the header also at the end of the block, referred to there as the ***"trailer"***

The Format of a Block (with a Trailer Too)

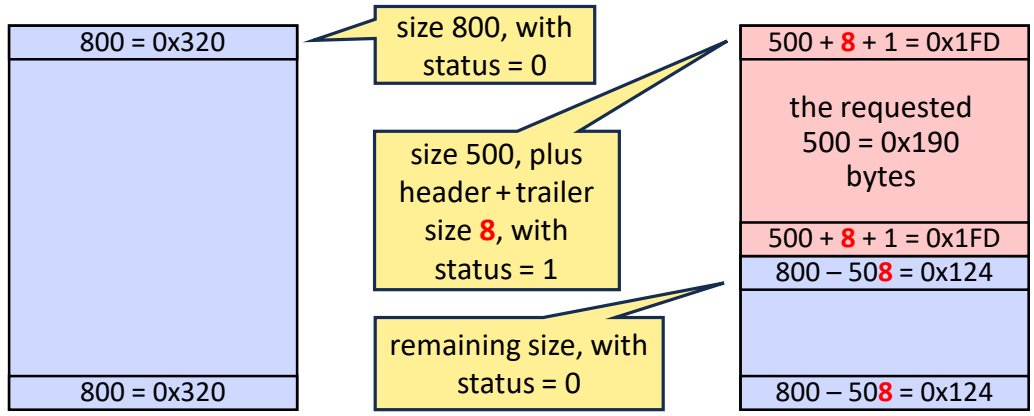
Must add 2 × the size of a header to the size requested

- The space for the header before the actual "allocated" space
- The space for the trailer after the actual "allocated" space



Example: Splitting a Block (with header and trailer)

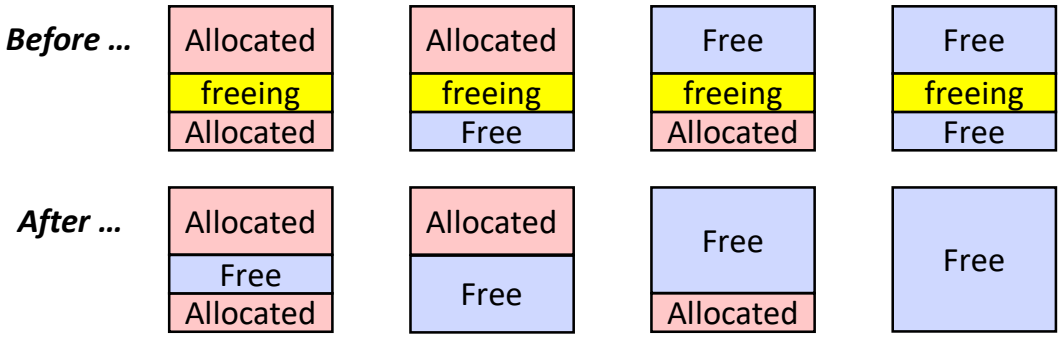
- Requested size = decimal 500 = 0x190
- Suppose existing size (including header and trailer) = decimal 800 = 0x320



Adding a Block Being Freed Back Into the Free List

To keep down external fragmentation, "coalesce" it with neighboring free blocks

- ***Now easy, with the header of following block and trailer of preceding block***
- Consider the four possible cases for the block in memory before and after



Implicit Free List: Summary

Advantages

- Easy to implement, since no actual pointers need to be manipulated
- Allows efficient, constant time coalescing, with use of a ***header*** and a ***trailer*** in ***every*** block (all free and all allocated blocks)
- Relatively space efficient, since no pointers need to be stored in each block

Disadvantages

- Allocation time can be linear in the number of blocks (the number of free ***and*** allocated blocks), not just in the number of ***free*** blocks
- No real way to improve on that with an ***implicit*** free list
  - Can't excluded allocated blocks from the list
  - Can't control the order or segregation of the free list

Some Cautions in Using the malloc Package

- What happens if you write ***past*** the end of an allocated block?
  - This overwrites control information that affects the rest of the ***free list***
  - Potentially overwrites the contents of ***allocated*** blocks in memory after that
- What happens if you free() a block that you had ***already*** earlier freed?
  - The apparent preceding ***header*** and/or following ***footer*** are "wrong", if the earlier free() coalesced with blocks in memory then
  - If the block now being free()d had been ***allocated*** after the earlier first free(), then this second free() messes up this block's state (and size)

These (and other similar) problems can be hard to debug

- These problems don't generally show themselves when the real bug occurs
- But instead, e.g., on later use of other allocated blocks or later malloc() or free()