Optimizing Malloc and Free

Professor Jennifer Rexford

http://www.cs.princeton.edu/~jrex

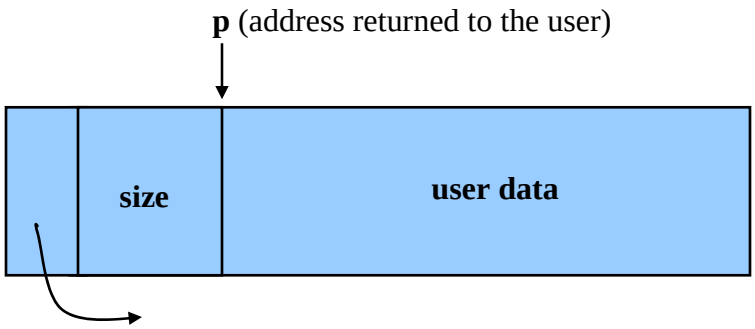
Goals of This Lecture

- Brief review of K&R implementation
  - Circular linked list of free blocks, with pointer and size in header
    - Malloc: first-fit algorithm, with splitting
    - Free: coalescing with adjacent blocks, if they are free
  - Limitations
    - Fragmentation of memory due to first-fit strategy
    - Linear time to scan the list during **`malloc`** and **`free`**
- Optimizations related to assignment #6
  - Placement choice, splitting, and coalescing
  - Faster free
    - Size information in both header and footer
    - Next and previous free-list pointers in header and footer
  - Faster malloc
    - Separate free list for free blocks of different sizes
    - One bin per block size, or one bin for a range of sizes

Free Block: Pointer, Size, Data

- Free block in memory
  - Pointer to the next block
  - Size of the block
  - User data

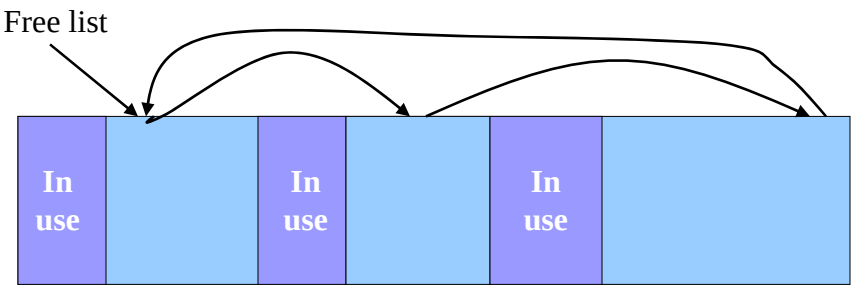
header



3

Free List: Circular Linked List

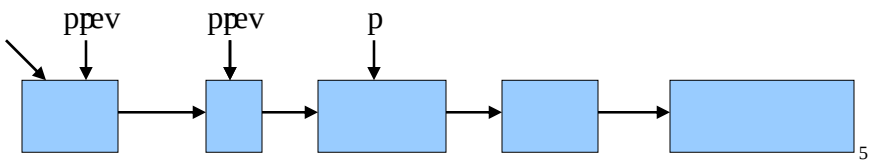
- Free blocks, linked together
  - Example: circular linked list
- Keep list in order of increasing addresses
  - Makes it easier to coalesce adjacent free blocks



4

Malloc: First-Fit Algorithm

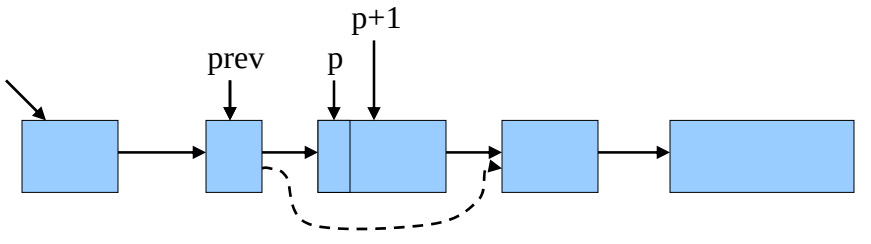
- Start at the beginning of the list
- Sequence through the list
  - Keep a pointer to the previous element
- Stop when reaching first block that is big enough
  - Patch up the list
  - Return a pointer to the user



5

Malloc: First Case, A Perfect Fit

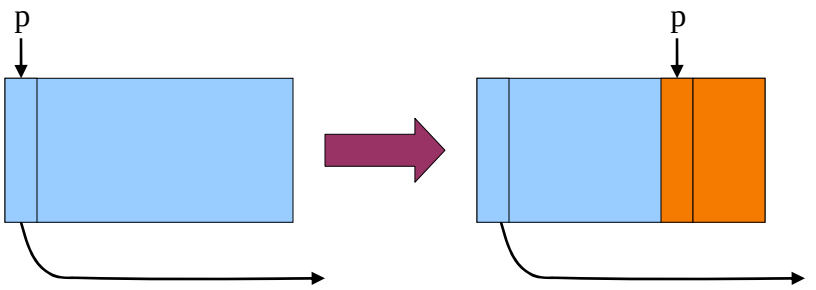
- Suppose the first fit is a perfect fit
  - Remove the block from the list
  - Link the previous free block with the next free block
  - Return the current to the user (skipping header)



6

Malloc: Second Case: Big Block

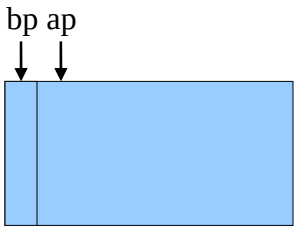
- Suppose the block is bigger than requested
  - Divide the free block into two blocks
  - Keep first (now smaller) block in the free list
  - Allocate the second block to the user



7

Free

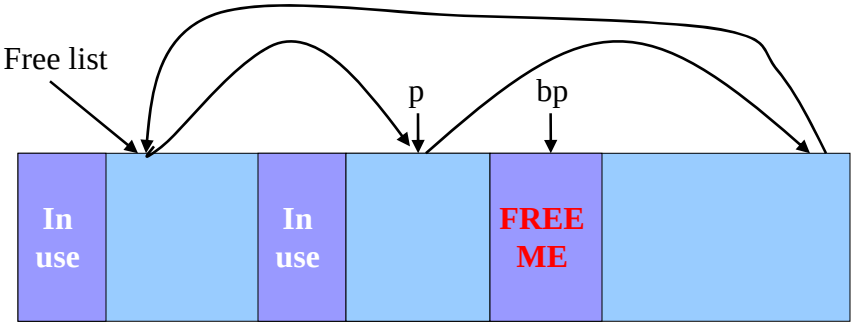
- User passes a pointer to the memory block
  - `void free(void *ap);`
- Free function inserts block into the list
  - Identify the start of entry
  - Find the location in the free list
  - Add to the list, coalescing entries, if needed



8

Free: Finding Location to Insert

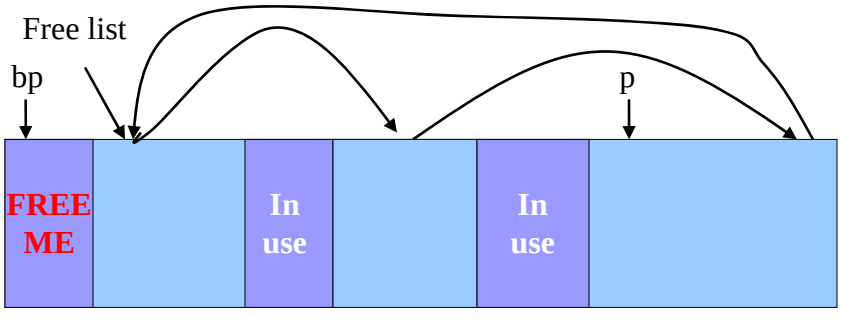
- Start at the beginning
- Sequence through the list
- Stop at last entry before the to-be-freed element



9

Free: Handling Corner Cases

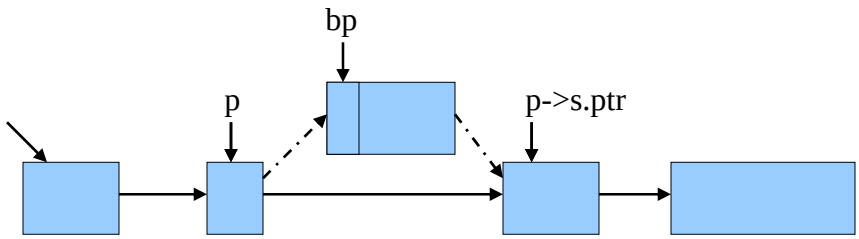
- Check for wrap-around in memory
  - To-be-freed block is before first entry in the free list, or
  - To-be-freed block is after the last entry in the free list



10

Free: Inserting Into Free List

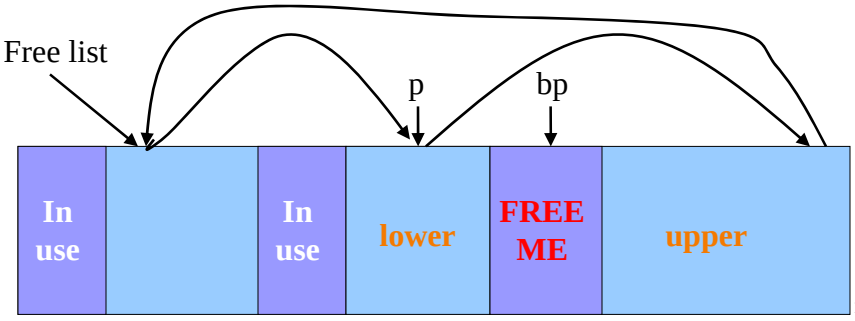
- New element to add to free list
- Insert in between previous and next entries
- But, there may be opportunities to coalesce



11

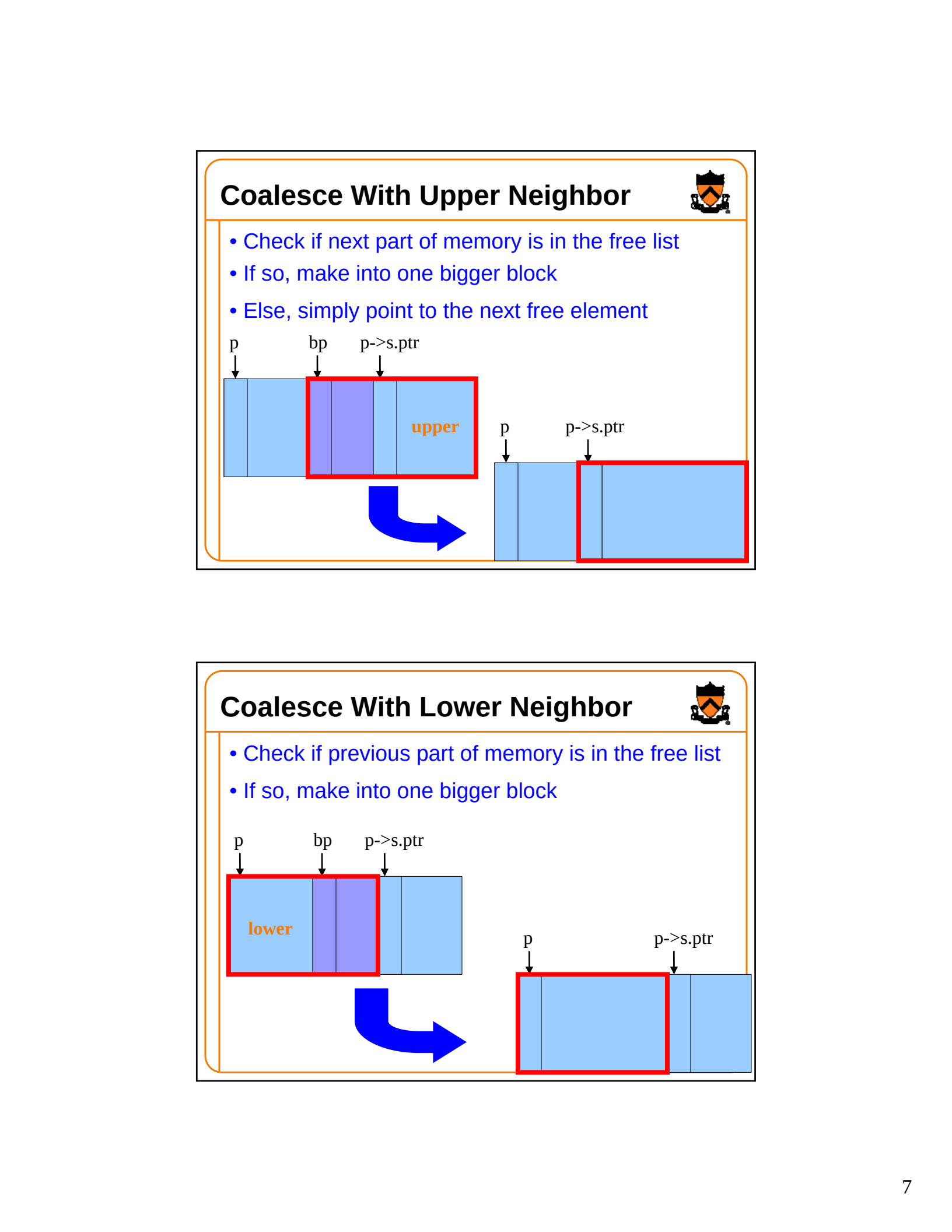
Coalescing With Neighbors

- Scanning the list finds the location for inserting
  - Pointer to to-be-freed element: **bp**
  - Pointer to previous element in free list: **p**
- Coalescing into larger free blocks
  - Check if contiguous to upper and lower neighbors



12

Coalesce With Upper Neighbor

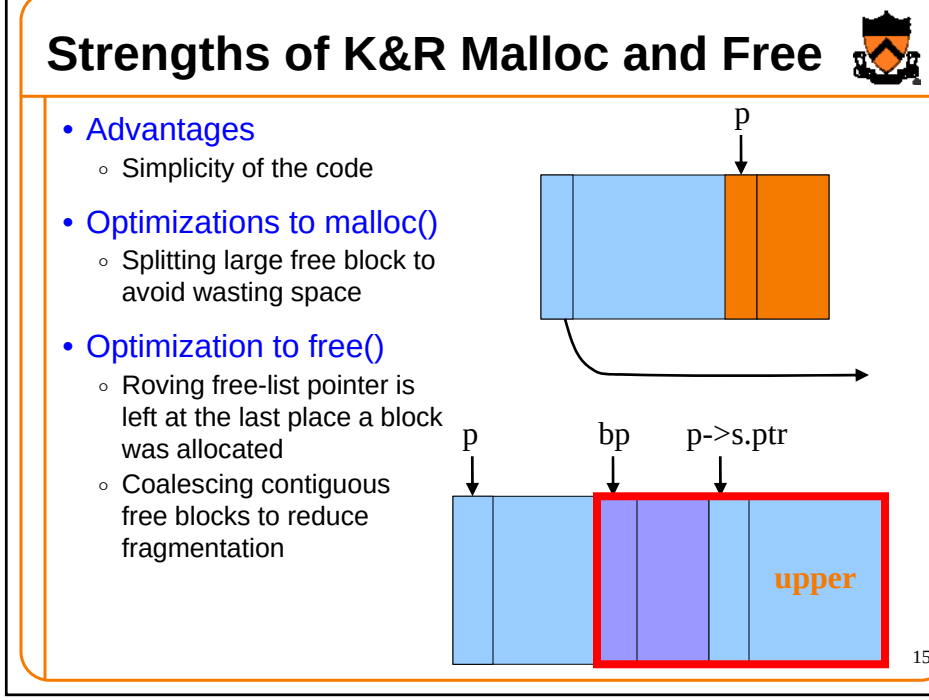


- Check if next part of memory is in the free list
- If so, make into one bigger block
- Else, simply point to the next free element

Coalesce With Lower Neighbor

- Check if previous part of memory is in the free list
- If so, make into one bigger block

Strengths of K&R Malloc and Free

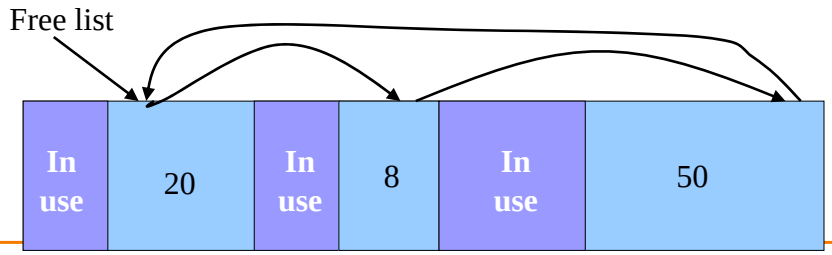


- Advantages
  - Simplicity of the code
- Optimizations to malloc()
  - Splitting large free block to avoid wasting space
- Optimization to free()
  - Roving free-list pointer is left at the last place a block was allocated
  - Coalescing contiguous free blocks to reduce fragmentation

15

Weaknesses of K&R Malloc and Free

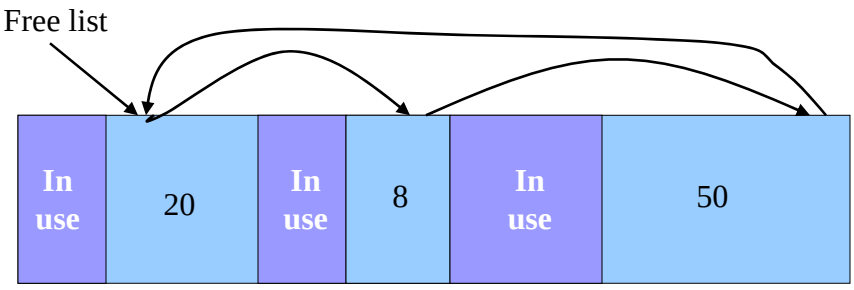
- Inefficient use of memory: fragmentation
  - Best-fit policy can leave lots of “holes” of free blocks in memory
- Long execution times: linear-time overhead
  - Malloc scans the free list to find a big-enough block
  - Free scans the free list to find where to insert a block
- Accessing a wide range of memory addresses in free list
  - Can lead to large amount of paging to/from the disk



16

Improvements: Placement

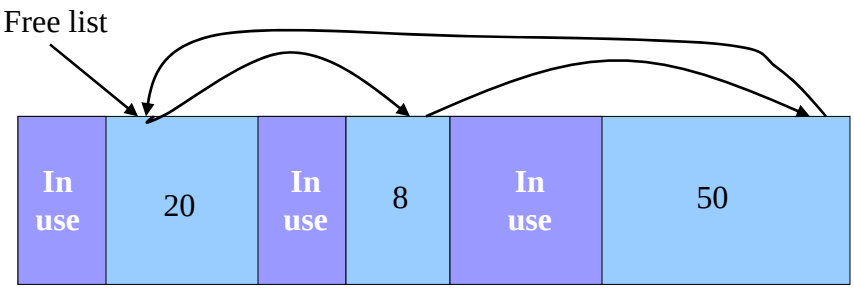
- Placement: reducing fragmentation
  - Deciding which free block to use to satisfy a **`malloc()`** request
  - K&R uses “first fit” (really, “next fit”)
    - Example: **`malloc(8)`** would choose the 20-byte block
  - *Alternative*: “best fit” or “good fit” to avoid wasting space
    - Example: **`malloc(8)`** would choose the 8-byte block



17

Improvements: Splitting

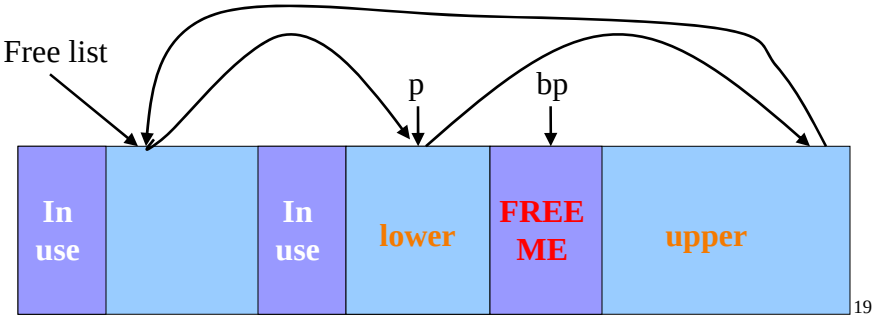
- Splitting: avoiding wasted memory
  - Subdividing a large free block, and giving part to the user
  - K&R **`malloc()`** does splitting whenever the free block is too big
    - Example: **`malloc(14)`** splits the 20-byte block
  - *Alternative*: selective splitting, only when the savings is big enough
    - Example: **`malloc(14)`** allocates the entire 20-byte block



18

Improvements: Coalescing

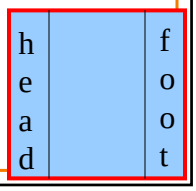
- Coalescing: reducing fragmentation
  - Combining contiguous free blocks into a larger free blocks
  - K&R does coalescing in **`free()`** whenever possible
    - Example: combine free block with lower and upper neighbors
  - *Alternative*: deferred coalescing, done only intermittently
    - Example: wait, and coalesce many blocks at a time later



19

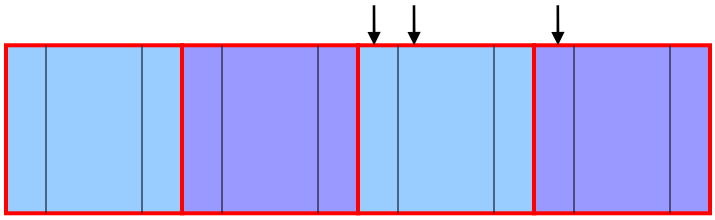
Improvements: Faster Free

- Performance problems with K&R **`free()`**
  - Scanning the free list to know where to insert
  - Keeping track of the "previous" node to do the insertion
- Doubly-linked, non-circular list
  - Header
    - Size of the block (in # of units)
    - Flag indicating whether the block is free or in use
    - If free, a pointer to the next free block
  - Footer
    - Size of the block (in # of units)
    - If free, a pointer to the previous free block



Size: Finding Next Block

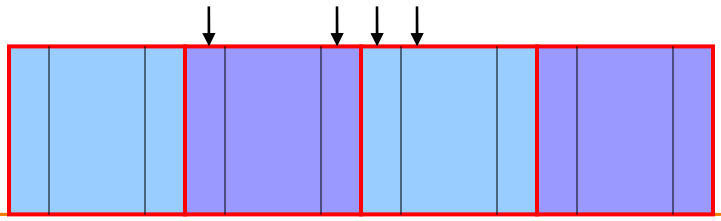
- Go quickly to next block in memory
  - Start with the user's data portion of the block
  - Go backwards to the head of the block
    - Easy, since you know the size of the header
  - Go forward to the head of the next block
    - Easy, since you know the size of the current block



21

Size: Finding Previous Block

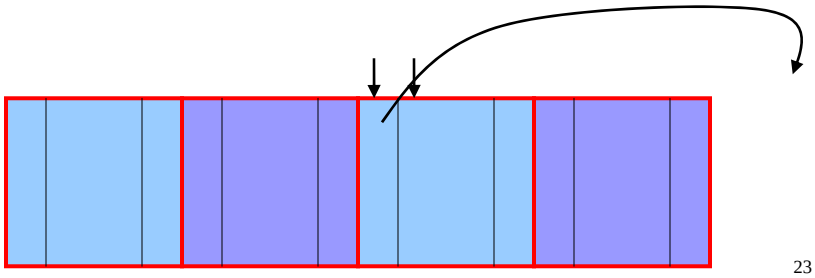
- Go quickly to previous chunk in memory
  - Start with the user's data portion of the block
  - Go backwards to the head of the block
    - Easy, since you know the size of the header
  - Go backwards to the footer of the previous block
    - Easy, since you know the size of the footer
  - Go backwards to the header of the previous block
    - Easy, since you know the size from the footer



22

Pointers: Next Free Block

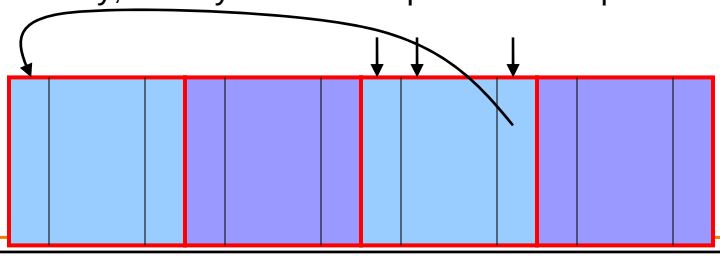
- Go quickly to next free block in memory
  - Start with the user's data portion of the block
  - Go backwards to the head of the block
    - Easy, since you know the size of the header
  - Go forwards to the next free block
    - Easy, since you have the next free pointer



23

Pointers: Previous Free Block

- Go quickly to previous free block in memory
  - Start with the user's data portion of the block
  - Go backwards to the head of the block
    - Easy, since you know the size of the header
  - Go forwards to the footer of the block
    - Easy, since you know the block size from the header
  - Go backwards to the previous free block
    - Easy, since you have the previous free pointer



24

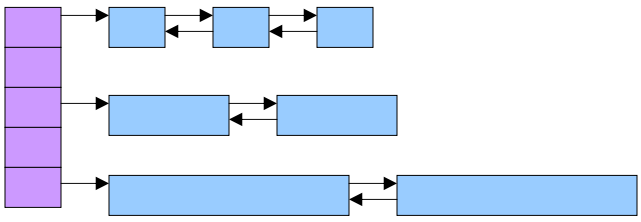
Efficient Free

- Before: K&R
  - Scan the free list till you find the place to insert
    - Needed to see if you can coalesce adjacent blocks
  - Expensive for loop with several pointer comparisons
- After: with header/footer and doubly-linked list
  - Coalescing with the previous block in memory
    - Check if previous block in memory is also free
    - If so, coalesce
  - Coalescing with the next block in memory the same way
  - Add the new, larger block to the front of the linked list

25

But Malloc is Still Slow…

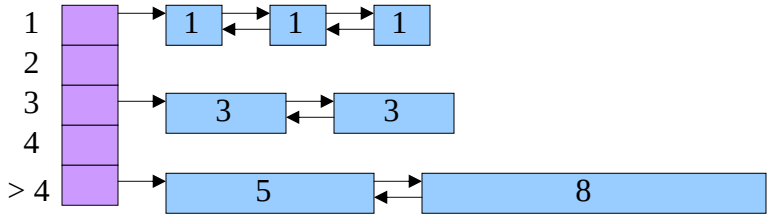
- Still need to scan the free list
  - To find the first, or best, block that fits
- Root of the problem
  - Free blocks have a wide range of sizes
- Solution: binning
  - Separate free lists by block size
  - Implemented as an array of free-list pointers



26

Binning Strategies: Exact Fit

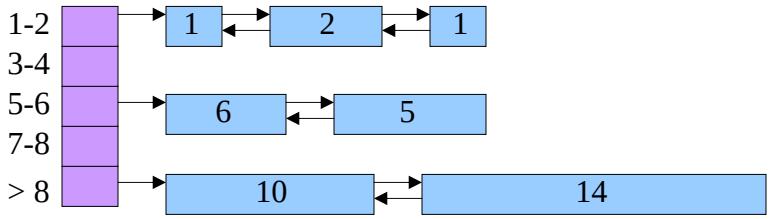
- Have a bin for each block size, up to a limit
  - Advantages: no search for requests up to that size
  - Disadvantages: many bins, each storing a pointer
- Except for a final bin for all larger free blocks
  - For allocating larger amounts of memory
  - For splitting to create smaller blocks, when needed



27

Binning Strategies: Range

- Have a bin cover a range of sizes, up to a limit
  - Advantages: fewer bins
  - Disadvantages: need to search for a big enough block
- Except for a final bin for all larger free chunks
  - For allocating larger amounts of memory
  - For splitting to create smaller blocks, when needed



28

Suggestions for Assignment #6

- Debugging memory management code is hard
  - A bug in your code might stomp on the headers or footers
  - … making it very hard to understand where you are in memory
- Suggestion: debug carefully as you go along
  - Write little bits of code at a time, and test as you go
  - Use assertion checks very liberally to catch mistakes early
  - Use functions to apply higher-level checks on your list
    - E.g,. all free-list blocks are marked as free
    - E.g., each block pointer is within the heap range
    - E.g., the block size in header and footer are the same
- Suggestion: draw lots and lots of pictures

29

Conclusions

- K&R **`malloc`** and **`free`** have limitations
  - Fragmentation of the free space
    - Due to the first-first strategy
  - Linear time for malloc and free
    - Due to the need to scan the free list
- Optimizations
  - Faster **`free`**
    - Headers and footers
    - Size information and doubly-linked free list
  - Faster **`malloc`**
    - Multiple free lists, one per size (or range of sizes)

30

Backup Slides

31

Stupid Programmer Tricks

- Inside the malloc library

```
if (size < 32)
  size = 32;
else if (size > 2048)
  size = 4096 * ((size+4095)/4096);
else if (size & (size-1)) {
  find next larger power-of-two
}
```

32

Stupid Programmer Tricks

- Inside the malloc library
- Why 4096?
  - Use mmap() instead of sbrk()
- Mmap (memory map) – originally intended to “map” a file into virtual address space
  - Often better than malloc+read. Why?
  - If no file specified, mapping becomes “anonymous” – temporary
  - Map/unmap at finer granularity (within reason)
  - Recycling – unmapped pages might get used by next sbrk()

33