



Optimizing Dynamic Memory Management

1



Goals of this Lecture

- Help you learn about:
 - Details of K&R heap mgr
 - Heap mgr optimizations related to Assignment #6
 - Faster `free()` via doubly-linked list, redundant sizes, and status bits
 - Faster `malloc()` via binning
 - Other heap mgr optimizations
 - Best/good fit block selection
 - Selective splitting
 - Deferred coalescing
 - Segregated data
 - Segregated meta-data
 - Memory mapping

2



Part 1: Details of the K&R Heap Manager

3

An Implementation Challenge



Problem:

- Need information about each free block
 - Starting address of the block of memory
 - Length of the free block
 - Pointer to the next block in the free list
- Where should this information be stored?
 - Number of free blocks is not known in advance
 - So, need to store the information on the *heap*
- But, wait, this code is what *manages* the heap!
 - Can't call `malloc()` to allocate storage for this information
 - Can't call `free()` to deallocate the storage, either

4

Store Information in the Free Block



Solution:

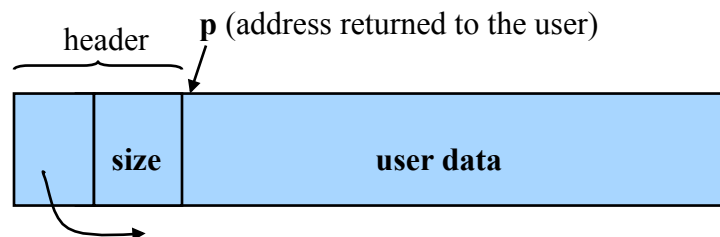
- Store the information directly in the block
 - Since the memory isn't being used for anything anyway
 - And allows data structure to grow and shrink as needed

5

Block Headers



- Every free block has a **header**, containing:
 - Pointer to (i.e., address of) the next free block
 - Size of the free block



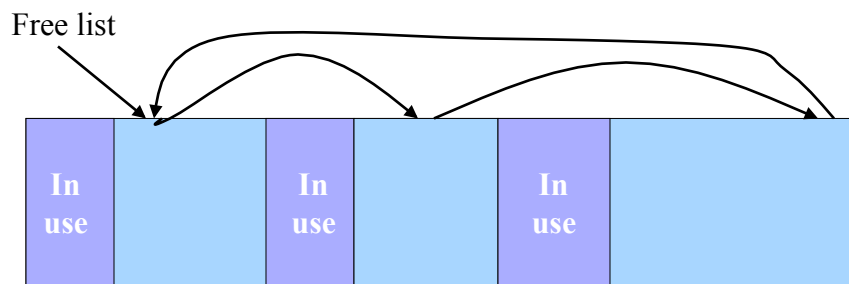
- Challenge: programming outside the type system

6

Free List: Circular Linked List



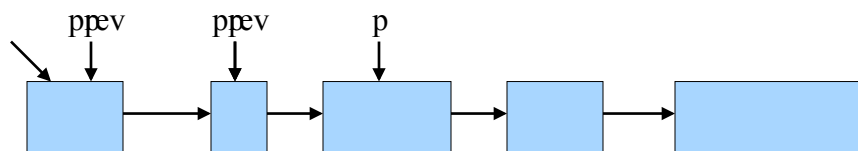
- Free blocks, linked together
 - Example: circular linked list
- Keep list in order of increasing addresses
 - Makes it easier to coalesce adjacent free blocks



Malloc: First-Fit Algorithm



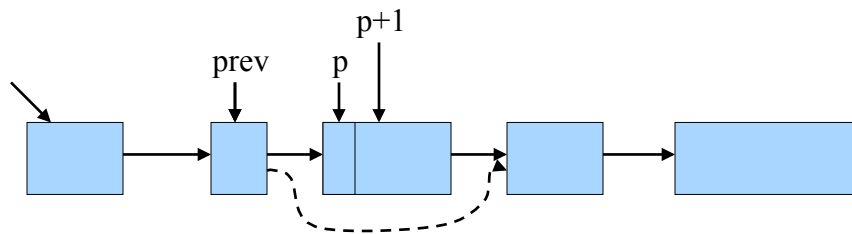
- Start at the beginning of the list
- Sequence through the list
 - Keep a pointer to the previous element
- Stop when reaching first block that is big enough
 - Patch up the list
 - Return a pointer to the user



Malloc: First Case: 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)

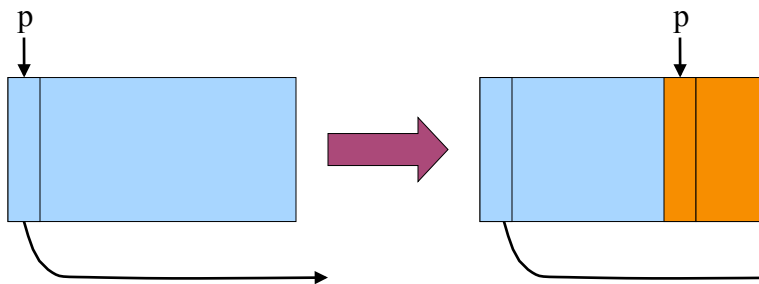


9

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
 - Bonus: No need to manipulate links

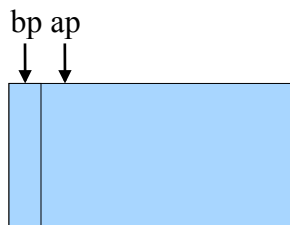


10

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

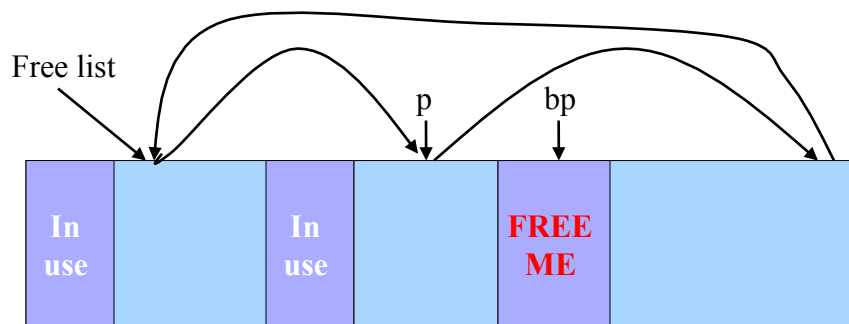


11

Free: Finding Location to Insert



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

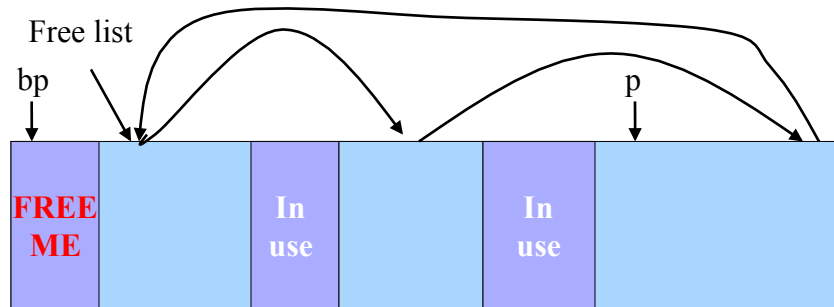


12

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

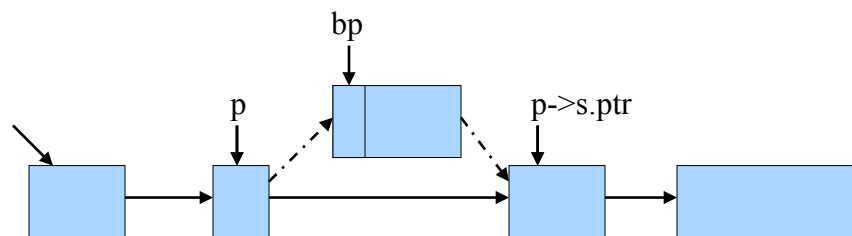


13

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

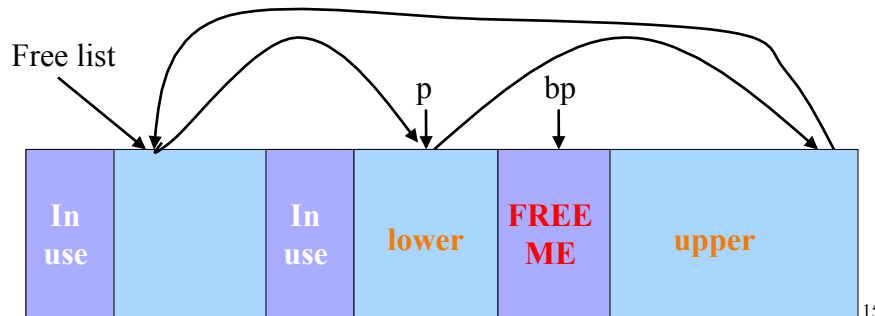


14

Coalescing With Neighbors



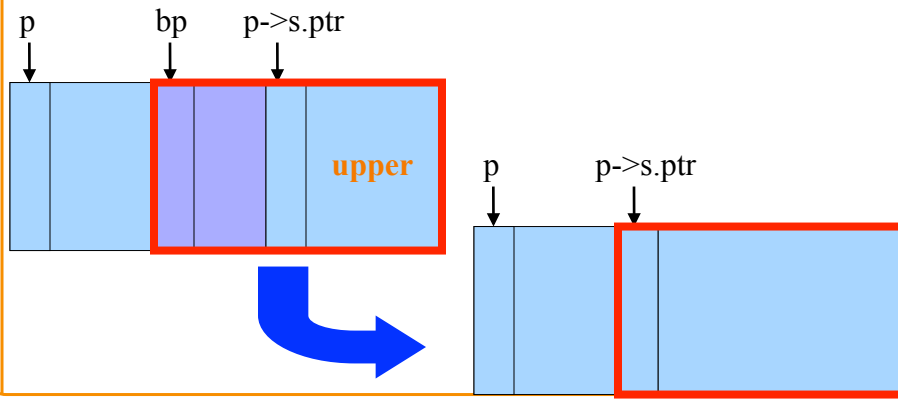
- Scanning the list finds the location for inserting
 - Pointer to to-be-freed element: bp
 - Pointer to previous element in free list: p
- Coalescing into larger free blocks
 - Check if contiguous to upper and lower neighbors



Coalesce With Upper Neighbor



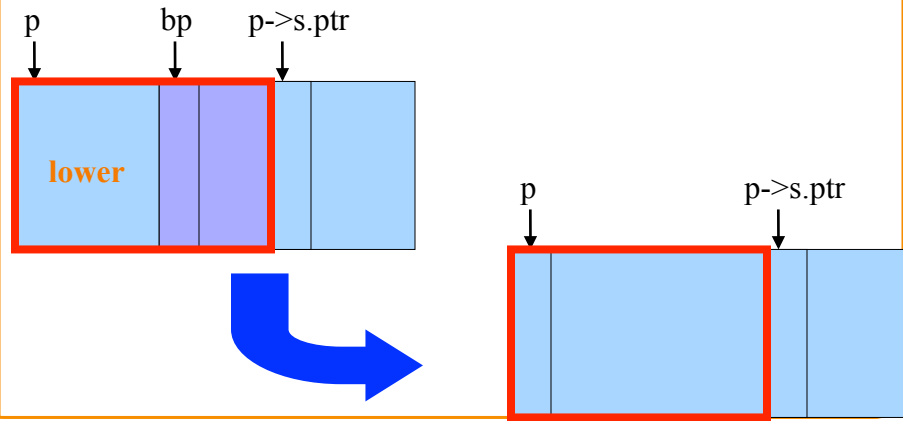
- Check if next part of memory is in the free list
- If so, make into one bigger block
- Else, simply point to the next free element



Coalesce With Lower Neighbor



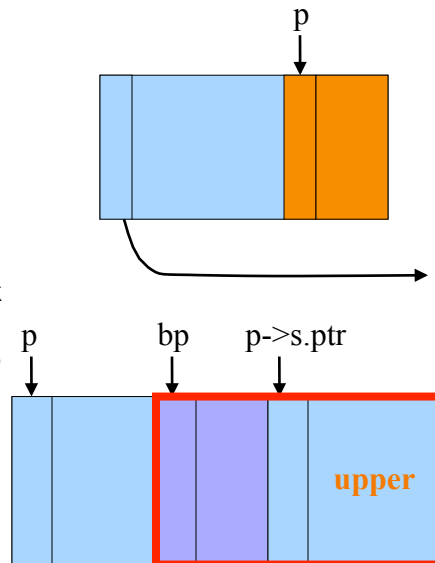
- Check if previous part of memory is in the free list
- If so, make into one bigger block



Strengths of K&R Approach



- 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

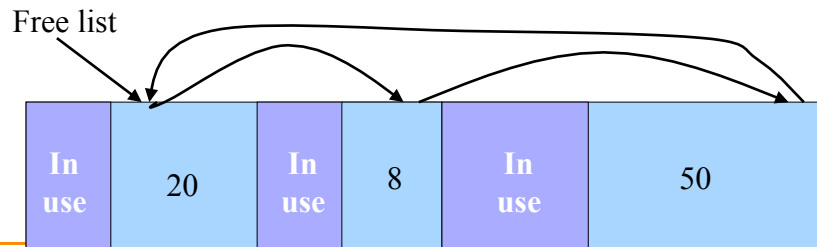


18

Weaknesses of K&R Approach



- **Inefficient use of memory: fragmentation**
 - First-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



Part 2: Optimizations Related to Assignment 6

Faster Free



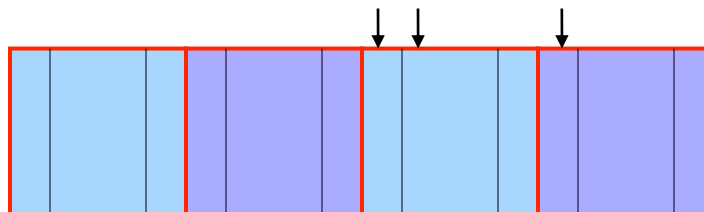
- Performance problems with K&R `free()`
 - Scanning the free list to know where to insert
 - Keeping track of the “previous” node to do the insertion
- Doubly-linked, non-circular list
 - Header
 - Size of the block (in # of units)
 - Flag indicating whether the block is free or in use
 - If free, a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

h		f
e		o
a		o
d		t

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

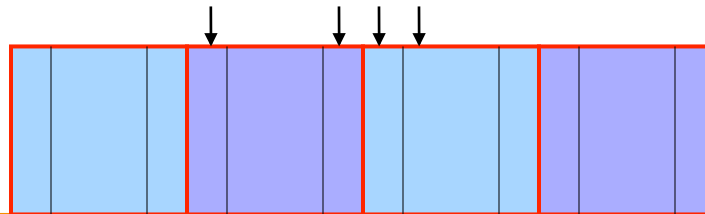


22

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

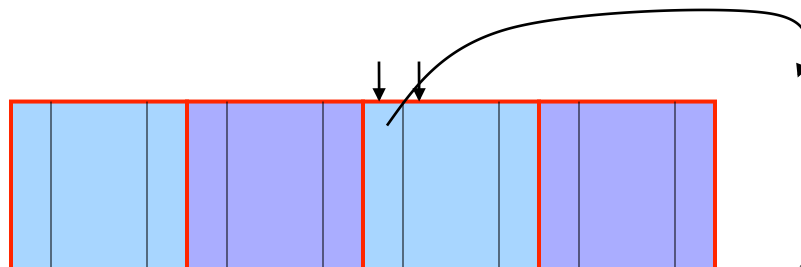


23

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

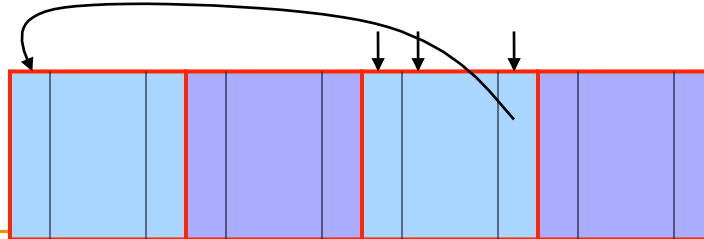


24

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



25

Efficient Free



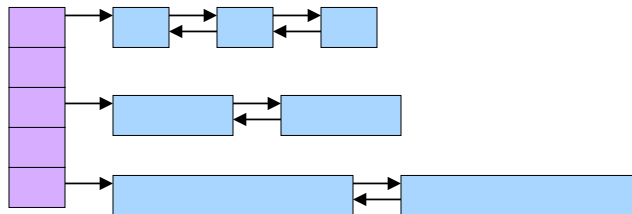
- Before: K&R
 - Scan the free list till you find the place to insert
 - Needed to see if you can coalesce adjacent blocks
 - Expensive for loop with several pointer comparisons
- After: with header/footer and doubly-linked list
 - Coalescing with the previous block in memory
 - Check if previous block 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

26

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

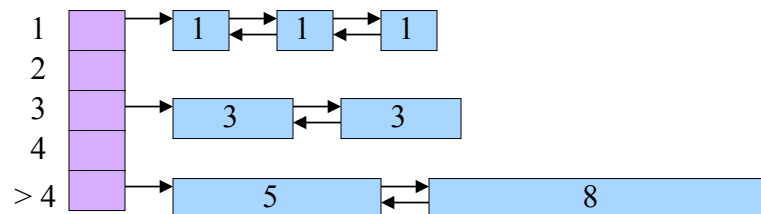


27

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

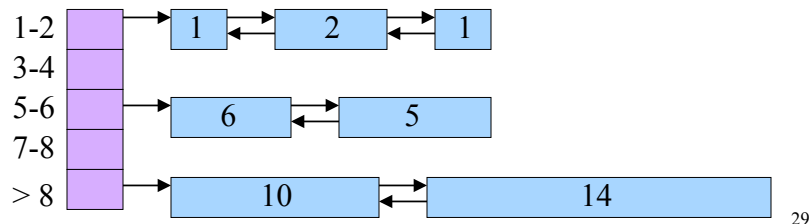


28

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



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

30



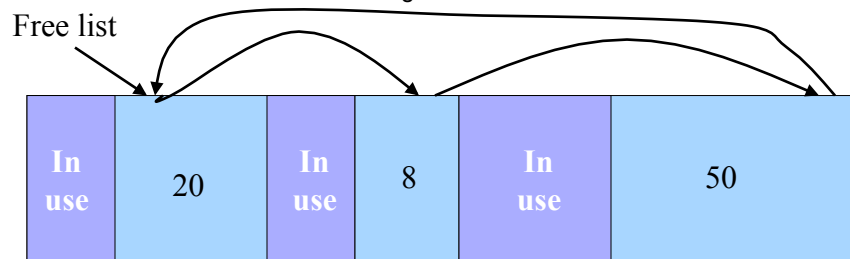
Part 3: Other Optimizations

31

Best/Good Fit Block Selection



- **Observation:**
 - K&R uses “first fit” (really, “next fit”) strategy
 - Example: `malloc(8)` would choose the 20-byte block
- **Alternative: “best fit” or “good fit” strategy**
 - Example: `malloc(8)` would choose the 8-byte block
 - Applicable if not binning, or if a bin has blocks of variable sizes
 - **Pro:** Minimizes internal fragmentation and splitting
 - **Con:** Increases cost of choosing free block

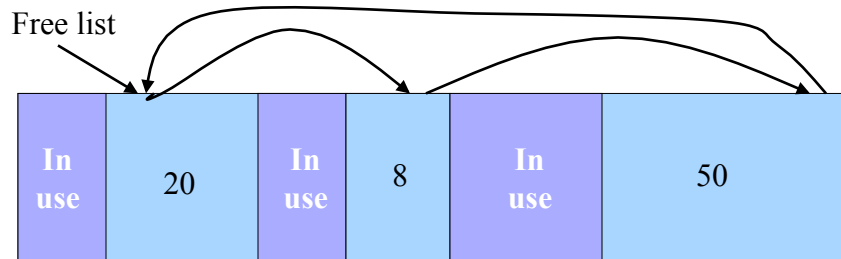


32

Selective Splitting



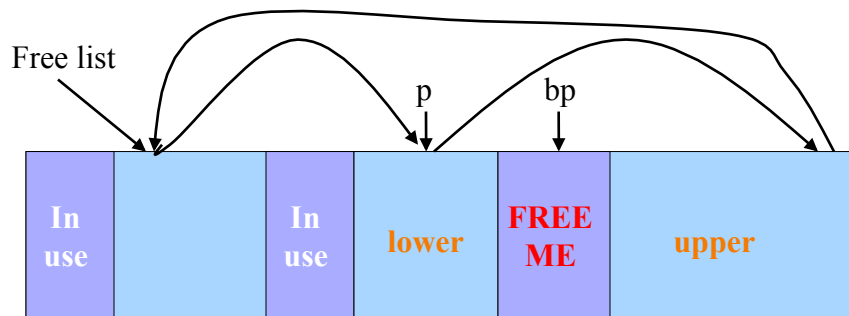
- **Observation:**
 - K&R `malloc()` splits whenever chosen block is too big
 - Example: `malloc(14)` splits the 20-byte block
- **Alternative: selective splitting**
 - Split only when the saving is big enough
 - Example: `malloc(14)` allocates the entire 20-byte block
 - **Pro:** Reduces external fragmentation
 - **Con:** Increases internal fragmentation



Deferred Coalescing



- **Observation:**
 - K&R does coalescing in `free()` whenever possible
- **Alternative: deferred coalescing**
 - Wait, and coalesce many blocks at a later time
 - **Pro:** Handles “`malloc(x); free(); malloc(x)`” sequences well
 - **Con:** Complicates algorithms



Segregated Data



- **Observation:**
 - Splitting and coalescing consume lots of overhead
- **Problem:**
 - How to eliminate that overhead?
- **Solution: Segregated data**
 - **Make use of the virtual memory concept...**
 - Store each bin's blocks in a distinct (segregated) virtual memory page
 - Elaboration...

35

Segregated Data (cont.)



- **Segregated data**
 - Each bin contains blocks of fixed sizes
 - E.g. 32, 64, 128, ...
 - All blocks within a bin are from same **virtual memory page**
 - Malloc never splits! Examples:
 - Malloc for 32 bytes => provide 32
 - Malloc for 5 bytes => provide 32
 - Malloc for 100 bytes => provide 128
 - Free never coalesces!
 - Free block => examine address, infer virtual memory page, infer bin, insert into that bin
 - **Pro:** Completely eliminates splitting and coalescing overhead
 - **Pro:** Eliminates most meta-data; only forward links are required (no backward links, sizes, status bits, footers)
 - **Con:** Some usage patterns cause excessive external fragmentation

36

Segregated Meta-Data



- **Observations:**
 - Meta-data (block sizes, status flags, links, etc.) are scattered across the heap, interspersed with user data
 - Heap mgr often must traverse meta-data
- **Problem 1:**
 - User error easily can corrupt meta-data
- **Problem 2:**
 - Frequent traversal of meta-data can cause excessive page faults
- **Solution: Segregated meta-data**
 - **Make use of the virtual memory concept...**
 - Store meta-data in a distinct (segregated) virtual memory page from user data

37

Memory Mapping



- **Observations:**
 - Heap mgr might want to release heap's physical memory to OS for unused virtual pages
 - Heap mgr can call `brk (currentBreak-x)` to release freed memory to OS, but...
 - Difficult to know when memory at high end of heap is free, and...
 - Often freed memory is not at high end of heap!
- **Problem:**
 - How can heap mgr effectively release freed memory to OS?
- **Solution: Memory mapping**
 - **Make use of virtual memory concept...**
 - Allocate memory via `mmap ()` system call
 - Free memory via `munmap ()` system call

38

mmap () and munmap ()



- Typical call of `mmap ()` call for allocating memory

```
p = mmap(NULL, size, PROT_READ|PROT_WRITE,  
MAP_PRIVATE|MAP_ANON, 0, 0);
```

- Asks the OS to map a new private read/write area of virtual memory containing `size` bytes
- Returns the virtual address of the new area on success, NULL on failure

- Typical call of `munmap ()`

```
status = munmap(p, size);
```

- Unmaps the area of virtual memory at virtual address `p` consisting of `size` bytes
- Returns 1 on success, 0 on failure

- See Bryant & O' Hallaron book and man pages for details

39

Using mmap () and munmap ()



Typical strategy:

- Allocate **small** block =>
 - Call `brk ()` if necessary
 - Manipulate data structures described earlier in this lecture
- Free **small** block =>
 - Manipulate data structures described earlier in this lecture
 - Do not call `brk ()`
- Allocate **large** block =>
 - Call `mmap ()`
- Free **large** block =>
 - Call `munmap ()`

40

Summary



- Details of K&R heap manager
- Heap mgr optimizations related to Assignment #6
 - Faster `free()` via doubly-linked list, redundant sizes, and status bits
 - Faster `malloc()` via binning
- Other heap mgr optimizations
 - Best/good fit block selection
 - Selective splitting
 - Deferred coalescing
 - Segregated data
 - Segregated meta-data
 - Memory mapping

41