Optimizing Malloc and Free

Prof. David August

COS 217

Reading: Section 8.7 in K&R book

http://gee.cs.oswego.edu/dl/html/malloc.html

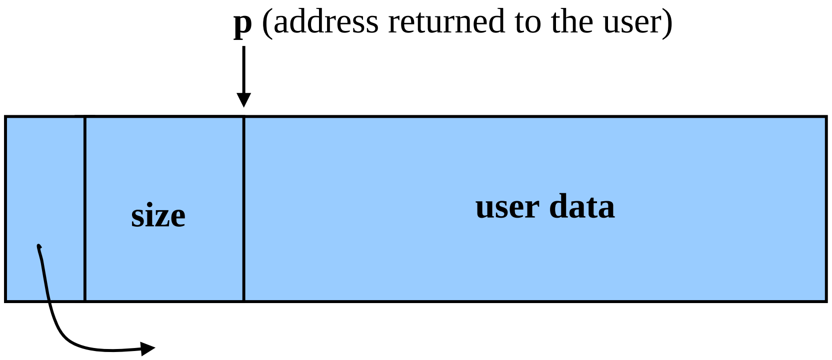
Goals of This Lecture

- Brief review of K&R implementation
  - Circular linked list of free chunks, with pointer and size in header
    - Malloc: first-fit algorithm, with splitting
    - Free: coalescing with adjacent chunks, 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 #4
  - 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 chunks of different sizes
    - One bin per chunk size, or one bin for a range of sizes

Free Chunk: Pointer, Size, Data

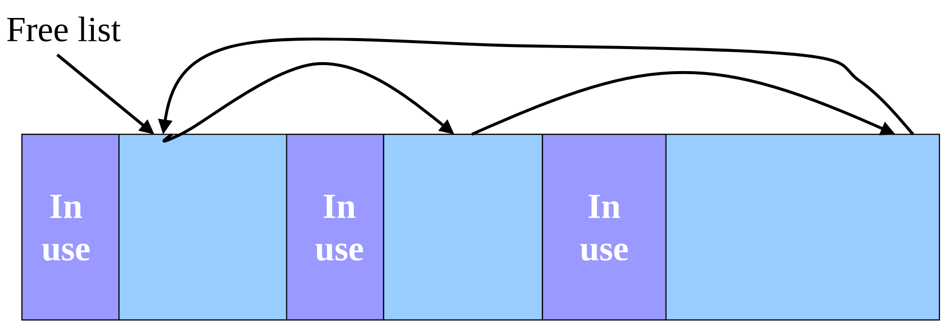
- Free chunk in memory
  - Pointer to the next chunk
  - Size of the chunk
  - User data

header



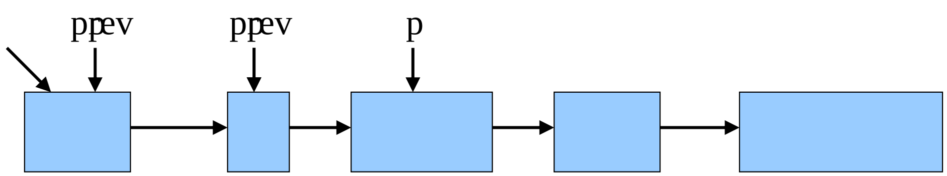
Free List: Circular Linked List

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



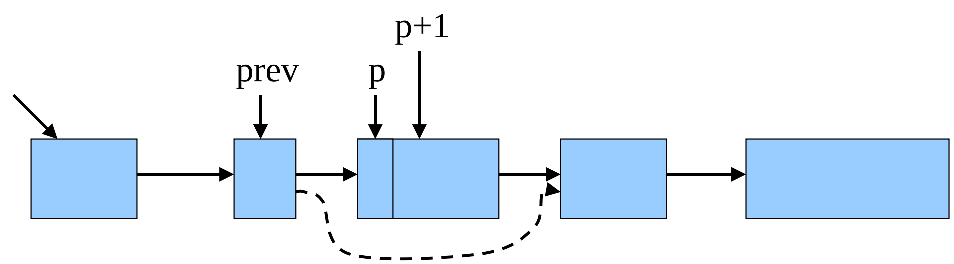
Malloc: First-Fit Algorithm

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



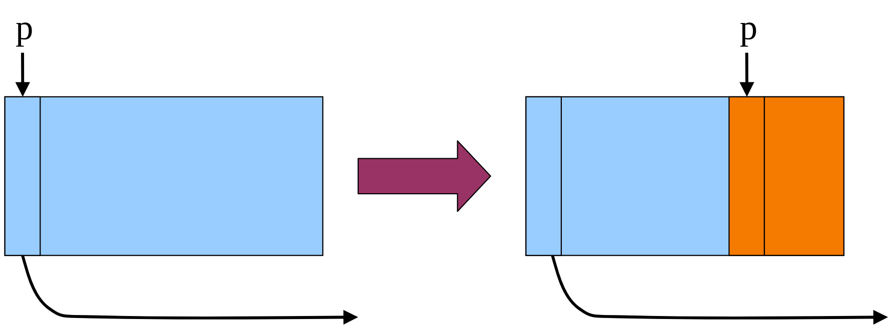
Malloc: First Case, A Perfect Fit

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



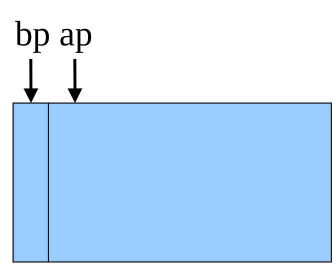
Malloc: Second Case: Big Chunk

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



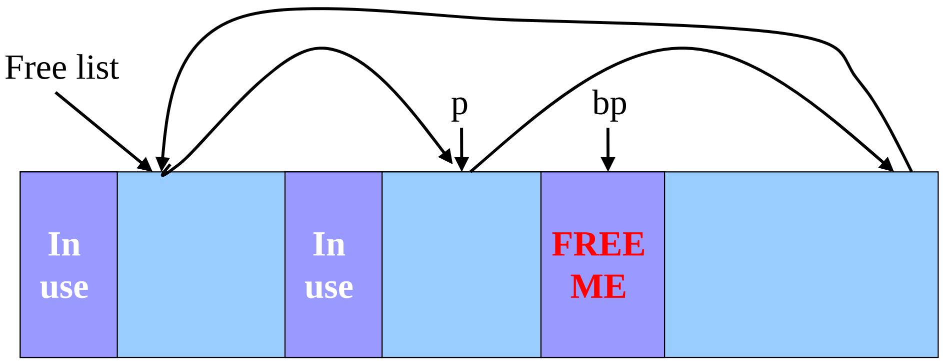
Free

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



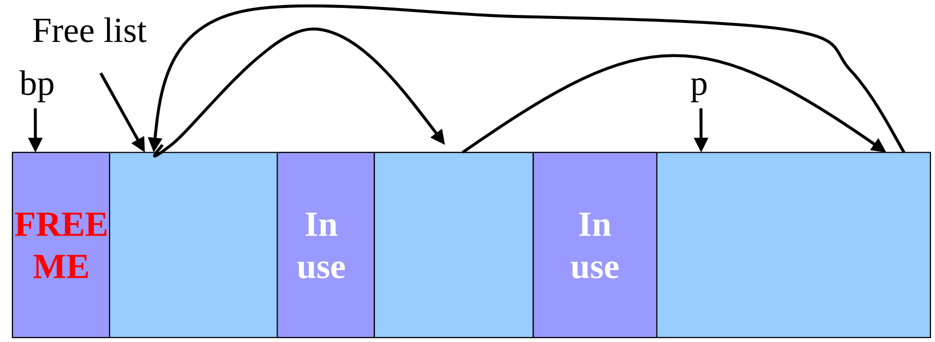
Free: Finding Location to Insert

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



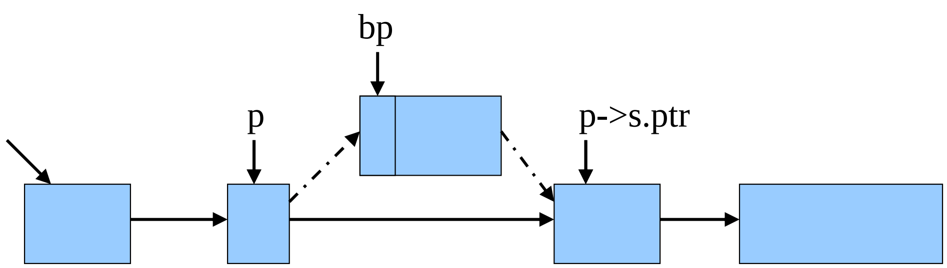
Free: Handling Corner Cases

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



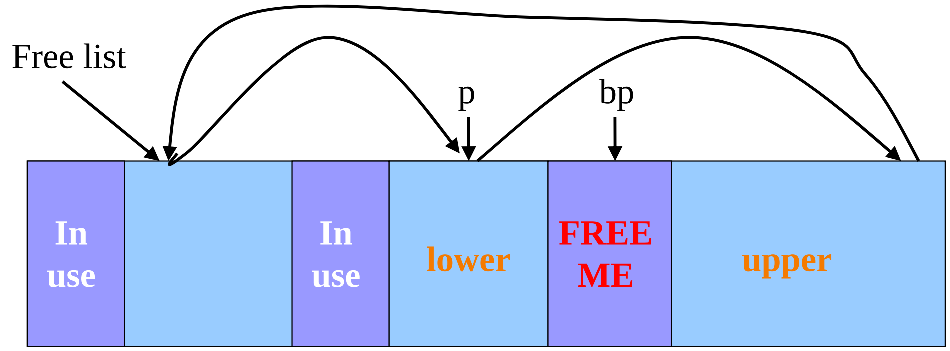
Free: Inserting Into Free List

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



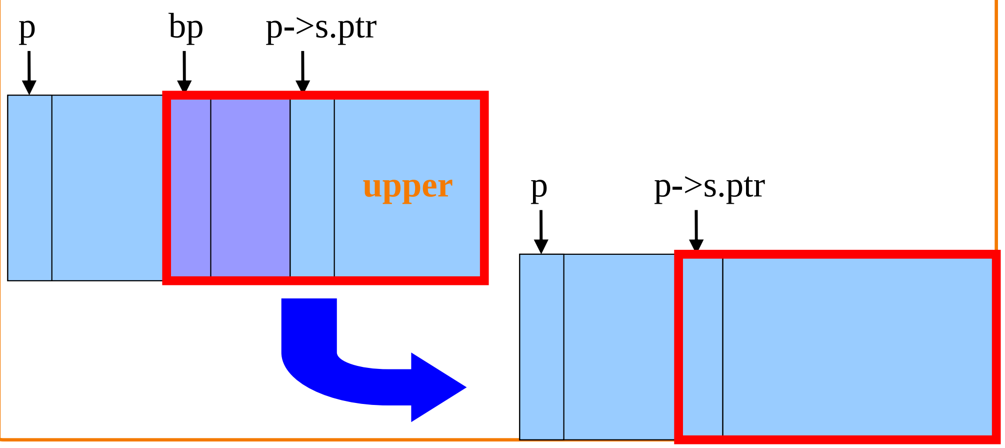
Coalescing With Neighbors

- 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 chunks
  - Check if contiguous to upper and lower neighbors



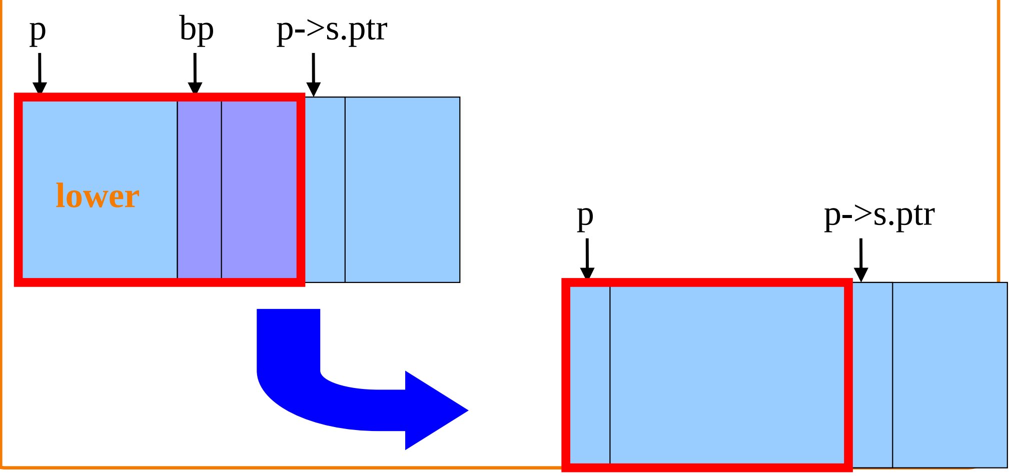
Coalesce With Upper Neighbor

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

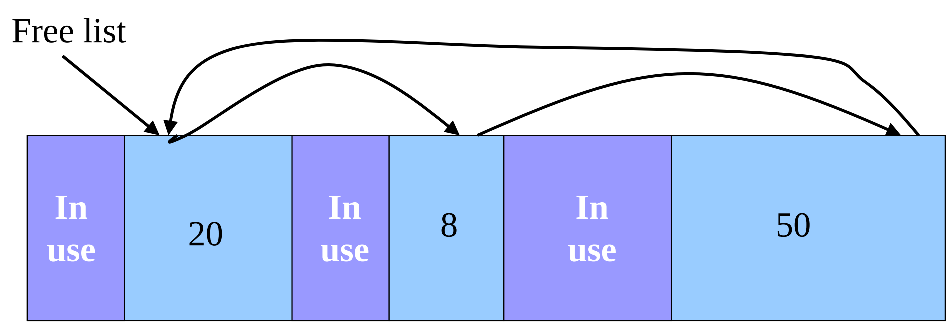


K&R Malloc and Free

- Advantages
  - Simplicity of the code
- Optimizations
  - Roving free-list pointer is left at the last place a chunk was allocated
  - Splitting large free chunks to avoid wasting space
  - Coalescing contiguous free chunks to reduce fragmentation
- Limitations
  - Inefficient use of memory: fragmentation
    - Best-fit policy can leave lots of “holes” of free chunks in memory
  - Long execution times: linear-time overhead
    - Malloc scans the free list to find a big-enough chunk
    - Free scans the free list to find where to insert a chunk

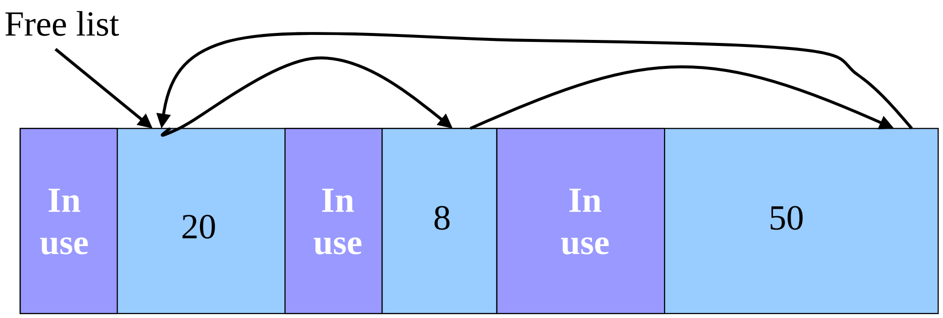
Improvements: Placement

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



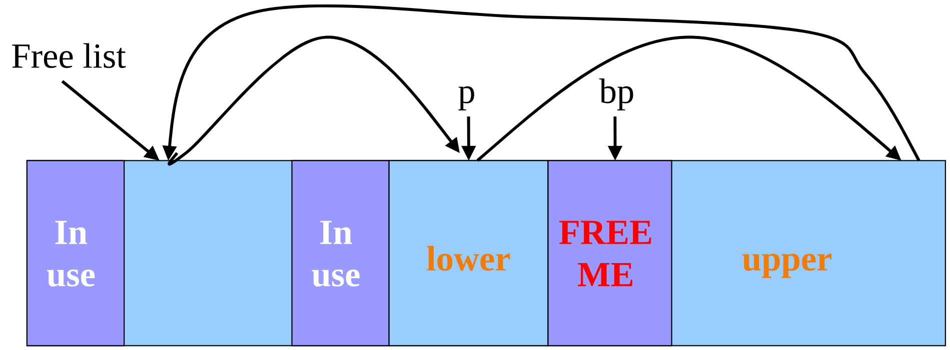
Improvements: Splitting

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



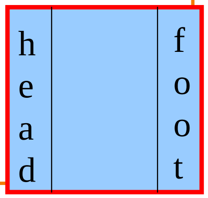
Improvements: Coalescing

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



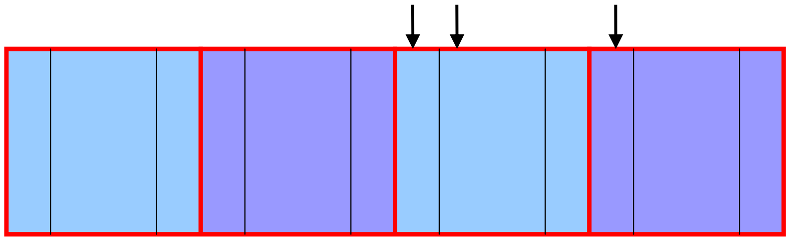
Improvements: Faster Free

- 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 chunk (in # of units)
    - Flag indicating whether the chunk is free or in use
    - If free, a pointer to the next free chunk
  - Footer in all chunks
    - Size of the chunk (in # of units)
    - If free, a pointer to the previous free chunk



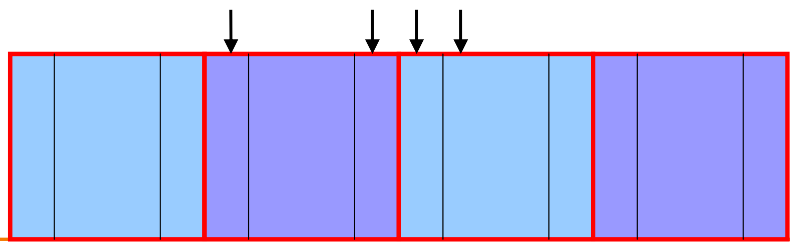
Size: Finding Next Chunk

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



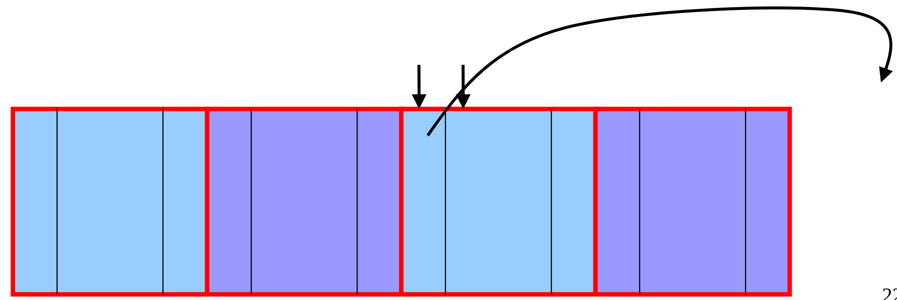
Size: Finding Previous Chunk

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



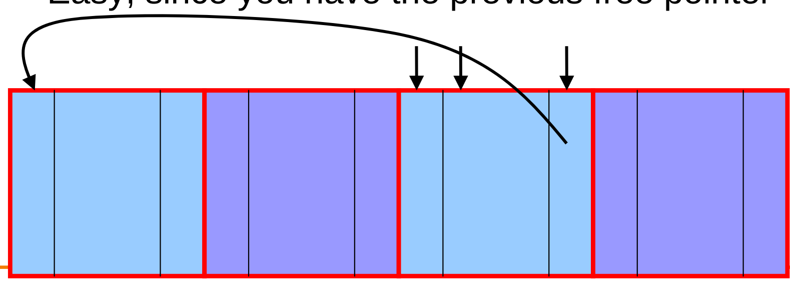
Pointers: Next Free Chunk

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



Pointers: Previous Free Chunk

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

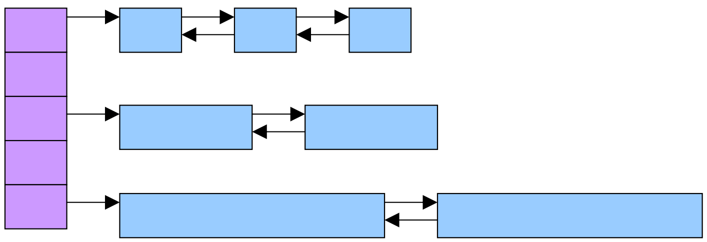


Efficient Free

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

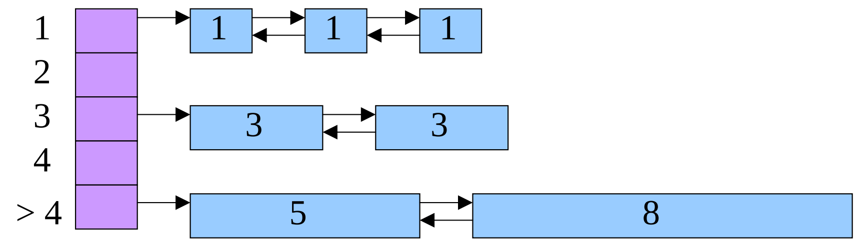
But Malloc is Still Slow…

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



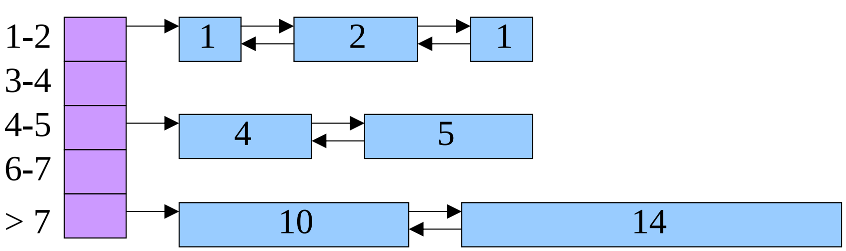
Binning Strategies: Exact Fit

- Have a bin for each chunk 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 chunks
  - For allocating larger amounts of memory
  - For splitting to create smaller chunks, when needed



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 chunk
- Except for a final bin for all larger free chunks
  - For allocating larger amounts of memory
  - For splitting to create smaller chunks, when needed



Stupid Programmer Tricks

- Reducing small allocs, especially strings

```
typedef struct Entry {
 struct Entry *e_next;
 int e_count;
 char e_string[1];
} Entry;
```

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
}
```


Stupid Programmer Tricks

- Defeating your malloc library

```
typedef struct MyData {
  struct MyData *md_nextFree;
  …
} MyData;
MyData *mdFreePtr;
void MyData_Free(MyData *ent) {ent->md_nextFree = mdFreePtr;
  mdFreePtr = ent;}
MyData *MyData_Alloc(void) {
  if (mdFreePtr != NULL)
    manipulate list, return first item
  else
    allocate array of items, add all to free list
}
```


Suggestions for Assignment #4

- 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 elements are marked as free
    - E.g., each chunk pointer is within the heap range
    - E.g., the chunk size in header and footer are the same
- Suggestion: working in pairs
  - Think (and discuss) how to collaborate together
- Suggestion: draw lots and lots of pictures

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)