Dynamic Memory Allocation: Explicit Free Lists

COMP 321

Dave Johnson

1

Why an Explicit Free List?

Problems with implicit free lists
- 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
- *The fundamental problem*
  - The heap is the free list is the heap ...
  - This includes all *allocated* blocks as well as all *free* blocks

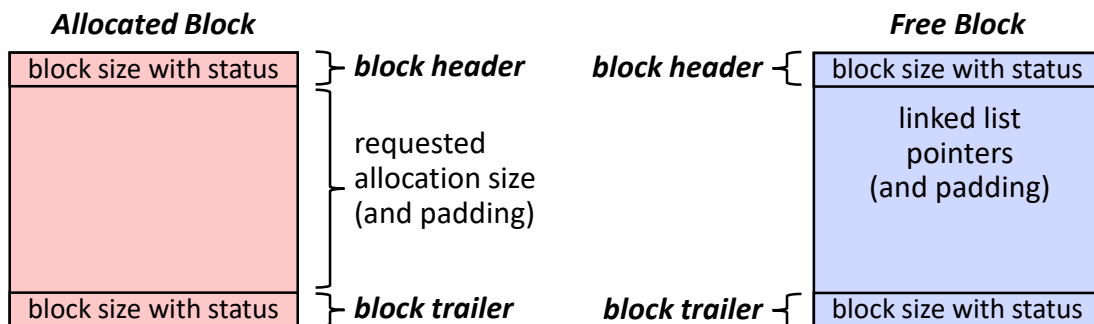*Creating and using an explicit free list lets us overcome this*

2

1

Basic Idea of an Explicit Free List?

The free list should contain only free blocks
- All blocks still contain header and trailer with size and allocation status
- No change to the format of an allocated block
- Each free block now also needs to contain pointer(s) to place it in the free list

Allocated Block

| block size with status | }— *block header* |

requested
allocation size
(and padding)

| block size with status | }— *block trailer* |

Free Block

block header —{ | block size with status |

linked list
pointers
(and padding)

block trailer —{ | block size with status |
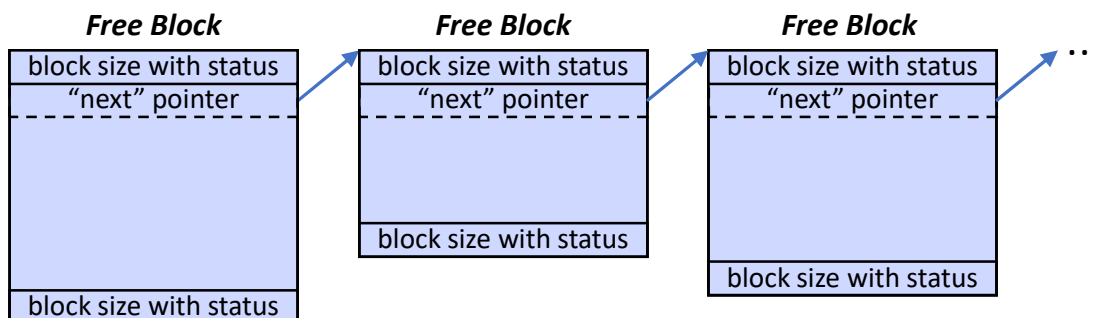
3

The Free Space in a Free Block <u>Stores</u> the Pointer(s)

The linked list pointer(s) take up no space to store
- A ***free*** block is by definition ***free*** – currently unused space
- Borrow some of that free space to store the pointer(s) for the free list

A simple example

Free Block

| block size with status |
| "next" pointer |

Free Block

| block size with status |
| "next" pointer |

| block size with status |

Free Block

| block size with status |
| "next" pointer |

| block size with status |

...

| block size with status |

4

The Minimum Size of a Block

An <u>allocated</u> block must be big enough to represent itself when it is a <u>free</u> block

- Meaning it must have room as a free block to hold the list pointer(s)
    - That extra room is not needed when it is an allocated block
- The minimum free block that can be represented contains
    - The free block's header
    - The free block's trailer
    - The free block's free list pointer(s)
- When allocating a block, must add extra padding inside the allocated block (in addition to padding for alignment) to make sure the allocated block is at least this minimum size
    - (May increase internal fragmentation)

5

The Order of the Free List

Many list ordering choices are possible, including

- FIFO – a freed block won't be reused until much later

- LIFO – a freed block will be reused on next allocation (depending on size)

- Ordered by addresses

- Ordered by size
    - ascending (increasing size) order
    - descending (decreasing size) order

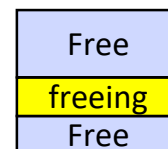***The order of free blocks in the free list need not be the order in the heap***

6

The Structure of the Free List

Many different list structures are possible for a "linked list"

- Singly linked or doubly linked
- Circular or not circular

Doubly linked circular is a good choice

- Can add or remove in constant time, regardless of the position in the list
- Example: for coalescing when freeing a block
  - May be able to merge yourself onto the previous block in place without removing it from the free list
  - But may need to remove following block from the free list, and you didn't find it by traversing the list

Free
freeing
Free

7

Segregated Free Lists

A remaining problem with our free list

- Allocation is still linear in the number of free blocks
  - Much better than linear in number of free or allocated blocks
  - But the search for a suitable free block can still be expensive

A solution: segregated free lists, with multiple explicit free lists

- A different free list for each different free block size (or range of block sizes), referred to as a ***size class*** (also called ***bins***)
- Makes finding a suitable block much faster (or even constant time)
  - First fit is now almost the same as best fit
- Small additional complexity (when splitting while allocating, or when coalescing while freeing) to move to correct new free list

8

4

Segregated Free Lists

Example: different size classes growing with powers of 2
- Size classes [1], [2], (2, 4], (4, 8], (8, 16], (16, 32], (32, 64], ..., (4096, ∞)
- All very large sizes are in the same (largest) size class

Example: different size classes for individual small sizes, then powers of 2
- Size classes [1], [2], [3], [4], [5], [6], ..., (1024, 2048], ..., (4096, ∞)
- All very large sizes are in the same (largest) size class

Some details
- What are the units here? Multiples of largest granularity or smallest block size
- What if the "correct" free list is empty? Can just look in the next larger class

9

Other Possibilities are Almost Endless

Once you have an explicit list, you can do other interesting forms of it

Example: a balanced binary tree free list rather than just a plain list
- Could, e.g., store in the space *inside* each *free* block

```
struct free_info {
        struct free_info *parent;    // pointer to parent in the tree
        struct free_info *left;      // subtree for smaller free blocks
        struct free_info *right;     // subtree for larger free blocks
        struct free_info *next;      // next free block of the same size
};
```

- Makes the minimum block size requirement be a bit larger
- With a flexible way to find the segregated list of the right size free blocks

10