The Hardware/Software Interface
CSE351 Winter 2013

Memory Allocation II

Knowing How Much to Free

- **Standard method**
  - Keep the length of a block in the word preceding the block
  - This word is often called the *header field* or *header*
  - Requires an extra word for every allocated block

```
<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
p0 = malloc(4)
```

```
<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
free(p0)
```

- How do we know how much memory to free given just a pointer?
- How do we keep track of the free blocks?
- How do we pick a block to use for allocation (when many might fit)?
- What do we do with the extra space when allocating a structure that is smaller than the free block it is placed in?
- How do we reinsert freed block into the heap?

Keeping Track of Free Blocks

- **Method 1: Implicit list** using length—links all blocks

```
<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
```

- **Method 2: Explicit list** among the free blocks using pointers

```
<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
```

- **Method 3: Segregated free list**
  - Different free lists for different size classes

- **Method 4: Blocks sorted by size**
  - Can use a balanced binary tree (e.g. red-black tree) with pointers within each free block, and the length used as a key
Implicit Free Lists

- For each block we need: size, is-allocated?
  - Could store this information in two words: wasteful!
- Standard trick
  - If blocks are aligned, some low-order size bits are always 0
  - Instead of storing an always-0 bit, use it as a allocated/free flag
  - When reading size, must remember to mask out this bit

Format of allocated and free blocks

- 1 word
- size
- a
- payload
- optional padding

E.g. with 8-byte alignment, sizes look like:

- 00000000
- 00001000
- 00001010
- 00001000

Implicit List: Finding a Free Block

- First fit:
  - Search list from beginning, choose first free block that fits:
    - *p gets the block header
    - *p & 1 extracts the allocated bit
    - *p & -2 masks the allocated bit, gets just the size
  - Can take time linear in total number of blocks (allocated and free)
  - In practice it can cause "splinters" at beginning of list
- Next fit:
  - Like first-fit, but search list starting where previous search finished
  - Should often be faster than first-fit: avoids re-scanning unhelpful blocks
  - Some research suggests that fragmentation is worse
- Best fit:
  - Search the list, choose the best free block: fits, with fewest bytes left over
  - Keeps fragments small—usually helps fragmentation
  - Will typically run slower than first-fit

Implicit List: Allocating in Free Block

- Allocating in a free block: splitting
  - Since allocated space might be smaller than free space, we might want to split the block

```c
void addblock(ptr p, int len) {
    int newsize = ((len + 1) >> 1) << 1; // round up to even
    int oldsize = *p & -2; // mask out low bit
    *p = newsize | 1; // set new length + allocated
    if (newsize < oldsize)
        *(p+newsize) = oldsize - newsize; // set length in remaining
}
```
Implicit List: Freeing a Block

- Simplest implementation:
  - Need only clear the "allocated" flag
    
    ```
    void free_block(ptr p) { *p = *p & -2 }
    ```
  - But can lead to "false fragmentation"

```
void free_block(ptr p) {
    *p = *p & -2;  // clear allocated bit
    next = p + *p; // find next block
    if ((*next & 1) == 0)  //    // not allocated
        *p = *p + *next; // add to this block if
} //    //    not allocated
```

```
<table>
<thead>
<tr>
<th>4</th>
<th>4</th>
<th>4</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>4</th>
<th>4</th>
<th>4</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

There is enough free space, but the allocator won't be able to find it

Implicit List: Coalescing

- Join (coalesce) with next/previous blocks, if they are free
  - Coalescing with next block

```
void free_block(ptr p) {
    *p = *p & -2;  // clear allocated bit
    next = p + *p; // find next block
    if ((*next & 1) == 0)  //    // not allocated
        *p = *p + *next; // add to this block if
} //    //    not allocated
```

```
<table>
<thead>
<tr>
<th>4</th>
<th>4</th>
<th>4</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>4</th>
<th>4</th>
<th>4</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

But how do we coalesce with the previous block?

Implicit List: Bidirectional Coalescing

- Boundary tags [Knuth73]
  - Replicate size/allocated word at "bottom" (end) of free blocks
  - Allows us to traverse the "list" backwards, but requires extra space
  - Important and general technique!

```
|4|4|4|4|6|4
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
|4|4|4|4|6|4
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Format of allocated and free blocks

```
<table>
<thead>
<tr>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>payload and padding</th>
</tr>
</thead>
</table>

Boundary tag (footer)

```
<table>
<thead>
<tr>
<th>size</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>payload: application data (allocated blocks only)</td>
<td></td>
</tr>
</tbody>
</table>
```

Constant Time Coalescing

<table>
<thead>
<tr>
<th>Case 1</th>
<th>Case 2</th>
<th>Case 3</th>
<th>Case 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>allocated</td>
<td>allocated</td>
<td>free</td>
<td>free</td>
</tr>
<tr>
<td>allocated</td>
<td>free</td>
<td>allocated</td>
<td>free</td>
</tr>
<tr>
<td>block being freed</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Case 1

```
<table>
<thead>
<tr>
<th>4</th>
<th>4</th>
<th>4</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>4</th>
<th>4</th>
<th>4</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Case 2

```
<table>
<thead>
<tr>
<th>4</th>
<th>4</th>
<th>4</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>4</th>
<th>4</th>
<th>4</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Case 3

```
<table>
<thead>
<tr>
<th>4</th>
<th>4</th>
<th>4</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>4</th>
<th>4</th>
<th>4</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Case 4

```
<table>
<thead>
<tr>
<th>4</th>
<th>4</th>
<th>4</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>4</th>
<th>4</th>
<th>4</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Constant Time Coalescing

Explicit Free Lists

- Allocating cost:
  - linear time (in total number of heap blocks) worst case
- Free cost:
  - constant time worst case
  - even with coalescing
- Memory utilization:
  - will depend on placement policy
  - First-fit, next-fit or best-fit
- Not used in practice for `malloc()`/`free()` because of linear-time allocation
  - used in some special purpose applications
- The concepts of splitting and boundary tag coalescing are general to all allocators

Keeping Track of Free Blocks

- Method 1: **Implicit free list** using length—links all blocks
  
- Method 2: **Explicit free list** among the free blocks using pointers

- Method 3: **Segregated free list**
  - Different free lists for different size classes

- Method 4: **Blocks sorted by size**
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key
Explicit Free Lists

- Logically (doubly-linked lists):
  
  ![Diagram of doubly-linked lists]

- Physically: blocks can be in any order
  
  ![Diagram of physically ordered blocks]

Alloca7ng From Explicit Free Lists

Before: conceptual graphic

After: (with splitting)

= malloc(…)

Freeing With Explicit Free Lists

- **Insertion policy:** Where in the free list do you put a newly freed block?
  - LIFO (last-in-first-out) policy
    - Insert freed block at the beginning of the free list
    - **Pro:** simple and constant time
    - **Con:** studies suggest fragmentation is worse than address ordered
  
  - Address-ordered policy
    - Insert freed blocks so that free list blocks are always in address order:
      
      \[
      \text{addr(prev)} < \text{addr(curr)} < \text{addr(next)}
      \]
    - **Con:** requires linear-time search when blocks are freed
    - **Pro:** studies suggest fragmentation is lower than LIFO

Freeing With a LIFO Policy (Case 1)

Before: conceptual graphic

After:

- **Insert the freed block at the root of the list**

Freeing in an Empty List

- Insert freed block as root

LIFO Policy (Case 2)

Before: conceptual graphic

After:

- **Insert the freed block at the root of the list**
Freeing With a LIFO Policy (Case 2)

- Splice out predecessor block, coalesce both memory blocks, and insert the new block at the root of the list

Freeing With a LIFO Policy (Case 3)

- Splice out successor block, coalesce both memory blocks and insert the new block at the root of the list

Freeing With a LIFO Policy (Case 4)

- Splice out predecessor and successor blocks, coalesce all 3 memory blocks and insert the new block at the root of the list

Explicit List Summary

- Comparison to implicit list:
  - Allocate is linear time in number of free blocks instead of all blocks
    - Much faster when most of the memory is full
    - Slightly more complicated allocate and free since needs to splice blocks in and out of the list
    - Some extra space for the links (2 extra words needed for each block)
    - Possibly increases minimum block size, leading to more internal fragmentation

- Most common use of explicit lists is in conjunction with segregated free lists
  - Keep multiple linked lists of different size classes, or possibly for different types of objects
Keeping Track of Free Blocks

- **Method 1:** *Implicit list* using length—links all blocks

- **Method 2:** *Explicit list* among the free blocks using pointers

- **Method 3:** *Segregated free list*
  - Different free lists for different size classes

- **Method 4:** *Blocks sorted by size*
  - Can use a balanced tree (e.g., Red-Black tree) with pointers within each free block, and the length used as a key

Segregated List (Seglist) Allocators

- Each *size class* of blocks has its own free list

- **Seglist Allocator**
  - Given an array of free lists, each one for some size class

  To allocate a block of size $n$:
  - Search appropriate free list for block of size $m > n$
  - If an appropriate block is found:
    - Split block and place fragment on appropriate list (optional)
  - If no block is found, try next larger class
  - Repeat until block is found

  If no block is found:
  - Request additional heap memory from OS (using `sbrk()`)  
  - Allocate block of $n$ bytes from this new memory
  - Place remainder as a single free block in largest size class

  **Advantages of seglist allocators**
  - Higher throughput
    - log time for power-of-two size classes
  - Better memory utilization
    - First-fit search of segregated free list approximates a best-fit search of entire heap.
    - Extreme case: Giving each block its own size class is equivalent to best-fit.
Summary of Key Allocator Policies

- **Placement policy:**
  - First-fit, next-fit, best-fit, etc.
  - Trades off lower throughput for less fragmentation
  - **Observation:** segregated free lists approximate a best fit placement policy without having to search entire free list

- **Splitting policy:**
  - When do we go ahead and split free blocks?
  - How much internal fragmentation are we willing to tolerate?

- **Coalescing policy:**
  - **Immediate coalescing:** coalesce each time `free()` is called
  - **Deferred coalescing:** try to improve performance of `free()` by deferring coalescing until needed. Examples:
    - Coalesce as you scan the free list for `malloc()`
    - Coalesce when the amount of external fragmentation reaches some threshold

More Info on Allocators

  - The classic reference on dynamic storage allocation

  - Comprehensive survey
  - Available from CS:APP student site (csapp.cs.cmu.edu)