Implementation Issues

- 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?

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
Keeping Track of Free Blocks

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

```
20  16  24  8
```

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

```
20  16  24  8
```

- **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

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

`a = 1: allocated block
a = 0: free block
size: block size
payload: application data
allocated blocks only`

Implicit Free List Example (32-bit)

Sequence of blocks in heap (size | allocated): 8 | 0, 16 | 1, 32 | 0, 16 | 1

```
Start of heap

8 bytes = 2 word alignment
```

- **8-byte alignment**
  - May require initial unused word
  - Causes some internal fragmentation

- **Special one-word marker (0 | 1) marks end of list**
  - Zero size is distinguishable from all real sizes

Implicit List: Finding a Free Block

- **First fit:**
  - Search list from beginning, choose *first* free block that fits:

```
p = heap_start;
while ((p < end) && ((p & 1) || (*p <= len)) { // not past end
  (*p & -2); // already allocated
  p = p + (*p & -2); // too small
} go to next block (UNSCALED +)
```
### 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

\[
\text{void split(ptr } b, \text{ int } \text{bytes}) \{
\text{ // bytes = desired block size}
\text{ int newsize = ((bytes + 7) >> 3) << 3;} \quad // \text{ round up to multiple of 8}
\text{ *b = newsize;} \quad // \text{ why not mask out low bit?}
\text{ if (newsize < oldsize)} \quad // \text{ initially unallocated}
\text{ *b = oldsize - newsize;} \quad // \text{ set length in remaining}
\text{ } \quad // \text{ part of block (UNSCALED +)}
\}
\]

\[
\text{malloc(12) } \rightarrow \text{ split(b, 16)}
\]

#### Implicit List: Coalescing

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

\[
\text{void free(ptr } p) \{
\text{ // p points to data}
\text{ ptr b = p - WORD;} \quad // \text{ b points to block}
\text{ *b = *b & -2;} \quad // \text{ clear allocated bit}
\text{ ptr next = b + *b;} \quad // \text{ find next block (UNSCALED +)}
\text{ if (**next & 1) == 0} \quad // \text{ not allocated}
\text{ *b = *b + *next;} \quad // \text{ add to this block if}
\text{ } \quad // \text{ logically gone}
\}
\]

- But how do we coalesce with the previous block?
**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>
</tbody>
</table>

block being freed

---

**Implicit Free Lists: Summary**

- **Implementation:** very simple
- **Allocate 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

  ![Diagram of Implicit Free List](image)

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

  ![Diagram of Explicit Free List](image)

- **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

- **Allocated block:**
  - size: \( a \)
  - payload and padding

- **Free block:**
  - size: \( a \)
  - next
  - prev

(same as implicit free list)

- Maintain list(s) of *free* blocks, rather than implicit list of *all* blocks
  - The “next” free block could be anywhere in the heap
    - So we need to store forward/back pointers, not just sizes
  - Luckily we track only free blocks, so we can use payload area for pointers
  - Still need boundary tags for coalescing

Explicit Free Lists

- **Logically (doubly-linked lists):**

- **Physically:** blocks can be in any order

Allocating From Explicit Free Lists

**Before**

- Conceptual graphic
Allocating From Explicit Free Lists

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

- Cache effects?
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.

Do we always need the boundary tag?

- Lab 5 suggests no...
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

  ![Diagram 1](image1.png)

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

  ![Diagram 2](image2.png)

- **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

  ![Diagram 3](image3.png)

- Often have separate classes for each small size

- For larger sizes: One class for each two-power size

Seglist Allocator

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

  ![Diagram 4](image4.png)

  **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
Seglist Allocator

- To free a block:
  - Coalesce and place on appropriate list (optional)

- 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)

Administrivia

- HW 3 solutions are posted
  - Link on the HW 3 page

- Optional ungraded memory allocation HW-style problems
  - On lab 5 page
  - Solutions available this Friday