Dynamic Memory Allocation

Dynamic memory allocators manage the heap.

- heap managed as a collection of variable-sized blocks, either allocated or free
  - OS manages pages
  - Application objects typically smaller than pages

**Manual (explicit)**
- C: `malloc`, `free`

**Automatic (implicit)**
- Java, ML, Python, JS, etc.: garbage collection

Assumptions

Block are word-aligned

```
<table>
<thead>
<tr>
<th>Allocated block</th>
<th>Free block</th>
</tr>
</thead>
<tbody>
<tr>
<td>(4 words)</td>
<td>(3 words)</td>
</tr>
</tbody>
</table>
```

**void** `malloc(size_t size);`:
- size: number of contiguous bytes required
- return: pointer to a newly allocated block of at least that size

**void** `free(void* ptr);`:
- ptr: pointer to allocated block to free

Allocation Example (32-bit words)

```
p1 = malloc(16)
p2 = malloc(20)
p3 = malloc(24)
free(p2)
p4 = malloc(8)
```

What about fragmentation? Unused memory that cannot be allocated.

Internal Fragmentation

*internal fragmentation* occurs if payload is smaller than block size

```
<table>
<thead>
<tr>
<th>Internal fragmentation</th>
<th>payload</th>
</tr>
</thead>
</table>
```

Caused by
- block metadata (inside block, outside payload)
- padding for alignment purposes
- explicit policy decisions (e.g., to return a big block to satisfy a small request)
- why would anyone do that?
External Fragmentation (32-bit)

Occurs when there is enough aggregate heap memory, but no single free block is large enough

\[
p_1 = \text{malloc}(16) \\
p_2 = \text{malloc}(20) \\
p_3 = \text{malloc}(24) \\
\text{free}(p_2) \\
p_4 = \text{malloc}(24) \quad \text{Oops! (what would happen now?)}
\]

Depends on the pattern of future requests

Thus, difficult to measure

Implementation Issues

How do we know how much to free given just a pointer?

How do we keep track of free blocks?

How do we pick a block to use for allocation?

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 make a freed block available for future reuse?

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

\[
p_0 = \text{malloc}(16) \\
\text{free}(p_0)
\]

Keeping Track of Free Blocks

Method 1: Implicit list using length—links all blocks

\[
\begin{array}{c}
\text{length} \\
|---|---|---|---|---|
\end{array}
\]

Method 2: Explicit list among the free blocks using pointers

\[
\begin{array}{c}
\text{length} \\
|---|---|---|---|---|
\end{array}
\]

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 aligned, low-order bits of size must be 0s.
Steal these bits for an allocated/free flag.
Mask off flag when using size.

Format of allocated and free blocks

<table>
<thead>
<tr>
<th>Size</th>
<th>Allocation Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>payload</td>
<td></td>
</tr>
<tr>
<td>optional padding</td>
<td></td>
</tr>
</tbody>
</table>

Implicit Free List Example (32-bit)
Sequence of blocks in heap (size|allocated): 8|0, 16|1, 32|0, 16|1

Start of heap

Free word
Allocated word
Allocated word closed

Implicit List: Finding a Free Block
First fit:
Search list from beginning, choose first free block that fits

Next fit:
Do first-fit starting where previous search finished

Best fit:
Search the list, choose the best free block: fits, with fewest bytes left over

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

malloc(12)
Implicit List: Freeing a Block

Simplest implementation:
Just clear the “allocated” flag, but can lead to “false fragmentation”

```c
free(p);
```

```
malloc()  Oops!
```

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
```
free(p);
```

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!

```
Header
```

<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;</code></th>
</tr>
</thead>
</table>

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

<table>
<thead>
<tr>
<th>size</th>
<th><code>&gt;</code></th>
</tr>
</thead>
</table>

Format of allocated and free blocks

Boundary tag (footer)

<table>
<thead>
<tr>
<th>size</th>
<th><code>=</code></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>payload: application data (allocated blocks only)</th>
</tr>
</thead>
</table>

**Constant Time Coalescing: 4 cases**
Implicit Free Lists: Summary
Implementation: very simple
Allocate: O(# blocks in heap)
Free: O(1)
Memory utilization: depends on placement policy
Not used in practice
used in some special purpose applications
Splitting, boundary tags, coalescing are general to all allocators.

Explicit Free Lists
Logically (doubly-linked lists):

Physically?

Explicit Free Lists
Logically (doubly-linked lists):

Physically: blocks can be in any order
Allocating From Explicit Free Lists

Before

Allocating From Explicit Free Lists

Before

After

= malloc(…)

Freeing With a LIFO Policy (Case 1)

Before

Insert the freed block at the root of the list

After

What if the freed block is adjacent to another?

Before

Root

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

After

Could be on either or both sides...
Explicit List Summary
Comparison to implicit list:
 Allocate: $O(\text{free blocks})$ vs. $O(\text{all blocks})$
   Much faster when most of the memory is full
 Splice blocks in and out of the list
 Minimum block size for next/prev links

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

Explicit Free Lists: Summary
Implementation: very simple
 Allocate: $O(\text{free blocks})$ vs. $O(\text{all blocks})$
 Free: $O(1)$
 Memory utilization: depends on placement policy
 Used in practice
   used in some special purpose applications
 Splitting, boundary tags, coalescing are general to all allocators.

Segregated List (Seglist) Allocators
Each size class of blocks has its own free list

<table>
<thead>
<tr>
<th>Size Class</th>
<th>Seglist</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
<tr>
<td>24-32</td>
<td></td>
</tr>
<tr>
<td>40-inf</td>
<td></td>
</tr>
</tbody>
</table>

Faster best-fit allocation...

Summary of Key Allocator Policies
All policies offer trade-offs in fragmentation and throughput.

Placement policy:
 First-fit, next-fit, best-fit, etc.
 Seglists approximate best-fit in low time

Splitting policy
 Always? Sometimes? Size bound?

Coalescing policy:
 Immediate coalescing
 Deferred coalescing