Dynamic Memory Allocation in the Heap

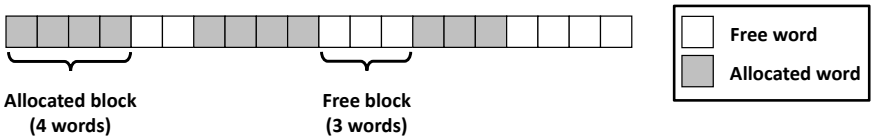
(malloc and free)

Explicit allocators (a.k.a. manual memory management)

Allocator Basics

Pages too coarse-grained for allocating individual objects.

Instead: **flexible-sized, word-aligned blocks.**



pointer to newly allocated block of at least that size

number of contiguous bytes required

```
void* malloc(size_t size);
```

pointer to allocated block to free

```
void free(void* ptr);
```


Allocator Goals: malloc/free

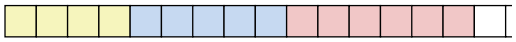
1. Programmer does not decide locations of distinct objects.

Programmer decides: what size, when needed, when no longer needed

2. Fast allocation.

mallocs/second or bytes malloc'd/second

3. High memory utilization.



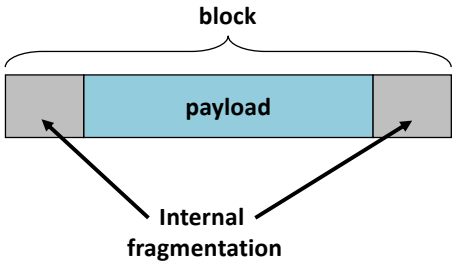
Most of heap contains necessary program data.

Little wasted space.

Enemy: **fragmentation** – unused memory that cannot be allocated.

Internal Fragmentation

payload smaller than block



Causes

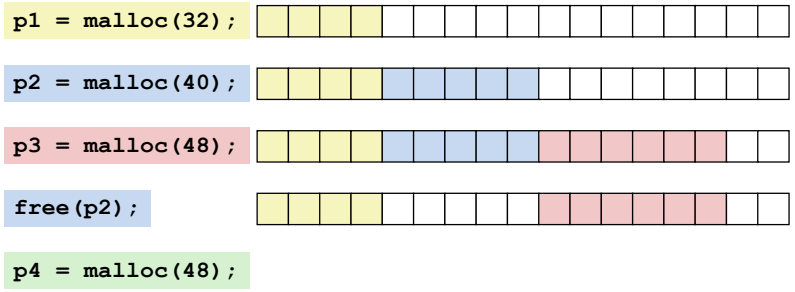
metadata

alignment

policy decisions

External Fragmentation (64-bit words)

Total free space large enough,
but no contiguous free block large enough



Depends on the pattern of future requests.

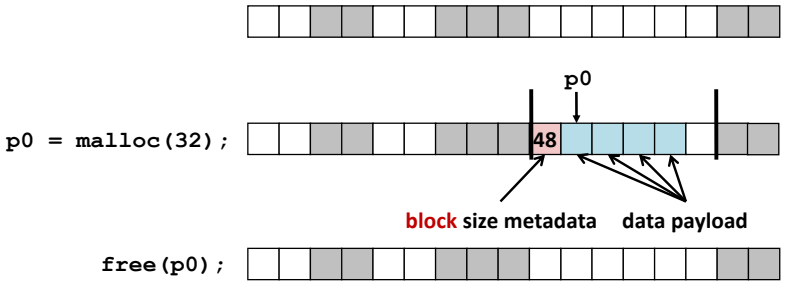
Implementation Issues

1. Determine **how much to free** given just a pointer.
2. **Keep track of free blocks.**
3. **Pick a block to allocate.**
4. Choose what do with **extra space when allocating** a structure that is **smaller than the free block used**.
5. **Make a freed block available for future reuse.**

Knowing How Much to Free

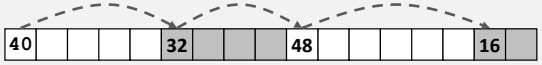
Keep length of block in ***header*** word preceding block

Takes extra space!

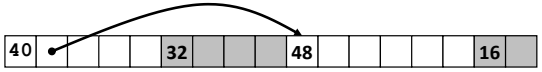


Keeping Track of Free Blocks

Method 1: *Implicit list* of all blocks using length



Method 2: *Explicit list* of free blocks using pointers



Method 3: *Seglist*

Different free lists for different size blocks

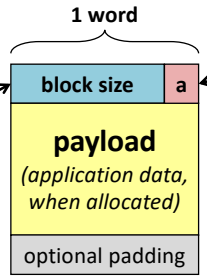
More methods that we will skip…

Implicit Free List: **Block Format**

Block metadata:

1. Block size
2. Allocation status

Store in one header word.

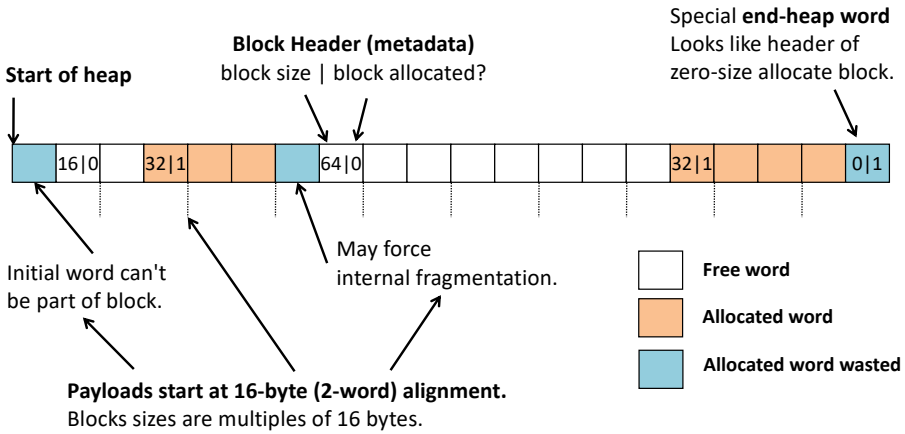


Steal LSB for status flag.
LSB = 1: allocated
LSB = 0: free

16-byte aligned sizes have 4 zeroes in low-order bits

00000000
00010000
00100000
00110000
…

Implicit Free List: **Heap Layout**



Payloads start at 16-byte (2-word) alignment.
Blocks sizes are multiples of 16 bytes.

Implicit Free List: **Finding a Free Block**

First fit:

Search list from beginning, choose ***first*** free block that fits

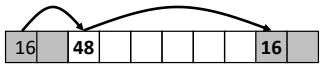
Next fit:

Do first-fit starting where previous search finished

Best fit:

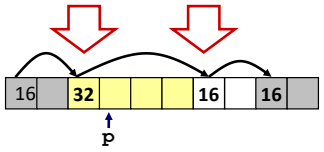
Search the list, choose the ***best*** free block: fits, with fewest bytes left over

Implicit Free List: **Allocating a Free Block**




```
p = malloc(24);
```

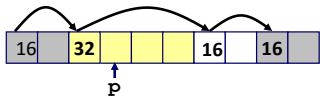
Allocated space ≤ free space.
Use it all? Split it up?



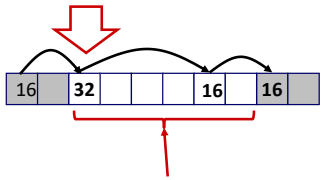
Block **Splitting**

Now showing allocation status flag implicitly with shading.

Implicit Free List: **Freeing a Block**

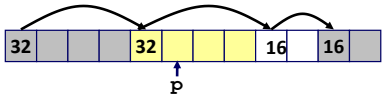


`free(p);` **Clear *allocated* flag.**

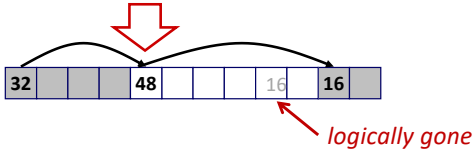


`malloc(40);` **External fragmentation!**
Enough space, not one block.

Coalescing Free Blocks



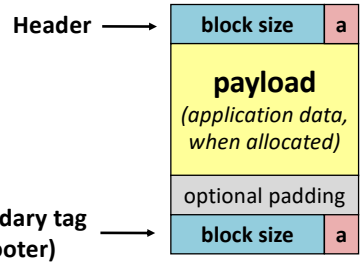
`free(p)` **Coalesce** with following *free* block.

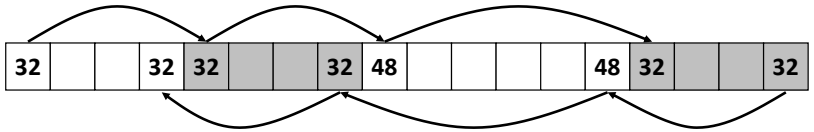


Coalesce with **preceding** *free* block?

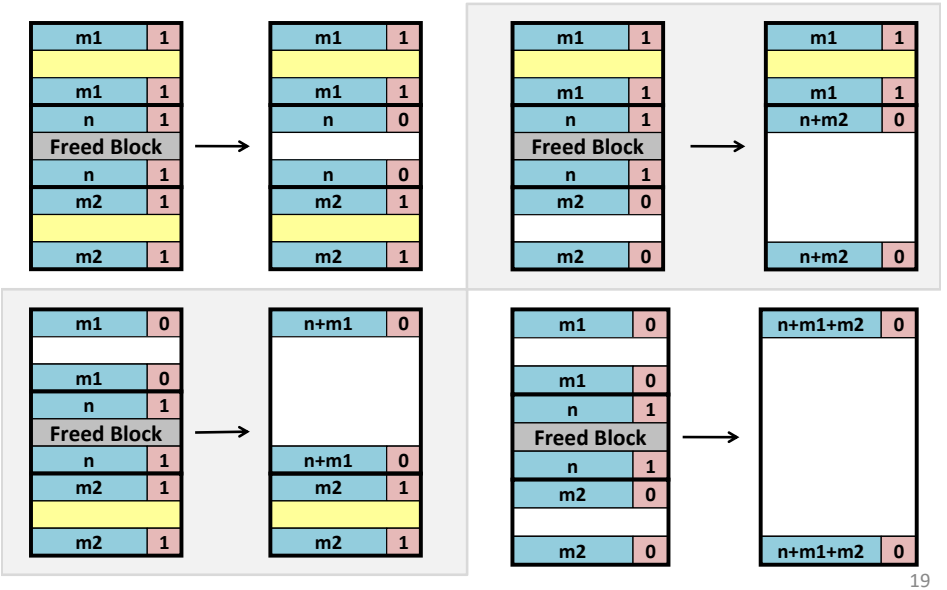
[Knuth73]

Bidirectional Coalescing: **Boundary Tags**





Constant-Time Coalescing: 4 cases



Summary: **Implicit Free Lists**

Implementation: **simple**

Allocate: **O(blocks in heap)**
Free: **O(1)**

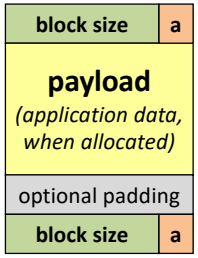
Memory utilization: depends on placement policy

Not widely used in practice
some special purpose applications

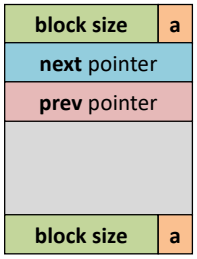
Splitting, boundary tags, coalescing are general to ***all*** allocators.

Explicit Free Lists

Allocated block:



Free block:

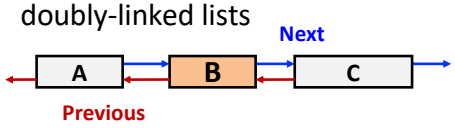


(same as implicit free list)

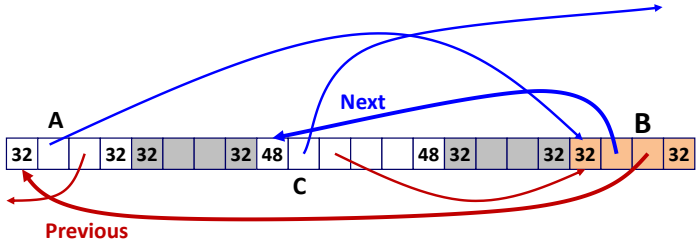
Explicit list of *free* blocks rather than **implicit list of *all* blocks.**

Explicit Free Lists: **List vs. Memory Order**

Abstractly: doubly-linked lists

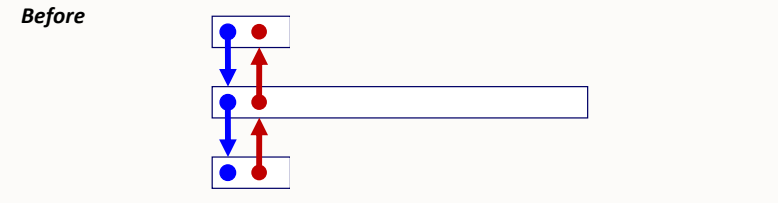


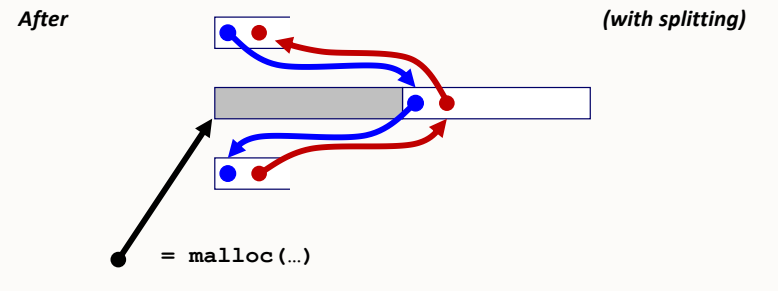
Concretely: free list blocks in any **memory** order



List Order ≠ Memory Order

Explicit Free Lists: **Allocating a Free Block**





Explicit Free Lists: **Freeing a Block**

***Insertion policy*:** Where in the free list do you add a freed block?

LIFO (last-in-first-out) policy

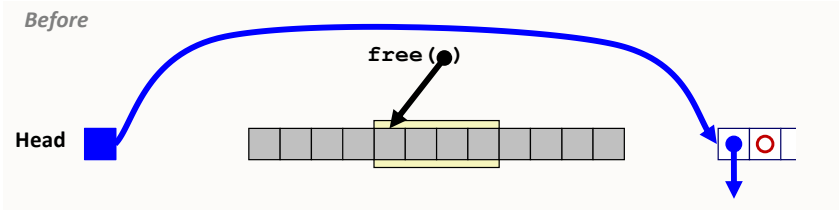
- ***Pro:*** simple and constant time
- ***Con:*** studies suggest fragmentation is worse than address ordered

Address-ordered policy

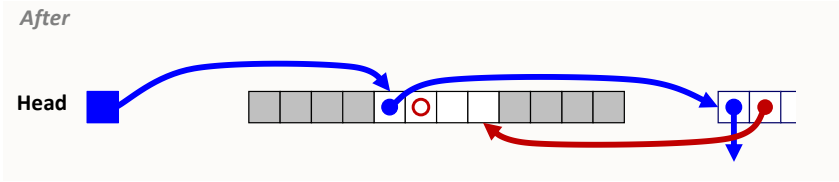
- ***Con:*** linear-time search to insert freed blocks
- ***Pro:*** studies suggest fragmentation is lower than LIFO

LIFO Example: 4 cases of freed block neighbor status.

Freeing with LIFO Policy: **between allocated blocks**



Insert the freed block at head of free list.



Summary: **Explicit Free Lists**

Implementation: **fairly simple**

Allocate:	**O(*free* blocks)**	vs. O(*all* blocks)
Free:	**O(1)**	vs. O(1)

Memory utilization:

depends on placement policy
larger minimum block size (next/prev) vs. implicit list

Used widely in practice, often with more optimizations.

Splitting, boundary tags, coalescing are general to ***all*** allocators.

Summary: **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 vs. deferred