CS 240 Spring 2020
Foundations of Computer Systems
Ben Wood

Dynamic Memory Allocation in the Heap

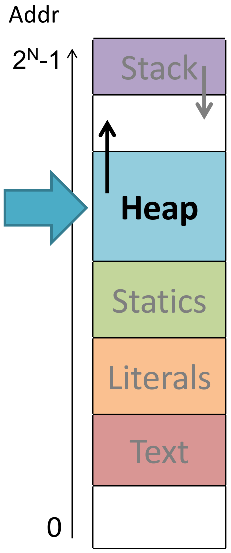
Explicit allocators

Manual memory management

C: implementing malloc and free

https://cs.wellesley.edu/~cs240/s20/

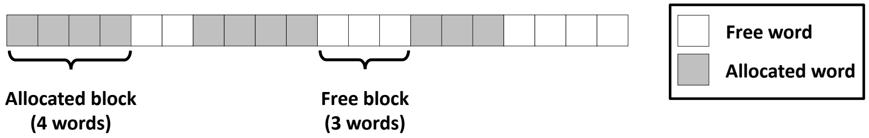
Heap Allocation



	Perm	Contents	Managed by	Initialized
Stack	RW	Procedure context	Compiler	Run-time
Heap	RW	**Dynamic data structures**	Programmer, **malloc/free,** new/GC	**Run-time**
Statics	RW	Global variables/ static data structures	Compiler/ Assembler/Linker	Startup
Literals	R	String literals	Compiler/ Assembler/Linker	Startup
Text	X	Instructions	Compiler/ Assembler/Linker	Startup

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

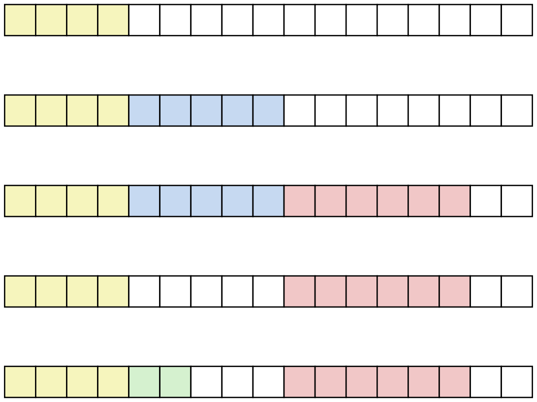
```
void* malloc(size_t size);
```

pointer to allocated block to free

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

Example (64-bit words)

```
p1 = malloc(32);
p2 = malloc(40);
p3 = malloc(48);
free(p2);
p4 = malloc(16);
```



Allocator goals: malloc/free

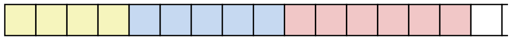
1. Programmer does not decide locations of distinct objects.

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

2. Fast allocation.

mallocs/second or bytes malloc'd/second

3. High memory utilization.



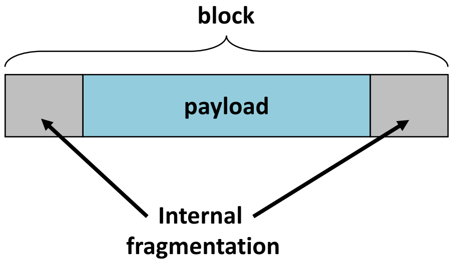
Most of heap contains necessary program data.

Little wasted space.

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

Internal fragmentation

payload smaller than block



Causes

metadata

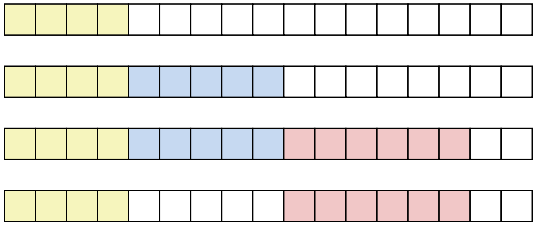
alignment

policy decisions

External fragmentation (64-bit words)

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

```
p1 = malloc(32);
p2 = malloc(40);
p3 = malloc(48);
free(p2);
p4 = malloc(48);
```



Depends on the pattern of future requests.

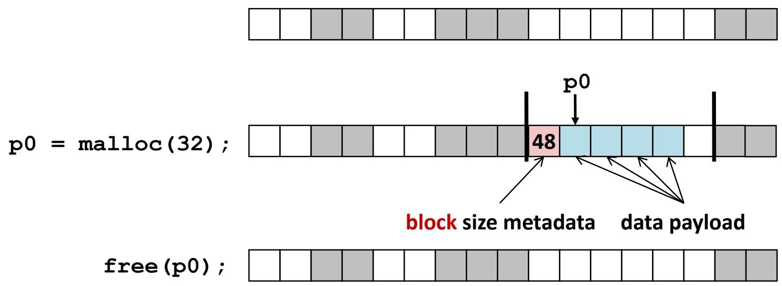
Implementation issues

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

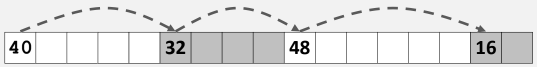
Keep length of block in *header* word preceding block

Takes extra space!

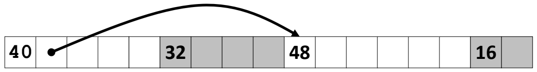


Keeping track of free blocks

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



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



Method 3: *Seglist*
Different free lists for different size blocks

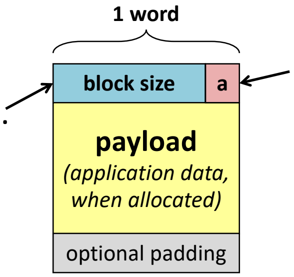
More methods that we will skip…

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

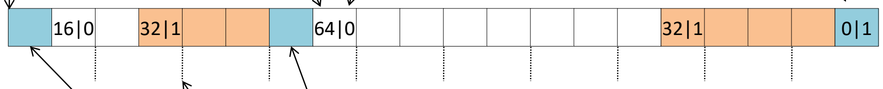
16-byte aligned sizes have
4 zeroes in low-order bits
0000**0000**
0001**0000**
0010**0000**
0011**0000**
…

Implicit free list: **heap layout**

Start of heap

Block Header (metadata)
block size | block allocated?

Special **end-heap word**
Looks like header of
zero-size allocate block.



Initial word cannot
be part of block.

May force
internal fragmentation.

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

□ **Free word**

🟧 **Allocated word**

🟦 **Allocated word wasted**

Implicit free list: **finding a free block**

First fit:

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

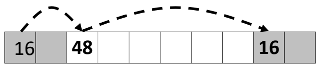
Next fit:

Do first-fit starting where previous search finished

Best fit:

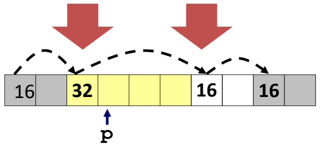
Search the list, choose the ***best*** free block: fits, with fewest bytes left over

Implicit free list: **allocating a free block**



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

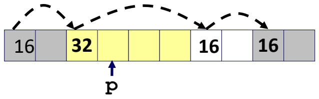
Allocated space ≤ free space.
Use it all? Split it up?



Block **Splitting**

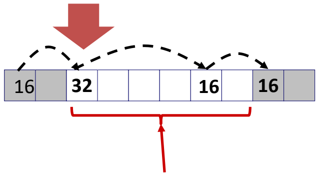
Now showing allocation status flag implicitly with shading.

Implicit free list: **freeing an allocated block**



```
free(p);
```

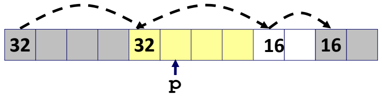
Clear *allocated* flag.



```
malloc(40);
```

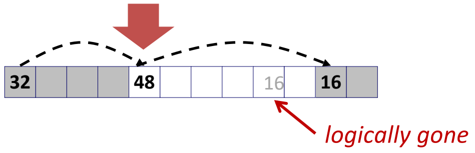
External fragmentation!
Enough space, not one block.

Coalescing free blocks



```
free(p)
```

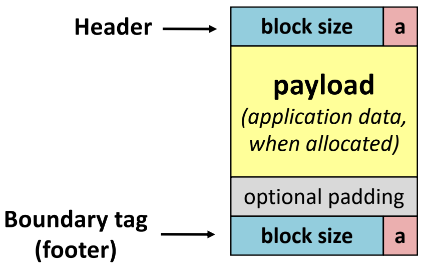
Coalesce with following *free* block.

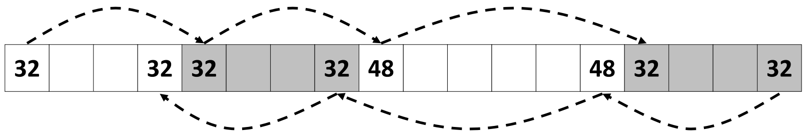


Coalesce with **preceding** *free* block?

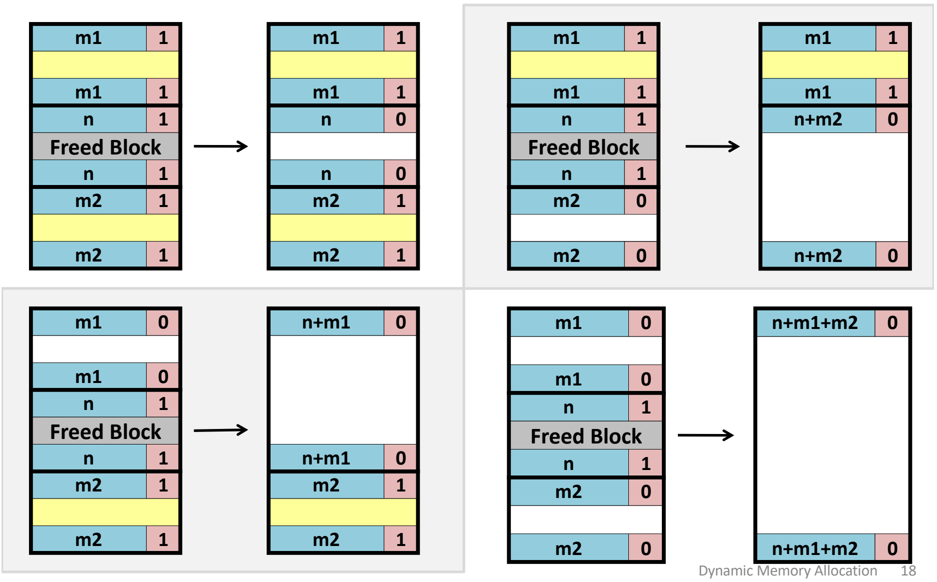
[Knuth73]

Bidirectional coalescing: boundary tags





Constant-time O(1) coalescing: 4 cases



Summary: **implicit free lists**

Implementation: simple

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

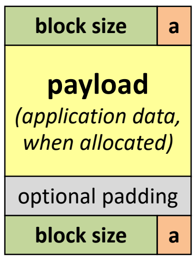
Memory utilization: depends on placement policy

Not widely used in practice
some special purpose applications

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

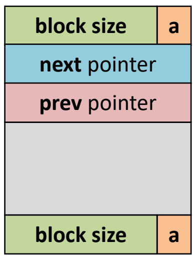
Explicit free list: block format

Allocated block:



(same as implicit free list)

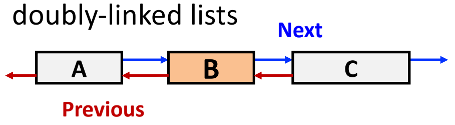
Free block:



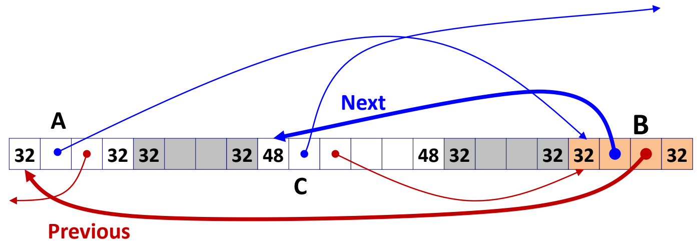
Explicit list of ***free*** blocks rather than implicit list of ***all*** blocks.

Explicit free list: **list vs. memory order**

Abstractly: doubly-linked lists

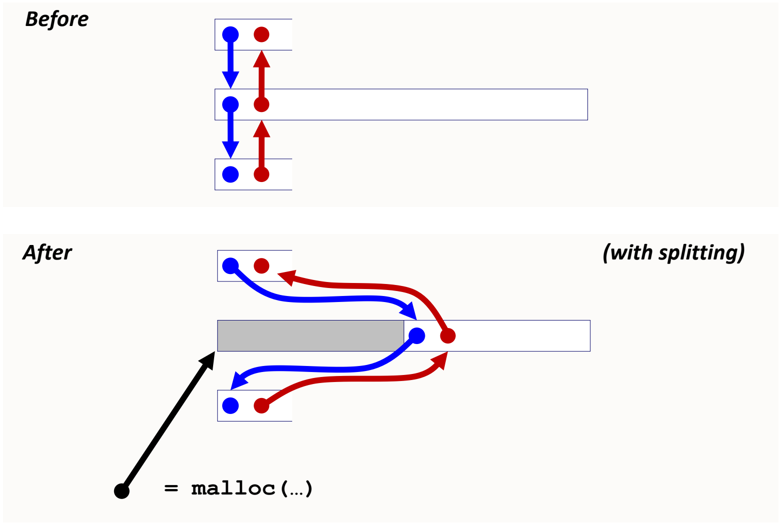


Concretely: free list blocks in any memory order



List Order ≠ Memory Order

Explicit free list: **allocating a free block**



Explicit free list: **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

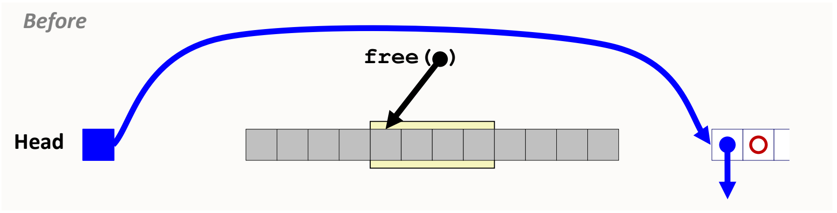
Address-ordered policy

Con: linear-time search to insert freed blocks

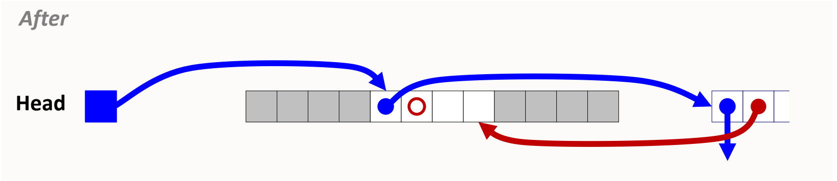
Pro: studies suggest fragmentation is lower than LIFO

LIFO Example: 4 cases of freed block neighbor status.

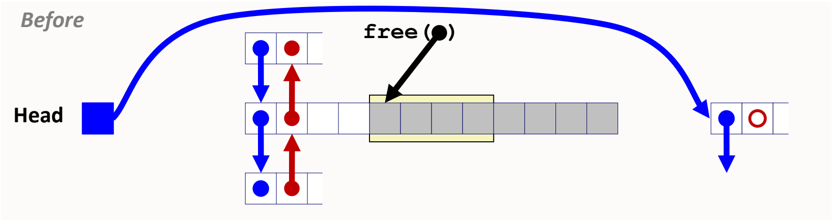
Freeing with LIFO policy: **between allocated blocks**



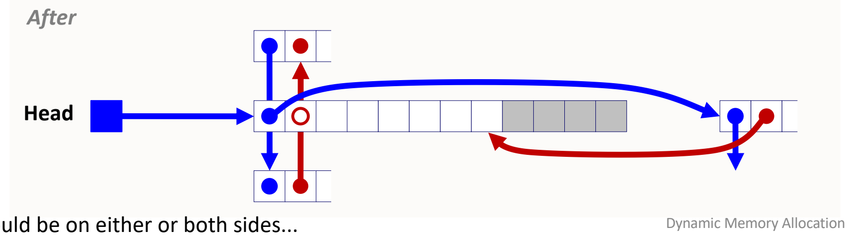
Insert the freed block at head of free list.



Freeing with LIFO policy: **between free and allocated**

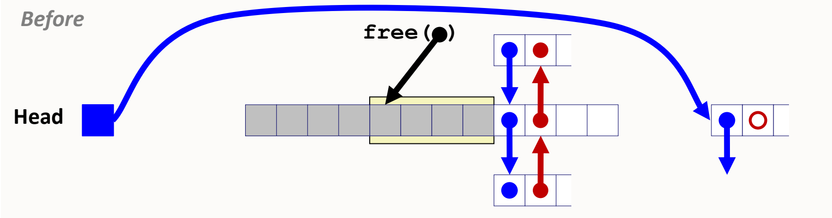


Splice out predecessor block, coalesce both memory blocks, and insert the new block at the head of the free list.

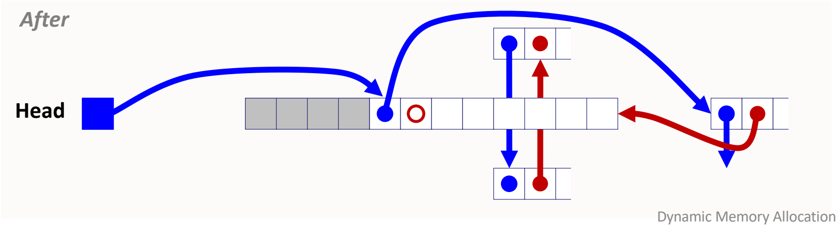


Could be on either or both sides...

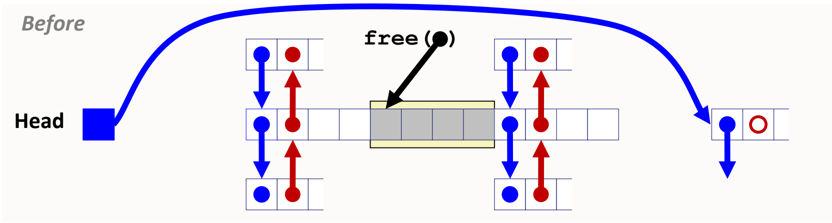
Freeing with LIFO policy: **between allocated and free**



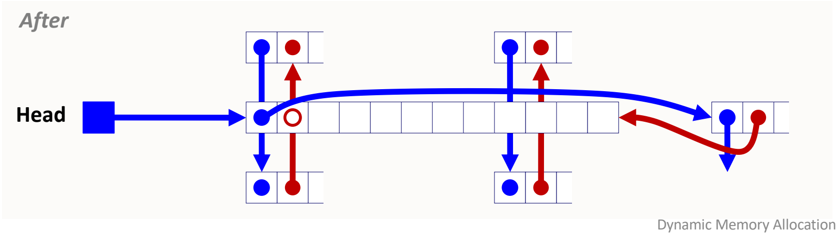
Splice out successor block, coalesce both memory blocks and insert the new block at the head of the free list.



Freeing with LIFO policy: **between free blocks**



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



Summary: **Explicit Free Lists**

Implementation: fairly simple

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

Memory utilization:

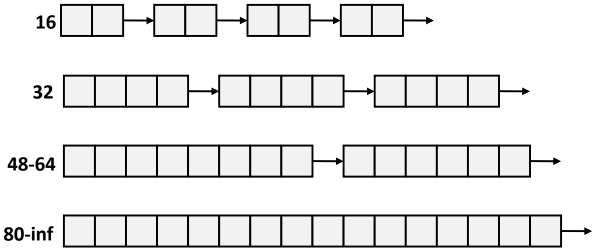
depends on placement policy
larger minimum block size (next/prev) vs. implicit list

Used widely in practice, often with more optimizations.

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

Seglist allocators

Each ***size bracket*** has its own free list



Faster best-fit allocation...

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:

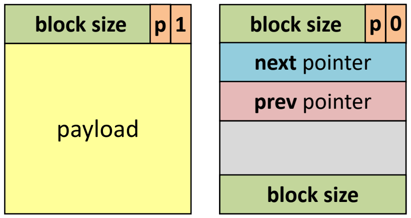
Always? Sometimes? Size bound?

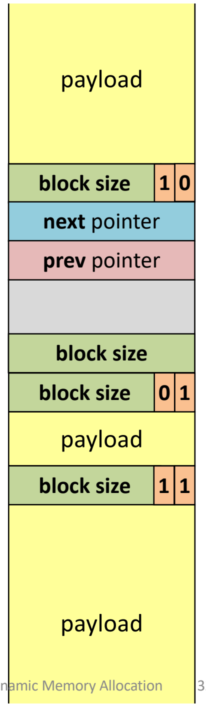
Coalescing policy:

Immediate vs. deferred

Remembrallocator block format

Allocated block: Free block:





Minimum block size?

- Implicit free list
- Explicit free list

Update headers of 2 blocks on each malloc/free.