CS 3214
Computer Systems

Dynamic Memory Management

Outline

- Memory management issues occur at multiple levels
  - User memory management (within the confines of a process)
    - Explicit vs implicit
  - Virtual address space management (by OS)
  - Physical memory management (by OS or hypervisor)
  - Interaction with virtual/physical address translation

Some of the following slides are taken with permission from
Complete Powerpoint Lecture Notes for
Computer Systems: A Programmer's Perspective (CS:APP)

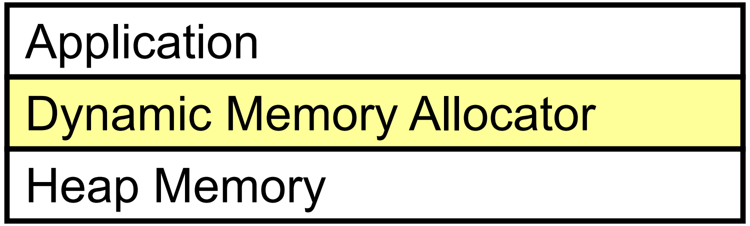
Randal E. Bryant and David R. O'Hallaron

http://csapp.cs.cmu.edu/public/lectures.html

Part 1

EXPLICIT MEMORY MANAGEMENT

Dynamic Memory Allocation

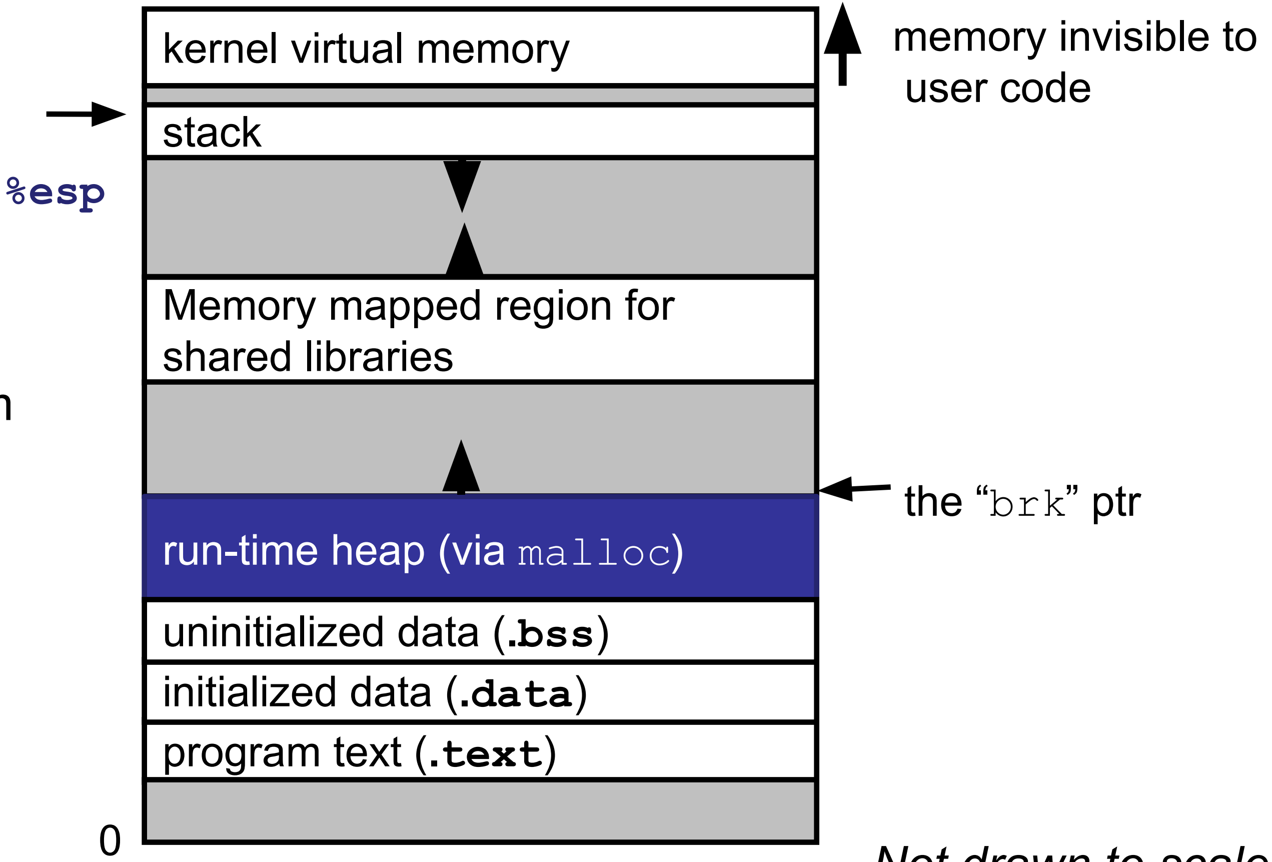


- Explicit vs. Implicit Memory Allocator
  - Explicit: application allocates and frees space
    - E.g., `malloc` and `free` in C
  - Implicit: application allocates, but does not free space
    - E.g. garbage collection in Java, ML or Lisp
- Allocation
  - In both cases the memory allocator provides an abstraction of memory as a set of blocks
  - Doles out free memory blocks to application
- Will discuss explicit memory allocation today

Process Memory Image

Allocators request additional heap memory from the operating system using the **`sbrk`** function.

Initial start of the heap is randomized
(a bit above end of .bss, usually)



Not drawn to scale

The Malloc API

`#include <stdlib.h>`
`void *malloc(size_t size)`

- If successful:
  - Returns a pointer to a memory block of at least `size` bytes, (typically) aligned to 8-byte boundary; use memalign() for other alignments
  - If `size == 0`, may return either NULL or a pointer that must be freed (platform-dependent)
- If unsuccessful: returns NULL (0) and sets `errno`.

`void free(void *p)`

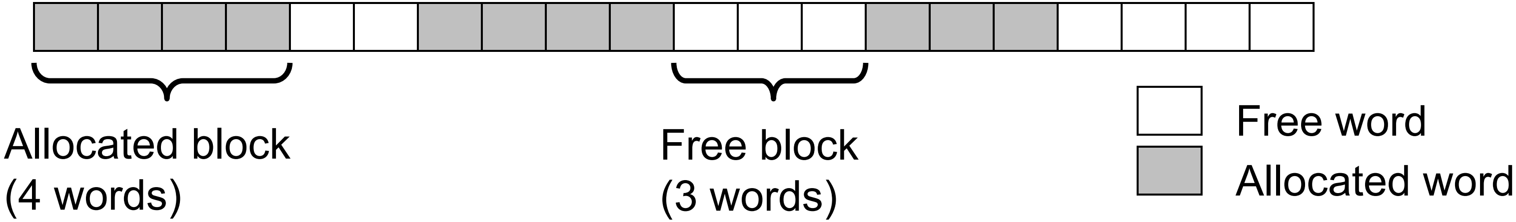
- Returns the block pointed at by `p` to pool of available memory
- `p` must come from a previous call to `malloc` or `realloc`.

`void *realloc(void *p, size_t size)`

- Changes size of block `p` and returns pointer to new block.
- Contents of new block unchanged up to min of old and new size.

Assumptions

- Assumptions made in this lecture
  - Memory is word addressed (each word can hold a pointer)



Allocation Examples

```
p1 = malloc(4)
```

```
p2 = malloc(5)
```

```
p3 = malloc(6)
```

```
free(p2)
```

```
p4 = malloc(2)
```

Constraints

- Applications: (clients)
  - Can issue arbitrary sequence of allocation and free requests
  - Free requests must correspond to an allocated block
- Allocators
  - Can’t control number or size of allocated blocks
  - Must respond immediately to all allocation requests
    - i.e., can’t reorder or buffer requests
  - Must allocate blocks from free memory
    - i.e., can place allocated blocks only in free memory
    - i.e., must maintain all data structures needed in memory they manage
  - Must align blocks so they satisfy all alignment requirements
    - 8 byte alignment for GNU malloc (libc malloc) on Linux boxes
  - Can manipulate and modify only free memory
    - Must not touch allocated memory
  - Can’t move the allocated blocks once they are allocated
    - i.e., compaction is not allowed

Goals for malloc/free design

- Primary goals
  - Good time performance for **`malloc`** and **`free`**
    - Ideally should take constant time (not always possible)
    - Should certainly not take linear time in the number of blocks
  - Good space utilization
    - User allocated structures ("payload") should be large fraction of the heap.
    - Want to minimize "fragmentation"
- Additional goals
  - Good locality properties
    - Structures allocated close in time should be close in space
    - "Similar" objects should be allocated close in space
  - Robust
    - Can check that `free(p1)` is on a valid allocated object `p1`
    - Can check that memory references are to allocated space

Performance Goals: Throughput

- Given some sequence of malloc and free requests:
  - $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$
- Want to maximize *throughput* and *peak memory* utilization.
  - These goals are often conflicting
  - Performance of allocators depends on the specific nature of the requests
- Throughput:
  - Number of completed requests per unit time
  - Example:
    - 5,000 malloc calls and 5,000 free calls in 10 seconds
    - Throughput is 1,000 operations/second.

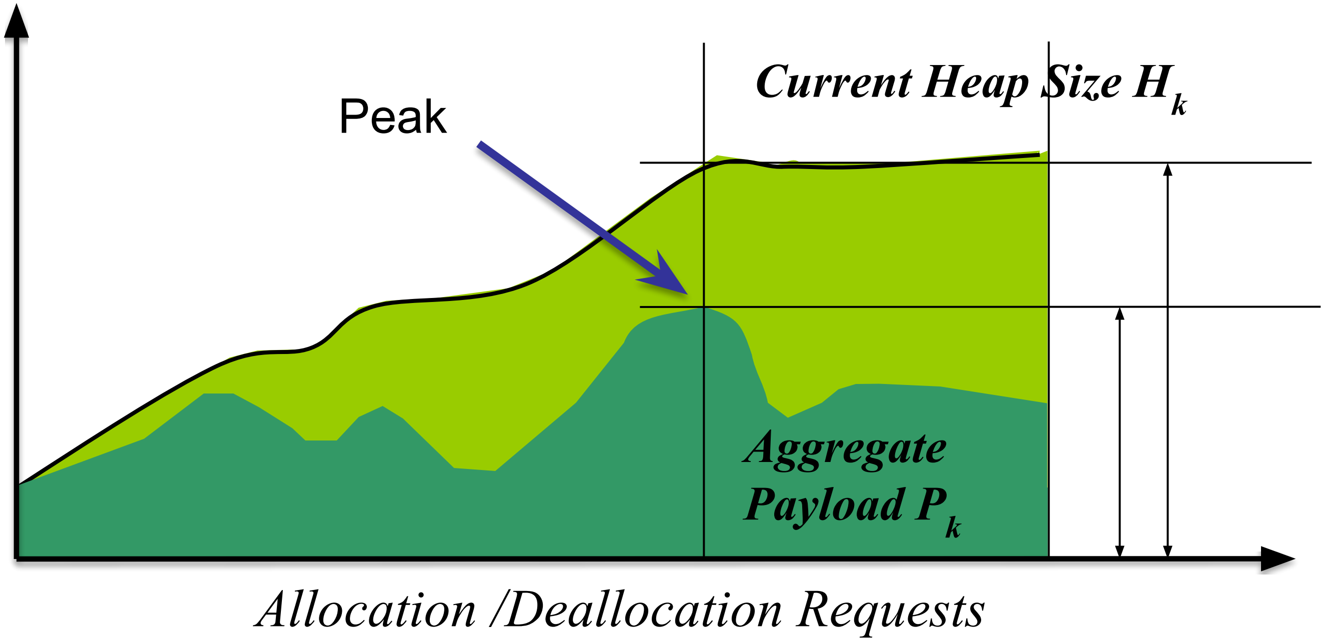
Performance Goals: Peak Memory Utilization

- Given some sequence of malloc and free requests:
  - $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$
- Def: Aggregate payload P_k:
  - malloc(p) results in a block with a payload of p bytes.
  - After request R_k has completed, the aggregate payload P_k is the sum of currently allocated payloads.
- Def: Current heap size is denoted by H_k
- Def: Peak memory utilization:
  - After *k* requests, peak memory utilization is:
    - $U_k = (\max_{i<k} P_i) / H_k$
  - Ratio of everything allocated and not yet free'd vs. how much space allocator is using, considered at the point where aggregate allocation was at its peak

Peak Memory Utilization

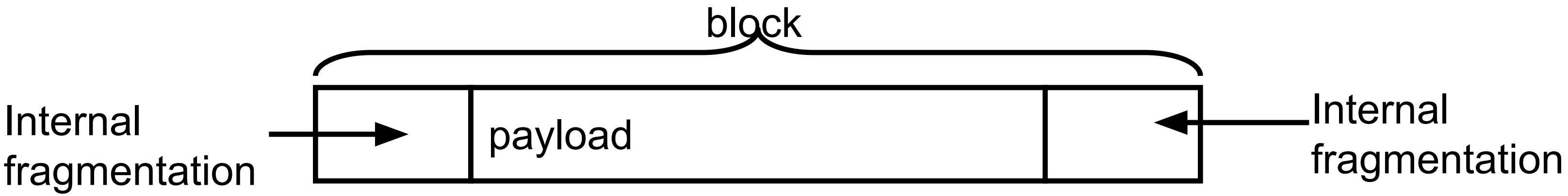
Lost to internal and external fragmentation

Used by application



Internal Fragmentation

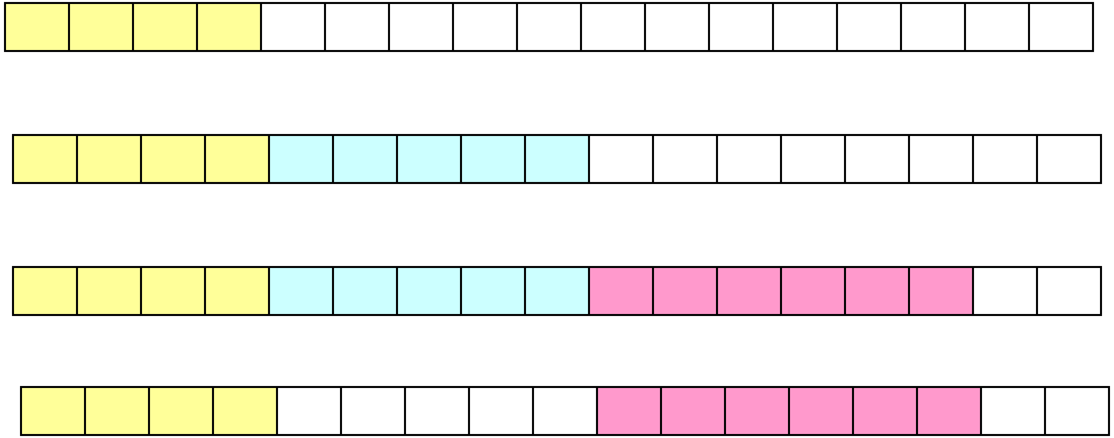
- Poor memory utilization caused by *fragmentation*.
  - Comes in two forms: internal and external fragmentation
- Definition: Internal fragmentation
  - For any block, **internal fragmentation** is the difference between the block size and the payload size.



  - Caused by overhead of maintaining heap data structures, padding for alignment purposes, or explicit policy decisions (e.g., not to split the block).
  - Depends only on the pattern of *previous* requests, and thus is easy to measure.

External Fragmentation

Occurs when there is enough aggregate heap memory, but no single free block is large enough; implies that allocator must obtain more memory via sbrk() and (eventually) may run out of memory



```
p1 = malloc(4)

p2 = malloc(5)

p3 = malloc(6)

free(p2)

p4 = malloc(6)
```

oops!

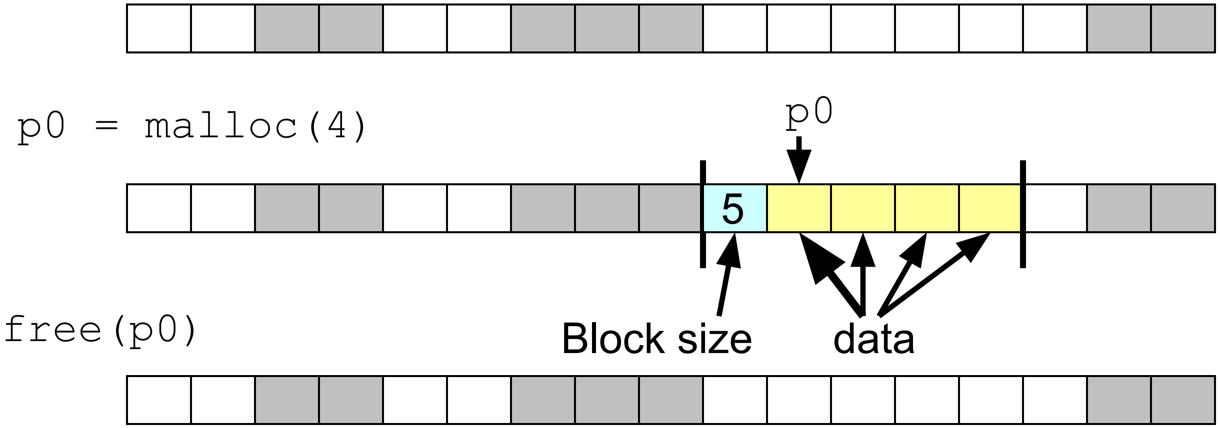
External fragmentation depends on the pattern of *future* requests, and thus is difficult to measure.

Implementation Issues

- How do we know how much memory to free just given a pointer?
  - free() takes no length!
- How do we keep track of the free blocks?
- What do we do with any extra space when allocating a structure that is smaller than the free block it is placed in?
- How do we pick a block to use for allocation -- many might fit the request?
- How do we reinsert freed block into heap?

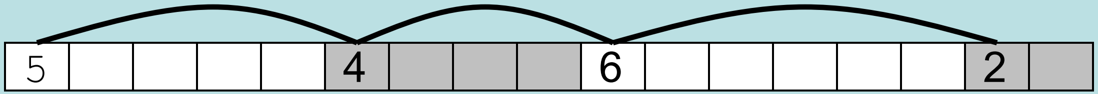
Knowing How Much to Free

- 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

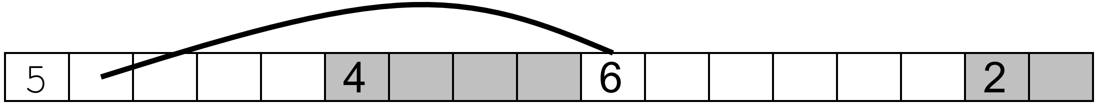


Keeping Track of Free Blocks

- *Method 1*: *Implicit list* using lengths -- links all blocks



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

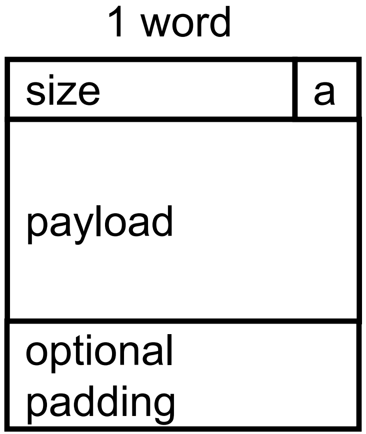


- *Method 3*: *Segregated free list*
  - Different free lists for different size classes
- *Method 4*: Blocks sorted by size
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

Method 1: Implicit List

- Need to identify whether each block is free or allocated
  - Don’t want to use extra word – steal last bit (can do that because size is a multiple of some power of two)
  - mask out low order bit when reading size.

Format of
allocated and
free blocks



1 word

size

a

payload

optional
padding

a = 1: allocated block
a = 0: free block

size: block size

payload: application data
(allocated blocks only)

Side Note

The following slides use explicit bit manipulation using C’s &, |, etc. operators.

Do not use those in your project.

Use bitfields instead, which modern compilers generally compile down to code that’s identical in performance.

Implicit List: Finding a Free Block

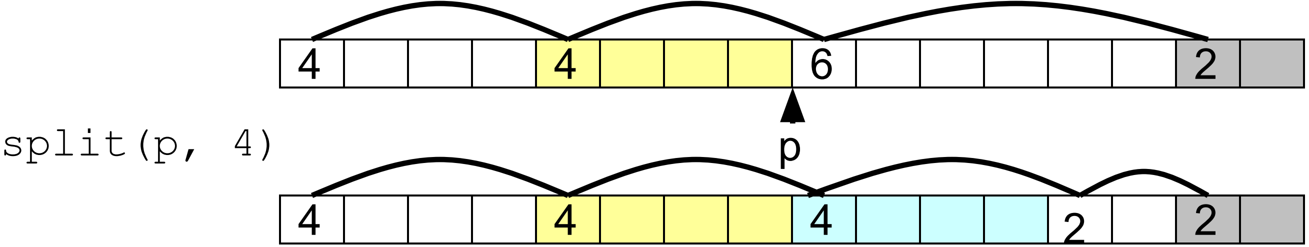
- *First fit:*
  - Search list from beginning, choose first free block that fits

```
p = start;
while ((p < end) ||     // not passed end
       (*p & 1) ||      // already allocated
       (*p <= len))     // too small
    p = p + (*p & ~1);
```

  - Can take linear time 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 from location of end of previous search
  - Research suggests that fragmentation is worse
- *Best fit:*
  - Search the list, choose the free block with the closest size that fits
  - 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

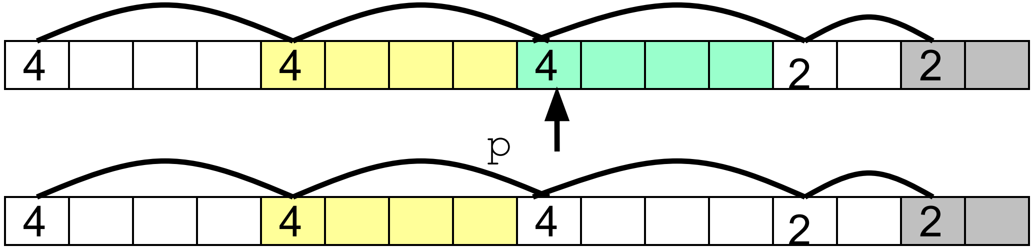


```
void split(ptr p, int len) {
  int newsize = ((len + 1) >> 1) << 1;  // add 1 and round up
  int oldsize = *p & ~1;                // mask out low bit
  *p = newsize | 1;                     // set new length
  if (newsize < oldsize)
    *(p+newsize) = oldsize - newsize;   // set length in remaining
}                                       // part of block
```

Implicit List: Freeing a Block

- Simplest implementation:
  - Only need to clear allocated flag

    `void free_block(ptr p) { *p = *p & -2}`
  - But can lead to "false fragmentation"



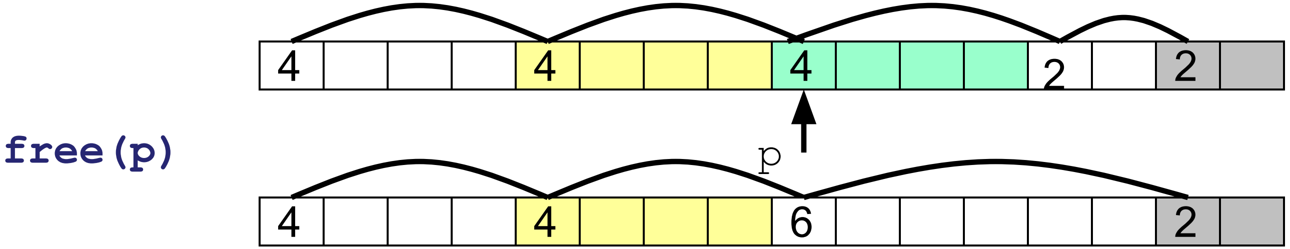
`free(p)`

`malloc(5)` Oops!

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

Implicit List: Coalescing

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

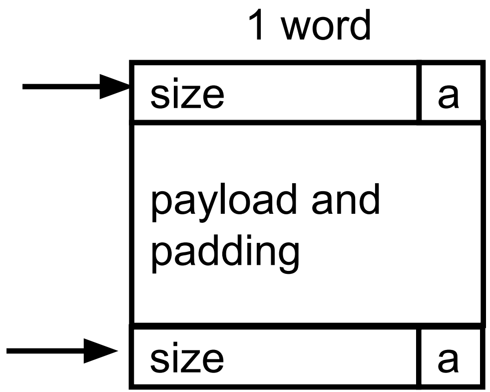


```
void free_block(ptr p) {
    *p = *p & -2;          // clear allocated flag
    next = p + *p;         // find next block
    if ((*next & 1) == 0)
      *p = *p + *next;    // add to this block if
}                         //    not allocated
```

But how do we coalesce with previous block?

Implicit List: Bidirectional Coalescing

- *Boundary tags* [Knuth73]
  - Replicate size/allocated word at bottom of free blocks
  - Allows us to traverse the “list” backwards, but requires extra space

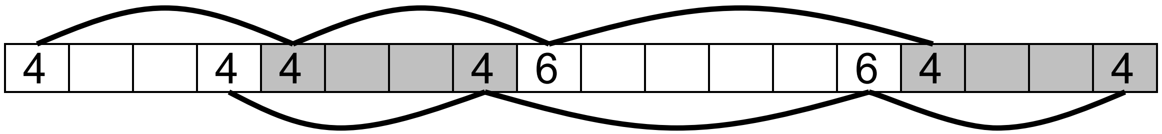


Format of allocated and free blocks

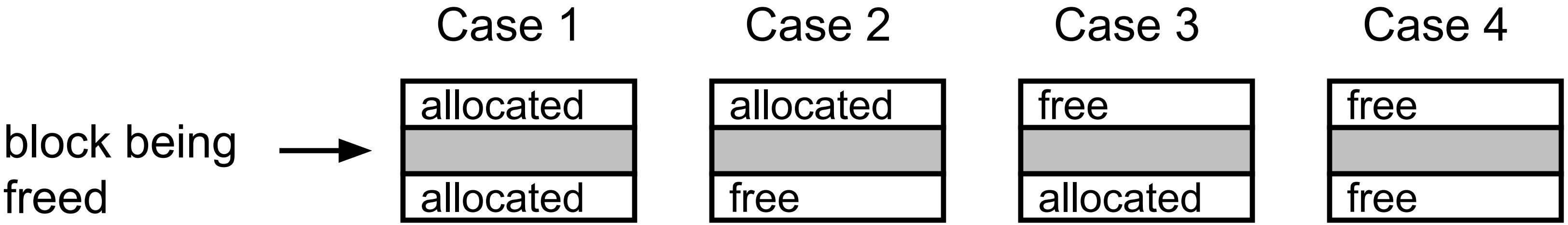
a = 1: allocated block
a = 0: free block

size: total block size

payload: application data (allocated blocks only)

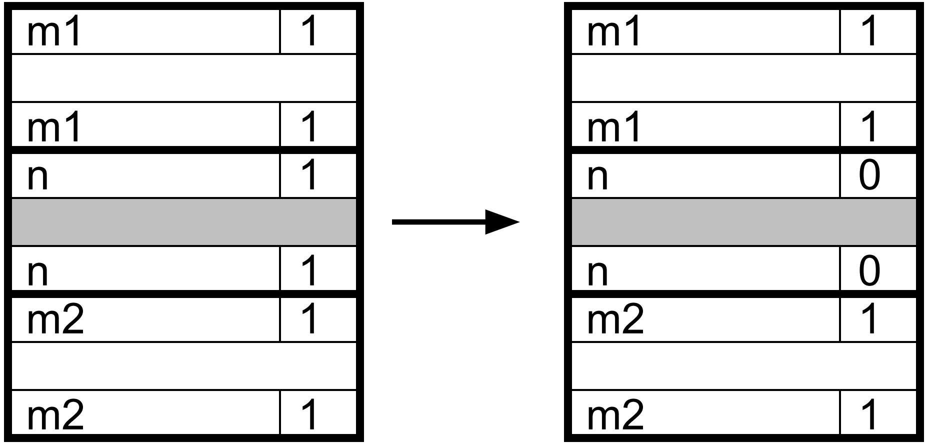


Constant Time Coalescing



	Case 1	Case 2	Case 3	Case 4
	allocated	allocated	free	free
block being freed →				
	allocated	free	allocated	free

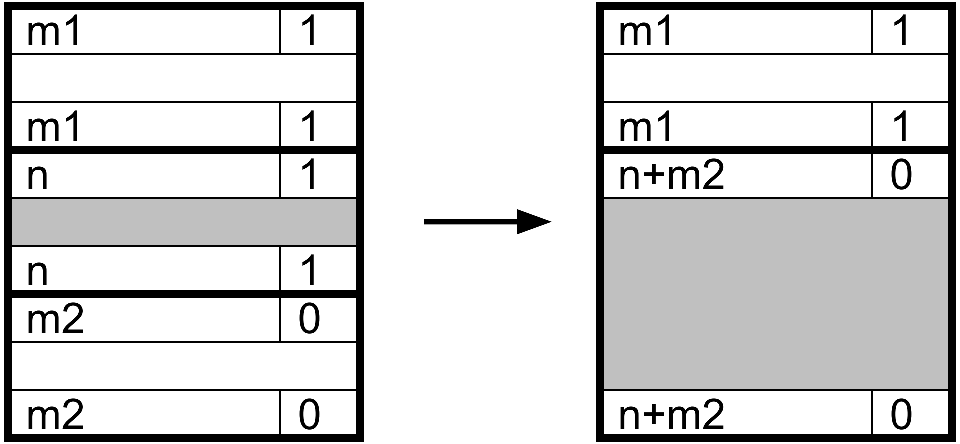
Constant Time Coalescing (Case 1)



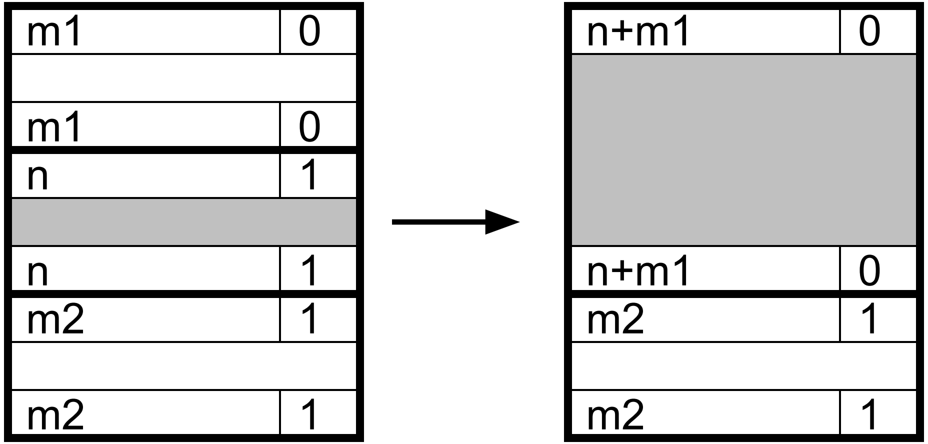
m1	1
m1	1
n	1
n	1
m2	1
m2	1

m1	1
m1	1
n	0
n	0
m2	1
m2	1

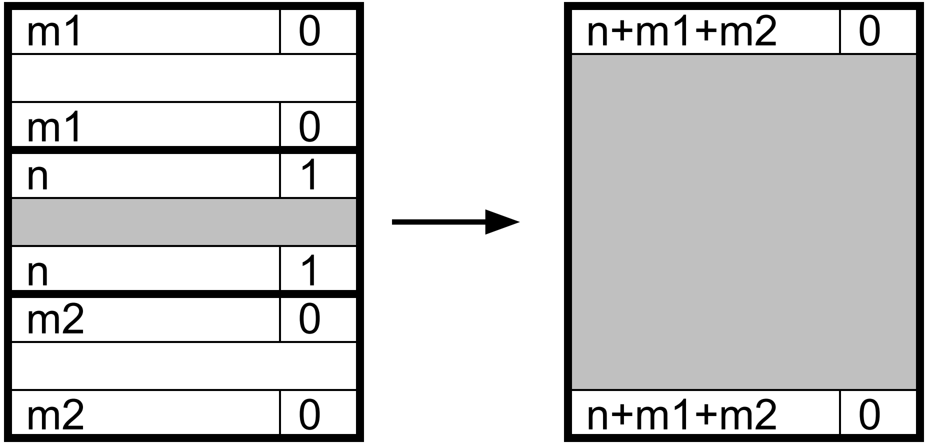
Constant Time Coalescing (Case 2)



Constant Time Coalescing (Case 3)



Constant Time Coalescing (Case 4)



Summary of Key Allocator Policies

- Placement policy:
  - First fit, next fit, best fit, etc.
  - Trades off lower throughput for less fragmentation
    - Interesting observation: segregated free lists (discussed later) approximate a best fit placement policy without having the 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 adjacent blocks each time free is called
  - Deferred coalescing: try to improve performance of free by deferring coalescing until needed. e.g.,
    - Coalesce as you scan the free list for malloc.
    - Coalesce when the amount of external fragmentation reaches some threshold.

Implicit Lists: Summary

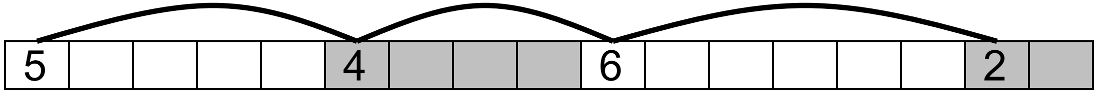
- Implementation: very simple
- Allocate: linear time worst case
- Free: constant time worst case -- even taking coalescing into account
- Memory usage: will depend on placement policy
  - First fit, next fit or best fit

- Not used in practice for malloc/free because of linear time allocate
  - Used in many special purpose applications

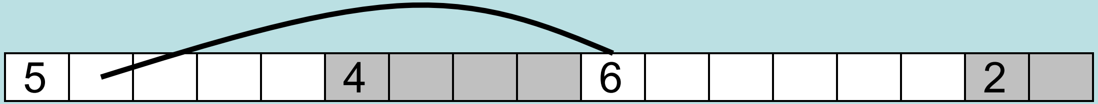
- However, the concepts of splitting and boundary tag coalescing are general to *all* allocators

Keeping Track of Free Blocks

Method 1: Implicit list using lengths -- links all blocks



Method 2: Explicit list among the free blocks using pointers within the free blocks



Method 3: Segregated free lists

Different free lists for different size classes

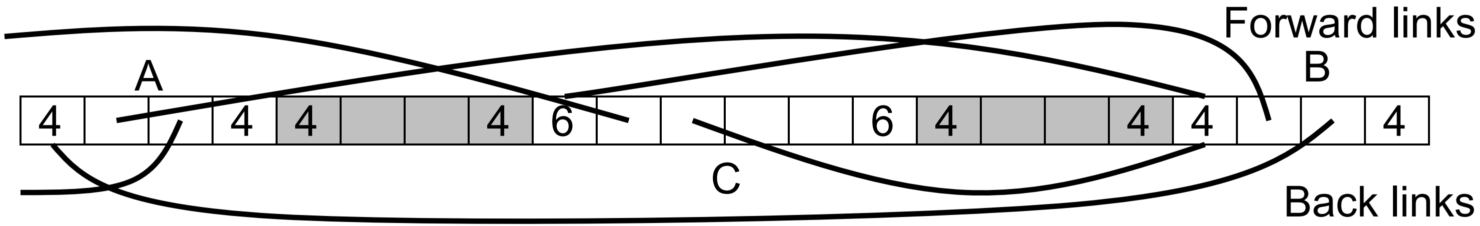
Method 4: Blocks sorted by size (not discussed)

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

Logical View

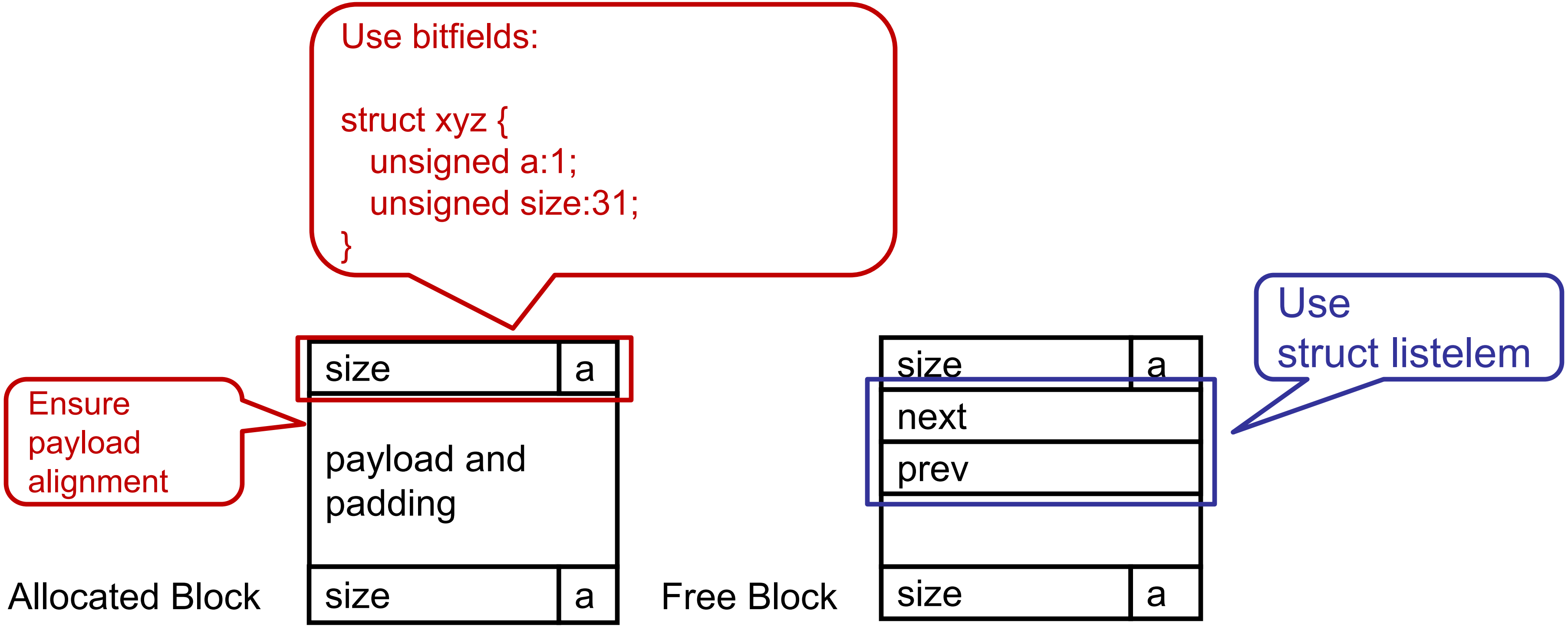
- Use data space for link pointers
  - Typically doubly linked
  - Still need boundary tags for coalescing



Physical View

  - Links are not necessarily in the same order as the blocks

Allocated vs. Free Blocks



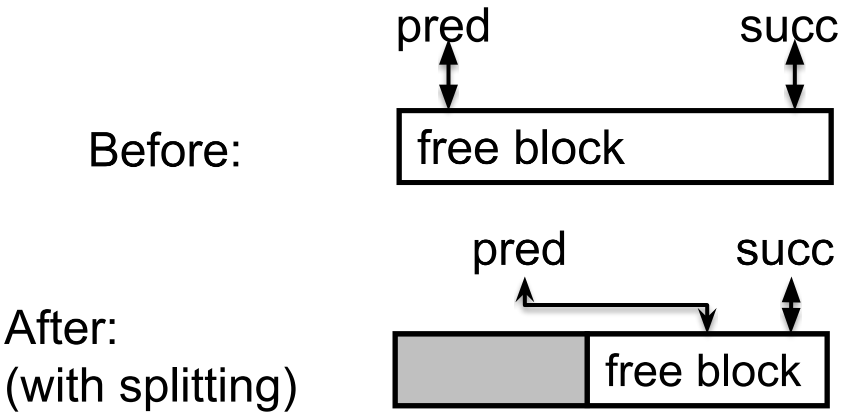
Use bitfields:

```
struct xyz {
   unsigned a:1;
   unsigned size:31;
}
```

Ensure payload alignment

Use struct listelem

Splitting & Explicit Free Lists



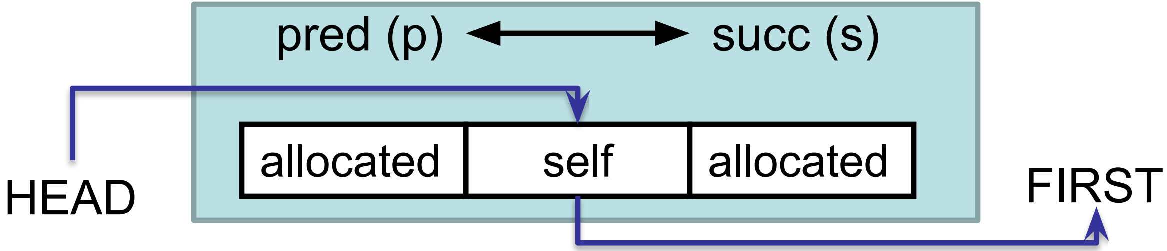
Note: if free block is left at same position in free list, can also split off bottom of block – then no pointer manipulation necessary

Freeing With Explicit Free Lists

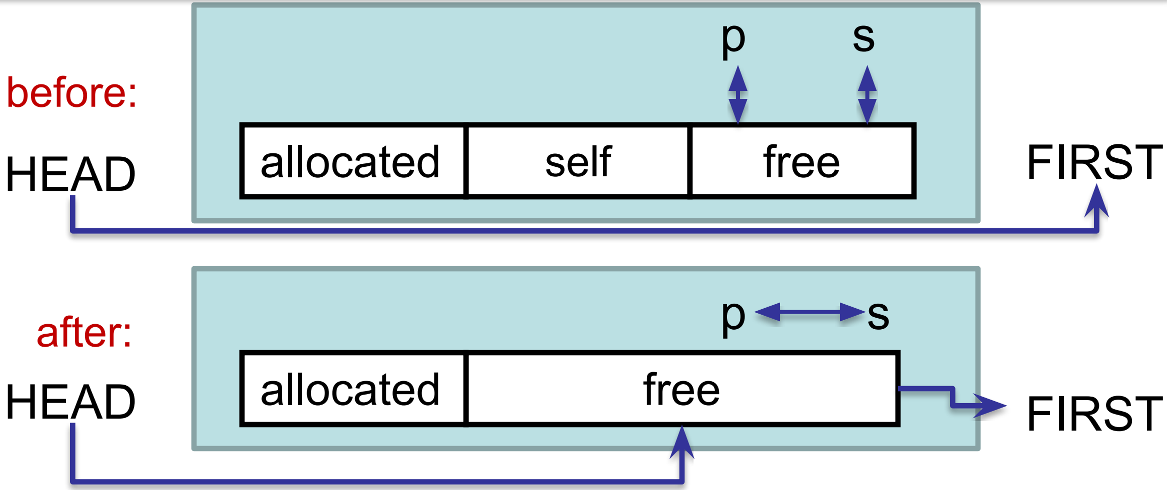
- *Insertion policy*: Where in the free list do you put a newly freed block?
  - LIFO (last-in-first-out) policy
    - Insert freed block at the beginning of the free list
    - Pro: simple and constant time
    - Con: studies suggest fragmentation is worse than address ordered
  - Address-ordered policy
    - Insert freed blocks so that free list blocks are always in address order
      - i.e. addr(pred) < addr(curr) < addr(succ)
    - Con: requires search
    - Pro: studies suggest fragmentation is better than LIFO

Freeing With a LIFO Policy

- Case 1: a-a-a
  - Insert self at beginning of free list

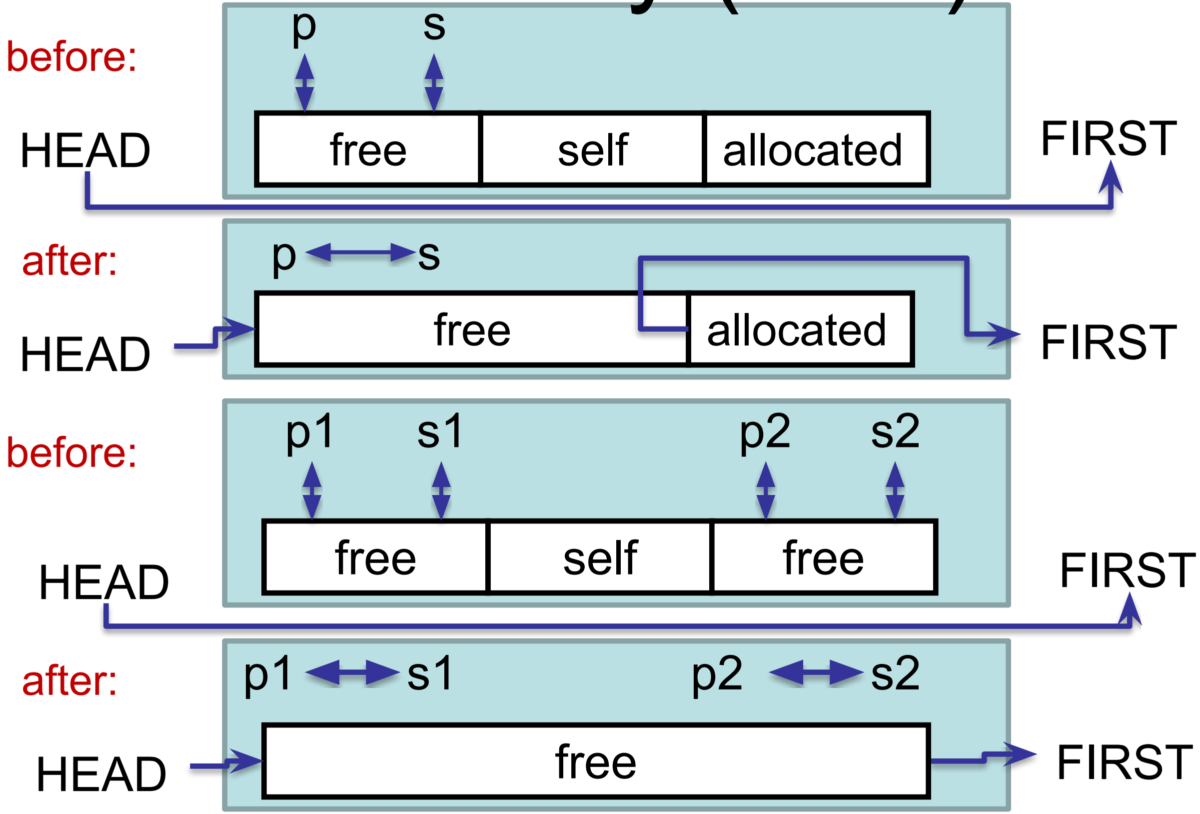


- Case 2: a-a-f
  - Splice out next, coalesce self and next, and add to beginning of free list



Freeing With a LIFO Policy (cont)

- Case 3: f-a-a
  - Splice out prev, coalesce with self, and add to beginning of free list
- Case 4: f-a-f
  - Splice out prev and next, coalesce with self, and add to beginning of list

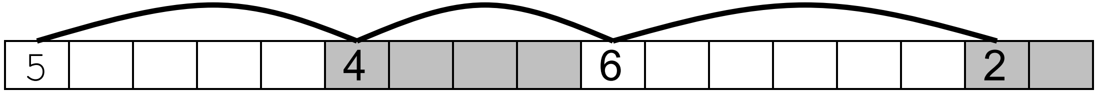


Explicit List Summary

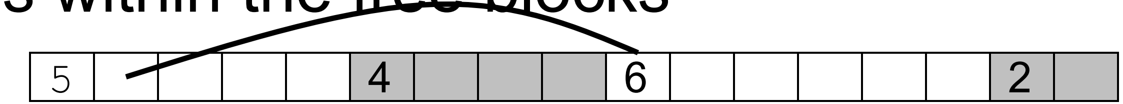
- Comparison to implicit list:
  - Allocate is linear time in number of free blocks instead of total blocks -- much faster allocates when most of the memory is full
  - Slightly more complicated allocate and free since needs to splice blocks in and out of the list
  - Some extra space for the links (2 extra words needed for each block)
- Main use of linked 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 lengths -- links all blocks



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



Method 3: *Segregated free list*

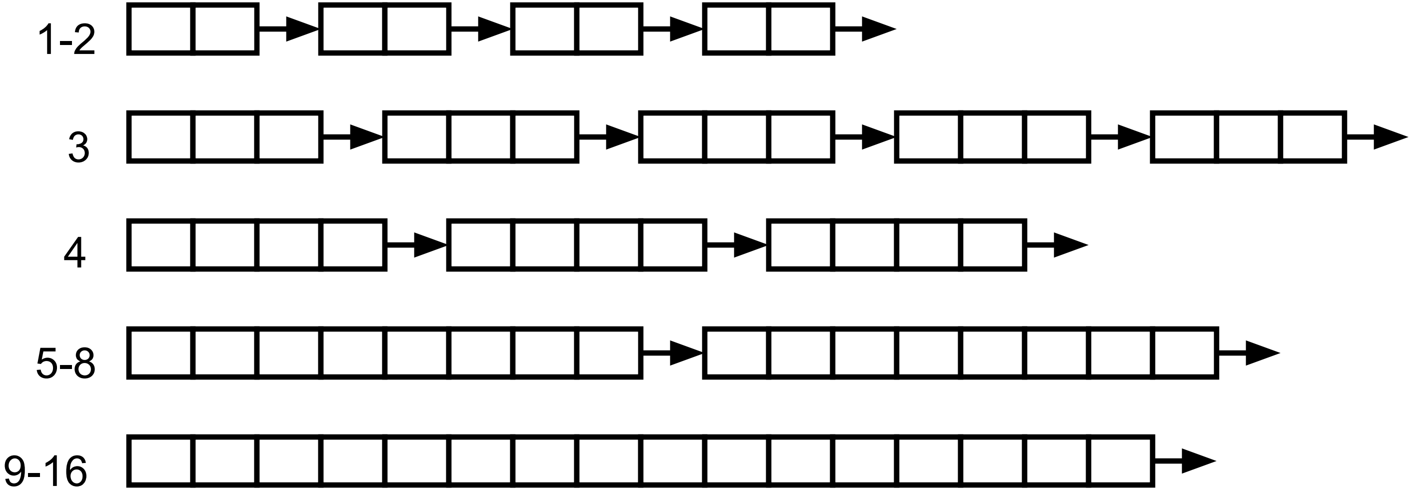
Different free lists for different size classes

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 Storage

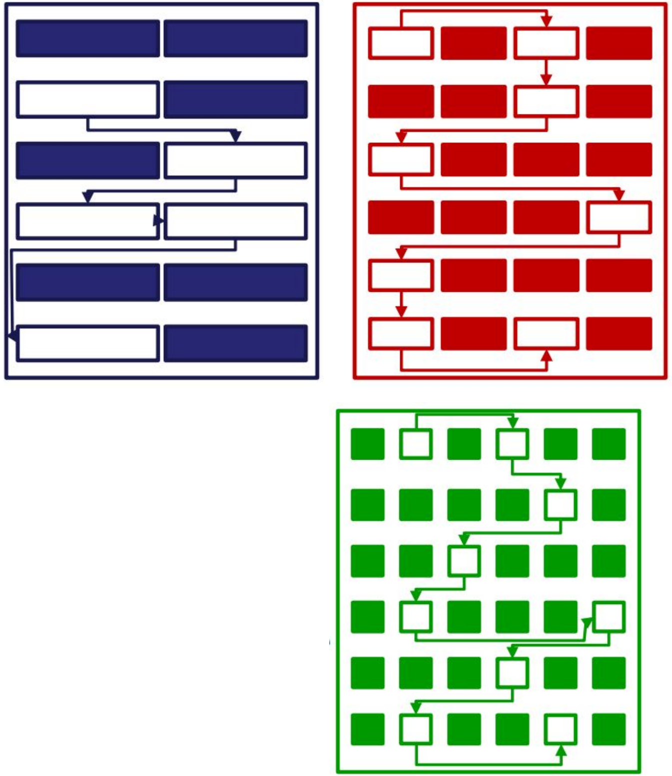
- Each *size class* has its own collection of blocks



  - Often have separate size class for every small size (2,3,4,…)
  - For larger sizes can have a size class for each power of 2

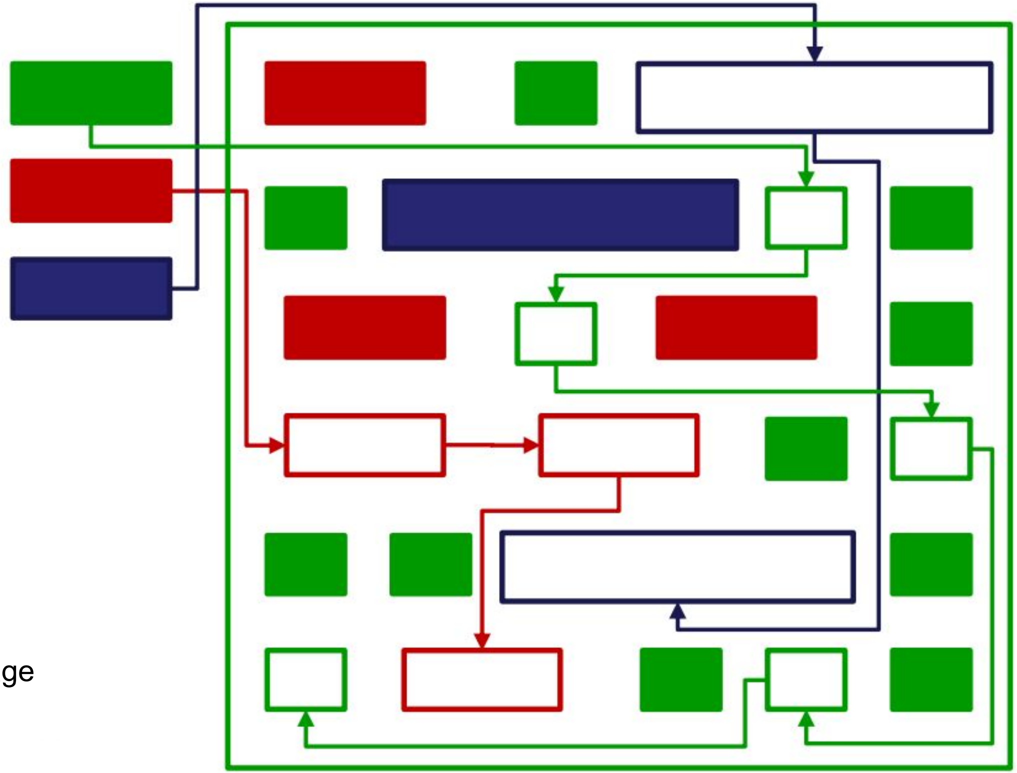
Simple Segregated Storage

- Separate heap and free list for each size class
- No splitting
- To allocate a block of size n:
  - If free list for size n is not empty,
    - allocate first block on list (note, list can be implicit or explicit)
  - If free list is empty,
    - get a new page
    - create new free list from all blocks in page
    - allocate first block on list
  - Constant time
- To free a block:
  - Add to free list
  - If page is empty, return the page for use by another size (optional)
- Tradeoffs:
  - Fast, but can fragment badly



Segregated Fits

- 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
- To free a block:
  - Coalesce and place on appropriate list (optional)
- Tradeoffs
  - Faster search than sequential fits (i.e., log time for power of two size classes)
  - Controls fragmentation of simple segregated storage
  - Coalescing can increase search times
    - Deferred coalescing can help



For More Info on Allocators

- D. Knuth, “The Art of Computer Programming, Second Edition”, Addison Wesley, 1973
  - The classic reference on dynamic storage allocation
- Wilson et al, “Dynamic Storage Allocation: A Survey and Critical Review”, Proc. 1995 Int’l Workshop on Memory Management, Kinross, Scotland, Sept, 1995.
  - Comprehensive survey
- NB: the mechanics of dynamic memory allocation have remained largely unchanged; however, modern memory allocators must pay a lot of attention to scalability in multi-threaded scenarios, which is beyond our scope here