ECE 454 Computer Systems Programming

Dynamic Memory

Ashvin Goel, Ding Yuan
ECE Dept, University of Toronto

Contents

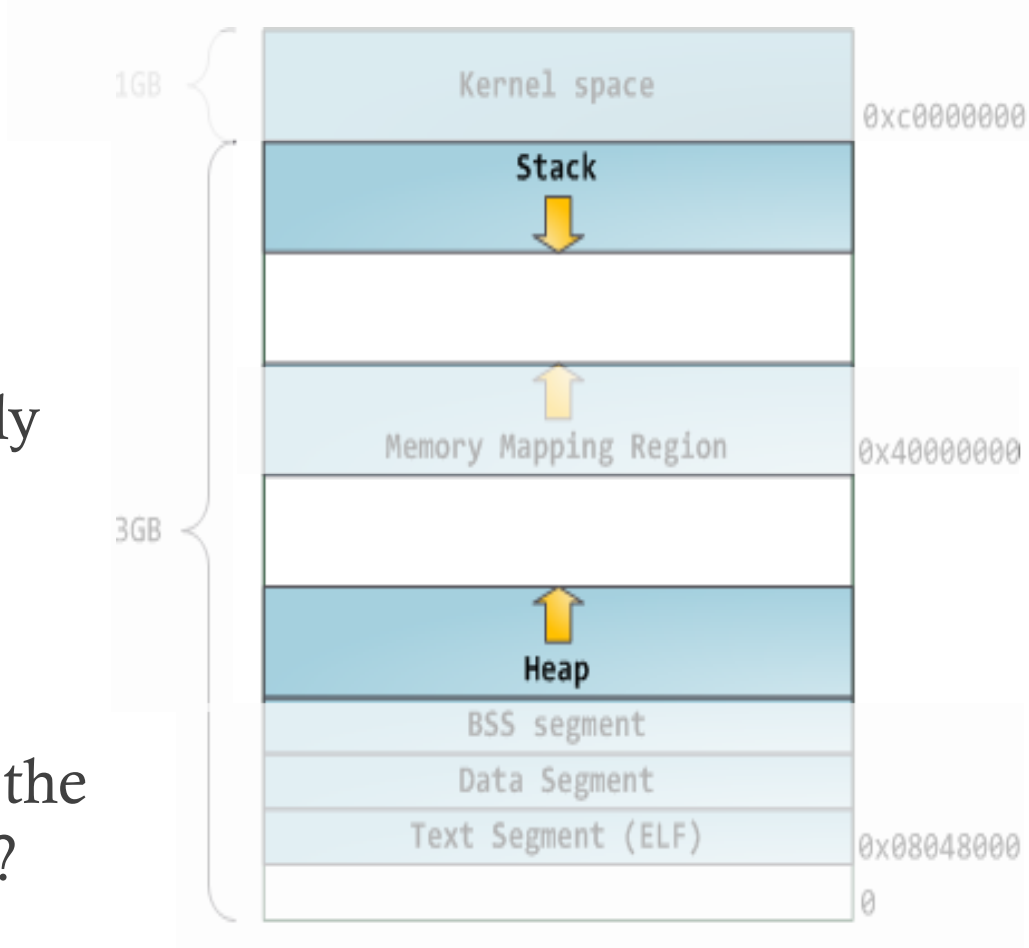
- Introduction to dynamic memory management
  - Alignment
  - Memory management API
  - Constraints, goals
  - Fragmentation
- Basic dynamic memory allocation
  - Implicit free list
  - Explicit free list
  - Segregated free lists
  - Buddy allocation
- Other memory management considerations

Why Dynamic Memory Allocation?

- Some data structure sizes are not known in advance
  - Read and store n values from file, where n is user specified
- Even today DRAM (main memory) is precious
  - Would like programs to request more memory when needed and give it back when no longer needed, to be re-used!

Aside: When to Use Stack vs. Heap

- Stack used to allocate
  - Local variables
  - Parameters
  - Return values
- Heap used for dynamically allocate memory
  - Memory allocated using malloc()
- Why can't we always use the stack to allocate memory?

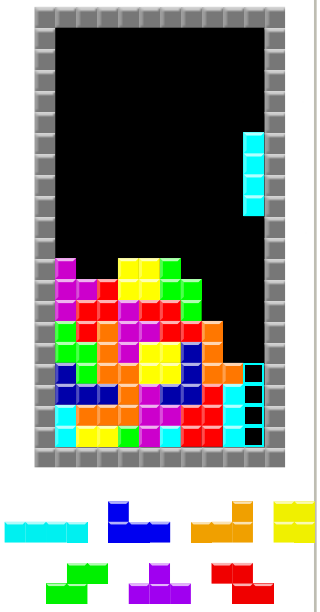


Why Learn about Dynamic Memory Allocation?

- Performance of dynamic memory allocation can significantly impact overall program performance
  - Programming guru: "don't use malloc, manage memory yourself!"
  - Today, many smart malloc implementations available
    - You should know how to use them effectively (or build one yourself …)
- Dynamic memory allocation is challenging/interesting
  - Good memory allocation algorithms are quite involved
  - Scalable memory allocation is essential for multi-core performance
- Gain a full understanding of systems "under-the-hood"
- Think you know pointers? Well, you'll learn pointers ☺

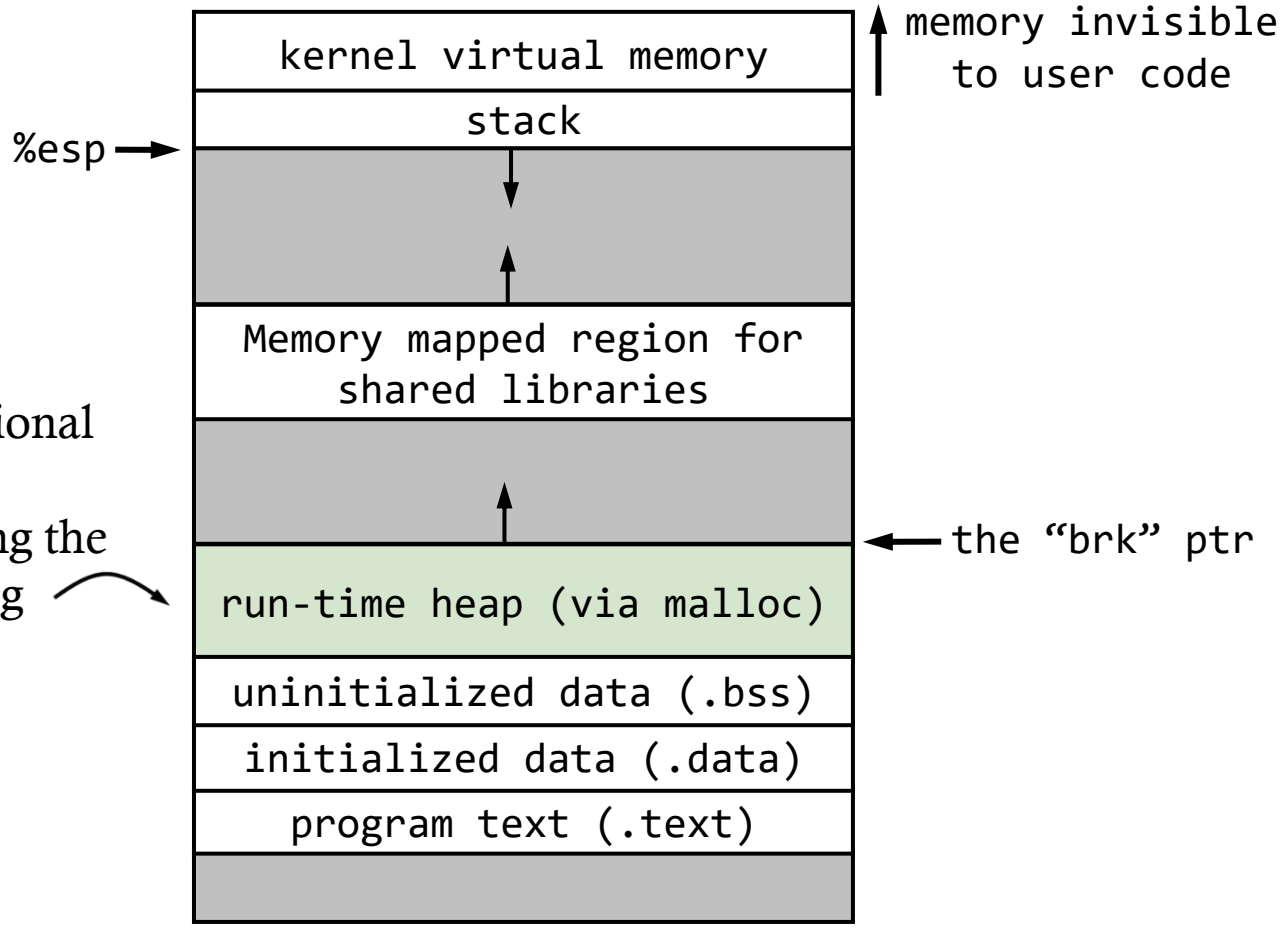
Dynamic Memory Allocators

- Provide an abstraction of memory as a set of blocks
  - A block is *variable* sized, *contiguous* memory
  - Provide free memory blocks to application



- Explicit: application allocates and frees space
  - E.g., `malloc` and `free` in C, `new` and `delete` in C++
- Implicit: application allocates, but does not free space
  - E.g., garbage collection in Java, ML or Lisp

Typical Process Memory Image



Allocators request additional heap memory from the operating system by using the sbrk or memory mapping system calls.

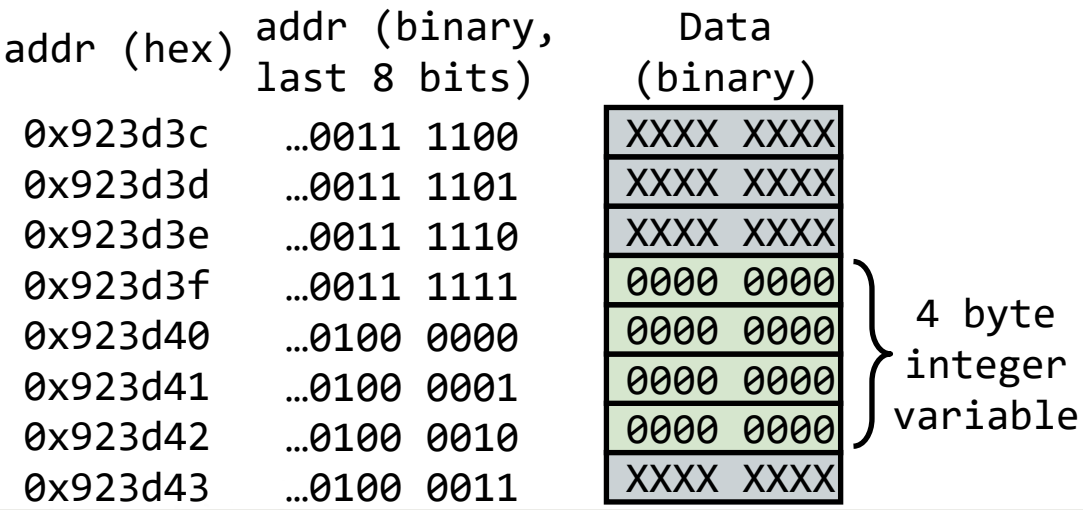
Background: Alignment

What is Alignment?

- Starting address of object must be multiple of K
  - K is typically a multiple of WORD size
  - 32-bit system
    - Word is 4 bytes, malloc returns objects with 8-byte alignment
  - 64-bit system
    - Word is 8 bytes, malloc returns objects with 16-byte alignment

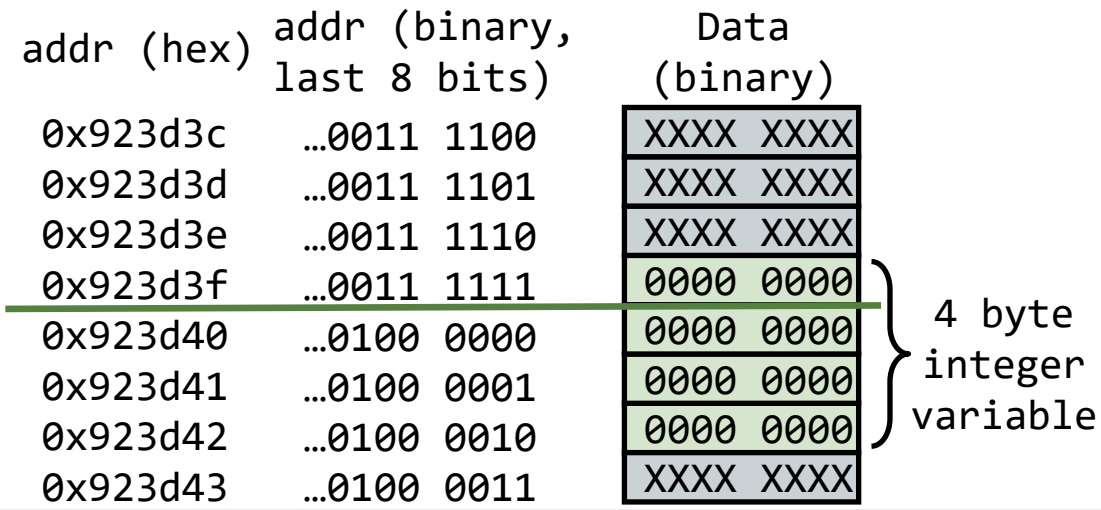
Why Alignment?

- Let's assume there is no alignment requirement
  - i.e., a data structure can start at any address
- E.g., suppose 4-byte integer starts at address 0x923d3f
  - Assume each cache block can hold 64 bytes
  - How many cache blocks do we need to read for this integer?



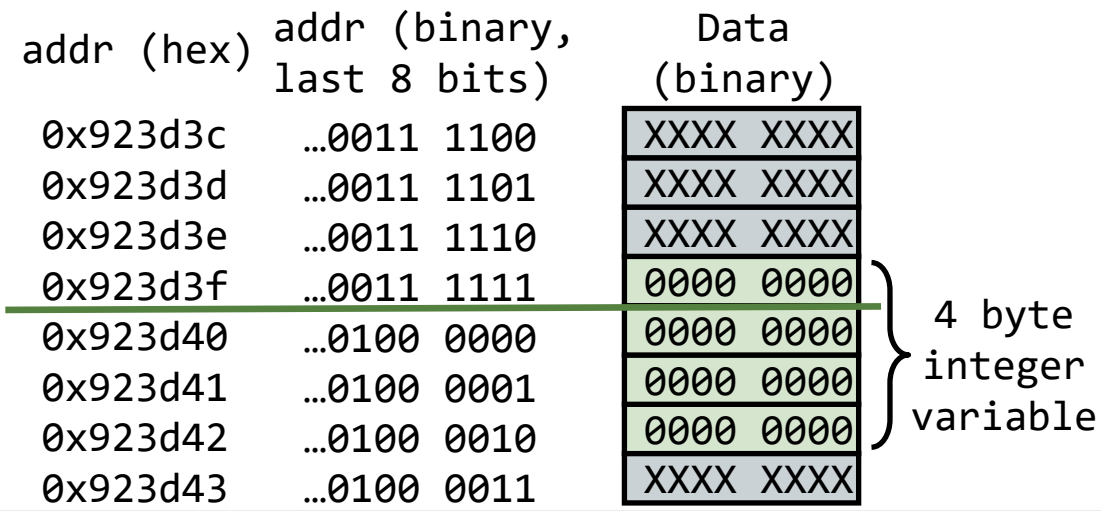
Why Alignment? (Cont.)

- 2 cache blocks!
  - A cache block contains data aligned at cache block size
  - So, starting address of a block has 0 in lower 6 bits (64 bytes)
  - Avoid crossing cache block boundaries for better performance



Why Alignment? (Cont.)

- Similar to cache accesses at 64B granularity, CPU accesses data at WORD granularity
  - When data is not aligned at WORD size, reading a simple data structure (e.g., short, int, pointer, etc.) can take two CPU reads
  - On 32-bit machine, align integer to 4 bytes for good performance
    - I.e., lower 2 bits are 0 (data stored at addresses …00, …01, …10, …11)



How to Align?

- Compilers
  - Insert gaps within structure to ensure correct alignment of fields
- Libraries (e.g., malloc)
  - Return aligned addresses
- Programmer
  - Can use compiler provided alignment directive for efficient access

```
// gcc allocates 6 bytes
struct S { short f[3]; }

// gcc allocates 8 bytes
struct S { short f[3]; } __attribute__ ((aligned (8)));
```

Specific Cases of Alignment

- By Data Type:
  - 1 byte (e.g., `char`)
    - no restrictions on address
  - 2 bytes (e.g., `short`)
    - lowest 1 bit of address is $0_{(2)}$, i.e., 2-byte aligned
  - 4 bytes (e.g., `int`, `float`, etc.)
    - lowest 2 bits of address are $00_{(2)}$, i.e., 4-byte aligned
  - 8 bytes (e.g., `double`)
    - lowest 3 bits of address are $000_{(2)}$, i.e., 8-byte aligned
  - Pointer (e.g., `char *`, `int *`, `void *`)
    - 4 or 8 bytes depending on 32 or 64 bit architecture

Satisfying Alignment of Structures

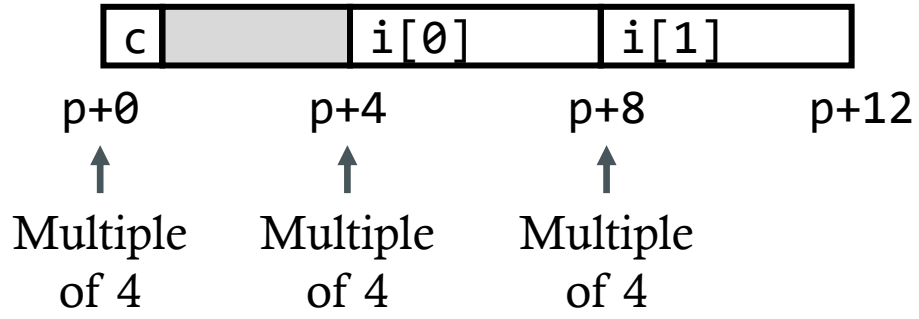
- Within structure
  - Offsets of elements satisfy element's alignment requirement
- Structure placement and size
  - Say largest alignment requirement of any element in structure is K
  - Then starting address and structure length must be multiple of K

Example 1

```
struct S1 {
  char c;
  int i[2];
} *p;
```

1B
4B x 2
} 9B total

Largest alignment
K = 4



12B total considering alignment

Example 2

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

1B
4B x 2
8B

17B total

Largest alignment

K = 8

p+0 Multiple of 8

p+4 Multiple of 4

p+8 Multiple of 4

p+16 Multiple of 8

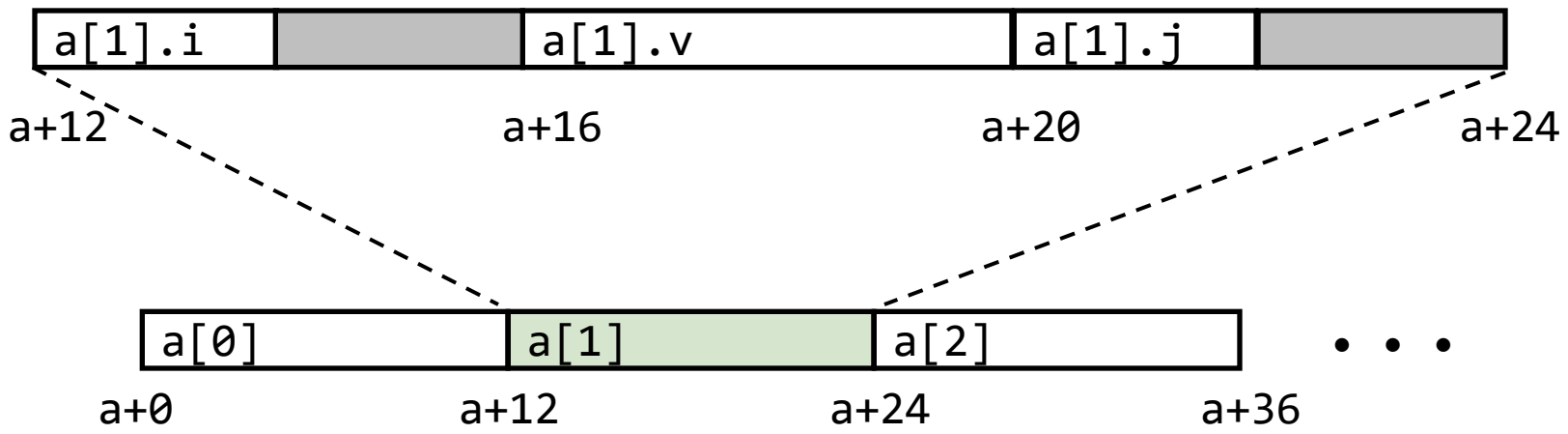
p+24 Multiple of 8

24B total considering alignment

Array of Structures

- Arrays of structures are allocated by repeating allocation for structure type

```
struct S3 {
  short i;
  int v;
  short j;
} a[10];
```

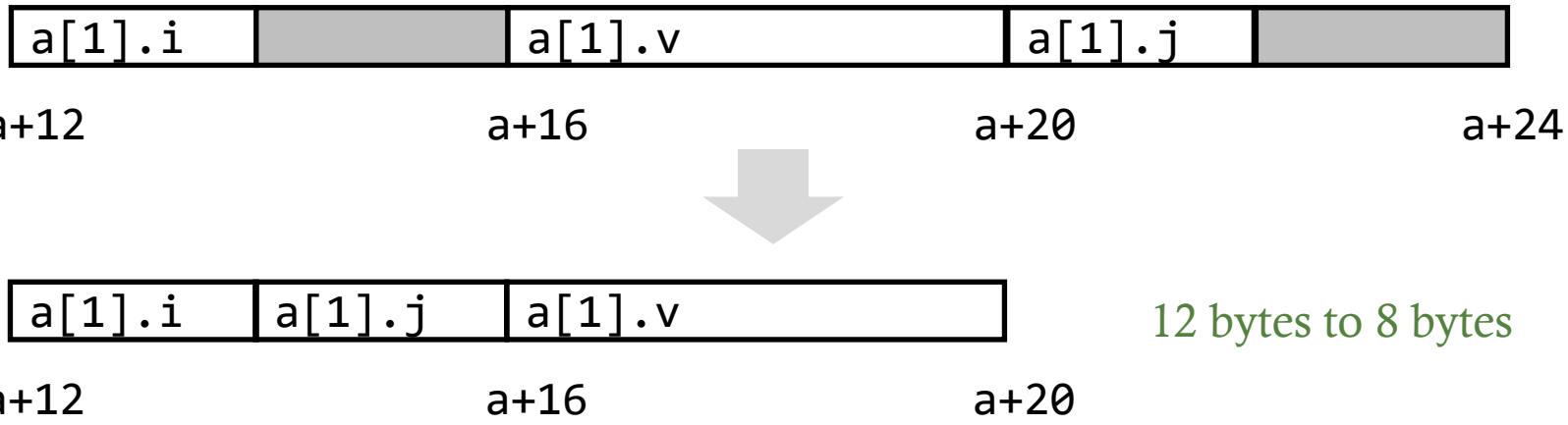


Saving Space

- Does the order of elements matter?

```
struct S3 {
  short i;
  int v;
  short j;
} a[10];
```

```
struct S3 {
  short i;
  short j;
  int v;
} a[10];
```



12 bytes to 8 bytes

- Demo of struct-alignment

Memory Management API

- `#include <stdlib.h>`
- `void *malloc(size_t size)`
  - If successful:
    - Returns a pointer to a memory block of at least size bytes
    - If `size == 0`, returns `NULL (0)`
  - If unsuccessful: returns `NULL` and sets `errno`
    - Note: a well-written program will check for unsuccessful mallocs!
- Typically, malloc returns double-word aligned address
  - 8-byte boundary on 32 bits machine, 16-byte on 64 bits machine
  - Why double word aligned?
- Demo of malloc alignment

Memory Management API

- `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`.

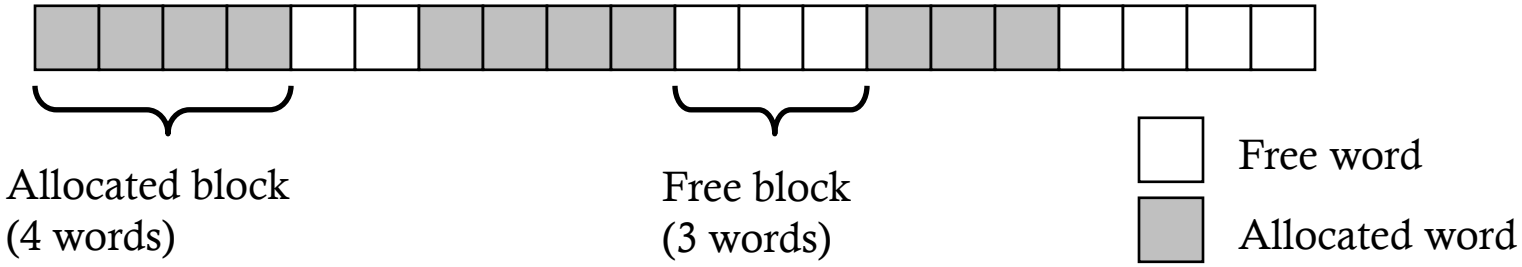
- `void *realloc(void *p, size_t size)`
  - Changes size of block `p` and returns pointer to new block
  - Contents of new block unchanged up to min of old and new size

Malloc Example

```
void foo(int n, int m) {
  int i, *p;

  /* allocate a block of n ints */
  if ((p = (int *) malloc(n * sizeof(int))) == NULL) {
    perror("malloc");
    exit(0);
  }
  for (i=0; i<n; i++)
    p[i] = i;

  /* add m bytes to end of p block */
  if ((p = (int *) realloc(p, (n+m) * sizeof(int))) == NULL) {
    perror("realloc");
    exit(0);
  }
  for (i=n; i < n+m; i++)
    p[i] = i;

  /* print new array */
  for (i=0; i<n+m; i++)
    printf("%d\n", p[i]);

  free(p); /* return p to available memory pool */
}
```

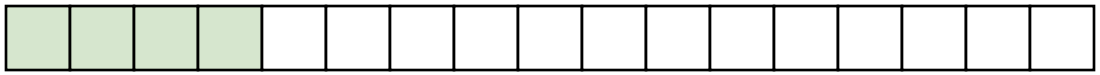
Assumptions

- Assumptions made in this lecture
  - Memory is word addressable (each word can hold a pointer)
  - Malloc returns word-aligned addresses (unless specified otherwise)
    - In practice GNU malloc returns double-word aligned address

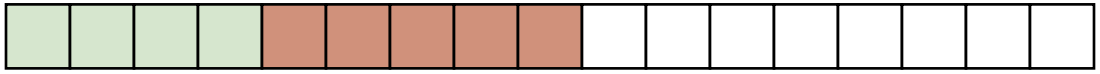


Allocation Examples

```
p1 = malloc(4 * sizeof (void *))
```



```
p2 = malloc(5 * sizeof (void *))
```



```
p3 = malloc(6 * sizeof (void *))
```

```
free(p2)
```

```
p4 = malloc(2 * sizeof (void *))
```

Constraints

- Applications
  - Can issue arbitrary sequence of allocation and free requests
  - Free requests must correspond to an allocated block
- Allocators
  - Must respond immediately to all allocation requests
    - i.e., can't buffer and reorder requests
  - Must allocate blocks from free memory
  - Must align blocks so they satisfy all alignment requirements
  - Can only manipulate and modify free memory
  - Can't move the allocated blocks once they are allocated
    - i.e., compaction is not allowed

Goals of Good malloc/free

- Primary goals
  - Good throughput
    - Ideally, `malloc`, `free` should take constant time (not always possible)
    - Should certainly not take time that is linear in the number of blocks
  - Good memory utilization
    - Malloc allocated structures should be a small fraction of the heap
    - Minimize fragmentation (defined later)
- One extreme example
  - malloc (N): find the next available N free blocks
  - free: do nothing
  - Great time performance, poor space utilization

Performance Goals: Throughput

- Given some sequence of malloc and free requests:
  - $R_0, R_1, ..., R_k, ..., R_{n-1}$
- Want to maximize throughput and peak memory utilization
  - These goals are often conflicting
- 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

- Aggregate payload is denoted by 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
  - A free request will decrease the aggregate payload
- Current (total) heap size is denoted by H_k
- Definition: Peak memory utilization U_k
  - After k requests, peak memory utilization is defined in terms of high watermarks (max values) of P_k and H_k (ranging from 0 to k)
    - $U_k = \max_{0 <= i <= k} (P_i) \ / \ \max_{0 <= j <= k} (H_j)$ (why use high watermarks?)
    - Higher is better

Fragmentation

- Poor memory utilization caused by unusable memory
  - Comes in two forms: internal and external fragmentation
- Internal fragmentation
  - Unutilized space within an allocation, i.e., padding
- External fragmentation
  - Unutilized space in the heap, external to allocations

Internal Fragmentation

```
Assume word size = sizeof (void *) = 4 bytes
p1 = malloc(13)
```

- payload: 13 bytes, returned allocation: 16 bytes, padding = 3
- internal fragmentation = internal fragmentation + 3 bytes
  - Depends only on the pattern of previous requests, easy to measure
- What causes internal fragmentation
  - Minimum size for any allocated block, padding for alignment
- Note: in-use header space affects heap size and thus peak memory utilization, but not internal fragmentation

External Fragmentation

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

```
p1 = malloc(4 * sizeof (void *))
```

```
p2 = malloc(5 * sizeof (void *))
```

```
p3 = malloc(6 * sizeof (void *))
```

```
free(p2)
```

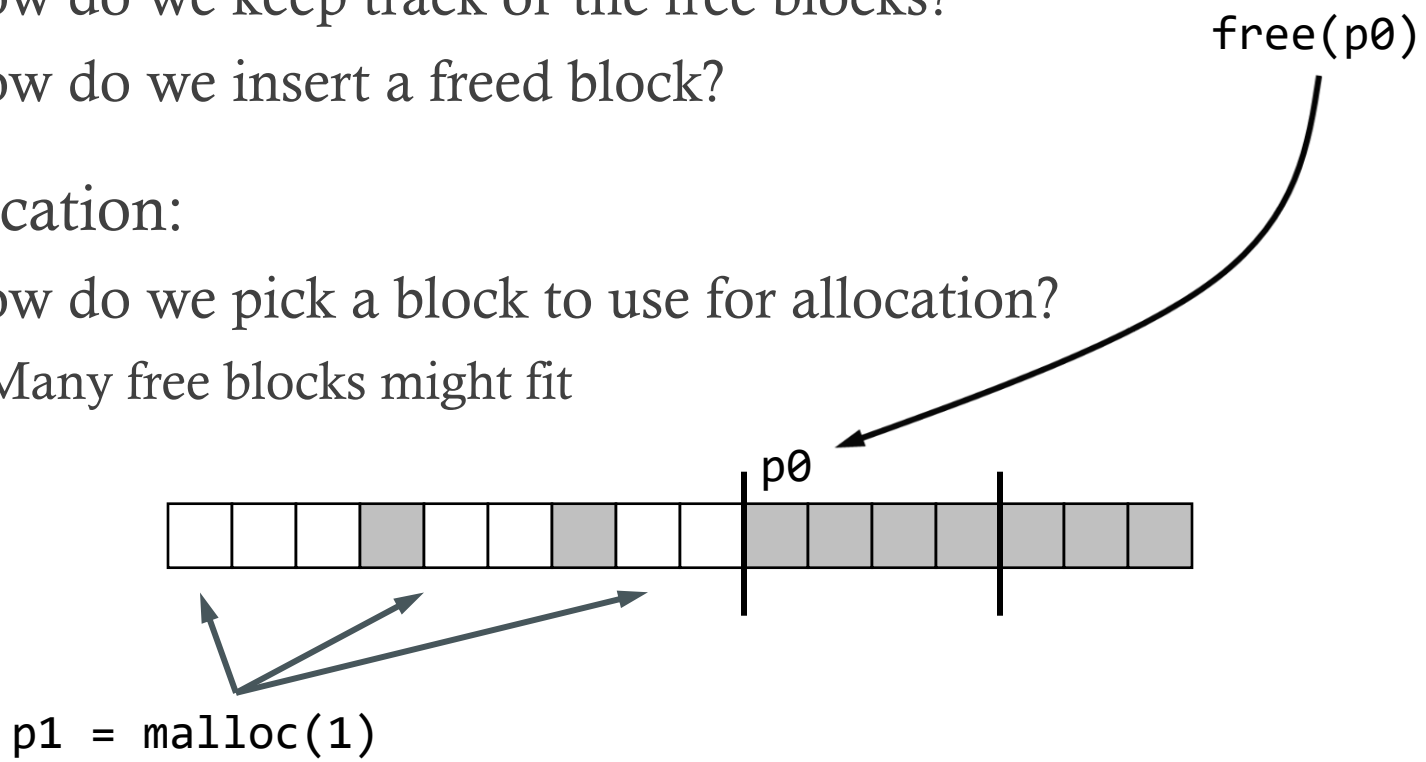
```
p4 = malloc(6 * sizeof (void *))
```
… oops!

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

Basic Dynamic Memory Allocation

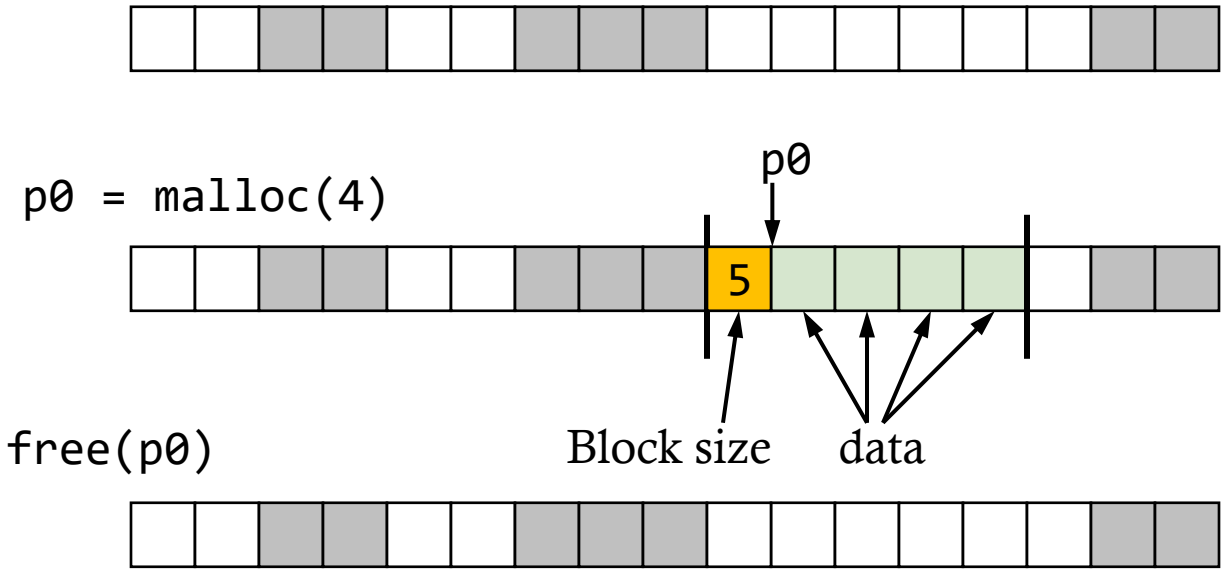
Implementation Issues

- Free:
  - When given a pointer, how much memory to free?
  - How do we keep track of the free blocks?
  - How do we insert a freed block?
- Allocation:
  - How do we pick a block to use for allocation?
    - Many free blocks might fit



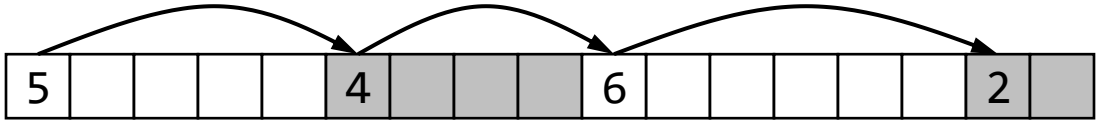
Knowing How Much to Free

- Simplest method
  - Keep the size 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

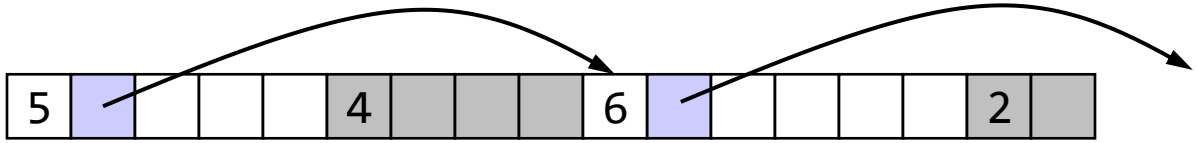


Keeping Track of Free Blocks

- Method 1: Implicit list using size field to links all blocks



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

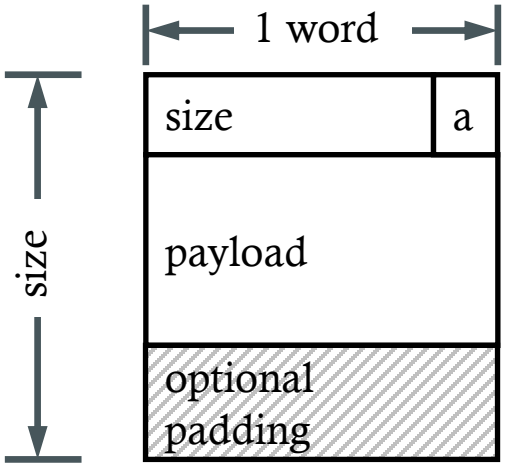


- Method 3: Segregated free list
  - Keep different free lists for different size classes

Method 1: Implicit List

- Need to identify whether each block is free or allocated
  - Use a bit, which can be put in the same word as the size field if block sizes are always multiples of two
  - Mask out low order bit when reading size
    - size = sizeword & ~0x1; // sizeword & 0b1111…1110

Format of allocated and free blocks



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

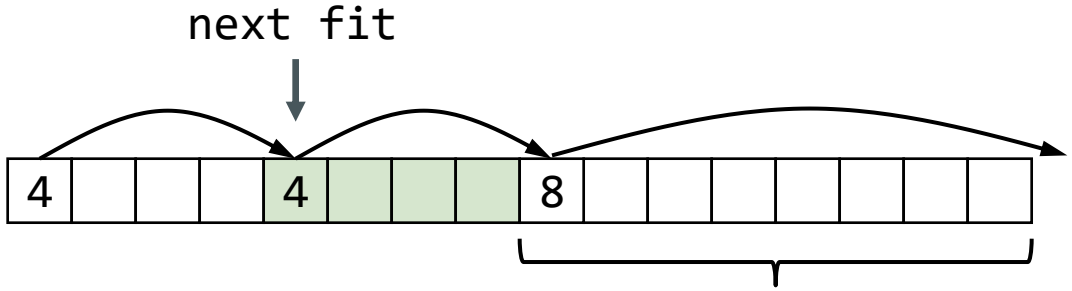
payload: application data in an allocated block

Implicit List: Finding a Free Block

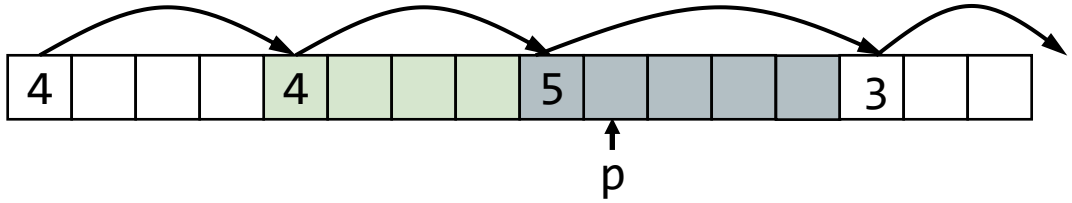
- First fit
  - Search list from beginning, choose first free block that fits
  - Takes linear time in total number of blocks (allocated and free)
  - In practice, may cause "splinters" at beginning of list
- Next fit
  - Like first-fit, but search list from end of previous search
  - Research suggests that fragmentation is worse
- Best fit
  - Search the list, choose the free block with the closest size that fits
  - Keeps fragments small, so usually helps with fragmentation
  - Will typically run slower than first-fit, next fit

Implicit List: Allocation from Free Block

- Allocate a block from a free block
  - Since allocated space might be smaller than free space, we may choose to split the free block

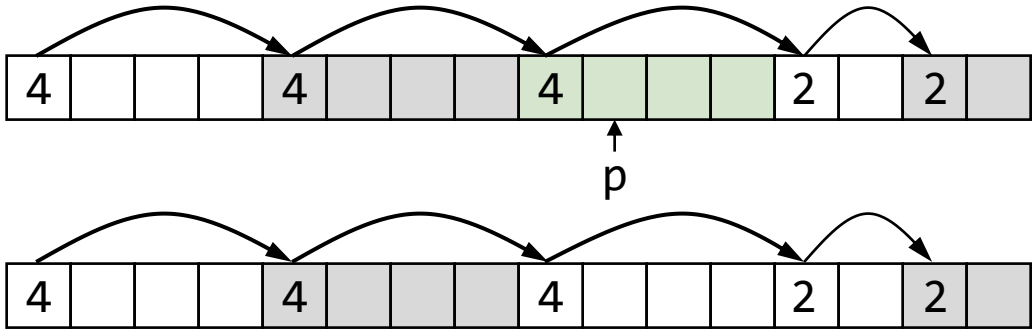



```
malloc(4 * sizeof (void *))
```



Implicit List: Freeing a Block

- Simplest implementation
  - Only need to clear allocated flag

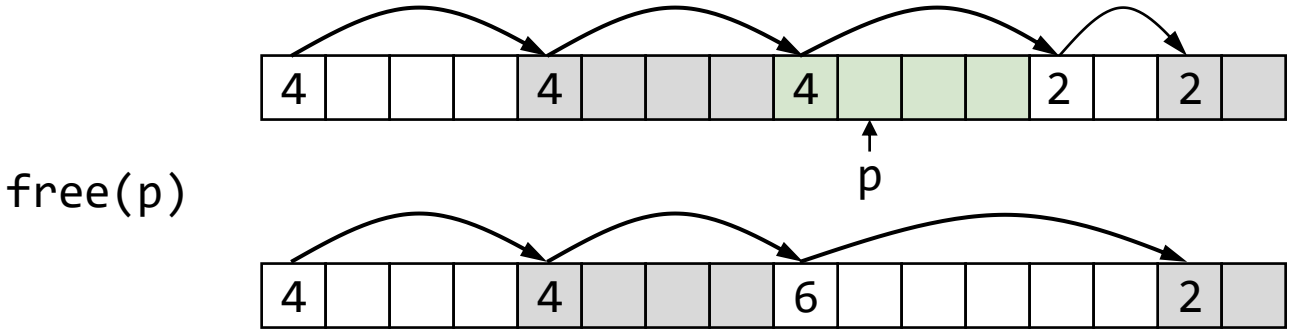


`malloc(5 * sizeof (void *))` Oops!

- Can lead to false external fragmentation
  - There is enough free space, but the allocator can't find it

Implicit List: Coalescing

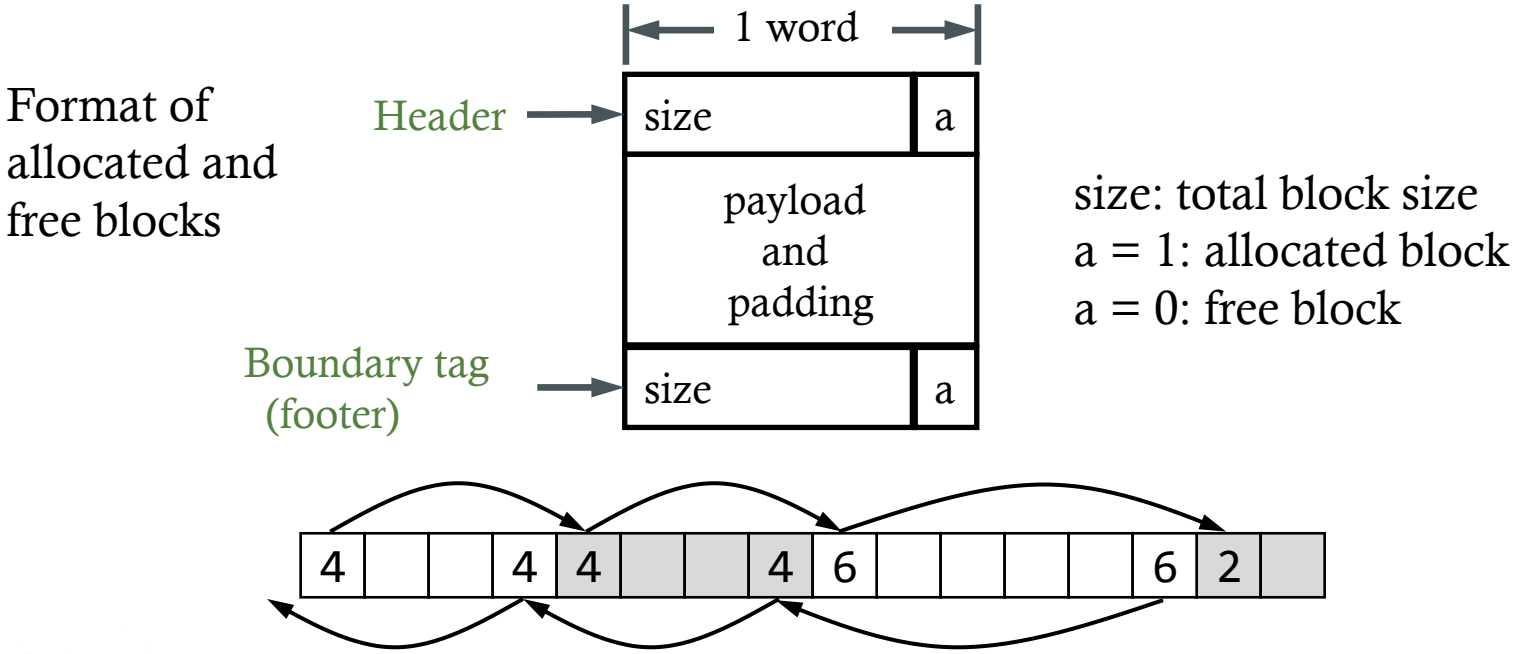
- Join (coelesce) with next and/or previous block if they are free
- Coalescing with next block



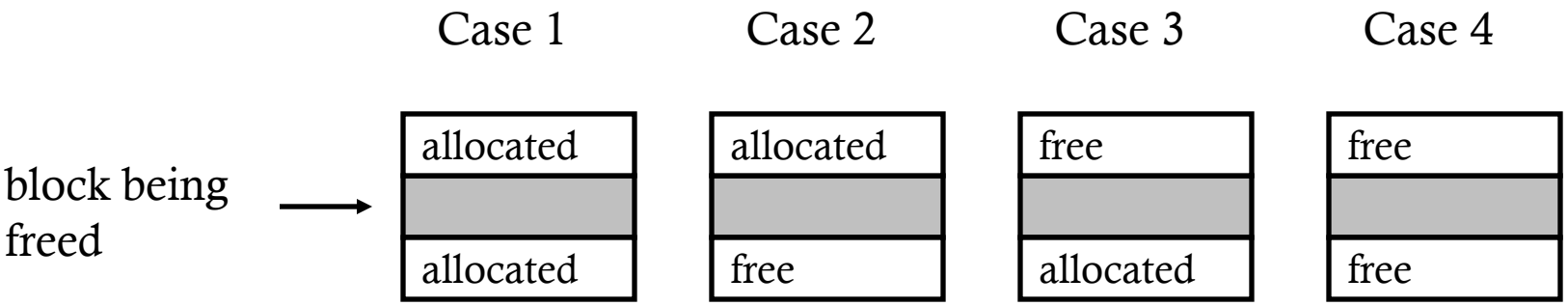
- But how do we coalesce with previous block?

Implicit List: Bidirectional Coalescing

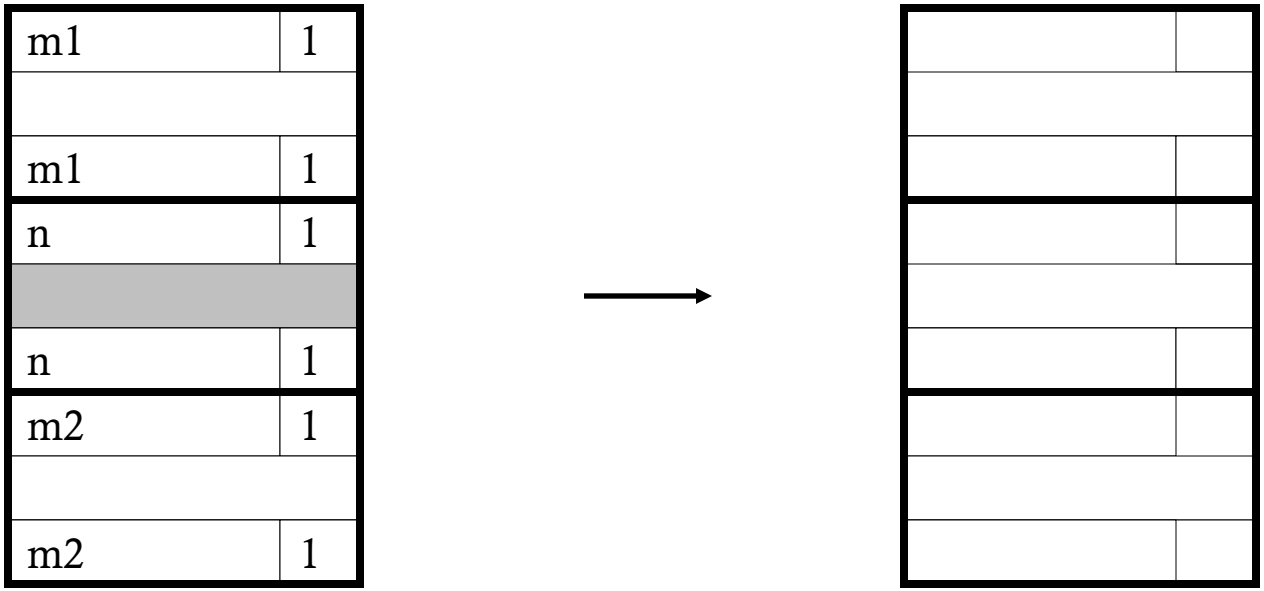
- Boundary tags [Knuth73]
  - Replicate size/allocated word at bottom of free blocks
  - Allows us to traverse "list" backwards, but requires extra space
  - Important and general technique!



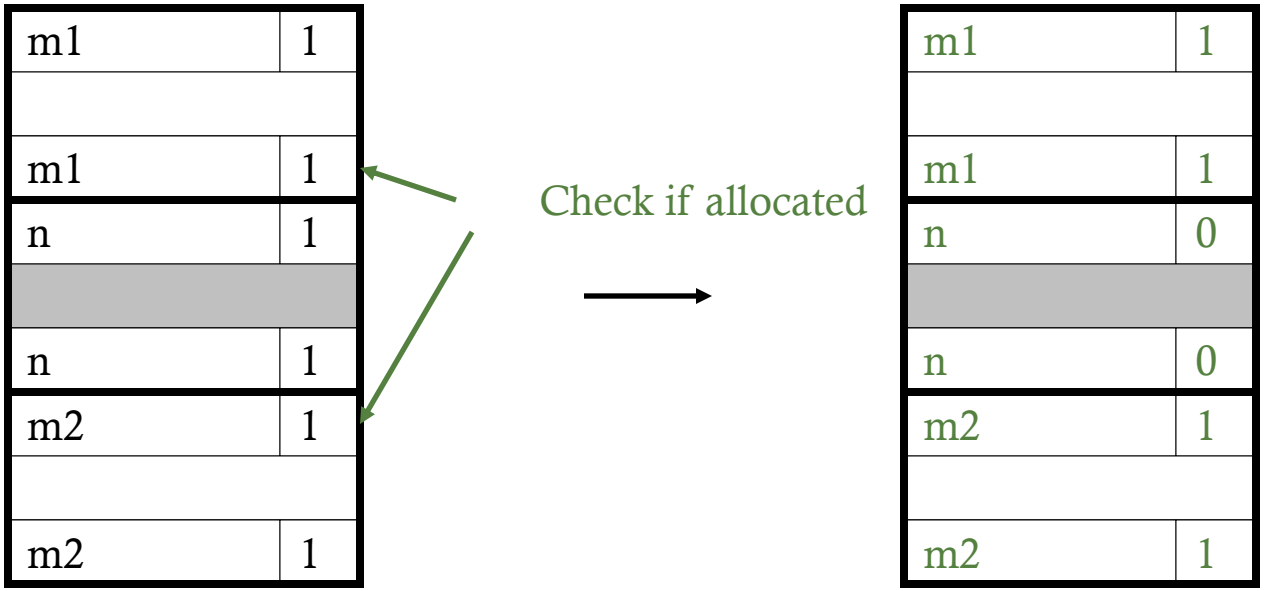
Constant Time Coalescing



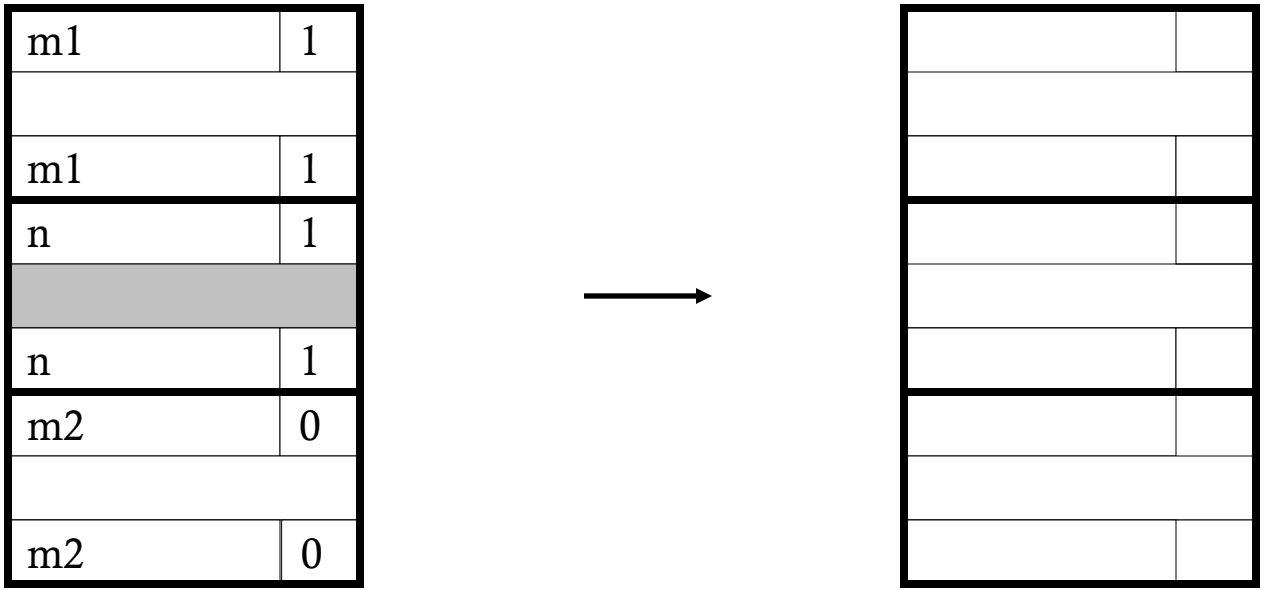
Constant Time Coalescing (Case 1)



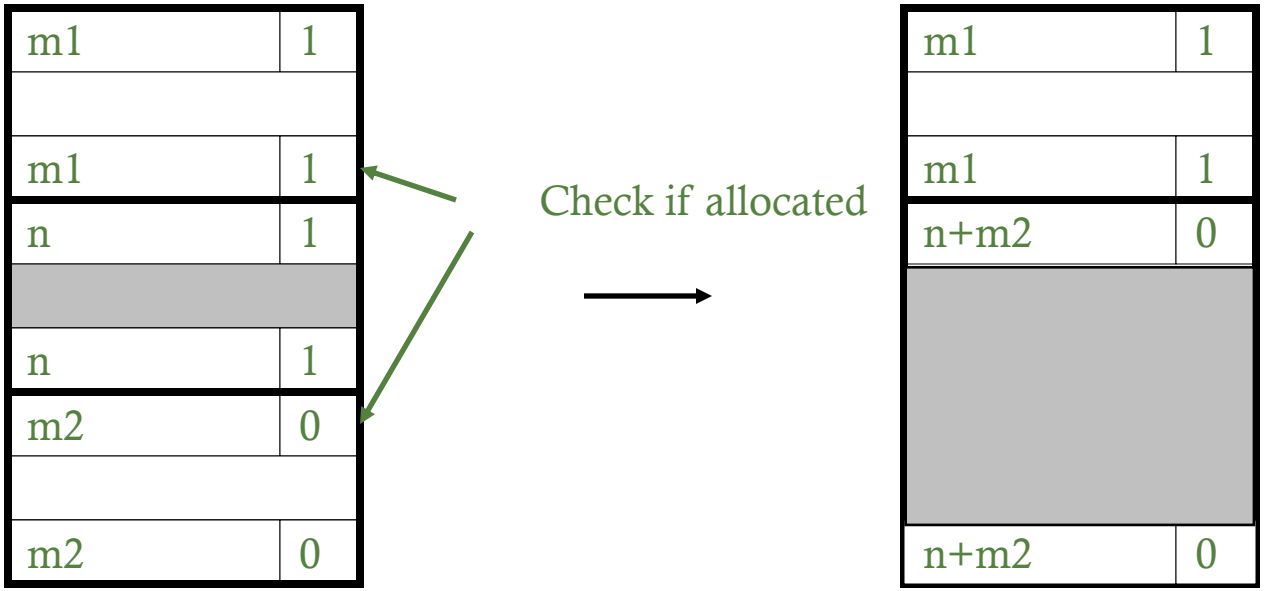
Constant Time Coalescing (Case 1)



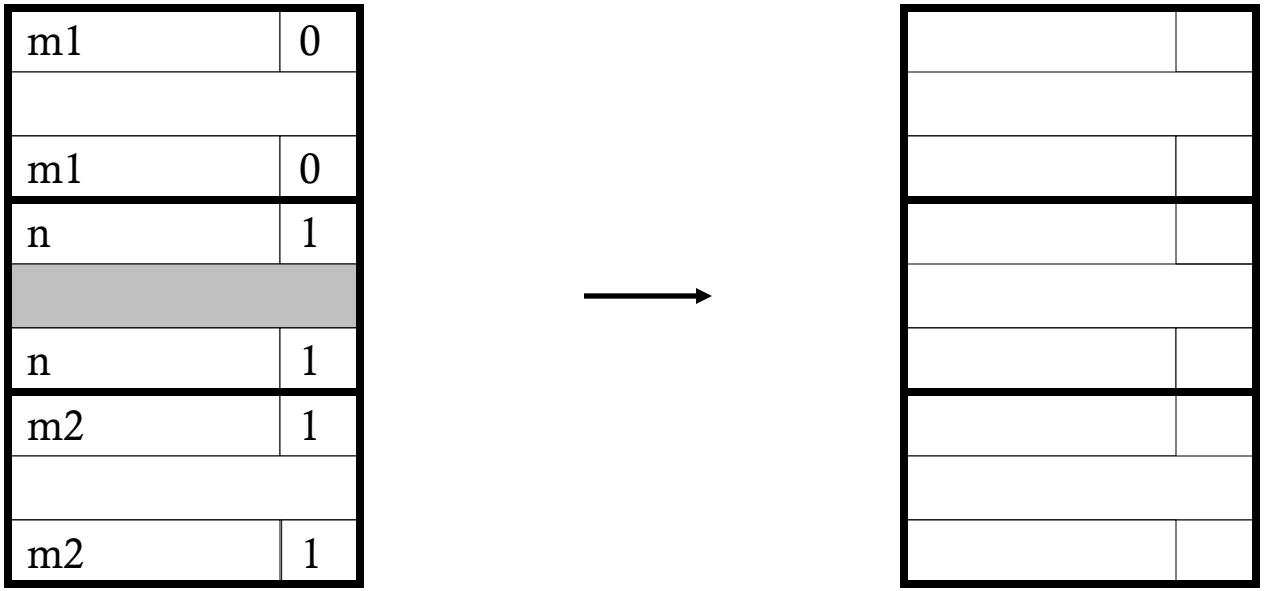
Constant Time Coalescing (Case 2)



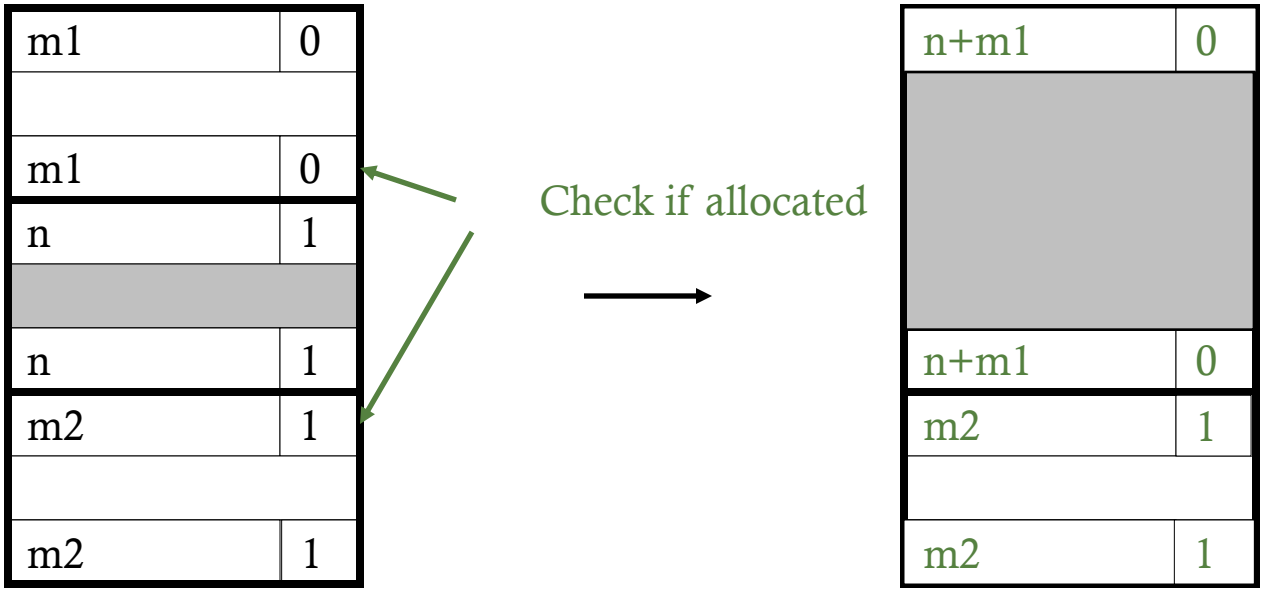
Constant Time Coalescing (Case 2)



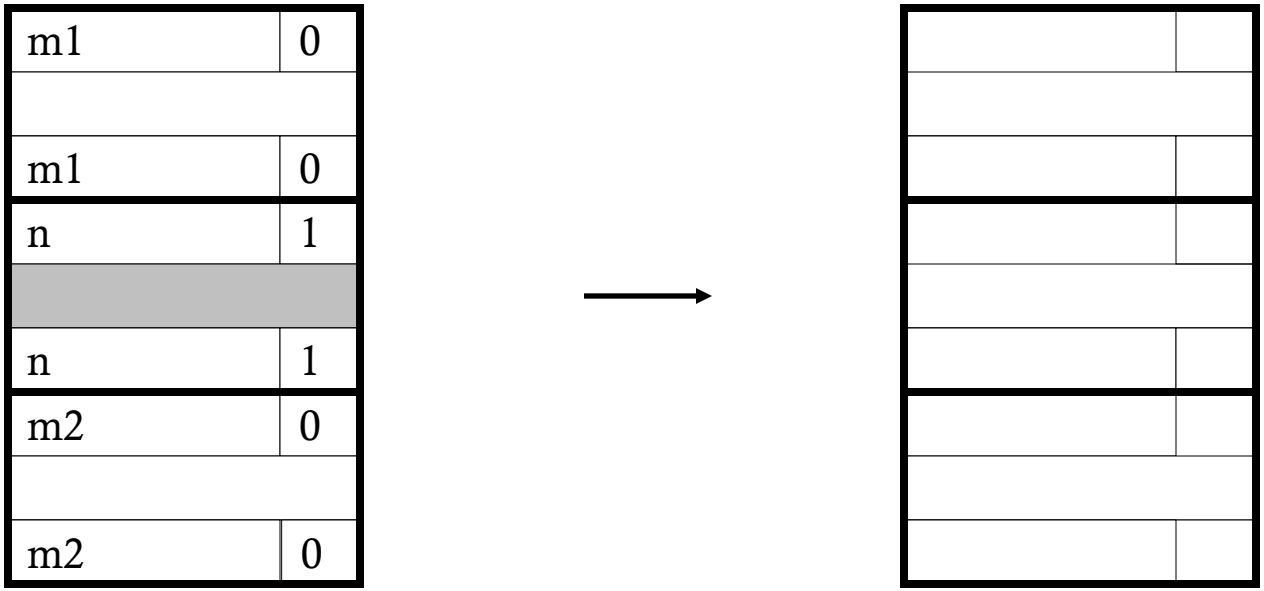
Constant Time Coalescing (Case 3)



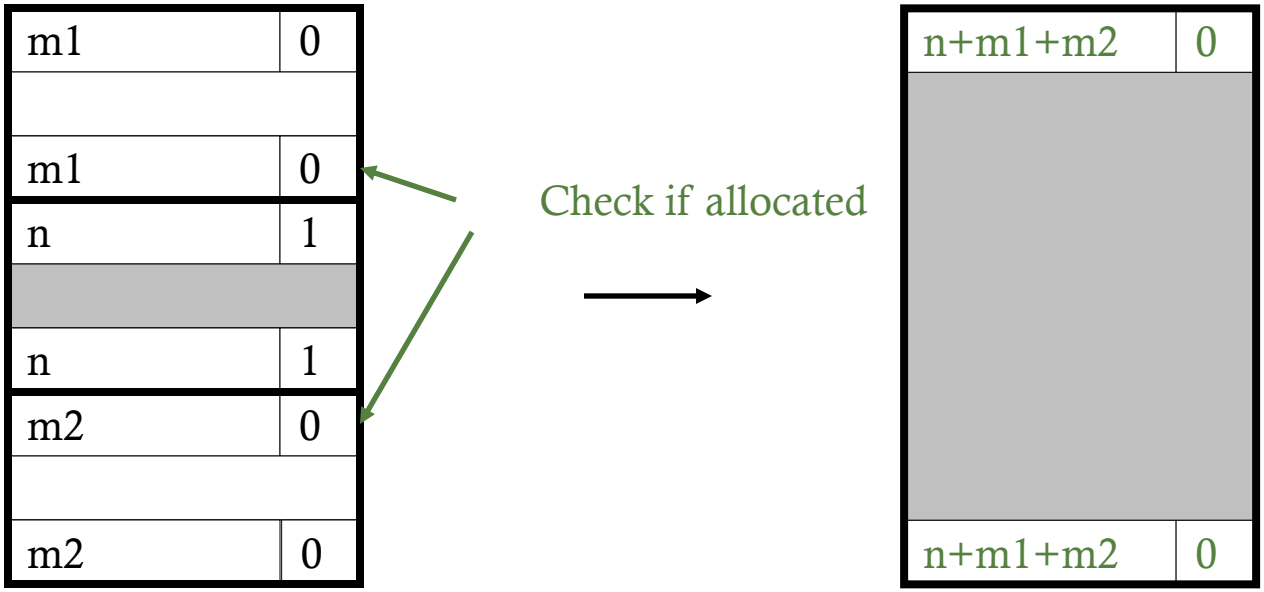
Constant Time Coalescing (Case 3)



Constant Time Coalescing (Case 4)



Constant Time Coalescing (Case 4)



Summary of Key Allocator Policies

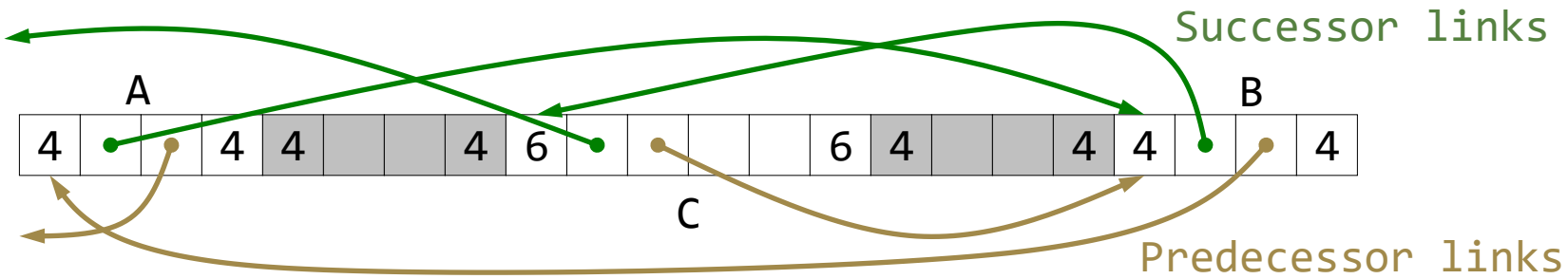
- Placement policy (how to find a free block during allocation):
  - First fit, next fit, best fit, etc.
- Coalescing policy (how to insert a block during free):
  - Immediate coalescing: coalesce adjacent blocks when free is called
  - Deferred coalescing: try to improve performance of free by deferring coalescing until needed
    - Coalesce as you scan the free list for malloc
    - Coalesce when external fragmentation reaches some threshold
    - Why might deferred coalescing be beneficial?

Implicit Lists: Summary

- Implementation: very simple
- Allocation: linear time in # of free and allocated blocks
- Free: constant time in all cases -- even with coalescing
- Memory usage: will depend on placement policy
  - First fit, next fit or best fit
- In practice:
  - Not used by modern allocators because of linear time allocation
  - However, splitting and boundary tag coalescing operations are used by many allocators

Method 2: Explicit List

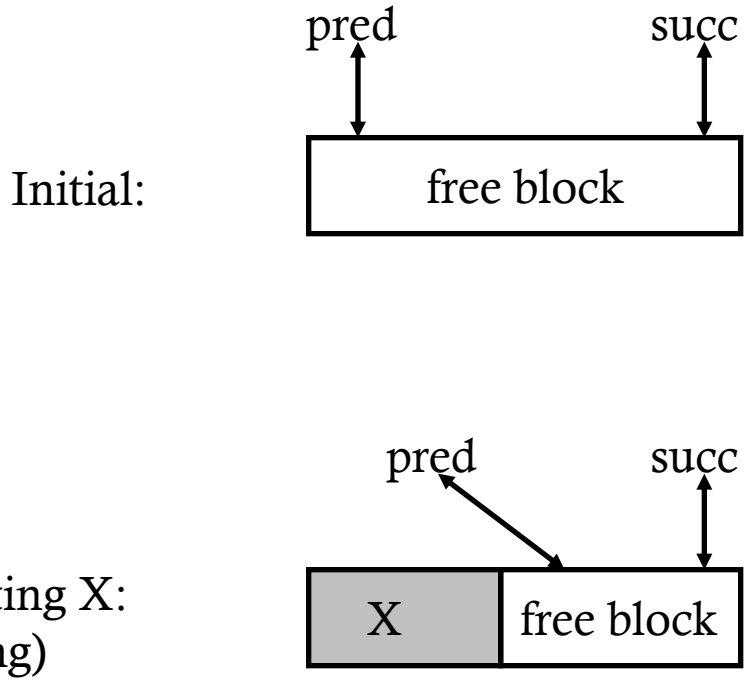
- Explicit list among the free blocks using pointers within the free blocks



- Use space in free regions for link pointers
  - Typically, doubly linked

  - Links can point anywhere, not necessarily to adjacent block
- Use boundary tags for constant-time coalescing of free blocks

Allocating From Explicit Free List



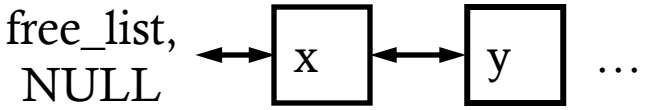
Allocation time is linear in the number of free blocks instead of total blocks

Freeing With Explicit Free List

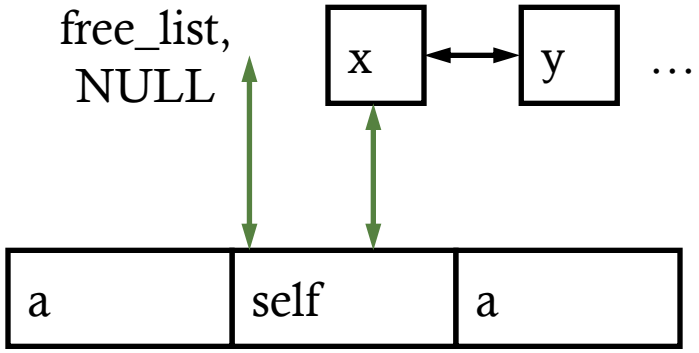
- Where should a freed block be inserted in free list?
  - LIFO (last-in-first-out) policy
    - Insert freed block at the beginning of the free list
      - ie., Latest block to be freed may be next one to be allocated
    - Pros: simple and constant time
    - Cons: studies suggest fragmentation is 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 for insertion
    - Pro: studies suggest fragmentation is better than LIFO

Freeing With a LIFO Policy

- Details:
  - a=allocated, f=freed
  - Assume free(self) in each example
  - Initially:
    - `free_list = x, x.pred = NULL`



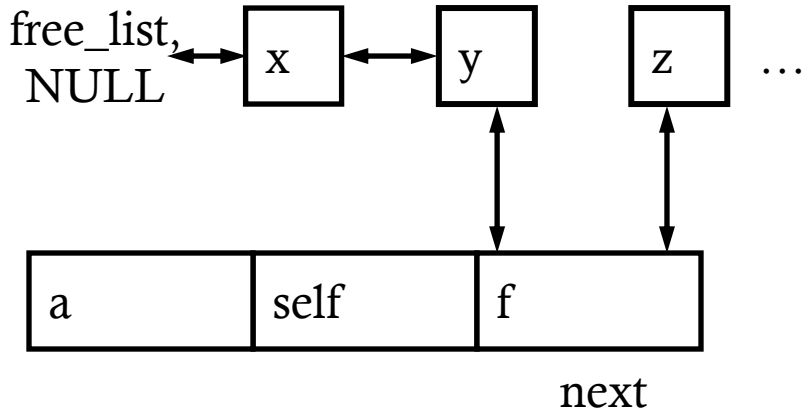
- Connect to head of free list:
  - `self.succ = free_list;`
  - `free_list.pred = self`
  - `free_list = self;`
  - `self.pred = NULL;`

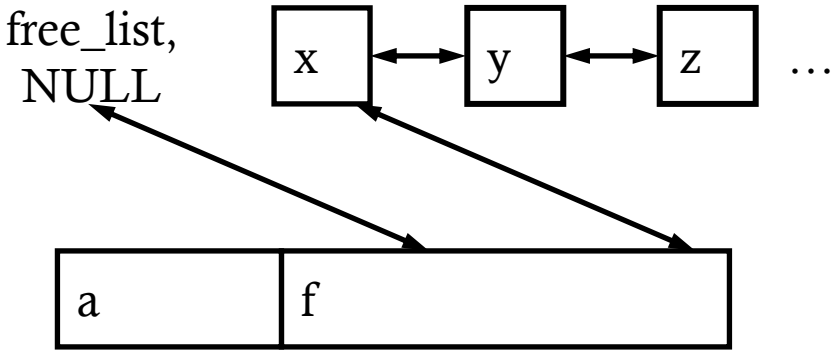


- How to coalesce?

LIFO: Coalescing

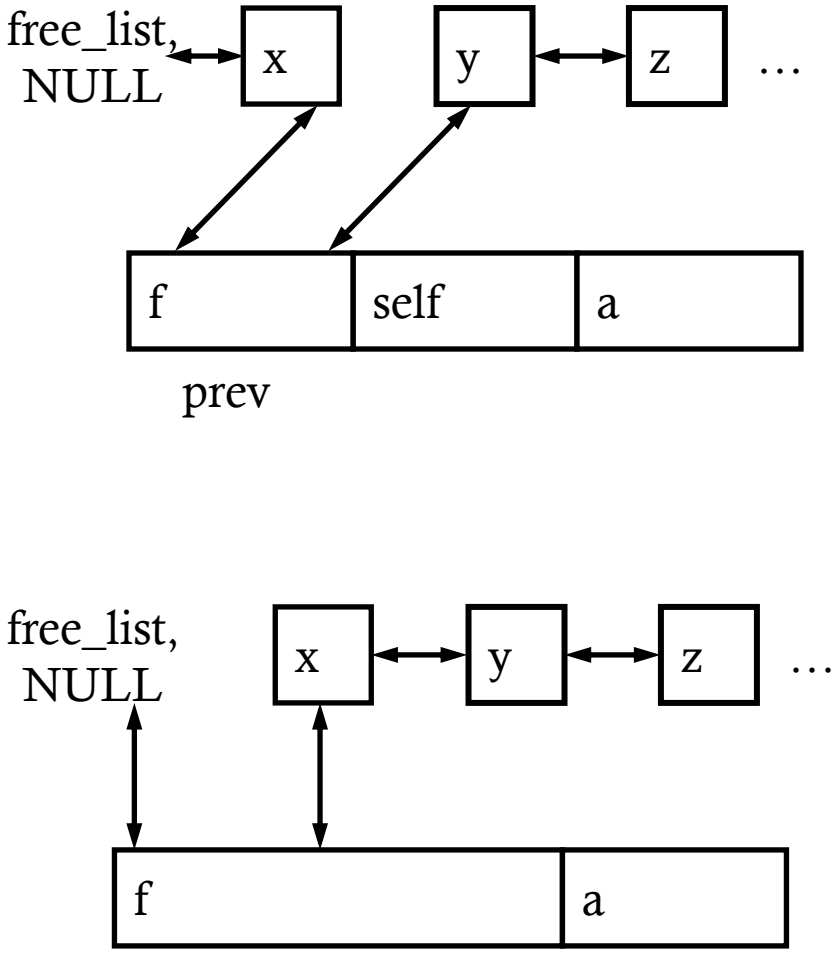
- Case 2: a-self-f
  - Splice out next, coalesce self and next, add to beginning of free list





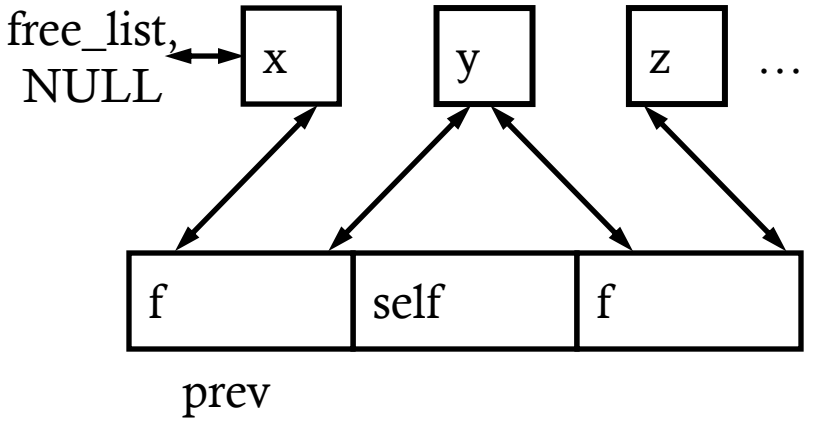
LIFO: Coalescing

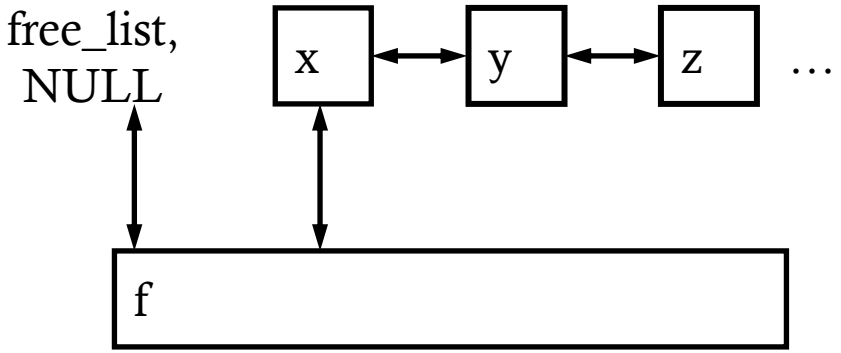
- Case 2: f-self-a
  - Splice out prev, coalesce self and prev, add to beginning of free list



LIFO: Coalescing

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



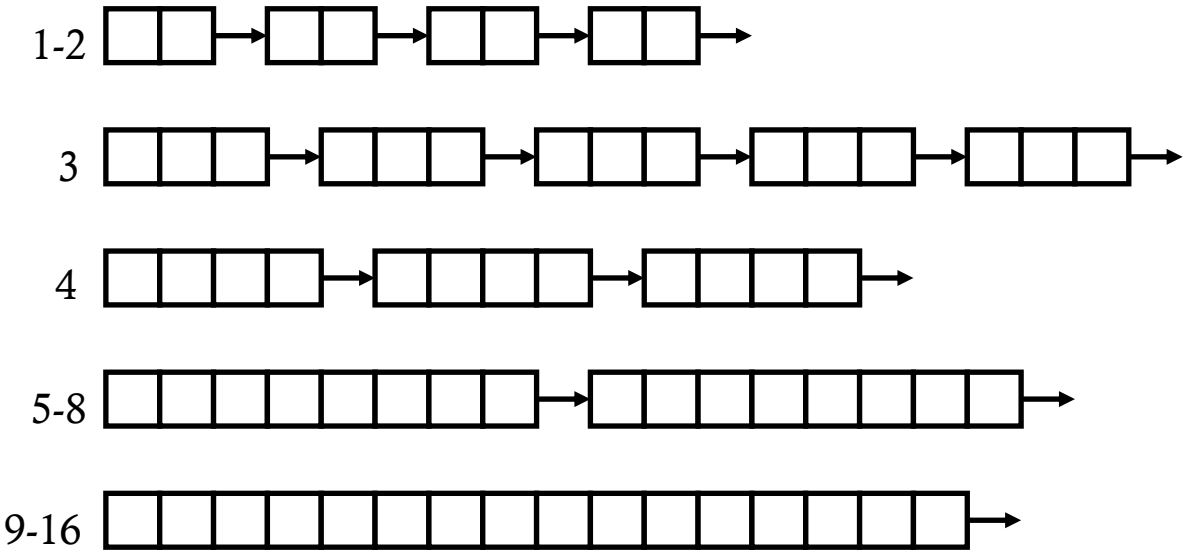


Explicit List Summary

- Comparison with implicit list
  - Allocation takes linear time in number of free blocks instead of total blocks
    - Much faster allocation when most of the memory is full
  - Slightly more complicated allocation and free since blocks need to be spliced in and out of the free list
- Main use of linked lists is with segregated free lists
  - Keep multiple linked lists of different size classes, or possibly for different types of objects (discussed next)

Method 3: Segregated Free List

- Each size class has its own collection of blocks



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

Simple Segregated Storage

- All blocks in a list have the SAME size N
- A bloc is allocated to a request of size in the range (M, N], where M is the block size in the previous list
- To allocate a block of size N
  - If free list for size N is not empty:
    - Allocate first block on list, no splitting required
  - If free list for size N is empty:
    - Grow heap, create new free blocks of size N from new heap space, add these blocks to free list, then allocate first block on list
- To free a block
  - Add the block to its free list

Simple Segregated Storage

- Advantages:
  - Constant time allocation and free
  - With same-sized blocks in each list:
    - No splitting or coalescing required
    - Low per-block memory overhead
      - Block size need not be maintained in the header (discussed later)
- Disadvantages:
  - Can lead to internal fragmentation
    - Since allocation is rounded up to next size
  - Can lead to high external fragmentation
    - Free blocks in a list cannot be used for other allocations
    - Blocks aren't coalesced

Segregated Best-Fit

- All blocks in a list lie within a size range
  - Blocks within the list can have different block sizes
- To allocate a block of size N
  - Search appropriate free list for block of size M > N
  - If an appropriate block is found:
    - (Optionally) split block and place fragment on appropriate size free list
  - If no block is found:
    - Try next larger class, repeat until block is found in a larger class
  - If block still not found, grow heap
- To free a block:
  - Coalesce and place on appropriate list for its new size

Segregated Best-Fit

- Advantages
  - Controls fragmentation of simple segregated storage
    - Mainly due to splitting and coalescing
    - Fragmentation similar to best fit
  - Faster than unsegregated best-fit
    - Doesn't require exhaustive search
- Tradeoffs
  - Slower allocation than segregated storage
  - Splitting and coalescing can increase search times
    - Deferred coalescing can help

Binary Buddy Allocator

- Variant of segregated best fit
  - Each list has fixed size blocks, block size is a power of 2
- `void *allocate(size)`
  - Round up a request size to 2^n size
  - If free block of that size is not available:
    - Find a larger block, recursively split it in half until block is available
- `free(p)`
  - Find address of buddy block by flipping bit for rounded size in the returned block address
  - Search for buddy in free list of that size, if found, coalesce and recursively repeat

Buddy Allocator Example

1. initial allocator state
2. a = allocate(34K);
3. b = allocate(66K);
4. c = allocate(35K);
5. d = allocate(67K);
6. free(b);
7. free(d);
8. free(a);
9. free(c);

Step	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K
1	2^4															
2.1	2^3								2^3							
2.2	2^2				2^2				2^3							
2.3	2^1		2^1		2^2				2^3							
2.4	2^0	2^0	2^1		2^2				2^3							
2.5	A: 2^0	2^0	2^1		2^2				2^3							
3	A: 2^0	2^0	B: 2^1		2^2				2^3							
4	A: 2^0	C: 2^0	B: 2^1		2^2				2^3							
5.1	A: 2^0	C: 2^0	B: 2^1		2^1		2^1		2^3							
5.2	A: 2^0	C: 2^0	B: 2^1		D: 2^1		2^1		2^3							
6	A: 2^0	C: 2^0	2^1		D: 2^1		2^1		2^3							
7.1	A: 2^0	C: 2^0	2^1		2^1		2^1		2^3							
7.2	A: 2^0	C: 2^0	2^1		2^2				2^3							
8	2^0	C: 2^0	2^1		2^2				2^3							
9.1	2^0	2^0	2^1		2^2				2^3							
9.2	2^1		2^1		2^2				2^3							
9.3	2^2				2^2				2^3							
9.4	2^3								2^3							
9.5	2^4															

Finding d's buddy in Step 7:

```
addr(d)          = 256K = 0x100 0000 0000 0000 0000
sizeof(d)        = 128K = 0x010 0000 0000 0000 0000
addr(d's buddy)  = 384K = 0x110 0000 0000 0000 0000
```

Other Considerations

- Allocation patterns
- Allocation data structures
  - Lists
  - Other structures

Allocation Patterns

- Block lifetimes are not random
  - Ramp – allocations throughout program lifetime without releases
  - Plateau – allocations, then lengthy usage, then releases
  - Peaks – bursty behavior and short object lifetimes
- Block sizes are not random
  - Zorn and Grunwald, 1992 study, six allocation-heavy C programs
  - Found that 53-93% of requests were for top two sizes
- Allocator can attempt to exploit patterns
  - Allocate blocks with similar lifetimes contiguously
  - Allocate blocks with same/similar object sizes contiguously

Linked Lists for Free Blocks

- We have seen linked list(s) of variable sized free blocks
  - Implicit – link allocated and free blocks
    - Not used due to linear time allocation
  - Explicit – link free blocks, use one or more lists
    - More commonly used
- Where is the list stored?
  - Integrated: use space within the free blocks to hold the links
    - Benefit: no need to separately manage space for links
    - Problem: poor locality when traversing the list (discussed later)
  - External: use space separate from allocated or free blocks
    - Benefit: better locality when traversing the list
    - Problem: need to manage this space, how is it grown (discussed later)

Linked Lists for Free Blocks

- What should be the order of free blocks in the list?
  - LIFO
    - Add freed block to beginning of list
    - Provides locality
  - FIFO
    - Add freed block to end of list
    - Benefits?
  - Sorted by block size
    - Limits traversal for smaller allocations
  - Sorted by address
    - Reduces heap fragmentation (we will see this later)

Other Data Structures for Free Blocks

- Single pointer for a region/arena
  - When related blocks can be released all at once
  - Use mmap to allocate large regions and maintain regions in list
  - No need to keep a free list within a region or to use headers in the allocated/free blocks
  - Allocate blocks by incrementing a single pointer
  - Release entire region when done

Other Data Structures for Free Blocks

- Bitmap for fixed-size contiguous blocks
  - Can be used for segregated storage
    - Each list maintains blocks of the same size
    - Blocks of the same size must be allocated contiguously
- Trees
  - Heap requires searching for a free block of a given size
  - Use ordered trees to reduce search times compared to linked list
    - E.g., use red-black tree to perform best fit in `log(n)` time, where `n` is number of free blocks