CS61: Systems Programming and Machine Organization
Harvard University, Spring 2008

Lecture 12:

Dynamic Memory Allocation 1:
Into the jaws of malloc()

Prof. Matt Welsh

March 11, 2008

Topics for today

- Dynamic memory allocation
- Implicit vs. explicit memory management
- Performance goals
- Fragmentation
- Free block list management
- Free block coalescing

Harsh Reality: Memory Matters!

Memory is not unlimited!

- It must be allocated and managed
- Many applications are memory dominated
  - *Especially those based on complex, graph algorithms*

Memory referencing bugs especially pernicious

- Effects are distant in both time and space

Memory performance is not uniform

- Cache and virtual memory effects can greatly affect program performance
- Adapting program to characteristics of memory system can lead to major speed improvements

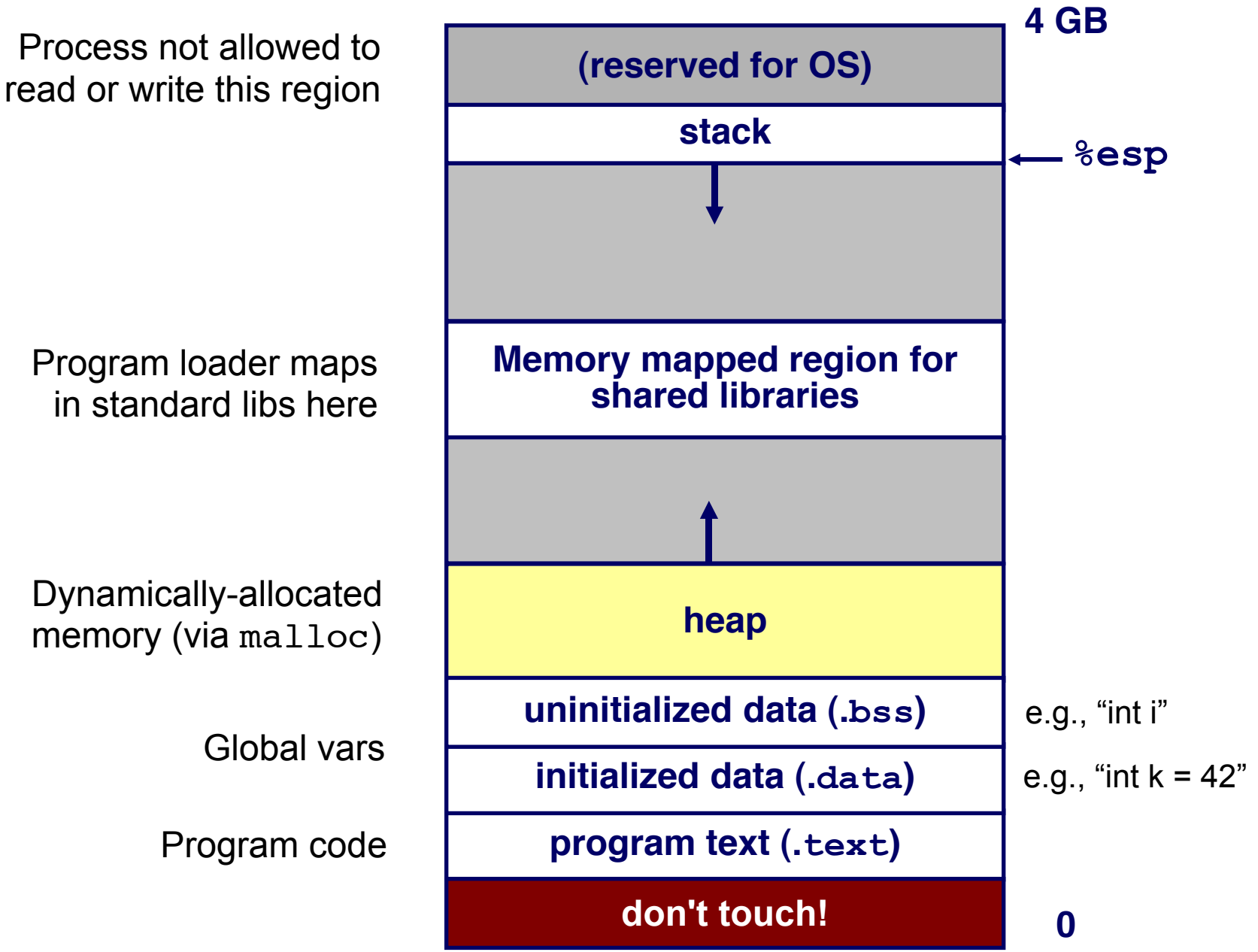
Dynamic Memory Management

- There are two broad classes of memory management schemes:
- Explicit memory management
  - Application code responsible for both explicitly **allocating** and **freeing** memory.
  - Example: `malloc()` and free()
- Implicit memory management
  - Application code can allocate memory, but **does not free memory explicitly**
  - Rather, rely on **garbage collection** to "clean up" memory objects no longer in use
- Advantages and disadvantages of each?

Dynamic Memory Management

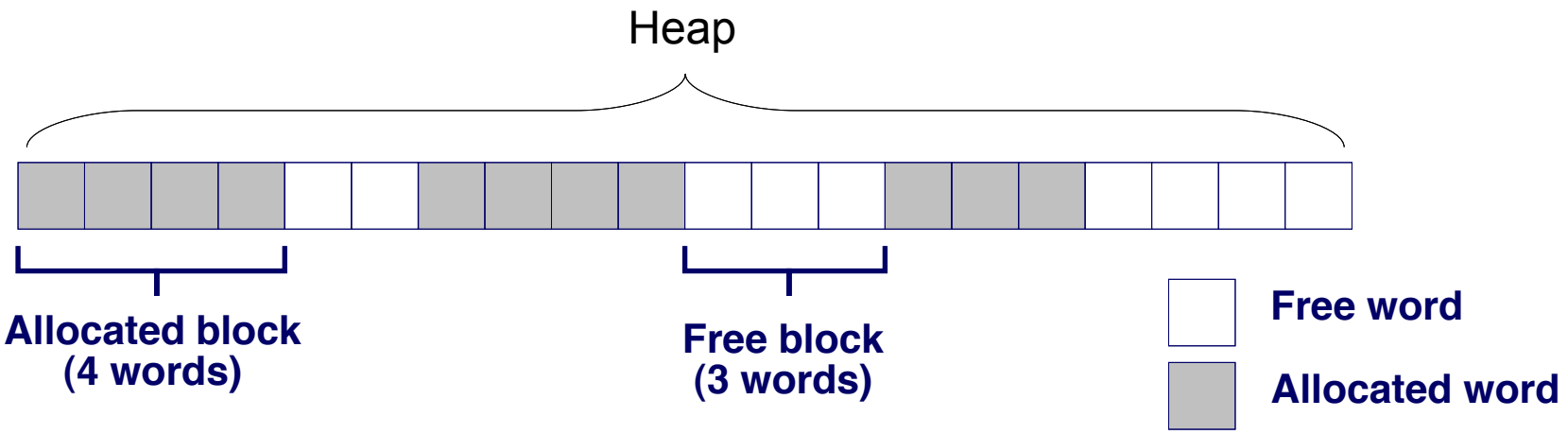
- There are two broad classes of memory management schemes:
- Explicit memory management
  - Application code responsible for both explicitly **allocating** and **freeing** memory.
  - Example: `malloc()` and free()
- Implicit memory management
  - Application code can allocate memory, but **does not free memory explicitly**
  - Rather, rely on **garbage collection** to "clean up" memory objects no longer in use
- Advantages and disadvantages of each?
  - Explicit management: Application has control over everything, possibly faster
  - Implicit management: Application can seriously screw things up
    - *Attempt to access a freed block*
    - *Freeing same block multiple times*
    - *Forgetting to free blocks (**memory leak**)*

A process's view of memory



The heap

The heap



The **heap** is the region of a program's memory used for dynamic allocation.

Program can allocate and free blocks of memory within the heap.

Heap starts off with a fixed size (say, a few MB).

The heap can grow in size, but never shrinks!

- Program can grow the heap if it is too small to handle an allocation request.
- On UNIX, the sbrk() system call is used to expand the size of the heap.
  - *Why doesn't it make sense to shrink the heap?*

Malloc Package

```
#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.*
  - *If `size == 0`, returns NULL*
- 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.

Malloc Example

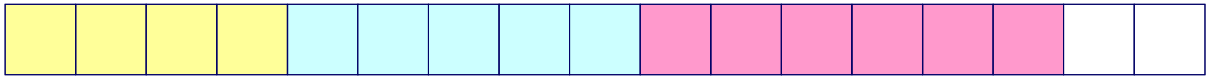
```
void foo(int n, int m) {
  int i, *p;

  /* allocate a block of n ints */
   p = (int *)malloc(n * sizeof(int));
   if (p == 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 */
}
```

Allocation Examples

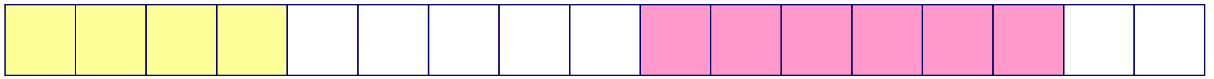
```
p1 = malloc(4)
```

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

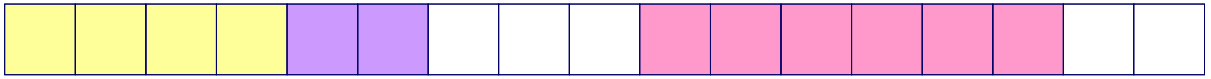
```
p3 = malloc(6)
```



```
free(p2)
```



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



Constraints

Applications:

- 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 requested 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 only place allocated blocks in free memory*
- Must align blocks so they satisfy all alignment requirements
  - *8 byte alignment for GNU malloc (`libc` malloc) on Linux boxes*
- Can only manipulate and modify free memory
- Can't move the allocated blocks once they are allocated
  - *i.e., compaction is not allowed*

Performance Goals: Allocation overhead

Want our memory allocator to be fast!

- Minimize the overhead of both allocation and deallocation operations.

One useful metric is **throughput**:

- Given a series of allocate or free requests
- Maximize the 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.

Note that a fast allocator may not be efficient in terms of memory utilization.

- Faster allocators tend to be “sloppier”
- To do the best job of space utilization, operations must take more time.
- Trick is to balance these two conflicting goals.

Performance Goals: Memory Utilization

Allocators rarely do a perfect job of managing memory.

- Usually there is some “waste” involved in the process.

Examples of waste...

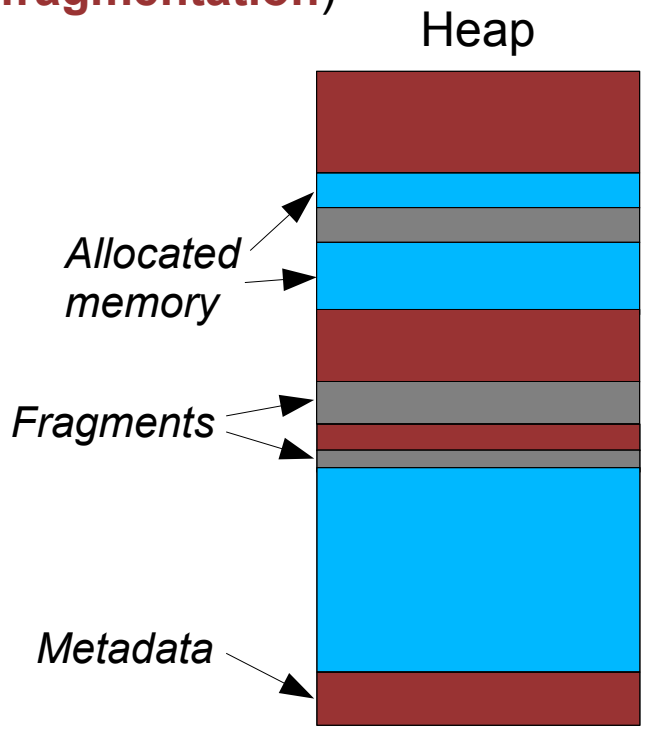
- Extra metadata or internal structures used by the allocator itself (example: Keeping track of where free memory is located)
- Chunks of free memory that are too small to be useful (**fragmentation**)

We define **memory utilization** as...

- The **total amount of memory allocated to the application** divided by the total **heap size**

Ideally, we'd like utilization to be to 100%

- In practice this is not possible, but would be good to get close.



Conflicting performance goals

Note that good throughput and good utilization are difficult to achieve simultaneously.

A fast allocator may not be efficient in terms of memory utilization.

- Faster allocators tend to be “sloppier” with their memory usage.

Likewise, a space-efficient allocator may not be very fast

- To keep track of memory waste (i.e., tracking fragments), the allocation operations generally take longer to run.

Trick is to balance these two conflicting goals.

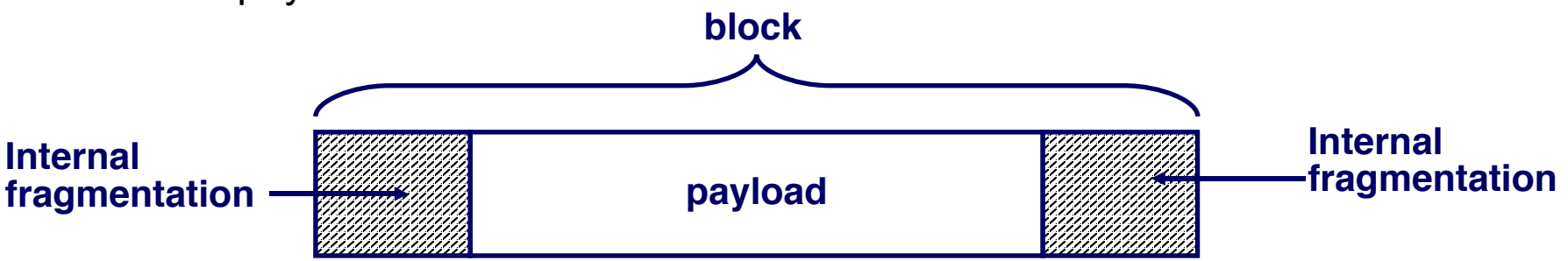
Internal Fragmentation

Poor memory utilization caused by **fragmentation.**

- Comes in two forms: **internal** and **external** fragmentation

Internal fragmentation

- **Internal fragmentation** is the difference between the block size and the payload size.



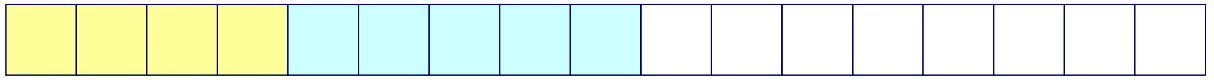
- Caused by overhead of maintaining heap data structures, padding for alignment purposes, or the policy used by the memory allocator
- Example: Say the allocator always “rounds up” to next highest power of 2 when allocating blocks.
  - *So malloc(1025) will actually allocate 2048 bytes of heap space!*

External Fragmentation

Occurs when there is enough aggregate heap memory, but no single free block is large enough to satisfy a given request.

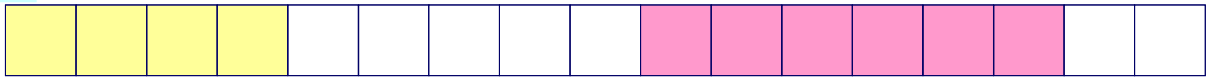
```
p1 = malloc(4)
```

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



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

```
free(p2)
```



```
p4 = malloc(6)
```

oops! - no free block large enough.

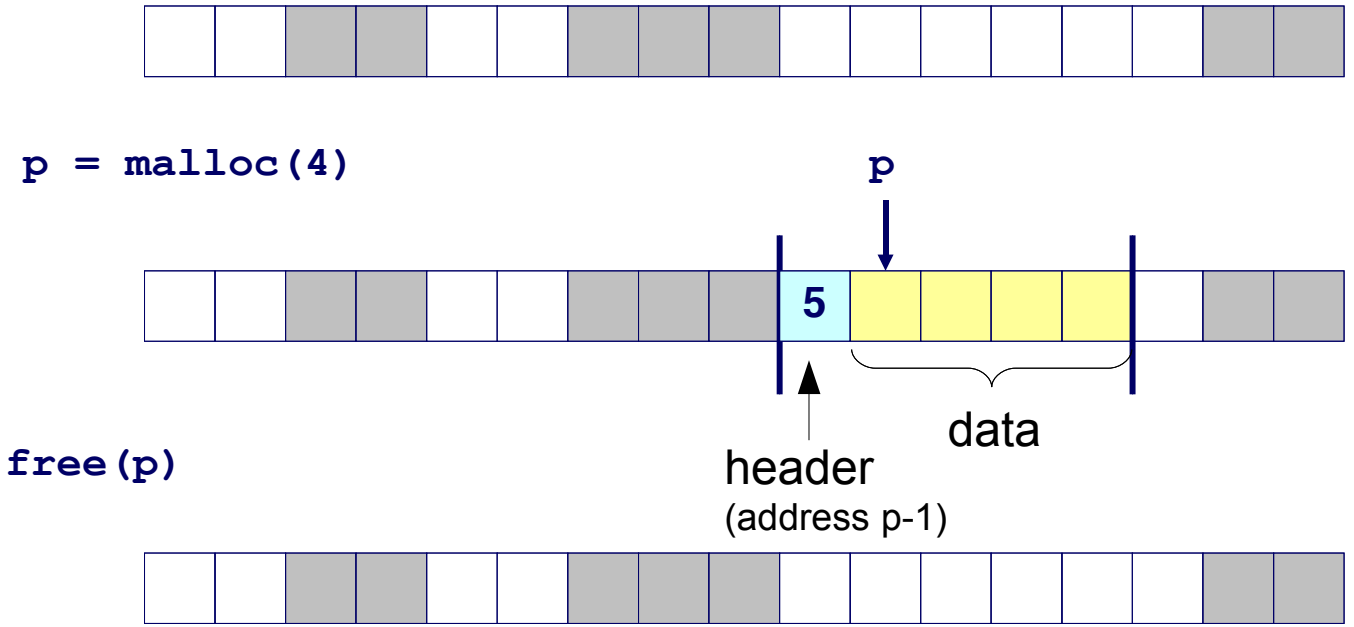
Implementation Issues

- How do we know how much memory to free just given a pointer?
- How do we keep track of the free blocks?
- 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 pick which free block to use for allocation?
- How do we reinsert freed block?

Knowing how much to free

Standard method

- Keep the length of a block in a **header** preceding the block.
- Requires an extra word for every allocated block



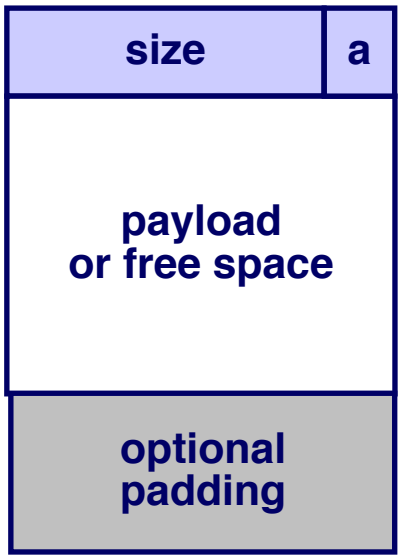
Keeping Track of Free Blocks

- One of the biggest jobs of an allocator is knowing where the free memory is.
- The allocator's approach to this problem affects...
  - Throughput – time to complete a malloc() or free()
  - Space utilization – amount of extra metadata used to track location of free memory.
- There are many approaches to free space management.
  - Today, we will talk about one: **Implicit free lists.**
  - Next time we will discuss several other approaches.

Implicit free list

Idea: Each block contains its **size** and an **allocated bit**

- Allocated bit indicates whether block is allocated or free.
- Trick: Use **low-order bit** of the size word in the header
- Implication: The block size must always be even.

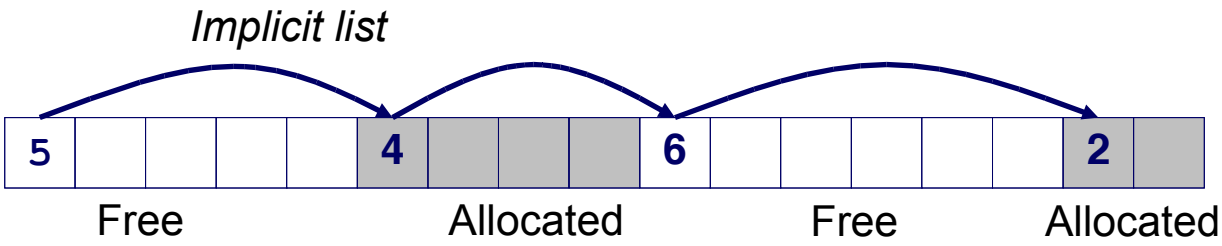


a = 1: block is allocated
a = 0: block is free

size: block size

payload: application data (allocated blocks only)

Implicit free list

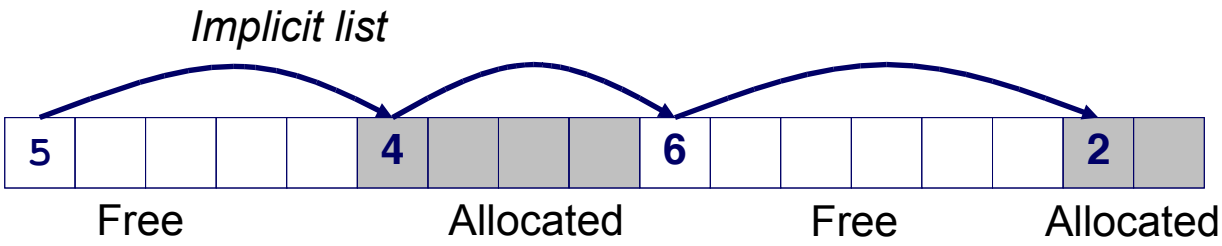


No **explicit** structure tracking location of free/allocated blocks.

- Rather, the size word (and allocated bit) in each block form an **implicit** "block list"

How do we find a free block in the heap?

Implicit free list



No **explicit** structure tracking location of free/allocated blocks.

- Rather, the size word (and allocated bit) in each block form an **implicit** "block list"

How do we find a free block in the heap?

Start scanning from the beginning of the heap.

Traverse each block until (a) we find a free block and (b) the block is large enough to handle the request.

This is called the **first fit** strategy.

Implicit List: Finding a Free Block

First fit strategy:

- Search list from beginning, choose first free block that fits
- Can take linear time in total number of blocks (allocated and free)
- In practice it can cause “splinters” at beginning of list

Next fit strategy:

- Like first-fit, but search list from location of end of previous search
- Research suggests that fragmentation is worse than first-fit

Best fit strategy:

- Search the list, choose the free block with the closest size that fits
- Keeps fragments small --- usually helps fragmentation
- Runs slower than first- or next-fit, since the entire list must be searched each time

Bitfields

How to represent the block header:

Masks and bitwise operators

```
#define SIZEMASK            (~0x7)
#define PACK(size, alloc)   ((size) | (alloc))
#define GET_SIZE(p)        ((p)->size & SIZEMASK)
```

Bitfields

```
struct {
    unsigned allocated:1;
    unsigned size:31;
} block_header;
```

Initial heap

Heap starts out as a single big “free block”

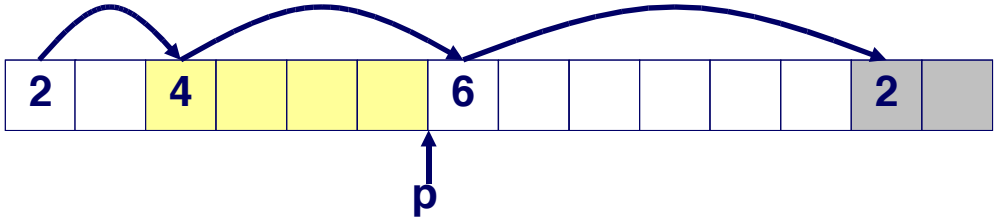
Each allocation **splits** the free space.

Over time the heap will contain a mixture of free and allocated blocks.

Implicit List: Allocating in Free Block

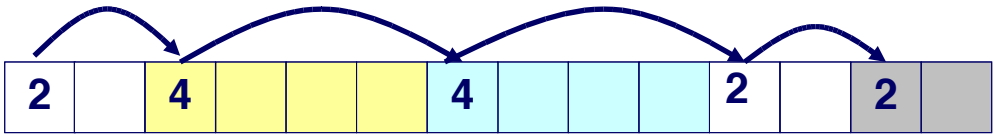
Splitting free blocks

- Since allocated space might be smaller than free space, we may need to **split** the block




```
void addblock(ptr p, int len) {
  int newsize = ((len + 1) >> 1) << 1;  // add 1 and round up
  int oldsize = *p & 0xfffffffe;        // mask out low bit
  *p = newsize | 0x1;                   // set new length + alloc bit
  if (newsize < oldsize)
    *(p+newsize) = oldsize - newsize;   // set length in remaining
}                                       //   part of block
```

```
addblock(p, 4)
```



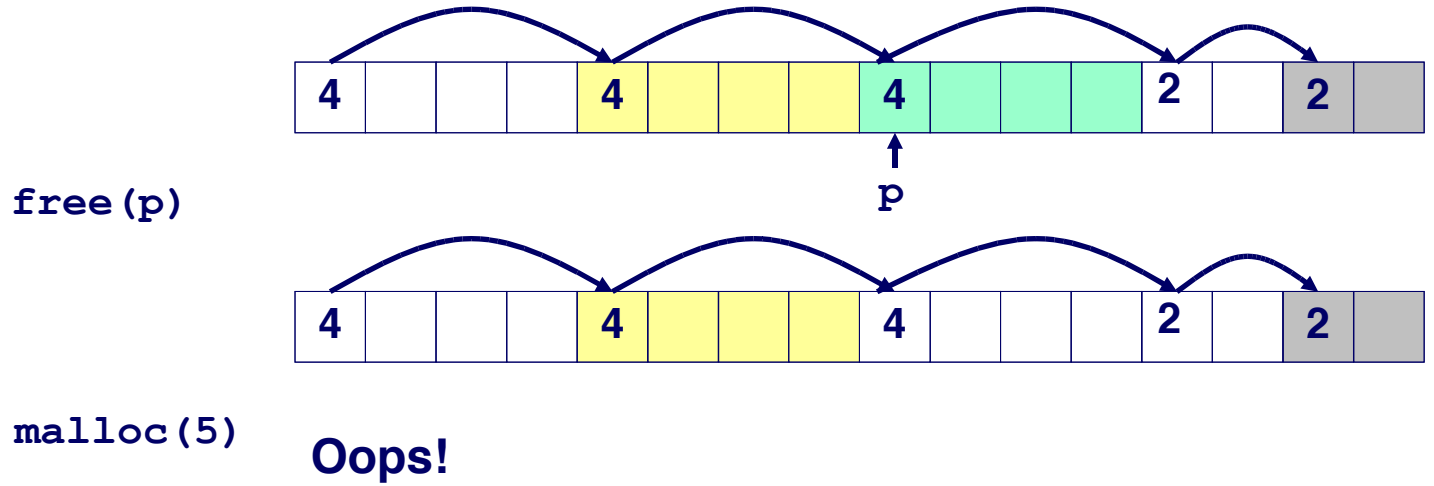
Implicit List: Freeing a Block

Simplest implementation:

- Only need to clear allocated flag

```
/* Here, p points to the block header. */
void free_block(ptr_t *p) { *p = *p & 0xfffffffe; }
```

- But can lead to “false fragmentation”

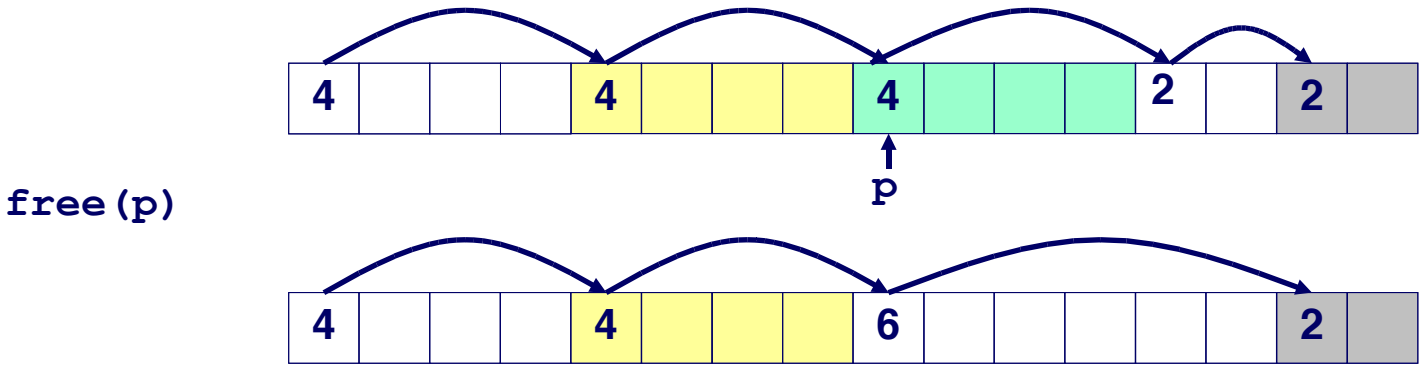


There is enough free space, but the allocator won’t be able to find it!

Implicit List: Coalescing

Coalesce with next and/or previous block if they are free

```
void free_block(ptr p) {
    *p = *p & 0xfffffffe;   // clear allocated flag
    next = p + *p;          // find next block
    if ((*next & 0x1) == 0)
      *p = *p + *next;      // add to this block if
}                           //    not allocated
```

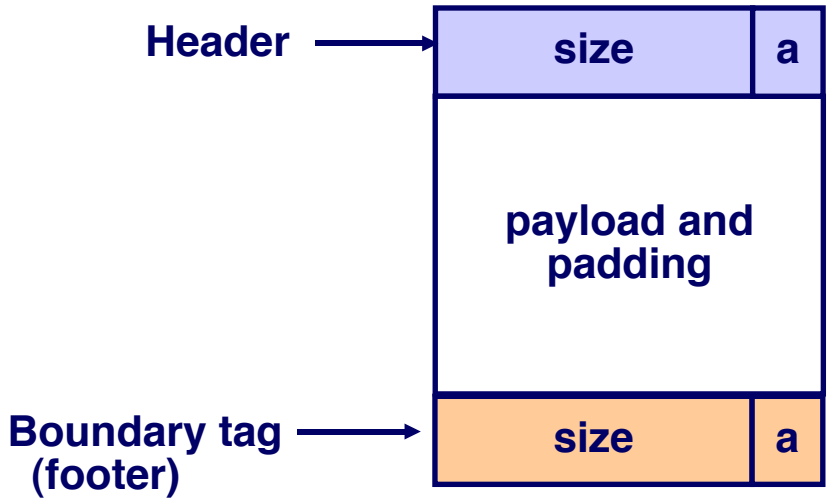


- This is coalescing with the next free block.
  How would we coalesce with the *previous* free block?

Implicit List: Bidirectional Coalescing

Boundary tags [Knuth73]

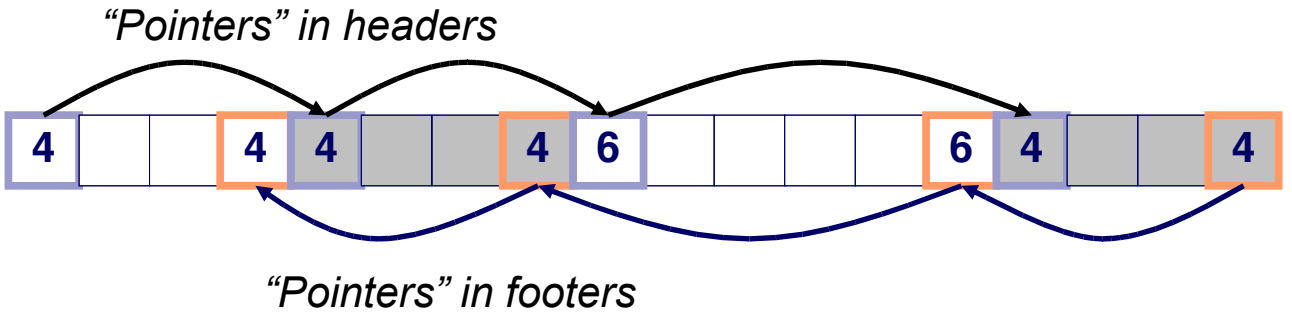
- Replicate size/allocated word at end of free blocks (a **footer**)
- Allows us to traverse the "block list" backwards, but requires extra space
- Important and general technique!



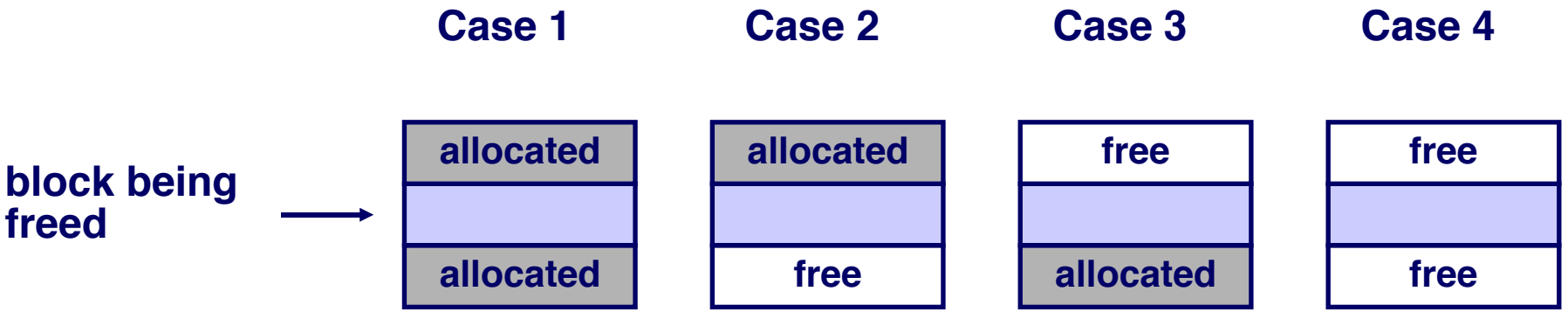
a = 1: block is allocated
a = 0: block is free

size: total block size

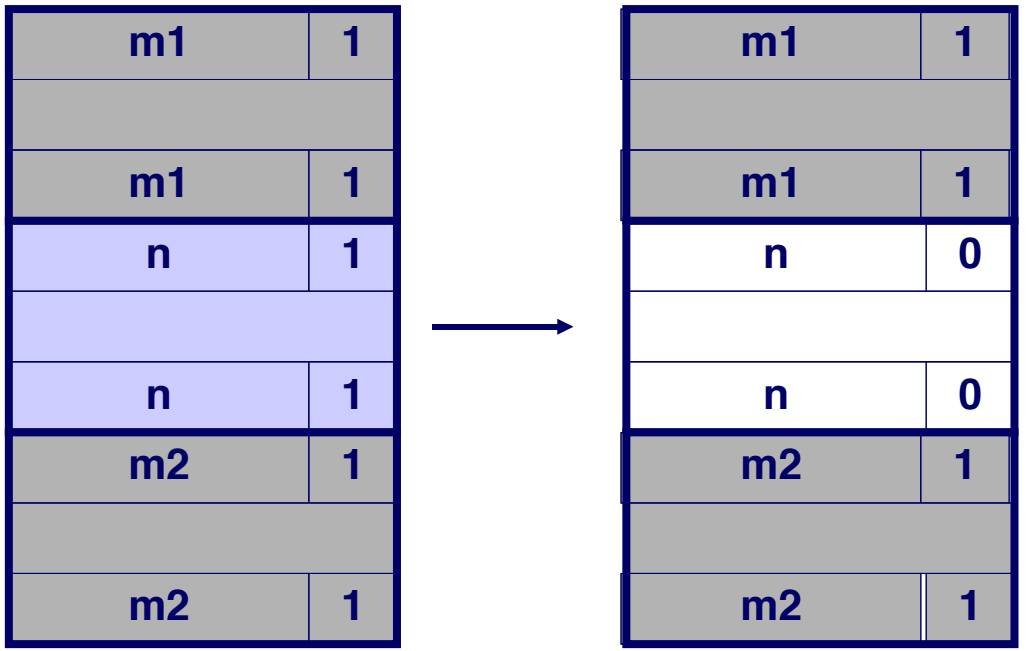
payload: application data (allocated blocks only)



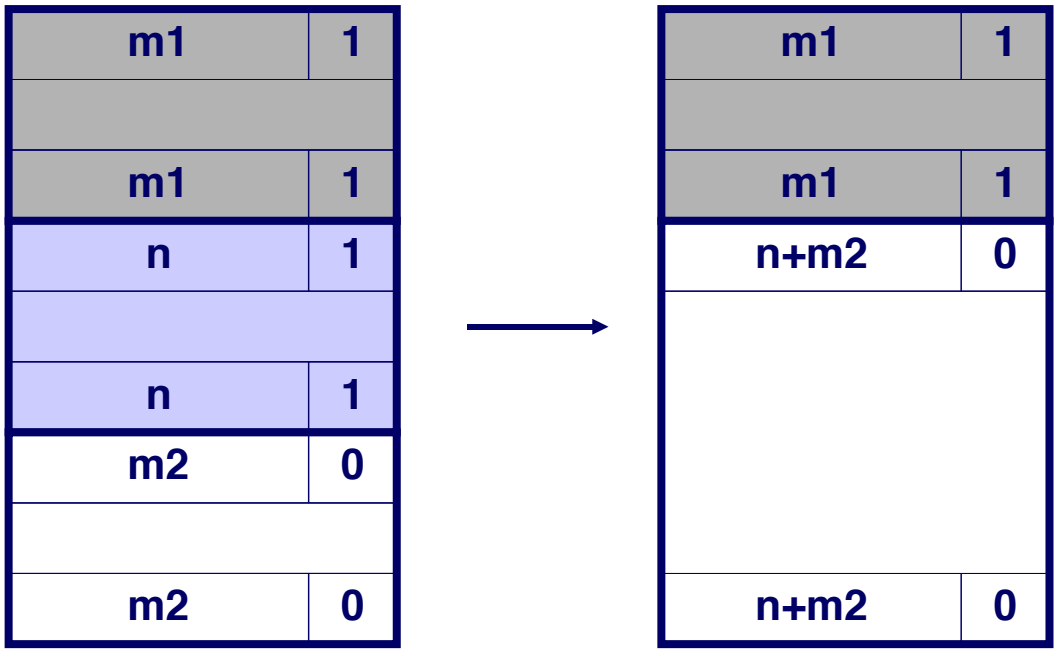
Constant Time Coalescing



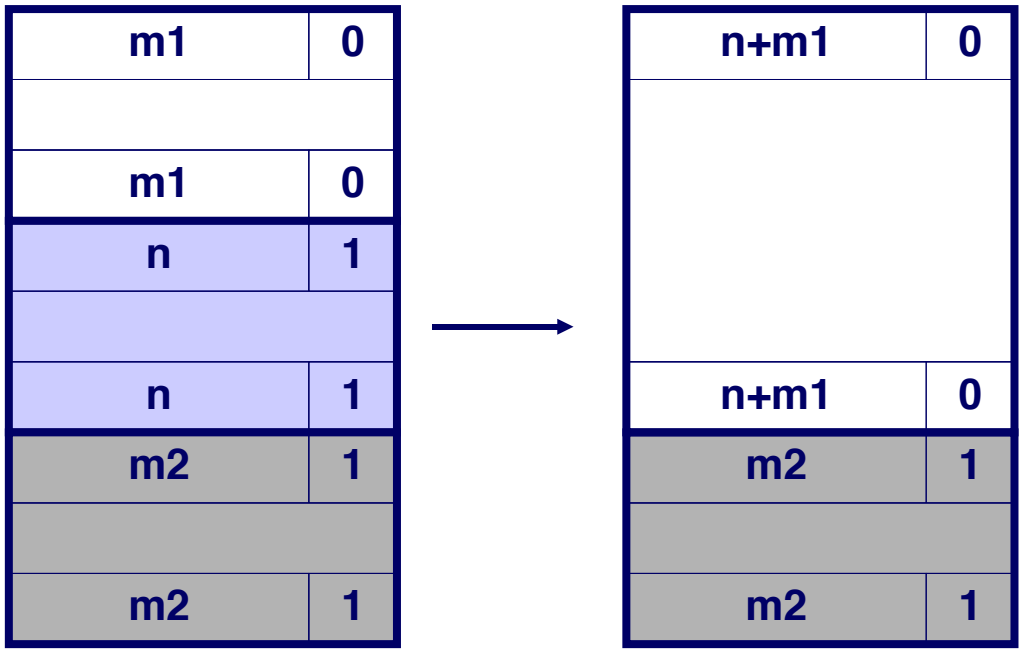
Constant Time Coalescing (Case 1)



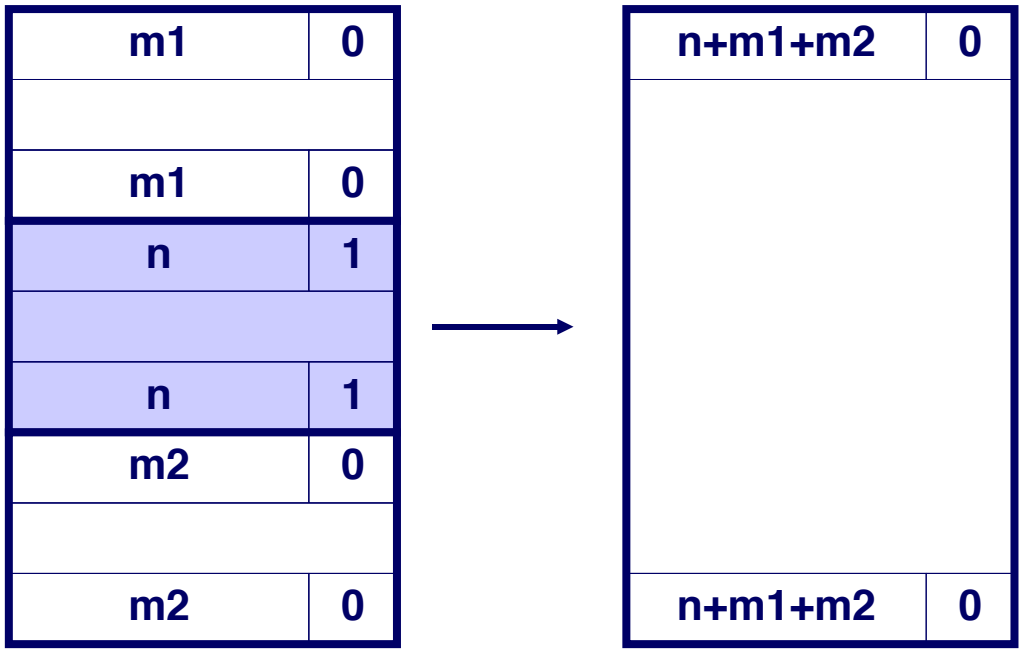
Constant Time Coalescing (Case 2)



Constant Time Coalescing (Case 3)



Constant Time Coalescing (Case 4)



Implicit Lists: Summary

- Implementation: Very simple.
- Allocation cost: Linear time worst case
- Free cost: Constant time, even with coalescing
- Memory usage: Depends on placement policy
  - First fit, next fit or best fit

Not used in practice for `malloc/free` because of linear time allocation.

The concepts of splitting and boundary tag coalescing are general to *all* allocators.

Allocation Policy Tradeoffs

Placement policy: First fit, next fit, or best fit

- Best fit has higher overhead, but less fragmentation.

Splitting policy: When do we split free blocks?

- Splitting leads to more internal fragmentation, since each block needs its own header.

Coalescing policy:

- *Immediate coalescing:* Coalesce each time `free` is called
- *Deferred coalescing:* Improve `free` performance by deferring coalescing until needed.
- Examples:
  - *Coalesce while scanning the free list for malloc()*
  - *Coalesce when the amount of external fragmentation reaches some threshold.*

Topics for next time

- Continuing discussion of dynamic memory allocation
- Explicit free list management
- Segregated free lists
- Implicit memory management: Garbage collection
- Common memory bugs