Dynamic Memory Allocation

CS 105: Computer Systems
Lecture 12

Melissa O'Neill

March 2, 2026

Learning Goals

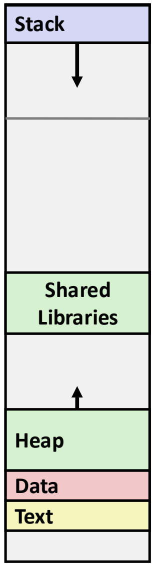
- **Consider the design space for a dynamic memory allocator**
  - How to allocate blocks
  - How to manage unallocated (free) blocks
- **Practice with the implicit free list approach**
- **Quiz 3 — Due Today**
  - Processes: understand what happens with fork()
  - Concurrency: how threads share variables in memory; semaphore basics

Memory allocation

- **Static memory allocation**
  - Allocated during compilation, e.g., on stack
  - Example: `int a[42];`
- **Dynamic memory allocation**
  - Allocated at run time *on heap*
  - Example: `int* a= (int *) malloc(sizeof(int)*42);`

`void *malloc(size_t size)`
- Returns a pointer to a memory block of at least `size` bytes, typically aligned on some boundary

`void free(void *p)`
- Releases block pointed at by **`p`** to pool of available memory; **`p`** must be from a previous call to `malloc`



Simple (inefficient) Implementation

- **Possible implementation for `malloc`**
  - Maintain pointer to top of heap
  - To service `malloc` request, allocate space at top of heap and return pointer to beginning of the allocation
  - Update pointer to top of heap for next `malloc` request
- **… implementation for `free`? Do nothing!**
- **Any issues with this design?**

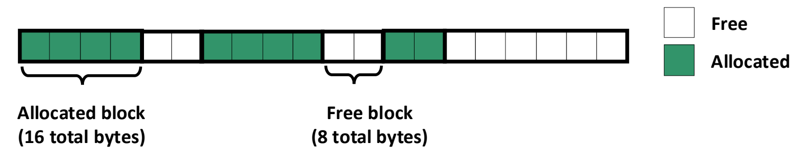
Dynamic Memory Allocator algorithm

- **Allocator maintains heap as collection of variable sized *blocks*, which are either *allocated* or *free***
- **Types of allocators**
  - ***Explicit allocator*:** application allocates and frees space
    - E.g., `malloc` and `free` in C
  - ***Implicit allocator:*** application allocates, but does not free space
    - E.g., garbage collection in Java

Our focus

Assumptions and Diagrams for Lecture

- **Allocate in 4-byte units, each square in diagram is 4 bytes**
  - Shaded squares are allocated, un-shaded squares are free
  - Memory addresses increase from left to right



- **The call `malloc(2)` means *allocate two 4-byte squares***

Simple Allocation Example

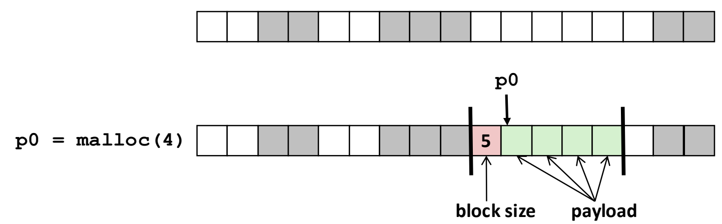
Allocator Design Space

- **Allocating and `Freeing`, a few aspects we'll look at today:**
  1. How does allocator know how much to free given just a pointer?
  2. How to keep track of which blocks are free?
  3. How should allocator reinsert a freed block?
  4. Which free block should allocator use for allocation request?

- **Other considerations**
  - Applications can issue arbitrary sequence of `malloc` and `free` calls
  - Allocator must respond immediately to requests (can't reorder)
  - Allocator cannot manipulate, modify, or move allocated blocks
  - Allocator doesn't know what data type request is for, must have general alignment consideration

1. Knowing How Much to Free

- **Keep the length of a block in the 4 bytes preceding the payload**
  - This 32 bits is often called the ***header field*** or ***header***
  - It is *overhead:* requires an extra 4 bytes for every allocated block



Simple Allocation With Headers

Exercise: Considering Alignment

- **Systems often require `malloc` to allocate a block size that satisfies an alignment constraint**
  - We'll use an **8-byte** alignment → block must start on address a **multiple of 8 bytes**
  - Recall each square in diagram is 4 bytes

- **Considering alignment in addition to headers, fill in the diagram to show how the heap would look after the three `malloc` requests.**
  - Draw arrows to indicate where each pointer would point
  - Include header and be sure to indicate the boundaries of each block
  - Assume the first square is aligned correctly
  - Use first free block that can satisfy the request

```
p1 = malloc(3)
p2 = malloc(4)
p3 = malloc(3)
```

Allocation with Header *and padding*

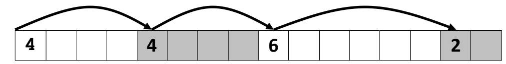
- **Alignment and padding**
  - Blocks aligned to 8 bytes
  - Blocks padded to multiple of 8 bytes
  - Use first free block that can satisfy request (in this example)

```
p1 = malloc(3)
p2 = malloc(4)
p3 = malloc(3)
```

For `malloc(n)`:
If n is odd, allocate block of n+1 words (header + payload)
If n is even, allocate block of n+2 words (header + payload + padding)

2. Keeping Track of Free Blocks

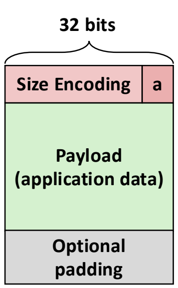
- **Simple Method: *Implicit list* using length—links all blocks**



- **Length of block includes header and any padding**

Keeping Track of Free Blocks: Implicit List

- **We need block size and allocation status (either *used* or *free*)**
  - Size is in header, the 4 bytes preceding payload
  - Should allocation status go in additional 4 bytes preceding payload?? Seems wasteful!
- **Idea: If the allocated size is always *even*, we can reuse its least significant bit to store the status**



***a* is one bit**
a = 1: Allocated block
a = 0: Free block

Size Encoding:
higher 31 bits of block size

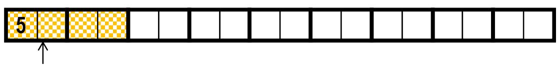
Example: Size and Allocation

- **Suppose we have a heap space of 18 4-byte units, initially it is entirely free.**
- **What value do we store in header (i.e., what is *h*)?**

Example: Size and Allocation

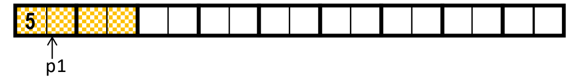
`p1 = malloc(3)`

- Step 1: Allocate new block

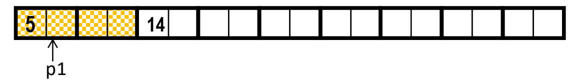


Example: Size and Allocation

`p1 = malloc(3)`



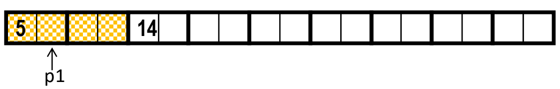
- Step 2: Compute header for remaining free block



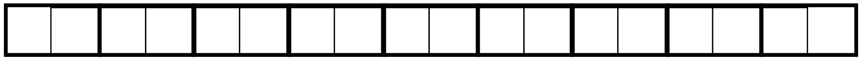
Note: suppose a header has value n

- **if n is odd, the block is allocated and actually has size n-1**
- **if n is even, the block is free and its size really is n**

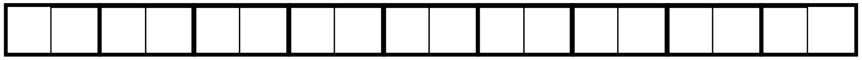
Exercise



1. **Update it after a request of `p2 = malloc(4)`:**



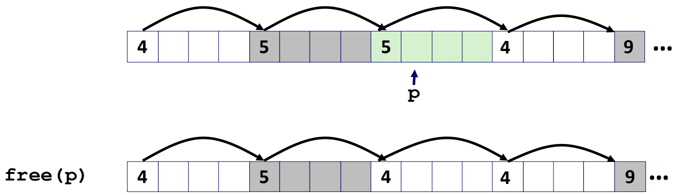
2. **Update it after an additional request of `p3 = malloc(3)`:**



3. **Update after a request of `free(p2)`:**

3. Implicit List: Freeing a Block

- **Simplest implementation:**
  - Need only clear the "allocated" flag
  - But can lead to **false fragmentation**

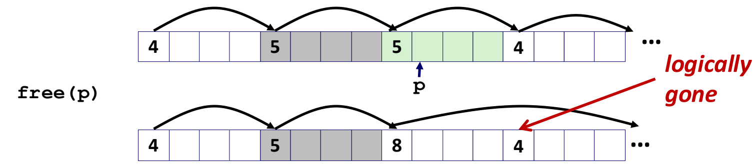


`malloc(5)` ***Oops!***

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

Implicit List: Coalescing

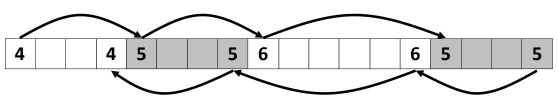
- **Join *(coalesce)* with next/previous blocks, if they are free**
  - Coalescing with next block

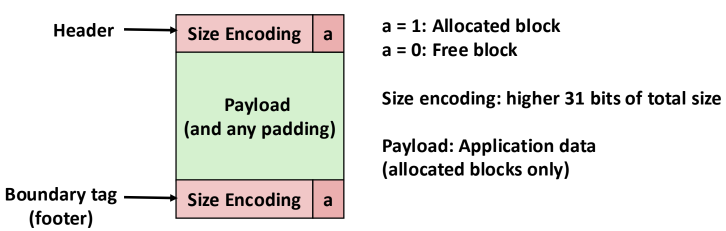


  - Next block is easy. But how do we coalesce with *previous* block?

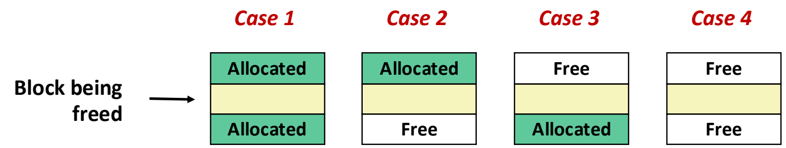
Implicit List: Bidirectional Coalescing

- ***Boundary tags***
  - Replicate size/allocated word at "bottom" (end) of blocks
  - Allows us to traverse the "list" backwards, but requires extra space

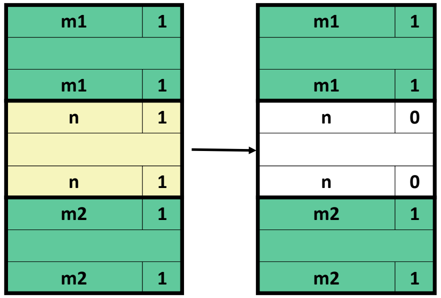




Coalescing in Constant Time

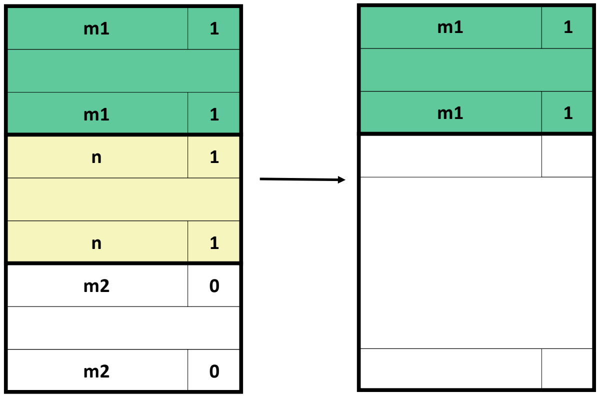


Constant Time Coalescing (Case 1)

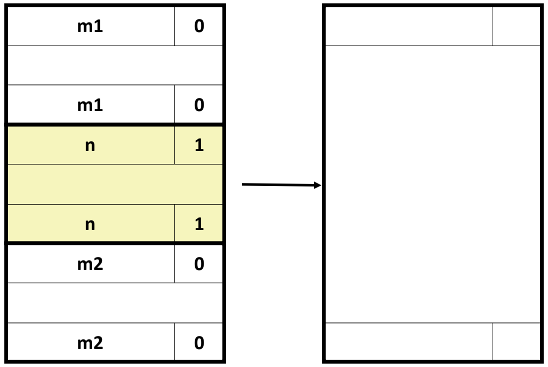


Exercise: Constant Time Coalescing (Case 2)

- Fill in the "after" picture with the header and footer contents showing after the middle block is freed



Constant Time Coalescing (Case 4)



4. Finding a Free Block: Performance Considerations

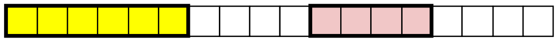
- **Given some sequence of `malloc` and `free` requests:**
  - $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$
  - Each allocation request has a ***payload***, i.e., number of bytes requested

- **Goals:**
  - Maximize ***throughput***: number of requests completed per unit time
  - Maximize ***memory utilization***: (sum of payloads)/(total heap size)

Finding a Free Block -- Fragmentation

- **Barrier to maximizing utilization? *fragmentation***

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



`malloc(6) ??`

- **Internal fragmentation**
  - Occurs if payload is smaller than block size
  - Caused by overhead in block, e.g., header/footer and padding

Implicit List: Finding a Free Block

- **So far in lecture: used first fit approach**

- **Good approach?**
  - Want to find **best fit** for a request → smallest free block that fits
  - Avoid a linear search through *all* blocks?