Memory Management

Adapted by Joseph Lunderville
from slides by Dr. Brian Fraser
and course material by Dr. Steve Ko

Topics

1) What is the *layout of memory?*
2) How does the *heap* work?
    a) Getting space from the OS
    b) Tracking free space
    c) Freeing allocated space

Context

- *Memory allocation/deallocation*
  - Heap is used for dynamically allocated memory
    - Usually use: `malloc()` or `calloc()`, and `free()`
  - How could *we* actually implement `malloc()`/`free()`?

    (This will help us really understand low-level memory management)
- We are not talking about physical memory here:

  User processes can only use virtual memory, not physical memory

Details

- *Can find more info in OSTEP book* (more depth than we require, really)
  - *Chapter 13,* The Abstraction: Address Spaces
    https://pages.cs.wisc.edu/~remzi/OSTEP/vm-intro.pdf
  - *Chapter 14,* Interlude: Memory API
    https://pages.cs.wisc.edu/~remzi/OSTEP/vm-api.pdf
  - *Chapter 15,* Free Space Management
    https://pages.cs.wisc.edu/~remzi/OSTEP/vm-freespace.pdf

Prerequisites

What You (Should) Already Know

- *This lecture assumes you know:*
  - Data structures used for memory management – **array**, **struct** (record type), **linked lists**
  - How to use `malloc()` and `free()` in C
  - How to implement a singly- and doubly-linked list
  - The **stack** and the **heap**
    - How variables are placed in stack and heap
    - Scope and lifetime of variables in stack and heap

Activity - Linked Lists

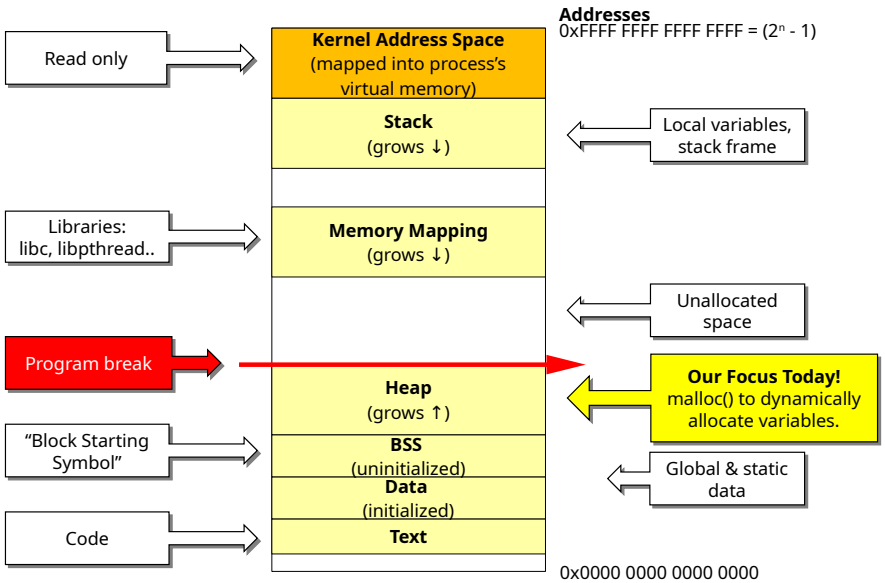
```c
struct Node {
    int data;
    struct Node *next;
};

// Create a new node with the given data
struct Node *createNode(int data) {
    struct Node *newNode
        = malloc(sizeof(*newNode));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}


// Insert a new node at the end of list
void append(struct Node **head, int data) {
    // Code together!
}

// Traverse and print the linked list
void traverse(struct Node *head) {
    // Code together!
}
```

```c
int main() {
    struct Node *head = NULL;

    // Append elements to the list
    append(&head, 1);
    append(&head, 2);
    append(&head, 3);

    // Traverse and print the list
    printf("Linked List: ");
    traverse(head);

    // Remember: free memory when done
    struct Node *current = head;
    while (current != NULL) {
        struct Node *temp = current;
        current = current->next;
        free(temp);
    }
    head = NULL;

    return 0;
}
```


Solution:

```c
// Function to insert a new node at the end of the linked list
void append(struct Node **head, int data) {
    struct Node *newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
    } else {
        struct Node *current = *head;
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = newNode;
    }
}

// Function to traverse and print the linked list
void traverse(struct Node *head) {
    struct Node *current = head;
    while (current != NULL) {
        printf("%d -> ", current->data);
        current = current->next;
    }
    printf("NULL\n");
```

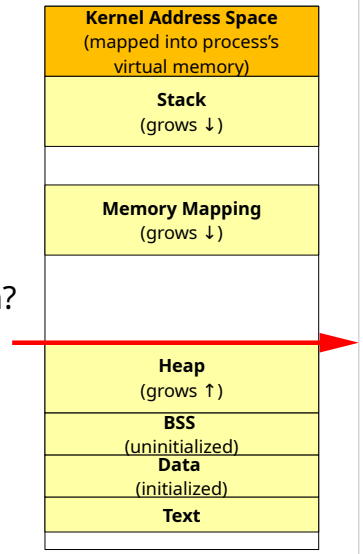
Memory Layout



brk() and sbrk()

Getting More Memory

- *Program Break*
  - Used by Linux to mark end of heap (actually end of BSS; but grows to be heap)
  - Above the **program break** is unallocated space
  - Consider – what does “unallocated” mean?
- *More Space*
  - OS moves the program break higher to expand the heap
  - Linux uses `brk()` and `sbrk()` to move the program break



man sbrk

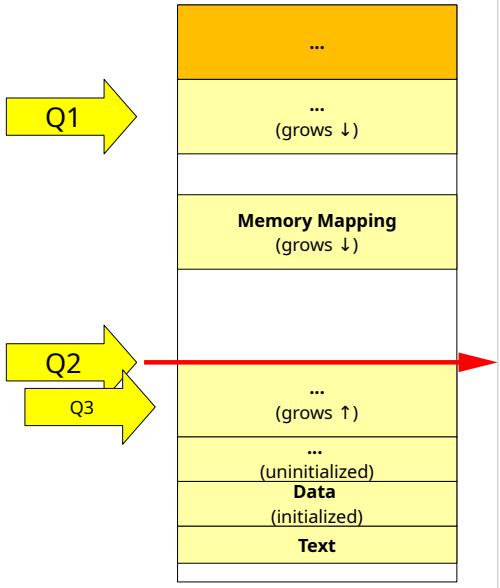
- *man sbrk*
  - OS increases size of heap
  - It's a syscall: overhead!
- *Don't call sbrk() often*
  - `malloc()` (user-level) calls `sbrk()` (kernel) to get big block of memory
  - `malloc()` hands out small pieces of memory for each request
- *How can `malloc()` do that?*
  - Allocation strategies
  - Deallocation strategies

```
brk(2)              System Calls Manual              brk(2)

NAME
       brk, sbrk - change data segment size

LIBRARY
       Standard C library (libc, -lc)

SYNOPSIS
       #include <unistd.h>

       int brk(void *addr);
       void *sbrk(intptr_t increment);

DESCRIPTION
       brk() and sbrk() change the location of the pro-
       gram break, which defines the end of the process's
       data segment (i.e., the program break is the first
       location after the end of the uninitialized data
       segment). Increasing the program break has the
       effect of allocating memory to the process; de-
       creasing the break deallocates memory.

       brk() sets the end of the data segment to the
       value specified by addr, when that value is rea-
       sonable, the system has enough memory, and the
       process does not exceed its maximum data size (see
       setrlimit(2)).

       sbrk() increments the program's data space by in-
       crement bytes. Calling sbrk() with an increment
       of 0 can be used to find the current location of
       the program break.

RETURN VALUE
       On success, brk() returns zero. On error, -1 is
       returned, and errno is set to ENOMEM.

       On success, sbrk() returns the previous program
       break. (If the break was increased, then this
       value is a pointer to the start of the newly allo-
       cated memory). On error, (void *) -1 is returned,
       and errno is set to ENOMEM.
```

Audience Participation - Memory Layout

- Identify the labeled part of the diagram

a) BSS
b) Heap
c) Program Break
d) Stack



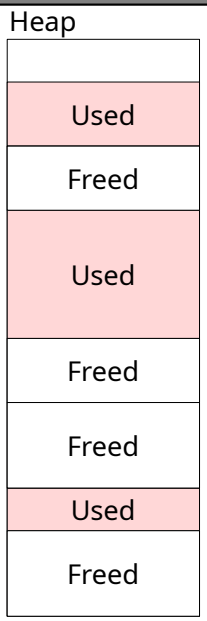
Managing Dynamic Memory
Overview

Sequential Fit Allocation

- *There are many ways to write an allocator*
  - Real allocators optimize for common cases, and include debug features
  - **Our discussion is naive:** we describe one allocation strategy in depth, and you will use it in an assignment
  - Still relevant because C-style allocators usually still operate in a *broadly* similar way, but details will vary, and in the real world *verify assumptions*
    - Why? “More robust” algorithms can perform worse (cache-unfriendliness, lock requirements, maintenance effort)
- *Alternative strategies not discussed (it’s a huge topic!)*
  - Segregated fit (buddy, small-object…)
  - Custom allocators (pools, arenas…)
  - Handle-based systems and defragmenting
  - Automatic garbage collection

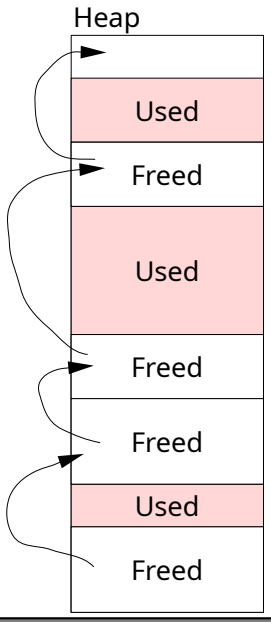
Memory Allocator

- ***Memory Allocator:*** *manages the heap*
  - For each allocation request, it returns a pointer to an unused (or free) region inside the heap
  - It tracks of which parts of the heap are not used
- ***Fragmentation***
  - Over time the application allocates and frees memory regions
  - This fragments memory into broken up pockets of used and freed memory (why?)
  - Handle-based strategies (which we won't go into)

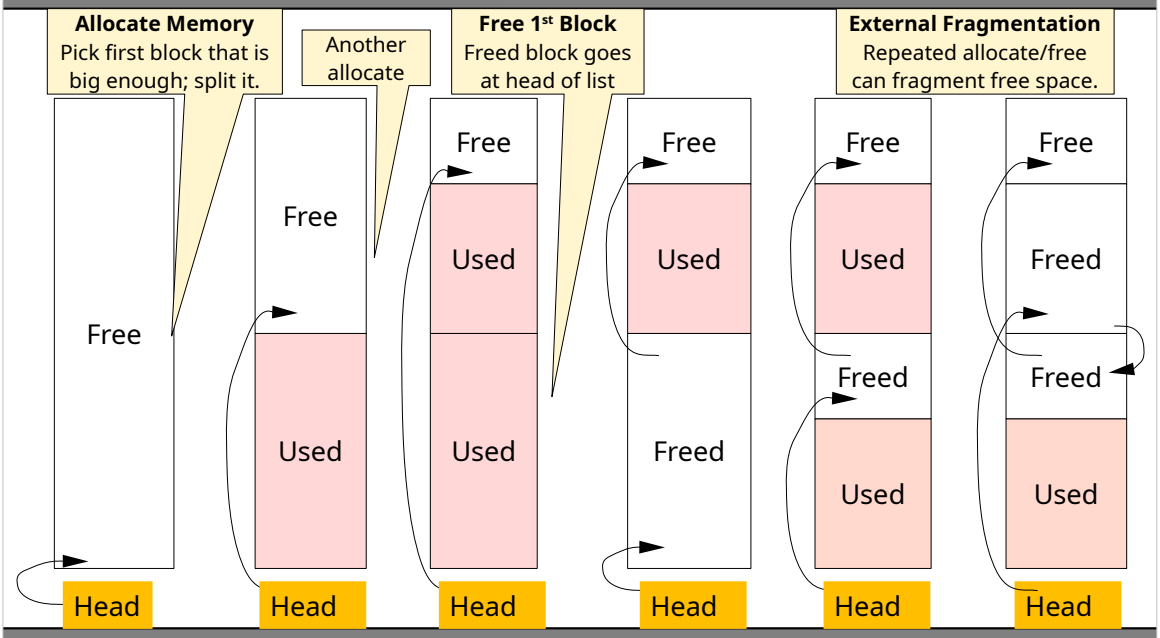


Track Free Space

- *Track free regions (blocks) in a linked list of free blocks*
  - We don't track used regions; we are given back regions from calls to `free()`



Linked List Management



Linked List Management

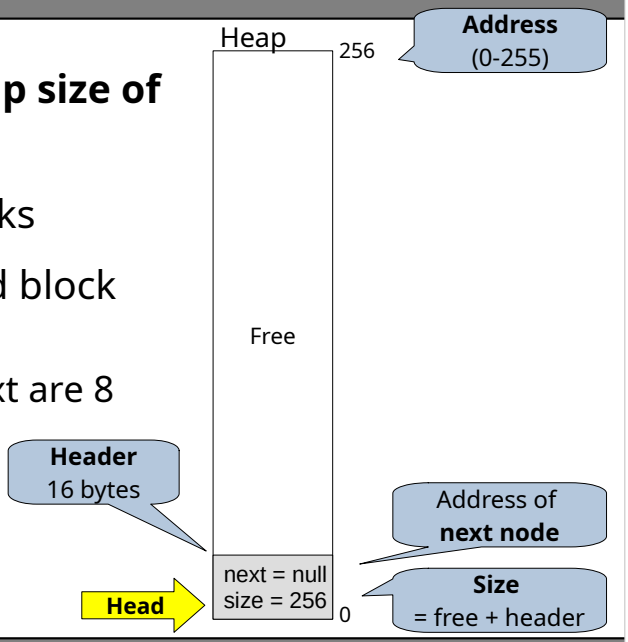
- *Free Blocks Linked List*
  - We have a linked list of free blocks
  - Head points to the most recent free block
- *Basics of Allocation – `malloc()`*
  - Pick a free block from the linked list
  - Remove it from the linked list
  - Split the free block into two blocks: **allocated** and **free**
  - Insert the *new* free block back into the head of the linked list
  - Return the allocated block to the caller
- *Basics of Deallocation – `free()`*
  - Insert the given (freed) block at head of the linked list

Linked List Without Dynamic Allocation

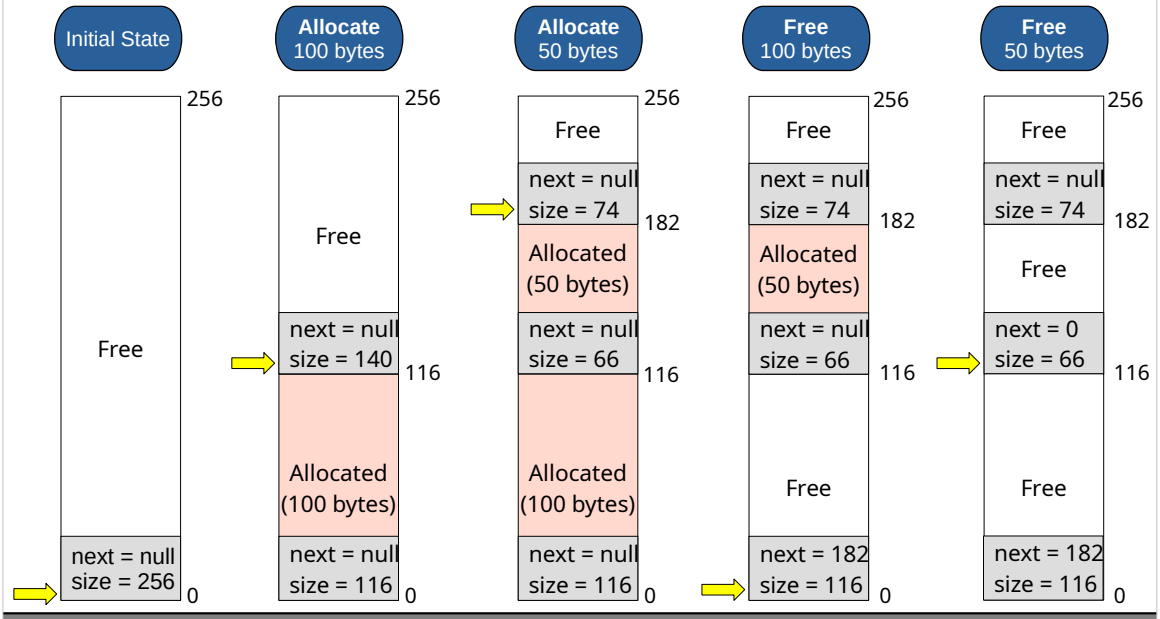
- *Linked List of Free Memory*
  - We've see how to manage free memory using a linked list of free blocks
  - But, how do we normally create nodes in a Linked List? Dynamic allocation!
  - So, how do we create a linked list without dynamic allocation?
- *In-Place Linked List (= Internal Nodes)*
  - Create a header on each free block to track size of the block and pointer to next free block
  - Perform coalescing: combine consecutive free blocks into a larger single free block

In-Place Linked List

- *Toy example* with a **heap size of 256 bytes**
- Build linked-list of blocks
- Each free and allocated block has a **header**
  - Assume size and next are 8 bytes each (64-bit)



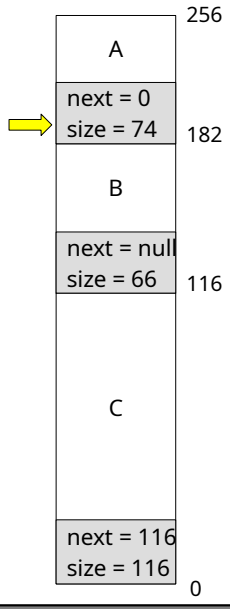
Example - In-Place Linked List



Audience Participation - Linked List

- What was the order in which these blocks were *freed*? (in order of first freed to last freed)

a) A then B then C
b) A then C then B
c) B then C then A
d) C then B then A



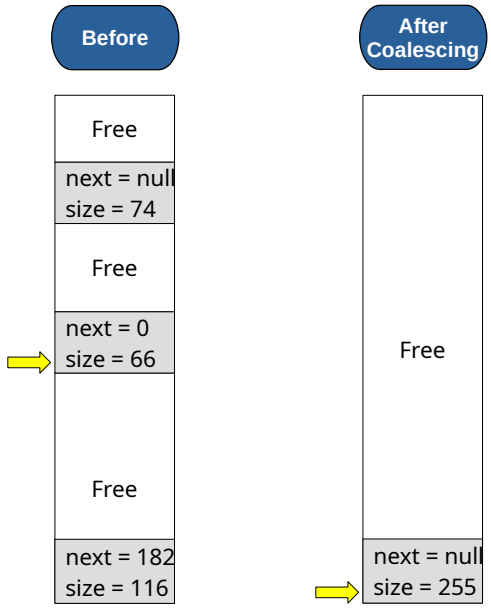
Answer: C
List is in order A --> C --> B
So it's the reverse order: BCA

External Fragmentation

- ***External*** *Fragmentation*
  - Free memory is fragmented into smaller blocks
  - But each allocation request can only be satisfied by a single block (cannot split it up)
  - Even if total free memory is enough, may not have one **contiguous** free block to satisfy an allocation request
- ***Internal*** *Fragmentation*
  - Problem of unused space **inside** blocks (more on this when we talk about virtual memory)

Coalescing

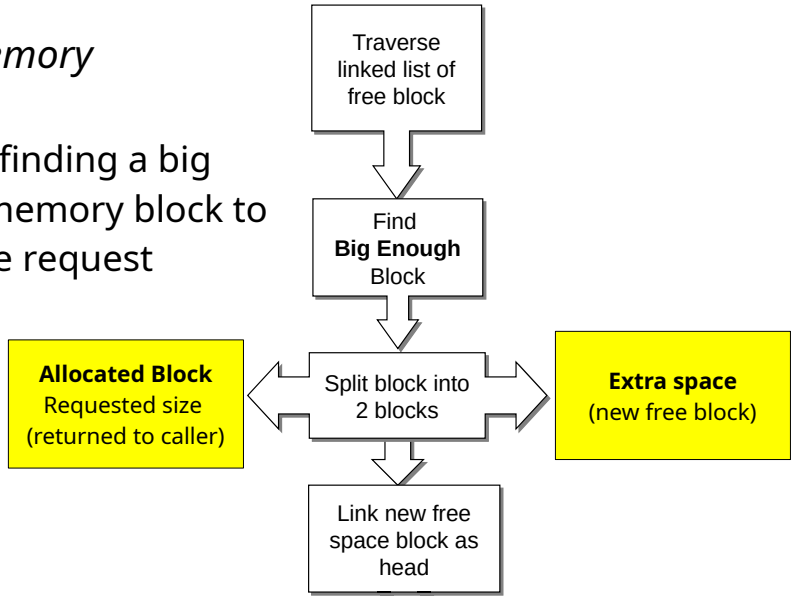
- *Coalescing*
  - Process of **combining consecutive free blocks** into bigger blocks
  - Some external fragmentation is unavoidable
  - Fix what we can fix easily



Finding a Free Block

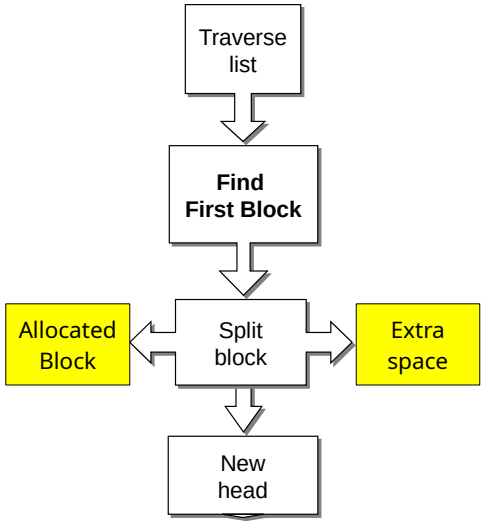
Allocation Policy

- *Allocating Memory (malloc ( ))*
  - Requires finding a big enough memory block to satisfy the request



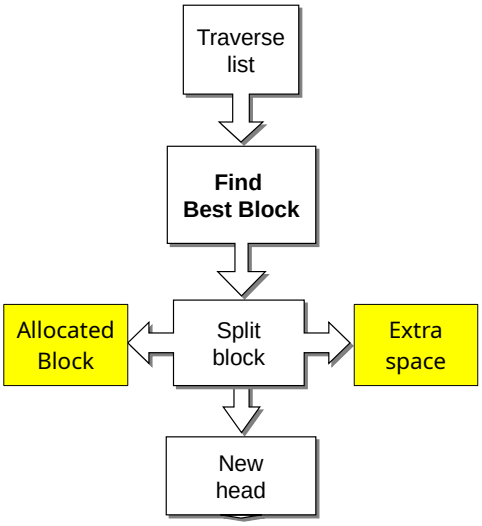
Allocation Policy: First Fit

- *First Fit*
  - Find the first block that is big enough
- *Advantage*
  - Implementation simplicity
  - **Fast:** greedy algorithm, don't consider a lot of alternatives
- *Disadvantage*
  - Can pollute the beginning of the free list with (too-)small blocks
  - Unpredictable, varies with list order
    - For us, list order determined by free order: FIFO
    - Other policies are possible (next fit/circular, LIFO, address order…)



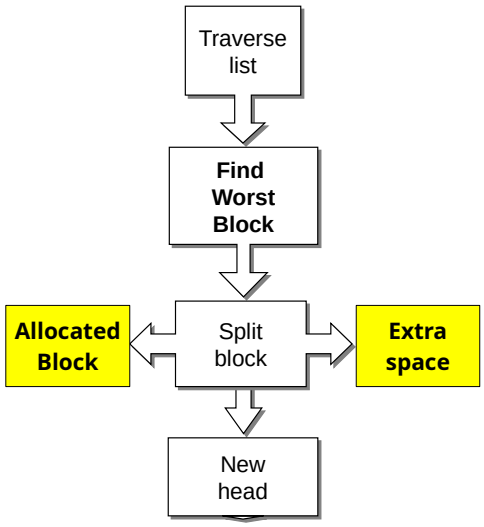
Allocation Policy: Best Fit

- *Best Fit*
  - Find the **smallest** free block that is big enough
- *Advantage*
  - Preserves large blocks for large allocations
  - Reuses exactly-same-size allocations well (why might this be a *big* advantage?)
  - Predictable
- *Disadvantage*
  - **Slow:** must search the entire list (or: use a more complex ordered data structure)



Allocation Policy: Worst Fit

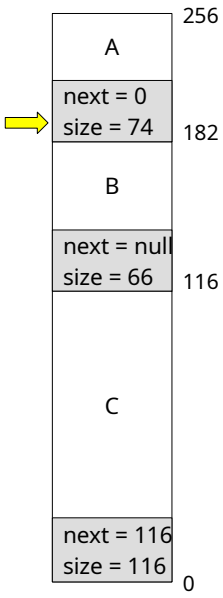
- *Worst Fit*
  - Find the largest free block
- Advantage
  - Avoids creating unusably small leftovers
    - ...this time
- Disadvantage
  - **Also slow:** must search the entire list, like best fit
  - Trends toward uniformly-spaced blocks (why is this not great?)



Audience Participation - Free Space

- A memory allocation system is asked to allocate 50 bytes. Which block is allocated if it is using…
  - First fit
  - Worst Fit
  - Best Fit

a) A
b) B
c) C
d) None of them.



First: A (actual free is 58 bytes)
Worst: C (actual free is 100 bytes)
Best: B (actual free is 50 bytes)

Summary

- *Memory Segments*
  - **text, data, BSS, heap, memory mapped, stack, kernel**
  - Program break and effect of `brk()` and `sbrk()`
- *Memory Allocator*
  - Linked list of free memory
  - New free blocks go first in the list
- *Fragmentation*
  - **External fragmentation**
  - Coalescing algorithm
- *Allocation Policy*
  - First/best (smallest)/worst (largest) fit