Dynamic Memory Allocation

malloc(woo!)

Ian Hartwig

Section F

April 1st, 2013

Outline

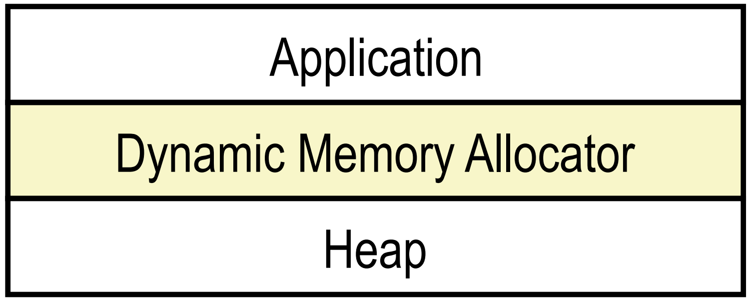
- **Schedule**
- **Dynamic Memory Allocation**
  - Keeping Track of Free Blocks
  - Finding a Free Block
  - Splitting Blocks
  - Freeing Blocks
- **MallocLab Tips**

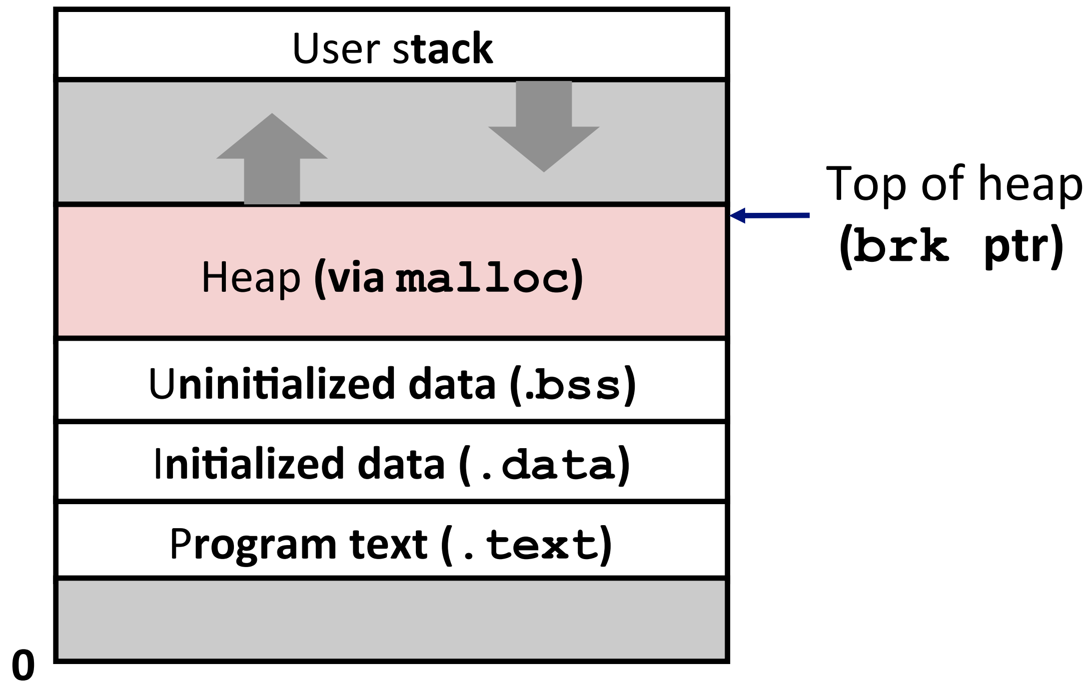
Schedule

- **Cachelab style grades should be up shortly (<1 day)**
  - Sorry for the delay.
- **Shell Lab was due Thursday, March 28th.**
  - Last turn in option was last night.
  - Should have style grades in ~2 weeks.
- **Malloc Lab out.**
  - Due April 11th.
  - Use your late days if you have them – the chances of using these days on proxy is slim.

Dynamic Memory Allocation

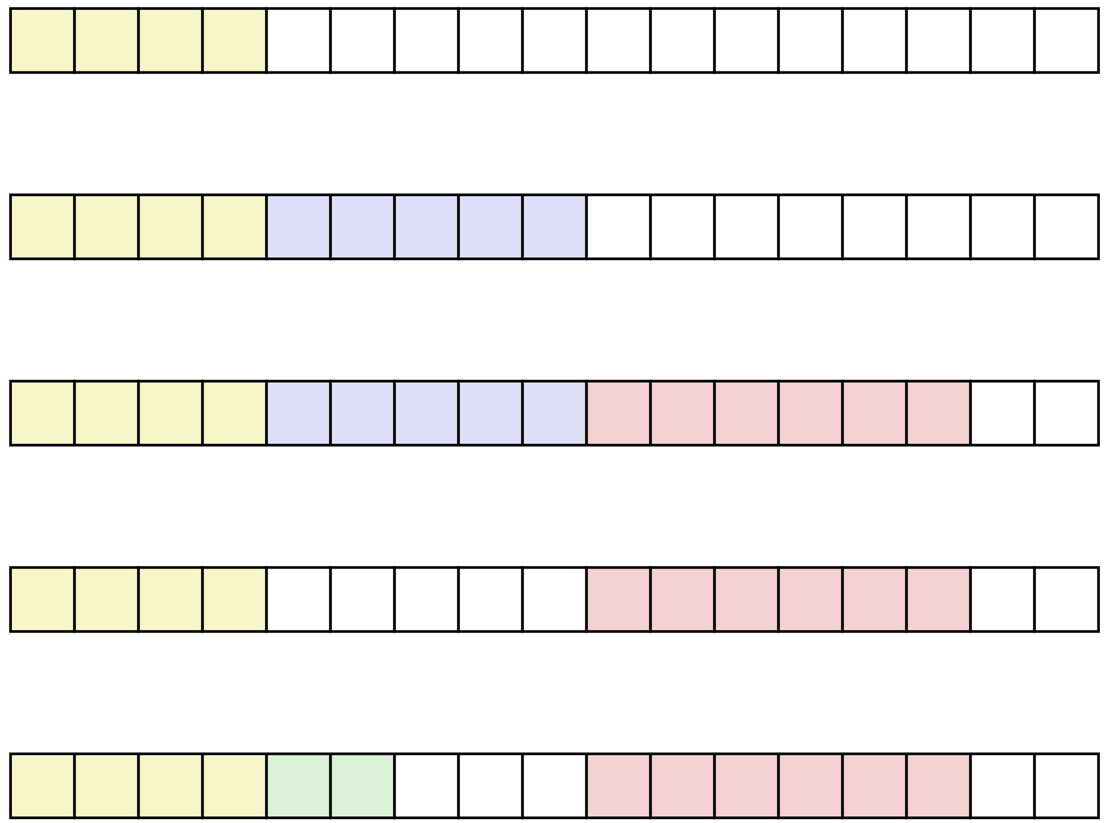
- **Programmers use *dynamic memory allocators* (such as `malloc`) to acquire VM at run time.**
  - For data structures whose size is only known at runtime.
- **Dynamic memory allocators manage an area of process virtual memory known as the *heap*.**





Dynamic Memory Allocation

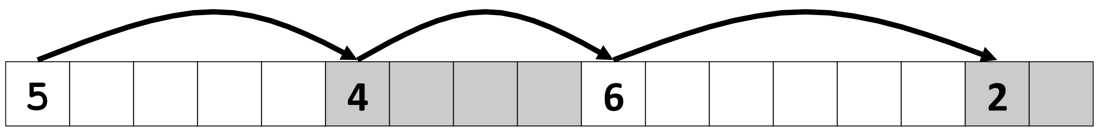
```
p1 = malloc(4)

p2 = malloc(5)

p3 = malloc(6)

free(p2)

p4 = malloc(2)
```



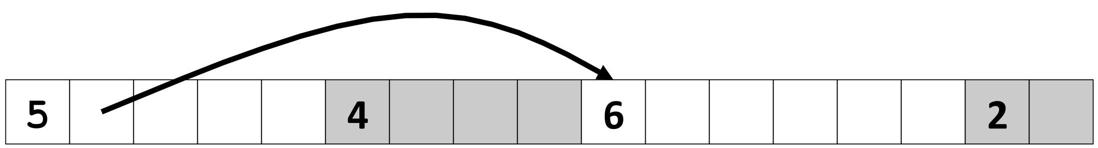
- How do we know where to put the next block?

Keeping Track of Free Blocks

- **Method 1:** ***Implicit list*** **using length—links all blocks**



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

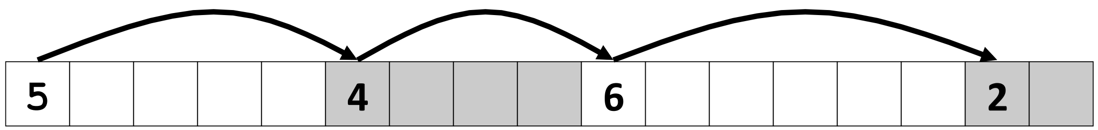


- **Method 3:** ***Segregated free list***
  - Different free lists for different size classes

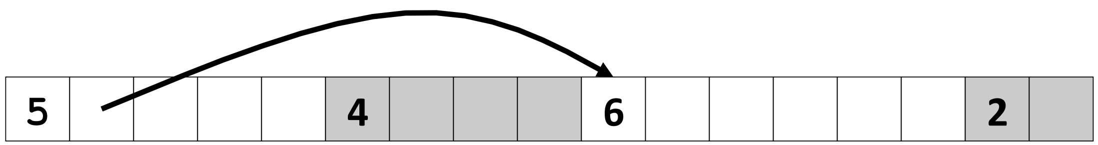
- **Method 4:** ***Blocks sorted by size***
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

Keeping Track of Free Blocks

- **Method 1: *Implicit list* using length—links all blocks**



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



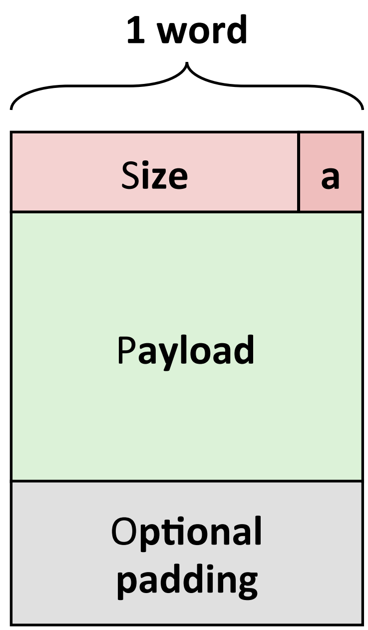
  - **Additionally: *Segregated free list***
    - Different free lists for different size classes

  - **Additionally: *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

- **For each block we need both size and allocation status**
  - Could store this information in two words: wasteful!
- **Standard trick**
  - If blocks are aligned, some low-order address bits are always 0
  - Instead of storing an always-0 bit, use it as a allocated/free flag
  - When reading size word, must mask out this bit

Format of allocated and free blocks

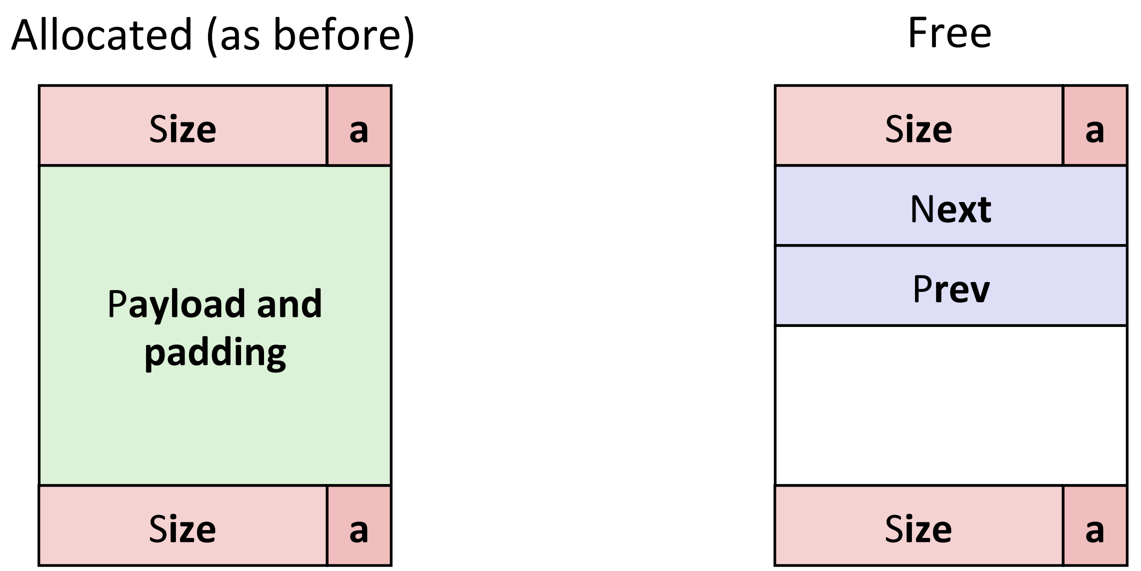


a = 1: Allocated block
a = 0: Free block

Size: **block size**

Payload: **application data (allocated blocks only)**

Method 2 - Explicit Free Lists

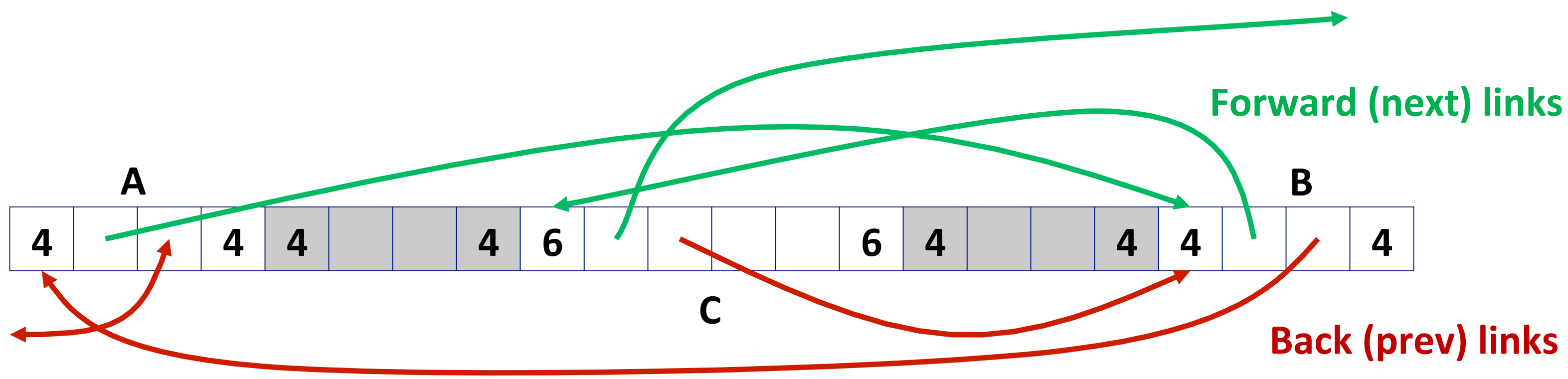


- **Maintain list(s) of *free* blocks, not *all* blocks**
  - The "next" free block could be anywhere
    - So we need to store forward/back pointers, not just sizes
  - Still need boundary tags for coalescing
  - *Luckily we track only free blocks, so we can use payload area*

Method 2 - Explicit Free Lists

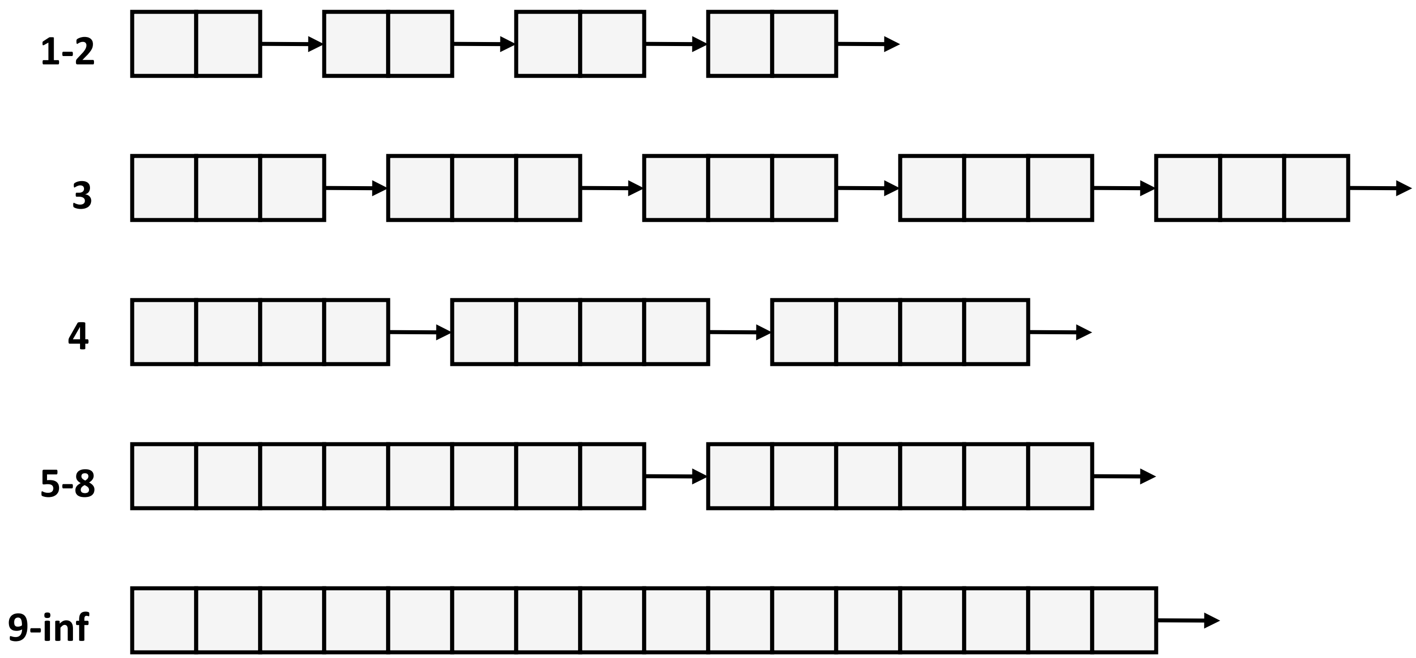
- **Logically:**

- **Physically: blocks can be in any order**



Segregated List (Seglist) Allocators

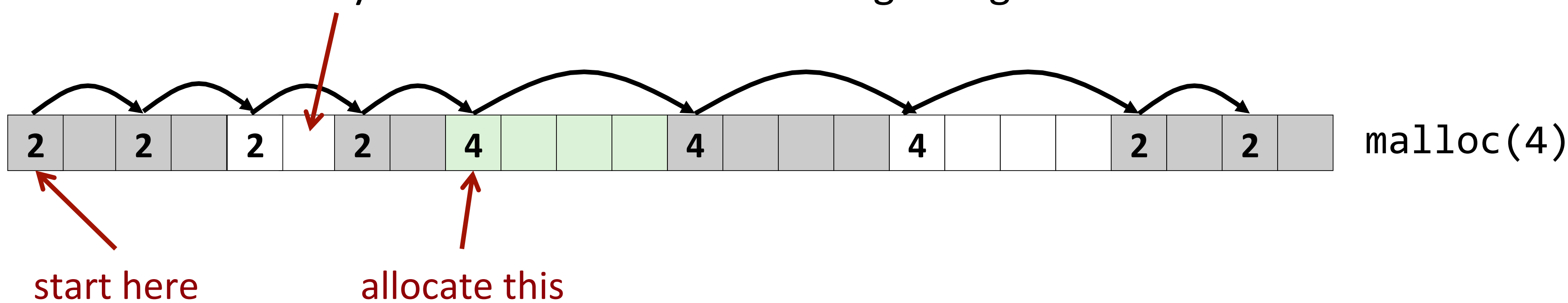
- Each *size class* of blocks has its own free list

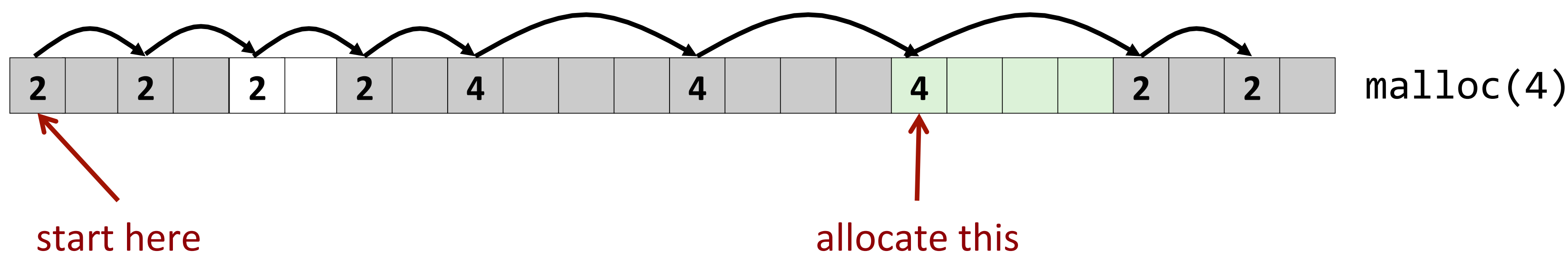


- Often have separate classes for each small size
- For larger sizes: One class for each two-power size

Finding a Free Block

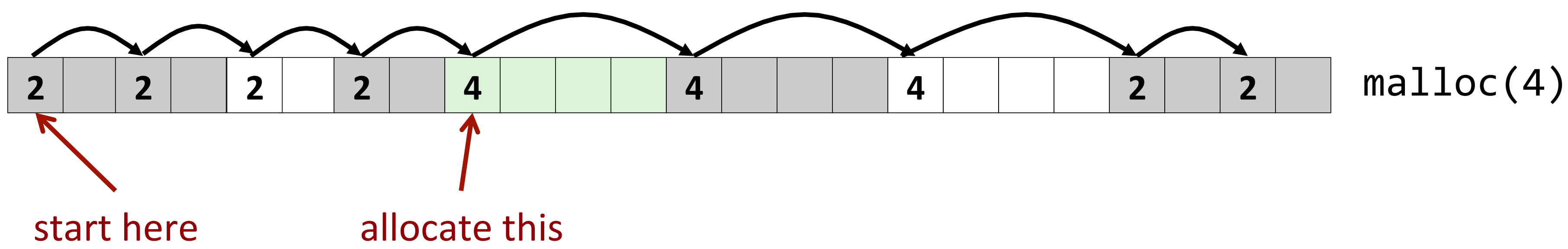
- **First fit:**
  - Search list from beginning, choose first free block that fits:
  - Can take linear time in total number of blocks (allocated and free)
  - In practice it can cause "splinters" at beginning of list
    - Many small free blocks left at beginning.

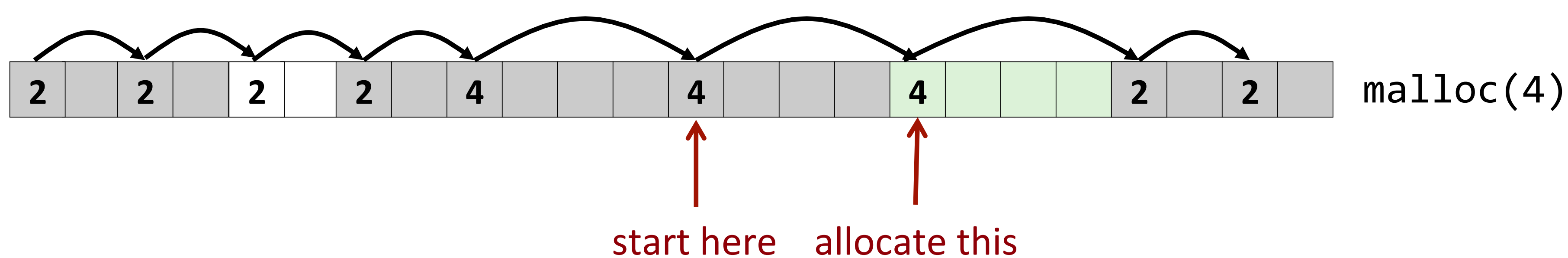




Finding a Free Block

- **Next fit:**
  - Like first fit, but search list starting where previous search finished
  - Should often be faster than first fit: avoids re-scanning unhelpful blocks
  - Some research suggests that fragmentation is worse





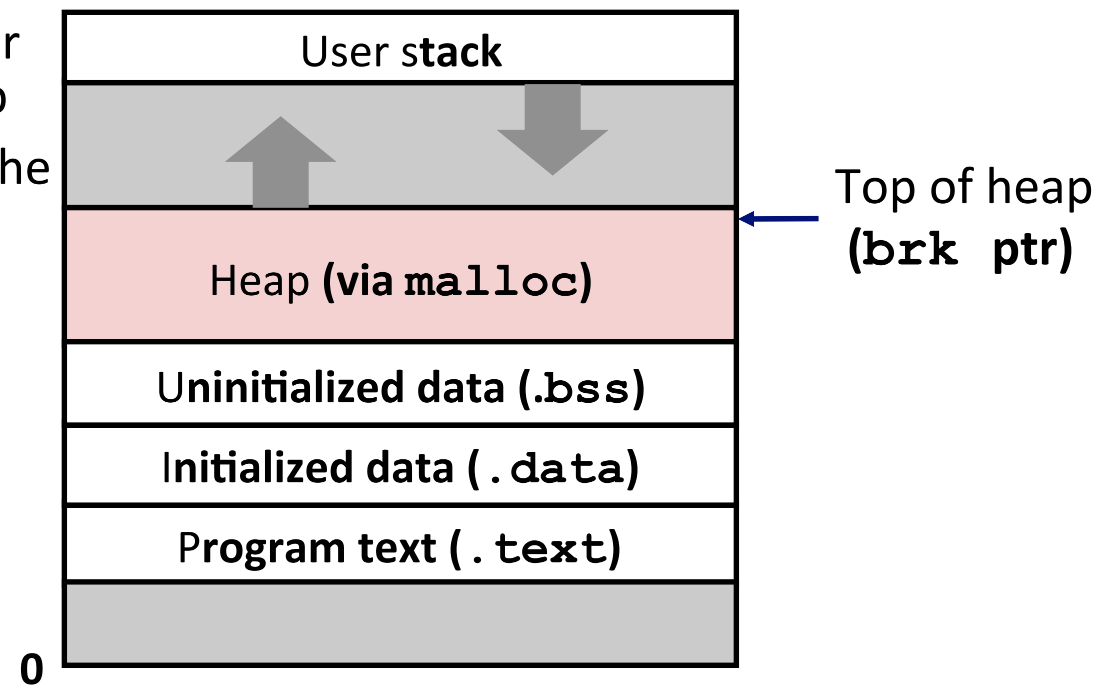
Finding a Free Block

- **Best fit:**
  - Search the list, choose the best free block: fits, with fewest bytes left over
  - Keeps fragments small—usually improves memory utilization
  - Will typically run slower than first fit

- **If the block we find is larger than we need, split it.**

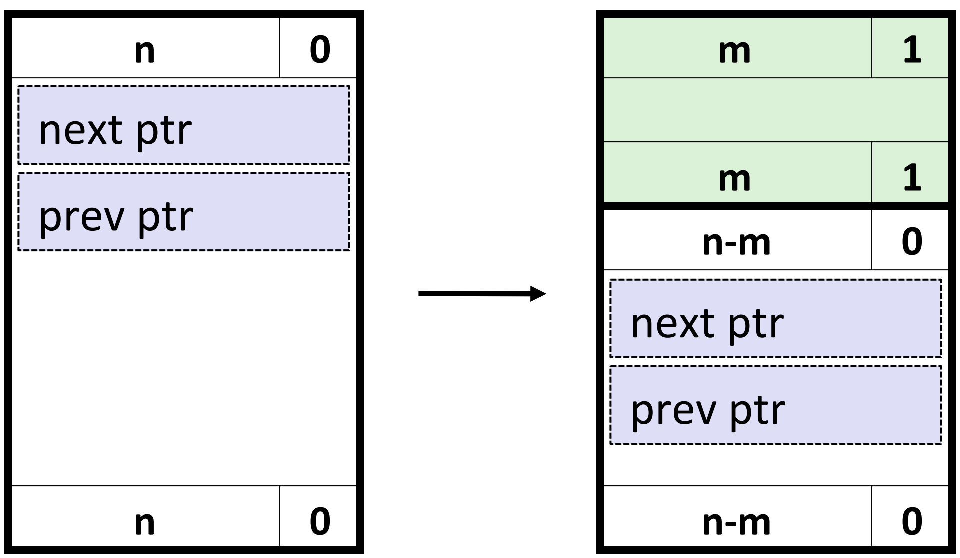
Finding a Free Block

- **What happens if we can't find a block?**
  - Need to extend the heap.
  - Use the brk() or sbrk() system calls.
    - In mallocLab, use mem_sbrk()
    - sbrk(*requested space*) allocates space and returns pointer to start of space
    - sbrk(0) returns pointer to top of current heap
  - Use what you need, add the rest as a free block.



Splitting a Block

- **What happens if the block we have is too big?**
  - Split between portion we need and free space.
  - For implicit lists: correct size maintains list
  - For explicit lists:
    - (if segregated) determine correct size list
    - Insert with insertion policy (we'll talk about this momentarily)

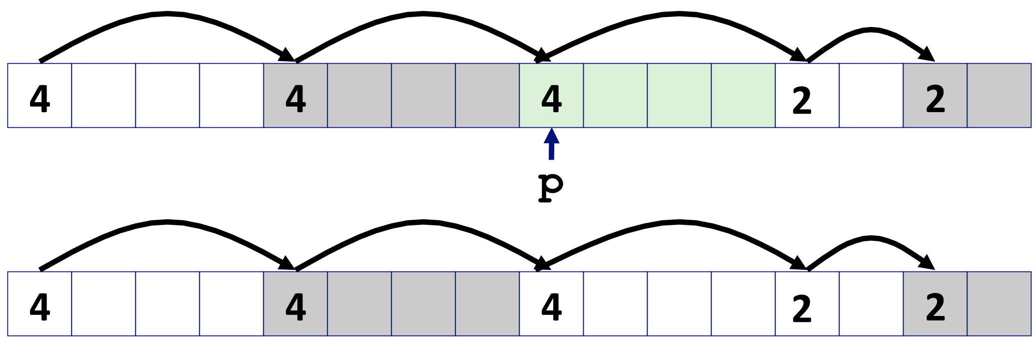


Freeing Blocks

- **Simplest implementation:**
  - Need only clear the "allocated" flag

```
void free_block(ptr p) { *p = *p & -2 }
```

  - But can lead to "false fragmentation"



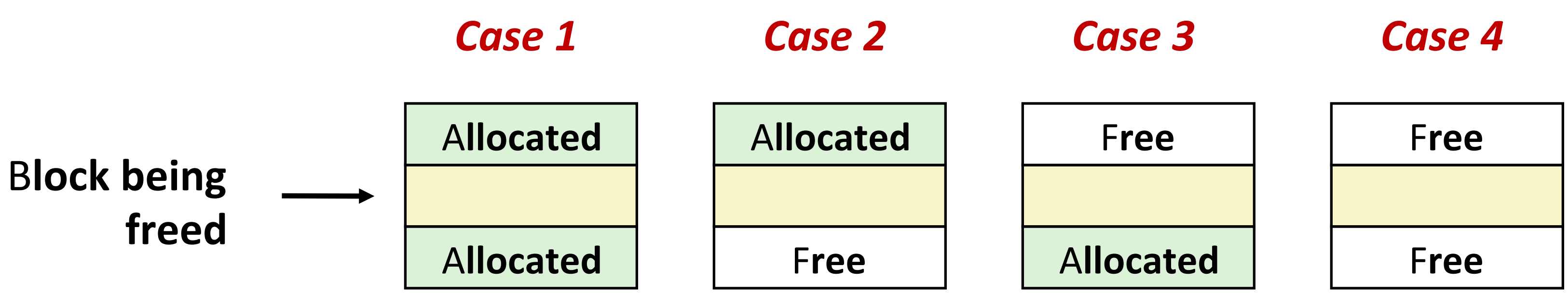
`free(p)`

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

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

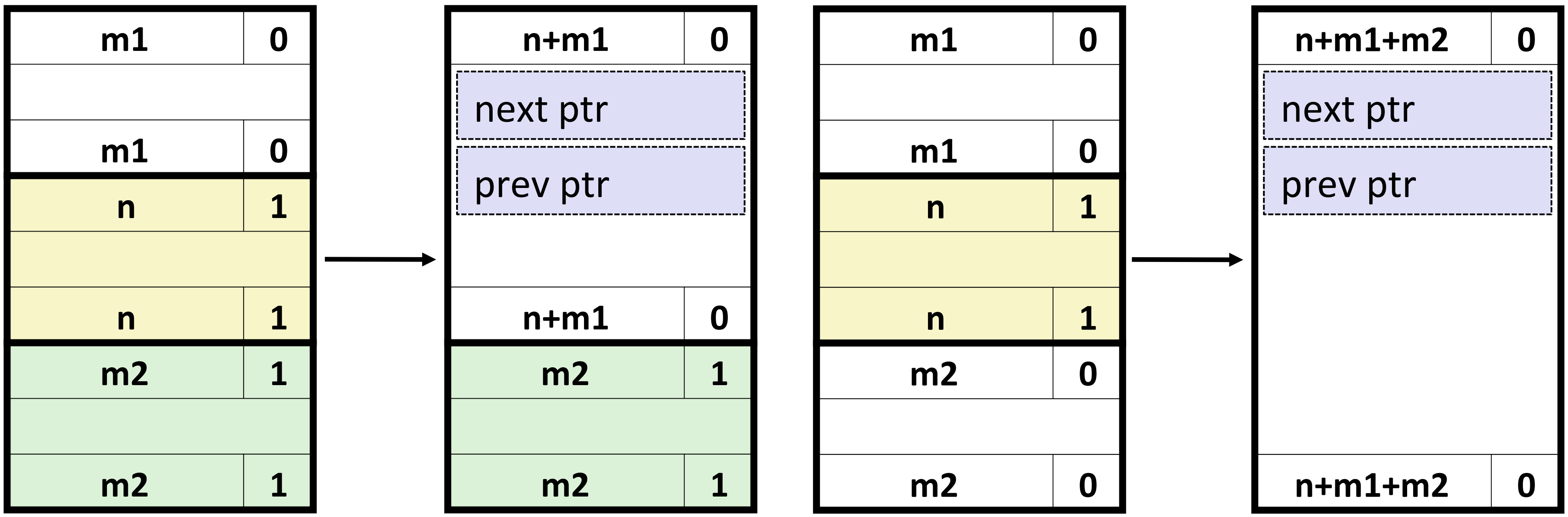
Freeing Blocks

- **Need to combine blocks nearby in memory.**
- **For implicit lists:**
  - Simply look backwards and forwards using block sizes.
- **For explicit lists:**
  - Look backwards/forwards using block sizes, not next/prev pointers.
  - If seg. list, use the size of new block to determine proper list
  - Insert back into list based on insertion policy (LIFO, FIFO)



Freeing Blocks

- **Graphical depiction (both implicit & explicit):**
  - (these are physical mappings)



Insertion Policy

- **Where in the free list do you put a newly freed block?**

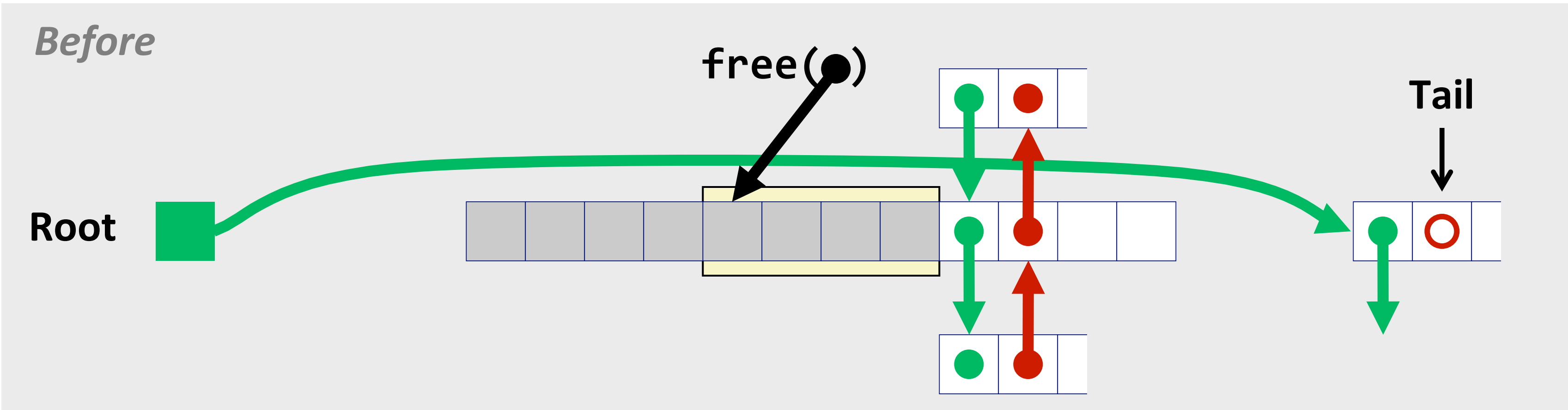
- **LIFO (last-in-first-out) policy**
  - Insert freed block at the beginning of the free list
  - ***Pro:*** simple and constant time
  - ***Con:*** studies suggest fragmentation is worse than address ordered

- **Address-ordered policy**
  - Insert freed blocks so that free list blocks are always in address order:

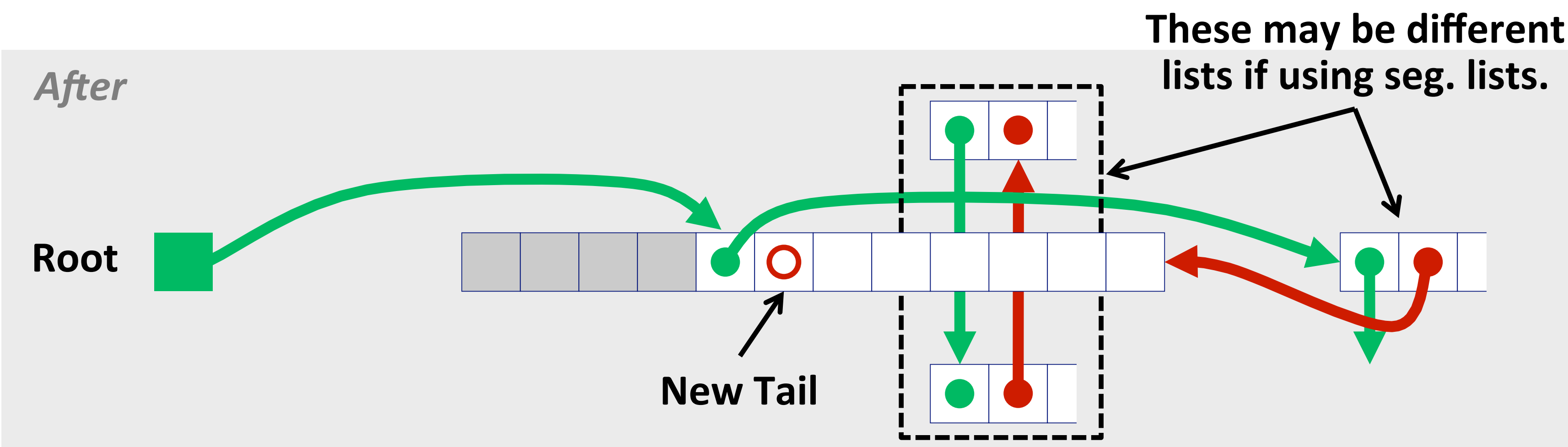
    addr(prev) < addr(curr) < addr(next)
  - ***Con:*** requires search
  - ***Pro:*** studies suggest fragmentation is lower than LIFO

Freeing Blocks (LIFO Policy)

conceptual graphic



- Splice out successor block, coalesce both memory blocks and insert the new block at the root of the list



MallocLab Tips

- **You need to implement the following functions:**

```
int mm_init(void);
void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
void *calloc (size_t nmemb, size_t size);
void mm_checkheap(int);
```

- **Scored on space efficiency and throughput**
- **Cannot call system memory functions**
- **Use helper functions (as static/inline functions)**
- **May want to consider practicing version control**

MallocLab Tips

- `void mm_checkheap(int)` **is critical for debugging.**
  - Write this early, and update it when you change cache topology
  - It should ensure that you haven't lost control of any part of heap memory (everything should either be allocated or listed)
  - Optionally test for consecutive free blocks. (This is bad.)
  - Look over lecture notes on garbage collection (particularly mark & sweep).
  - This function is meant to be correct, not efficient.

MallocLab Tips

- `inline`
  - Essentially copies function code into location of each function call.
  - Avoids overhead of stack discipline/function call (once assembled).
  - Can often be used in place of macros.
  - Strong type checking and input variable handling, unlike macros.
- `static`
  - We've discussed static variables – this is same.
  - Resides in a single place in memory
  - Limits scope of function to the current translations unit (file)
    - Should use this for helper functions only called locally
    - Avoids polluting namespace.
- `static inline`
  - Not surprisingly, can be used together.

Debugging

- **Using printf, assert, etc only in debug mode:**
-
```
#define DEBUG -or- //#define DEBUG
#ifdef DEBUG
  # define dbg_printf(...) printf(__VA_ARGS__)
  # define dbg_assert(...) assert(__VA_ARGS__)
  # define dbg(...)        __VA_ARGS__
#else
  # define dbg_printf(...)
  # define dbg_assert(...)
  # define dbg(...)
#endif
```

Debugging

- **Valgrind**
  - Powerful debugging and analysis technique
  - Rewrites text section of executable object file
  - Can detect all errors as debugging **`malloc`**
  - Can also check each individual reference at runtime
    - Bad pointers
    - Overwriting
    - Referencing outside of allocated block
- **GDB**
  - You know how to use this (hopefully).