Dynamic Memory Allocation

15-213: Introduction to Computer Systems
Recitation 11: Monday, March 31th, 2014

Donghui Luan

Section E

Today

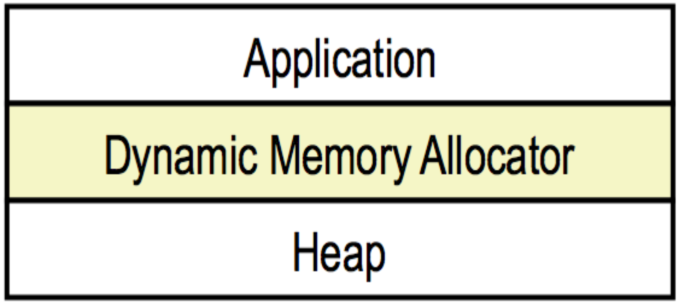
- Lecture Review
- Macros and Inline Functions
- Malloclab
- Heap Checker

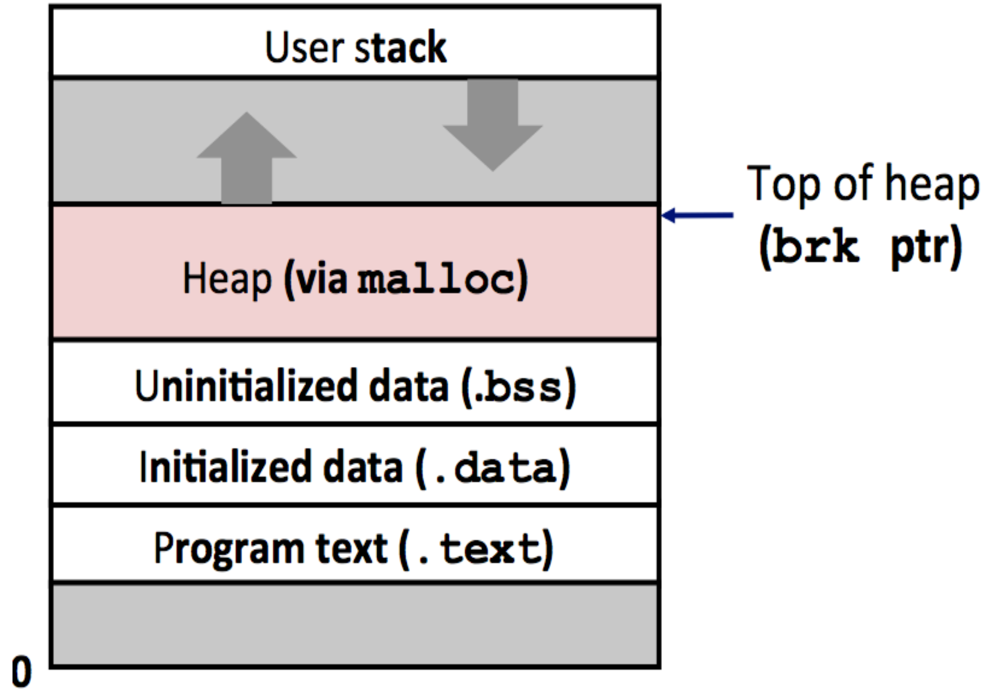
Today

- *Lecture Review*
- Macros and Inline Functions
- Malloclab
- Heap Checker

Dynamic Memory Allocation

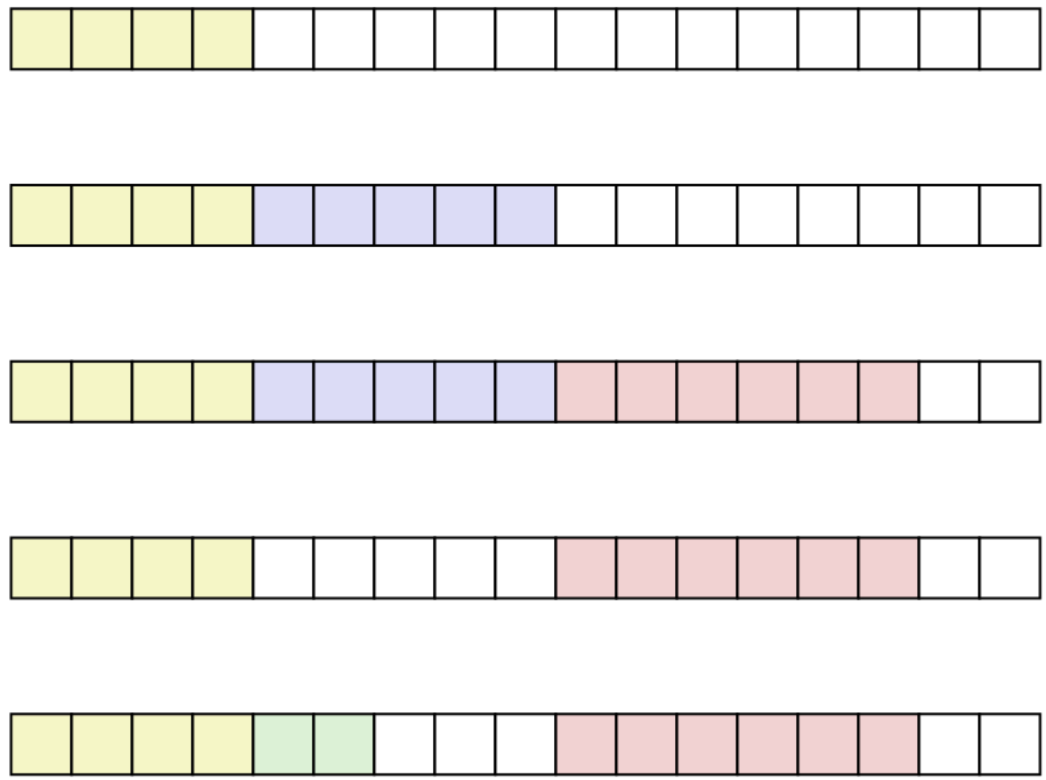
- Programmers use *dynamic memory allocators* (such as malloc) to acquire VM at run time.
- Dynamic memory allocators manage an area of process virtual memory known as the *heap*.





Dynamic Memory Allocation

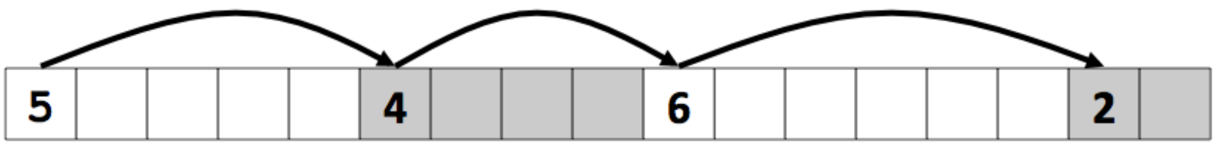
```
p1 = malloc(4)

p2 = malloc(5)

p3 = malloc(6)

free(p2)

p4 = malloc(2)
```



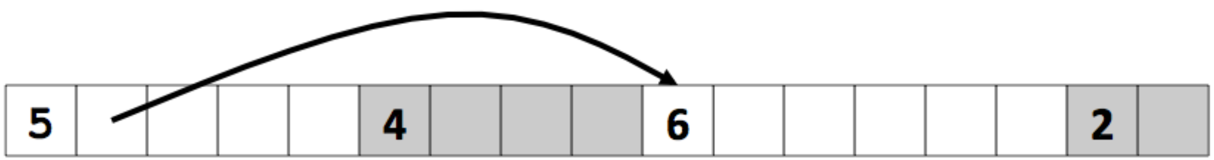
- **How do we know where to put the next block?**

Keeping Track of Free Blocks

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



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

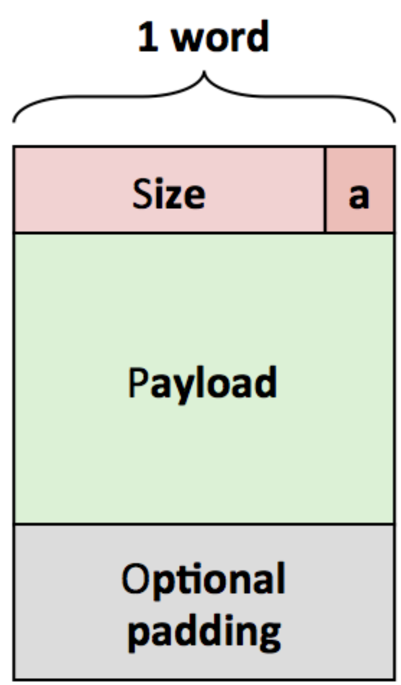


- Method 3: *Segregated free list*
  - *Different free lists for free blocks of different size classes*

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

Format of allocated and free blocks



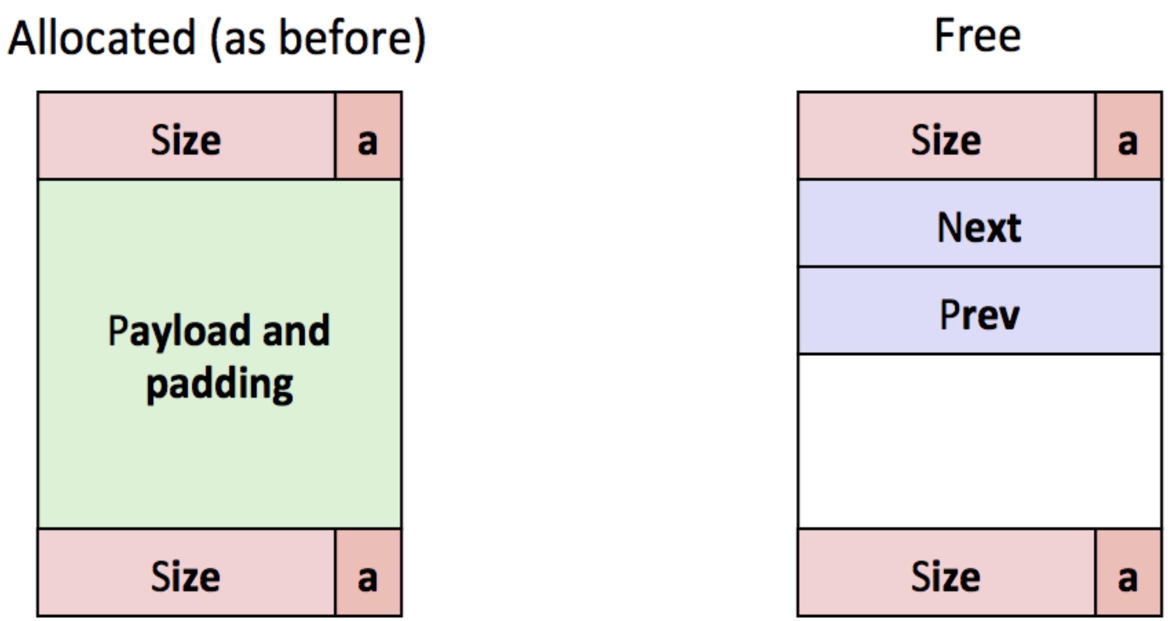
a = 1: Allocated block
a = 0: Free block

Size: block size

Payload: application data (allocated blocks only)

Method 2: Explicit List

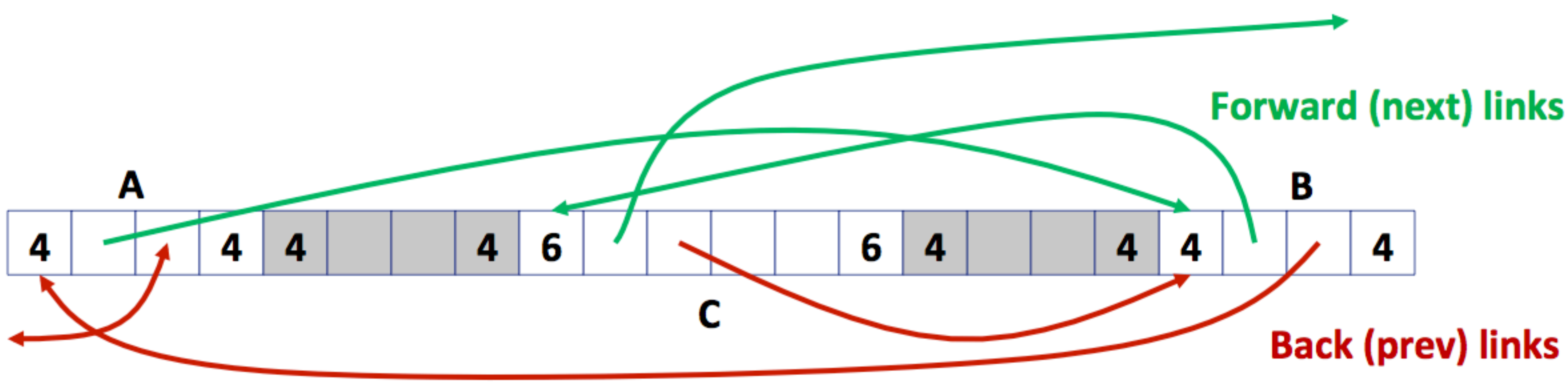
- **Maintain list(s) of *free* blocks instead of *all* blocks**
  - The "next" free block could be anywhere
  - So we need to store forward/back pointers, not just sizes
  - Still need boundary tags for coalescing
- *Luckily we track only free blocks, so we can use payload area*



Method 2: Explicit Free Lists

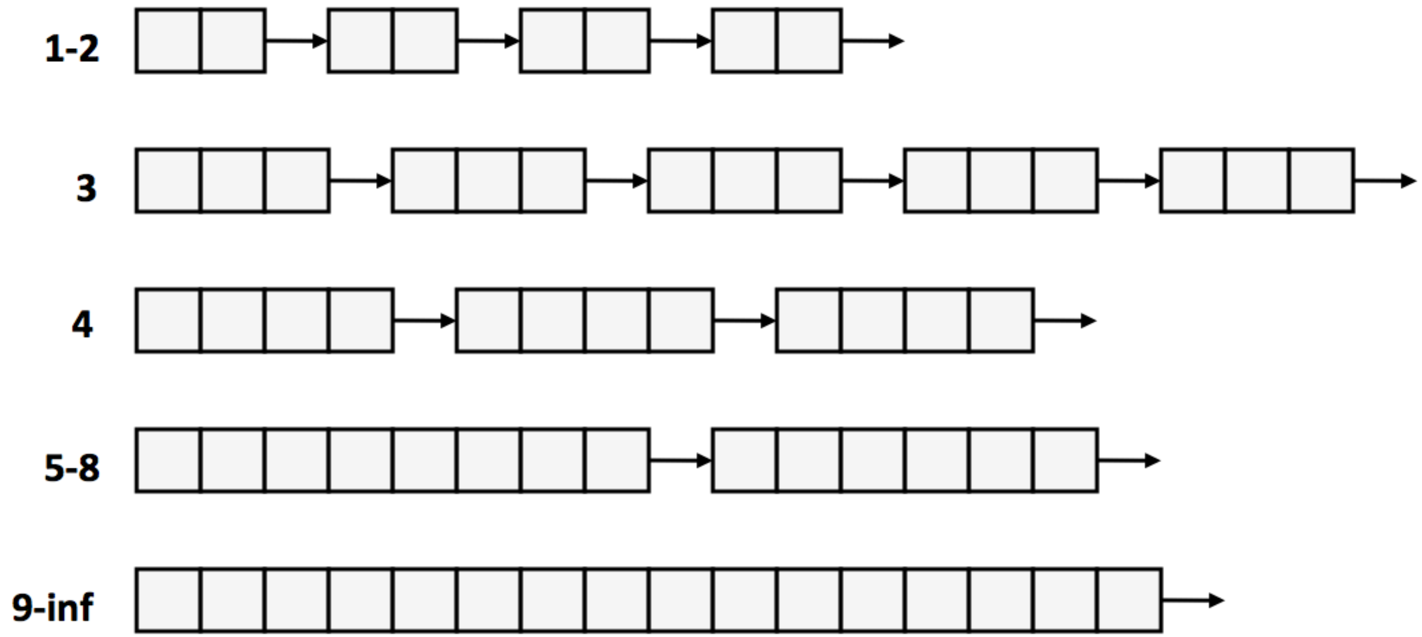
- Logically…

- But physically…



Method 3: Segregated List

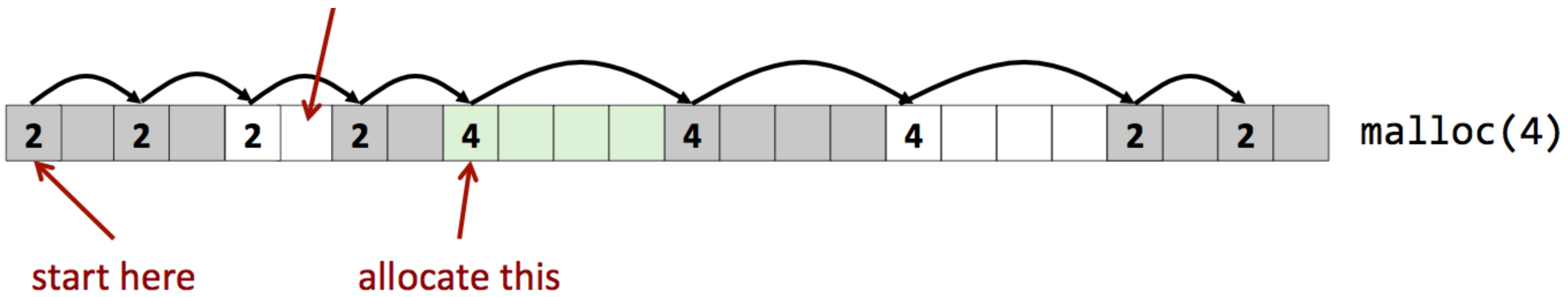
- **Each *size class* of blocks has its own free list**

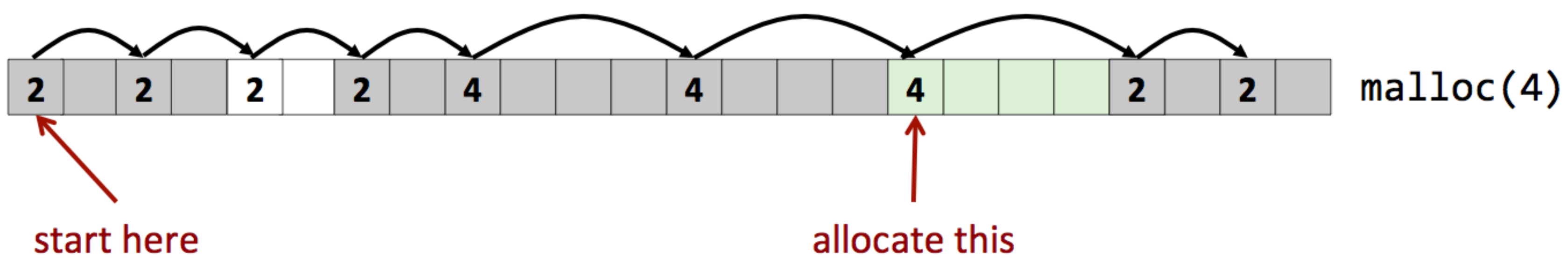


- Small sized blocks: more lists for separate classes
- Larger sizes: one class for each two-power size

Finding a Free Block

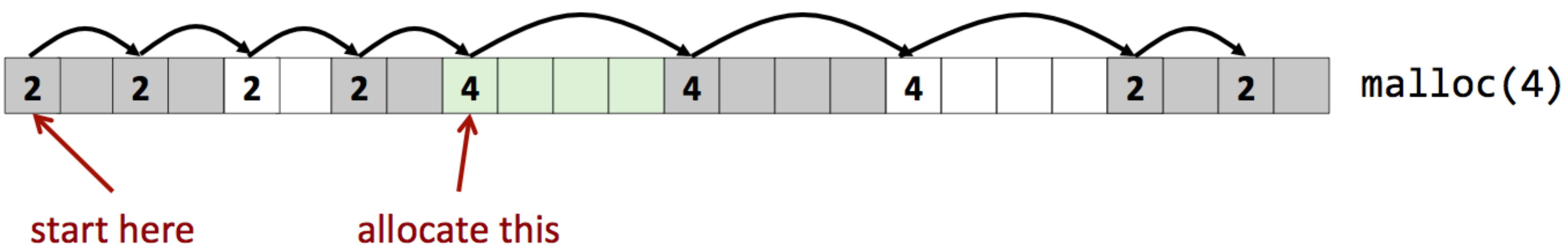
- **First fit:**
  - Search list from beginning, choose first free block that fits
  - Can take linear time in total number of blocks (allocated and free)
- In practice it can cause “splinters” at beginning of list
  - Many small free blocks left at beginning

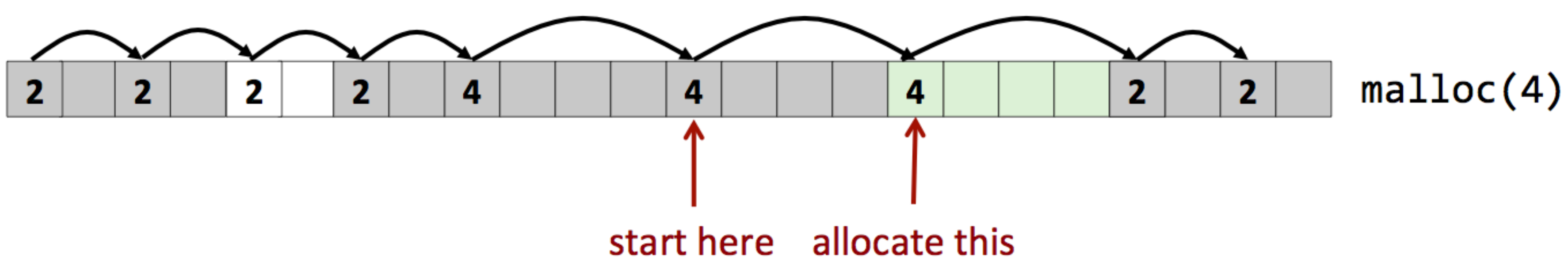




Finding a Free Block

- **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



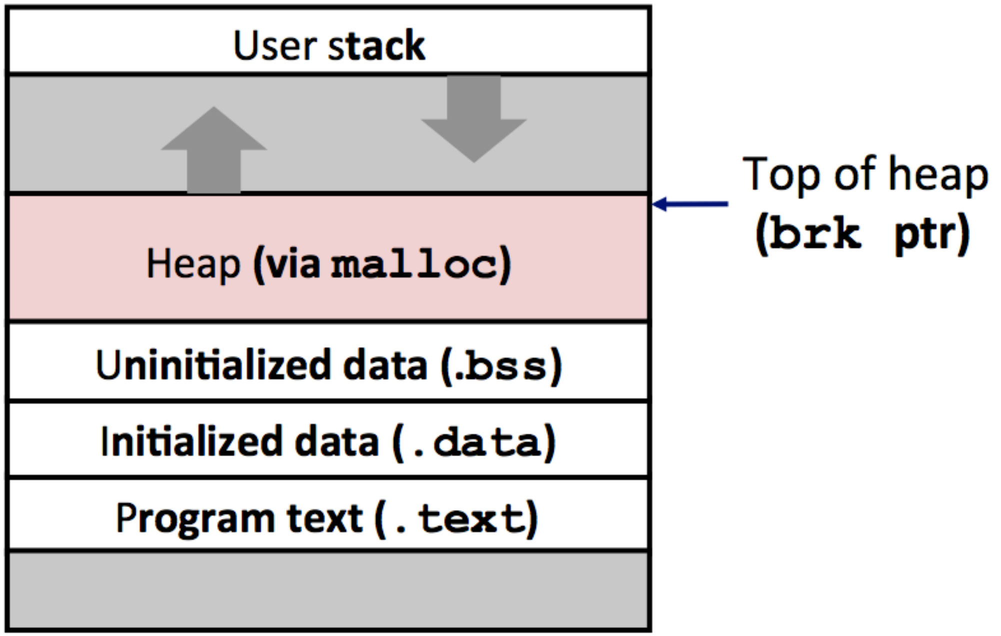


Finding a Free Block

- **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**

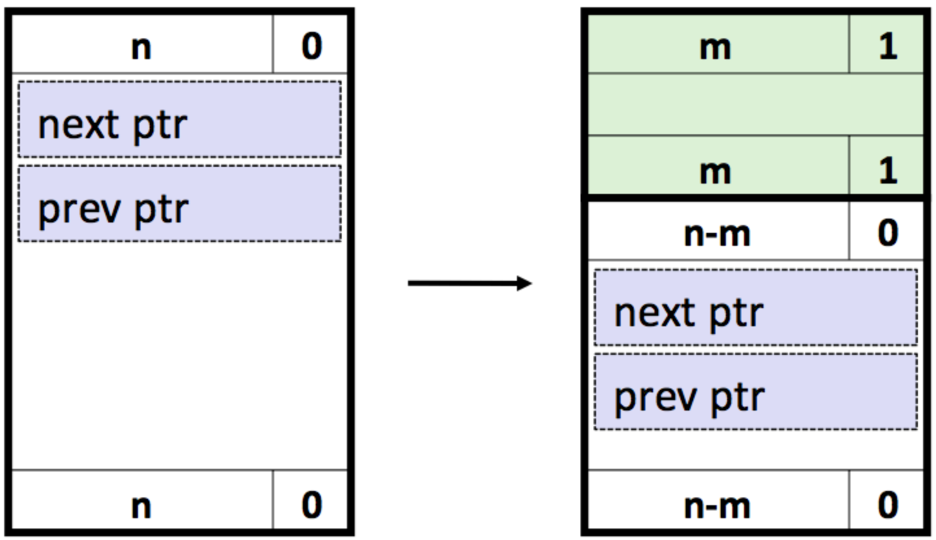
Finding a Free Block

- **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 new space
    - sbrk(0) returns pointer to top of current heap
  - Use what you need, add the rest as a whole free block



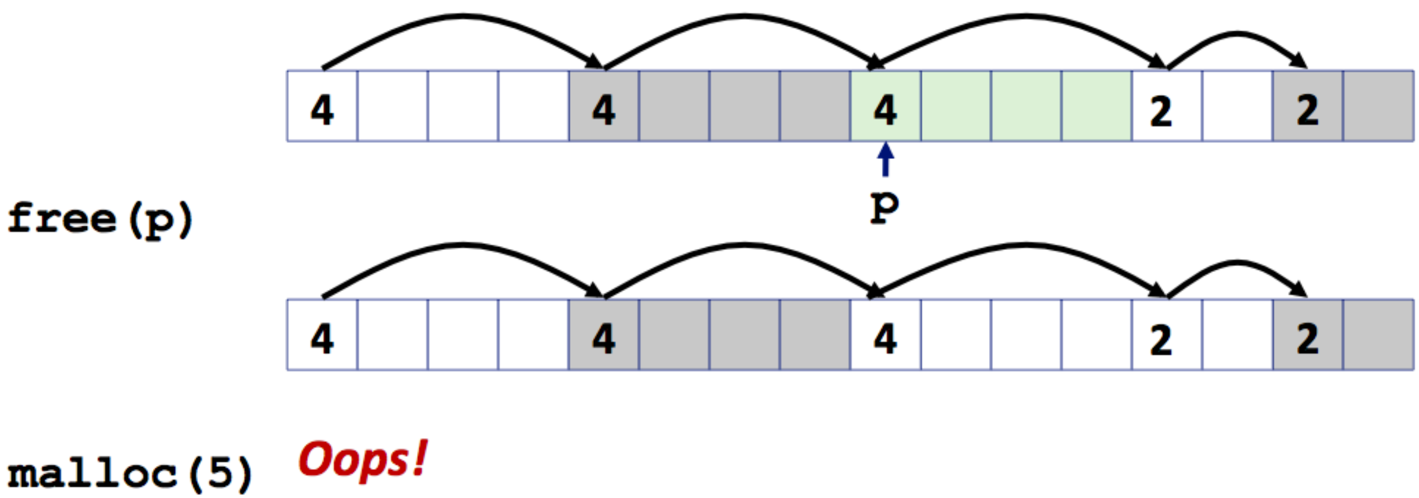
Splitting a Block

- **What happens if the block we have is too big?**
  - Split between portion we need and the leftover free space
  - For implicit lists: correct the block size
  - For explicit lists: correct the previous and next pointers
  - For segregated lists:
    - determine correct size list
    - Insert with insertion policy (more on this later)



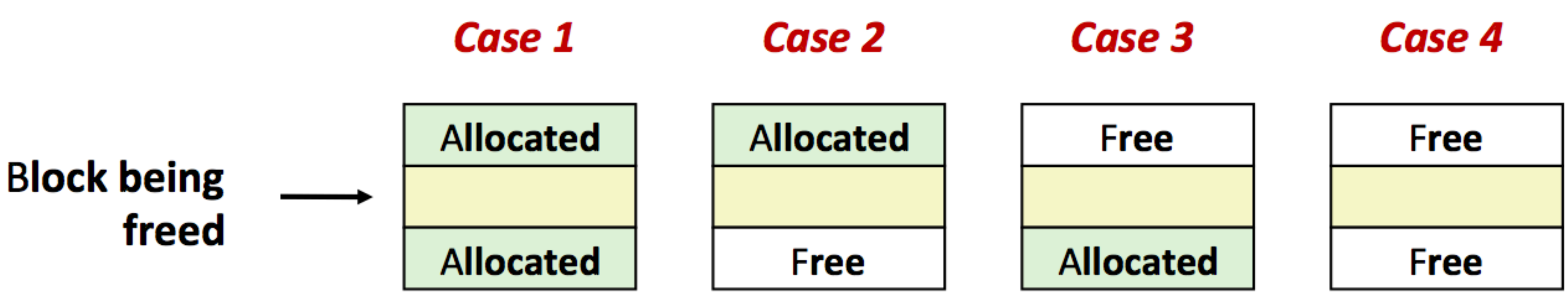
Freeing Blocks

- **Simplest implementation:**
  - Need only clear the "allocated" flag
    void free_block(ptr p) { *p = *p & -2 }
  - But can lead to external fragmentation:
    - There is enough free space, but the allocator can't find it



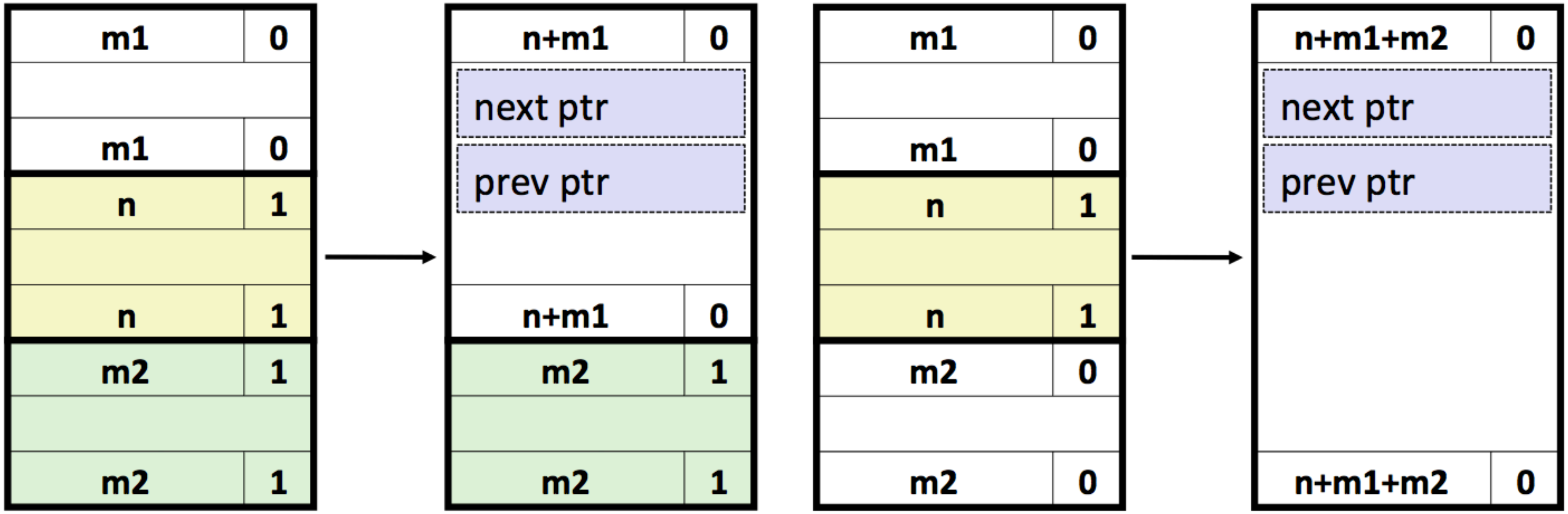
Freeing Blocks

- **Need to combine blocks nearby in memory (coalescing)**
- **For implicit lists:**
  - Simply look backwards and forwards using block sizes
- **For explicit lists:**
  - Look backwards/forwards using block sizes, not next/prev pointers
- **For segregated lists:**
  - 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:*** fragmentation is lower than LIFO

Today

- Lecture Review
- *Macros and Inline Functions*
- Malloclab
- Heap Checker

Macros

- C Preprocessor looks at macros in the preprocessing step of compilation
- Use #define to avoid magic numbers:
  - #define TRIALS 100
- Function like macros – short and heavily used code snippets
  - #define GET_BYTE_ONE(x) ((x) & 0xff)
  - #define GET_BYTE_TWO(x) (((x) >> 8) & 0xff)
- Inline functions
  - Ask the compiler to insert the complete body of the function in every place that the function is called (simply replacing code)
  - inline int fun(int a, int b)
  - Requests compiler to insert assembly of fun wherever a call to fun is made
- Both are useful for malloclab:D

Assert()

- assert(expr)
  - If expr is false, the calling process is terminated
  - If expr is true, it does nothing
- May be turned off at compile time with option –DNDEBUG
- As always, “Man is your friend.”
- For style points: you MUST use asserts in your code

Debugging

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

Today

- Lecture Review
- Macros and Inline Functions
- *Malloclab*
- Heap Checker

Malloclab

- **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 n, size_t size);
  - void mm_checkheap(int verbose);
- **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

- **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**
  - 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

Today

- Lecture Review
- Macros and Inline Functions
- Malloclab
- ***Heap Checker***

Heap Checker

- **Int mm_checkheap(int verbose) is critical for debugging**
  - Write this early
  - update it when you change your free list implementation
  - It should ensure that you haven't lost control of any part of heap memory (everything should either be allocated or listed)
- Look over lecture notes on garbage collection (particularly mark & sweep).
- This function is meant to be correct, not efficient.

Heap Checker

- Once you’ve settled on a design, write the heap checker that checks all the invariants of the particular design
- The checking should be detailed enough that the heap check passes if and only if the heap is truly well-formed
- Call the heap checker before/after the major operations whenever the heap should be well-formed
- Define macros to enable/disable it conveniently
  - e.g.

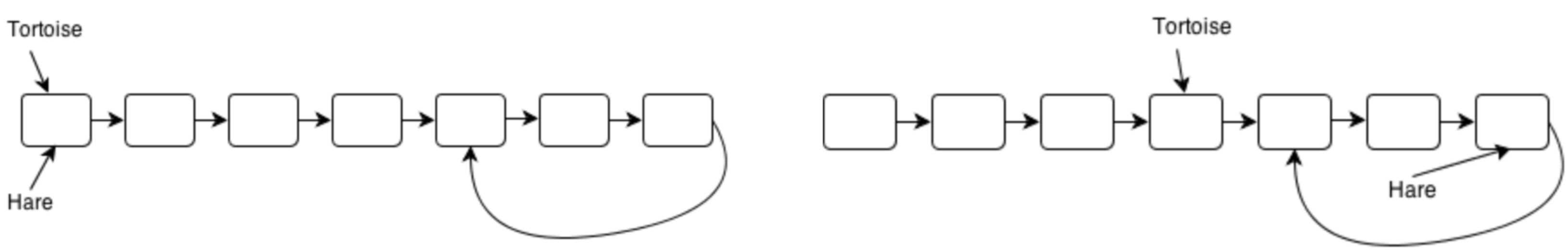
```
#ifdef DEBUG
#define CHECKHEAP(verbose) printf("%s\n", __func__); mm_checkheap(verbose);
#endif
```

Invariants (non-exhaustive)

- Block level:
  - Header and footer match
  - Payload area is aligned
- List level:
  - Next/prev pointers in consecutive free blocks are consistent
  - Free list contains no allocated blocks
  - All free blocks are in the free list
  - No contiguous free blocks in memory (unless you defer coalescing)
  - No cycles in the list (unless you use circular lists)
  - Segregated list contains only blocks that belong to the size class
- Heap level:
  - Prologue/Epilogue blocks are at specific locations (e.g. heap boundaries) and have special size/alloc fields
  - All blocks stay in between the heap boundaries
- And your own invariants (e.g. address order)

Hare and Tortoise Algorithm

- Detects cycles in linked lists
- Set two pointers "hare" and "tortoise" to the beginning of the list
- During each iteration, move the hare pointer forward two nodes and move the tortoise forward one node. If they are pointing to the same node after this, the list has a cycle.
- If the tortoise reaches the end of the list, there are no cycles.



Asking for help

- It can be hard for the TAs to debug your allocator, because this is a more open-ended lab
- Before asking for help, ask yourself some questions:
  - What part of which trace file triggers the error?
  - Around the point of the error, what sequence of events do you expect?
  - What part of the sequence already happened?
- If you can't answer, it's a good idea to gather more information…
  - How can you measure which step worked OK?
  - printf, breakpoints, watchpoints…

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)

Beyond Debugging: Error prevention

- It is hard to write code that are completely correct the first time, but certain practices can make your code less error -prone
- Plan what each function does before writing code
  - Draw pictures when linked list is involved
  - Consider edge cases when the block is at start/end of list
- Document your code as you write it

Questions?

- Good luck!:D