Michael's Fairly Satisfactory Recitation Slides

(Mostly adapted from Anita's Super Awesome Recitation slides)

15/18-213: Introduction to Computer Systems

Dynamic Memory Allocation

UPDATES

- Shell Lab due tonight
- Midterm next week
- Malloc Lab due Tuesday, July 23, 2013

Welcome to Malloc

- Dynamic Memory Allocation
  - Managing Free Blocks
  - Finding a Free Block
  - Splitting Blocks
  - Allocating/ Freeing Blocks
- Malloc Lab Tips

Mal = bad

Loc = place

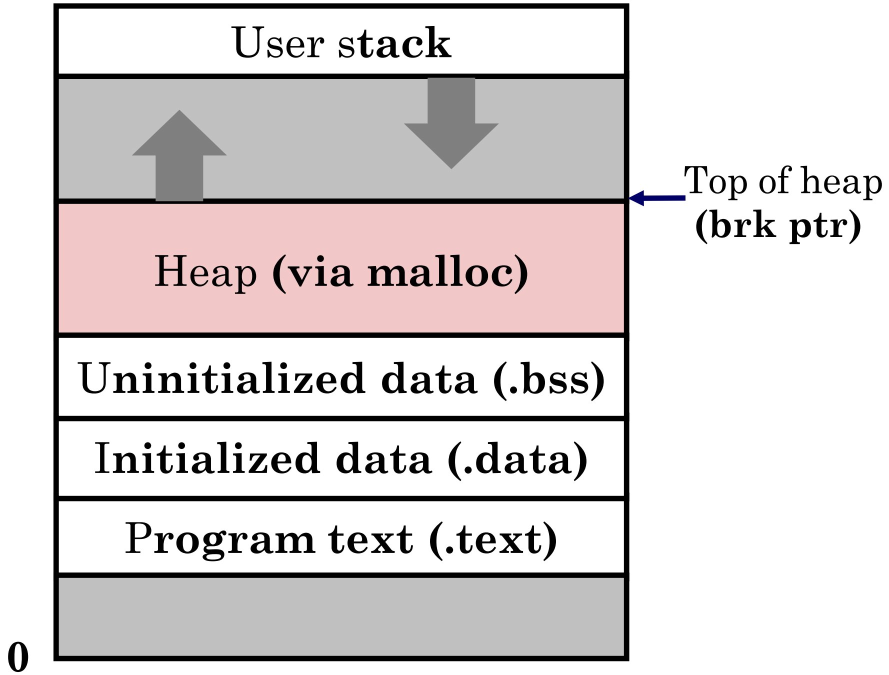
Bad Place™

Remember This?

Rubber Duck Debugging

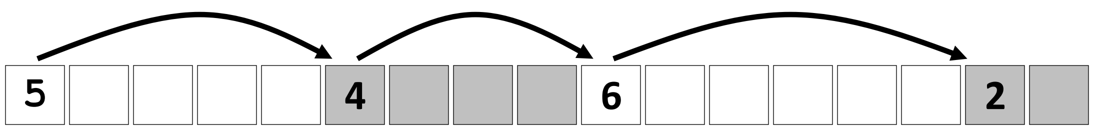
DYNAMIC MEMORY

- Programmers use dynamic memory allocators (i.e. `malloc`) to acquire memory
  - For sizes only known at runtime
- Dynamic memory allocators manage an area of process virtual memory known as the heap

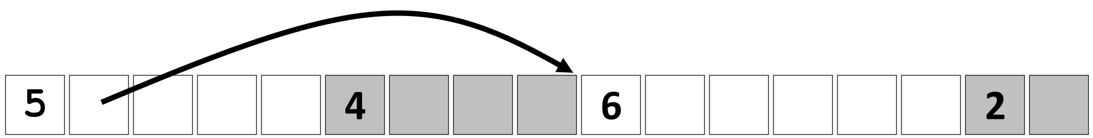


Managing Free Blocks

- Method 1: Implicit list using length– links all blocks



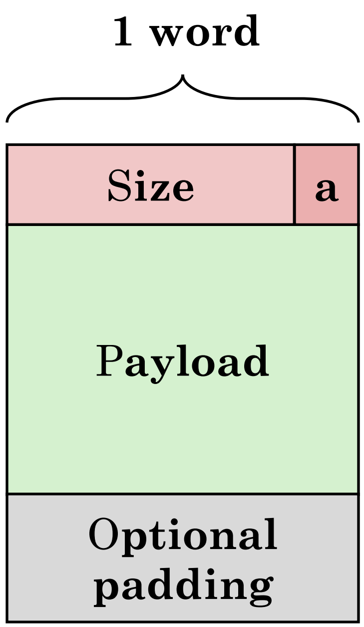
- Method 2: Explicit list among the free blocks using pointers



- 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

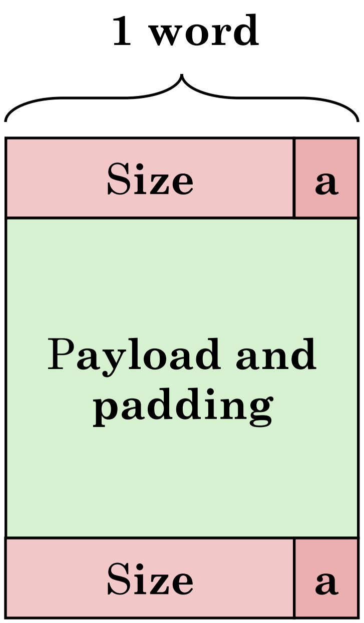
Comparing Management Formats

Implicit Free List

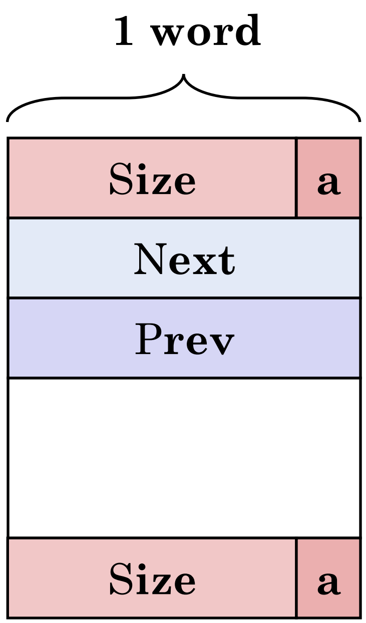


Format of allocated and free blocks

Explicit Free List



Allocated blocks

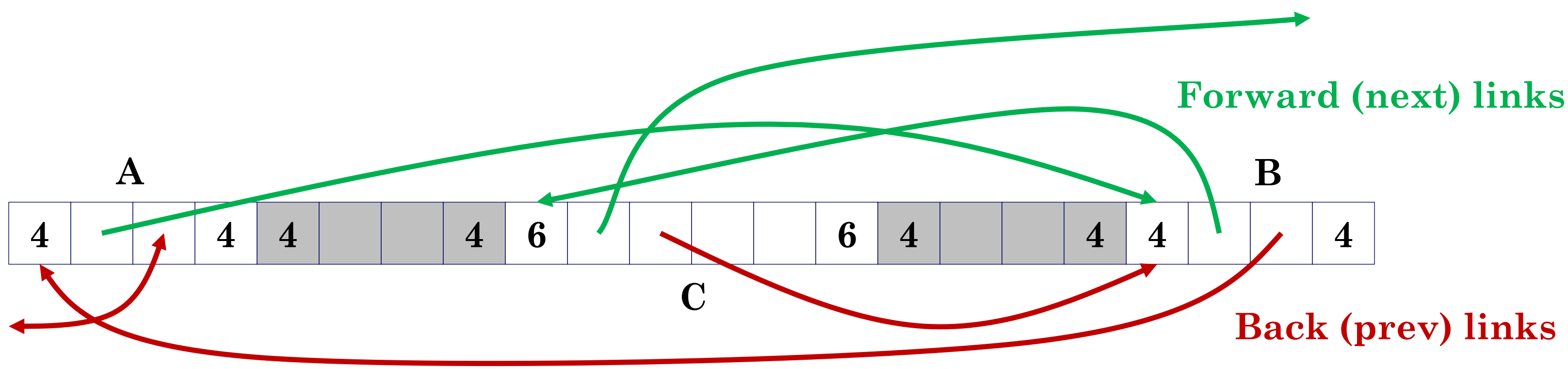


Free blocks

Visualizing Explicit Free Lists

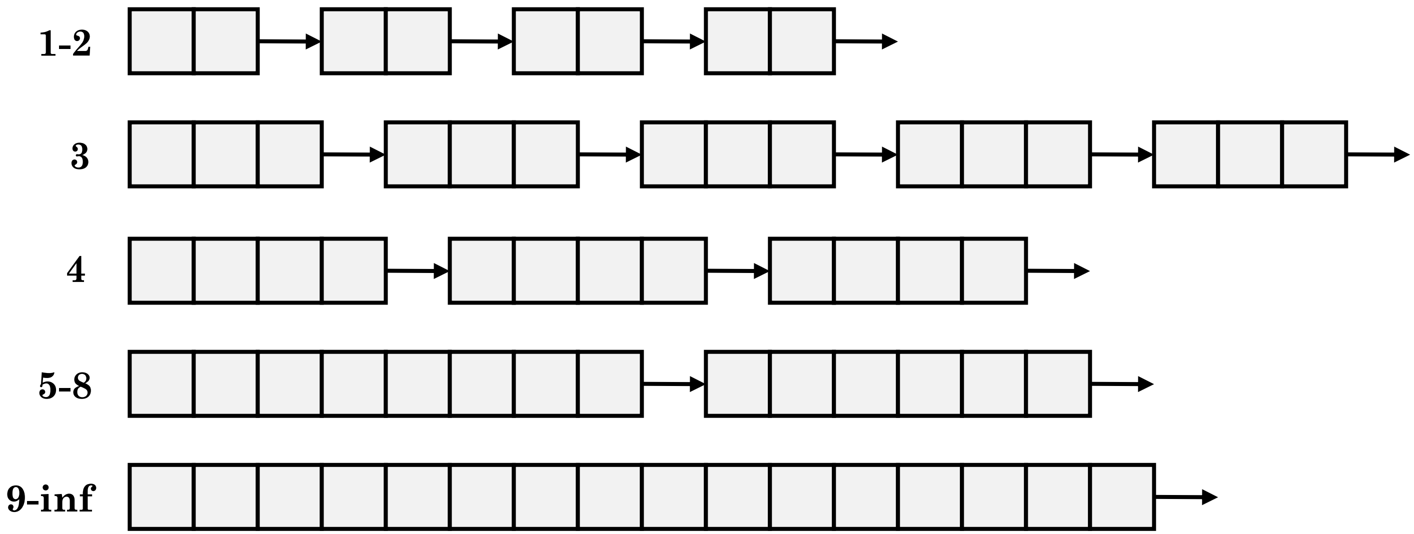
- Logically:

- Physically (any order):



Segregated Free Lists

- Each *size class* of blocks has its own free list
  - May also be called a “bucket”



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

Finding Free Blocks

- First fit:
  - Search from the beginning
  - Choose the first free block that fits
  - Can take linear time depending on the total number of blocks in the list
  - Can cause "splinters" at the beginning of list
    - Many small free blocks left at the beginning

Finding Free Blocks

- Next fit
  - Searches starting where previous search finished
  - Often faster than first fit
    - Avoids re-scanning blocks of the wrong size
  - Some research suggests that fragmentation is worse
  - K&R has an example of this

Finding Free Blocks

- Best fit
  - Chooses the “best” fitting free block
    - Fits with the fewest bytes left over
  - Keeps fragments small
    - Usually improves memory utilization
  - Will typically run slower than first fit
  - If the best block is larger than we need, may split it

Finding Free Blocks Overview

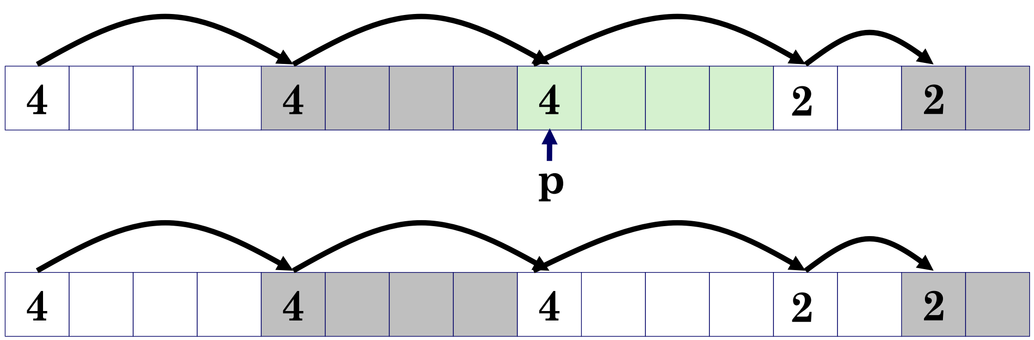
- 3 Methods
  - First Fit
  - Next Fit
  - Best Fit
- What if no blocks are large enough?
  - Extend the heap
    - Use brk() or sbrk() system calls
    - Malloc Lab: use mem_sbrk()
    - Allocates more bytes to the end of the heap; high overhead
    - sbrk(0) returns a pointer to top of the current heap
  - Key: Use what you need, save the rest as a free block

Splitting Blocks

- What happens if the block we have is too big?
  - Split it up
    - Key: Use what you need, save the rest as a free block
  - Implicit lists
    - Correct size maintains list
  - Explicit lists
    - (If segregated) determine correct bucket size
    - Follow insertion policy

Properly Freeing Blocks

- Simplest implementation:
  - Need only clear the "allocated" flag
    - `void free_block(ptr p) { *p = *p & -2 }`
  - ...But can lead to "false fragmentation"



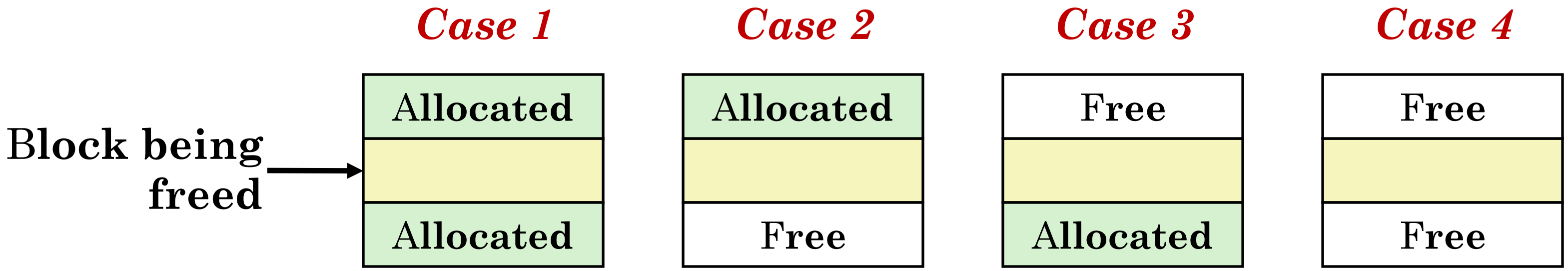
free(p)

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

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

Coalescing

- Combining blocks in nearby memory
- Implicit lists
  - Look backwards/ forwards using block sizes.
- Explicit lists
  - Look backwards/ forwards using block sizes
  - Seg. List: Use the new block size to find the bucket



Insertion Policy

- Where should freed blocks go?
- LIFO (last-in-first-out)
  - 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
  - Keep freed blocks list sorted in address order
  - **Pro**: Studies suggest fragmentation is lower than LIFO
  - **Con**: Requires searching

About Malloc Lab

- 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 efficiency and throughput
- Cannot call system memory functions
- Use helper functions
- Consider version control

Design Questions (in no order)

- How do we efficiently manage freed blocks?
- When should we coalesce?
- What are the ideal bucket sizes?
- How can we increase throughput? Latency?
- Which search algorithm is better?
- What insertion policy should I use?

Heap Checker

- `void mm_checkheap(int)`
  - Write it early; update it with your implementation
  - Ensures the heap is “sane”
    - Everything should either be allocated or listed
    - Your pointers are pointing to the correct blocks
  - Look over lecture notes on garbage collection
    - Particularly mark & sweep
  - This function is meant to be correct, not efficient

KEYWORDS

- `inline`
  - "Copies" function code into location of each function call
  - Avoids overhead of a function call (once assembled)
  - Can often be used in place of macros
  - Strong type checking and input handling, unlike macros
- `static`
  - Pretty much like static variables
    - Resides in a single place in memory
  - Limits scope of function to the current file
    - Should use this for helper functions only called locally
    - Avoids polluting namespace
- `static inline`
  - Combined effect

Debugging

- Using printf, assert, etc. only in debug mode
  - Comment out `#define` for the else case

```
#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
```

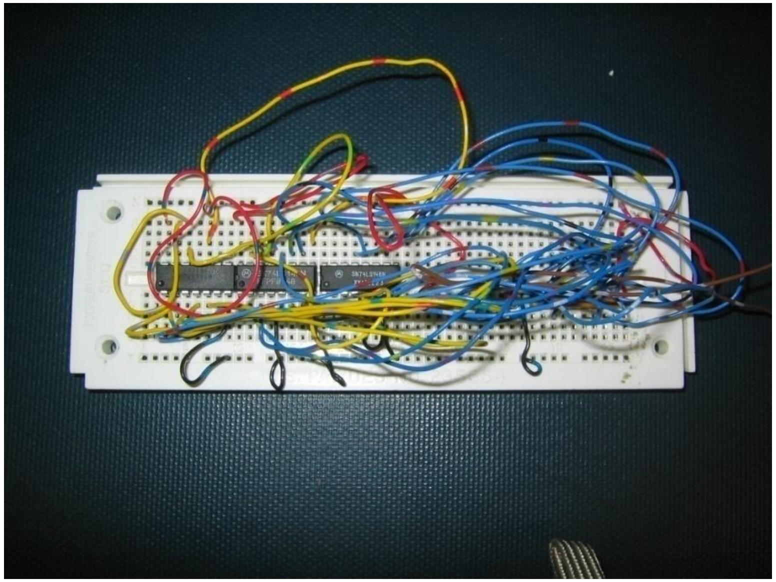
DEBUGGING

- Valgrind
  - Powerful debugging and analysis technique
  - Rewrites text section of executable object file
  - Can detect all errors as a "debugging malloc"
  - Can also check each individual reference at runtime
    - Bad pointers
    - Overwriting
    - Referencing outside of allocated block
- GDB
  - Pro Tip: The O2 flag is used in the Makefile
    - May give unexpected results when using GDB

Version Control

- Warning: You may have to rewrite your malloc once or twice in the next week
  - Use version control so you don't lose track
- Here's a good reference on Git

The Messy Circuit Analogy



Common Misconceptions

- "Global data structures" is not the same as declaring types
  - Use `mem_sbrk` to get space for your data structures

`typedef struct {`
`    int x;`
`    int y;`
`} point;`
	Vs.	`point a = {5, 6};`

- Casting is your friend in this lab
  - Data from `mem_sbrk` is like any other data
- The driver resets the heap by calling `mm_init`
  - May require you to update some of your pointers
- 64 bit addresses, but the heap $\leq 2^{32}$ bytes
  - Use this information as you see fit

Getting Started

- Read the 32 bit implicit list in CS:APP
  - Understand the macros, then steal them
- Don't copy and paste from the CS:APP website
  - Typing it yourself will give you epiphanies
  - The coalescing code provided is great
- Implement a 64 bit `malloc`
  - Super naïve and inefficient may be a good start
  - Implement `mm_checkheap` for this heap pattern

Getting More Points

- Implicit list malloc is worth no credit
  - Last checked it was worth ~40 points
- Explicit free list is expected
  - Gets you to the ~80 point range
- Update from explicit to segregated free lists
  - Puts you in the ~90 point range

Final Words From Previous Years

- Write `mm_checkheap`
- Write `mm_checkheap` well
- Write coalescing to make bugs more apparent, then fix bugs using `mm_checkheap`
- Start now
  - You'll be spending a lot of time pointer chasing
- Accelerate neutrinos past the speed of light, enabling you to start three days ago
- Good luck!

Questions and Credits Slide

- Rubber Duck
- Git Reference
- Some picture of a messy circuit