Malloc Lab

15-213: Introduction to Computer Systems
Recitation 11: Nov. 4, 2013
Marjorie Carlson
Recitation A
Weekly Update

- **Malloc lab is out**
  - Due Thursday, Nov. 14
  - Start early
  - Seriously... start early.
  - “It is possible to write an efficient malloc package with a few pages of code. However, we can guarantee that it will be some of the most difficult and sophisticated code you have written so far in your career.”
Agenda

- Malloc Overview
- Casting & Pointer Review
- Macros & Inline Functions
- Malloc Design
- Debugging & an Action Plan
Dynamic Memory Allocators

- Are used to acquire memory for data structures whose size is known only at run time.
- Manage area in a part of memory known as the heap.
Allocation Example

```c
p1 = malloc(4)
p2 = malloc(5)
p3 = malloc(6)
free(p2)
p4 = malloc(2)
```
Malloc Lab

- Create a general-purpose allocator that dynamically modifies the size of the heap as required.
- The driver calls your functions on various trace files to simulate placing data in memory.
- Grade is based on:
  - Space utilization (minimizing fragmentation)
  - Throughput (processing requests quickly)
  - Your heap checker
  - Style & correctness, hand-graded as always
Functions You Will Implement

- **mm_init**: initializes the heap before malloc is called.
- **malloc**: returns a pointer to a free block (\( \geq \text{req. size} \)).
- **calloc**: same, but zeros the memory first.
- **realloc**: changes the size of a previously allocated block. (May move it to another location.)
- **free**: marks allocated memory available again.
- **mm_checkheap**: debugging function (more on this later)
Functions You May Use

- **mem_sbrk**
  - Used for expanding the size of the heap.
  - Allows you to dynamically increase your heap size as required.
  - Helpful to initialize your heap.
  - Returns a pointer to first byte in newly allocated heap area.

- **mem_heap_lo**
  - Pointer to first byte of heap

- **mem_heap_hi**
  - Pointer to last byte of heap

- **mem_heapsize**

- **mem_pagesize**
Agenda

- Malloc Overview
- Casting & Pointer Review
- Macros & Inline Functions
- Malloc Design
- Debugging & an Action Plan
Pointer Arithmetic

*(arr + i) is equivalent to arr[i]

Thus the result of arithmetic involving pointers depends on the type of the data the pointer points at.

```
int *arr = 0x1000
arr + 1 = 0x1004
```

```
short *arr = 0x1000
arr + 1 = 0x1002
```

So ptr + i is really ptr + (i * sizeof(ptr-type))

example and pictures from http://www.cs.umd.edu/class/spring2003/cmsc311/Notes/BitOp-pointer.html
Pointer Casting

- Pointer casting can thus be used to make sure the pointer arithmetic comes out right.
- Since chars are 1 byte, casting a pointer as a char pointer then makes arithmetic on it work “normally.”

```c
int *ptr = 0x10203040

char *ptr2 = (char *)ptr + 2 = 0x10203042

char *ptr3 = (char *(ptr + 2)) = 0x10203048
```
Examples

1. int *ptr = (int *) 0x12341234;
   int *ptr2 = ptr + 1; = 0x12341238

2. char *ptr = (char *) 0x12341234;
   char *ptr2 = ptr + 1; = 0x12341235

3. void *ptr = (int *) 0x12341234;
   void *ptr2 = ptr + 1; = 0x12341235

4. int *ptr = (int *) 0x12341234;
   int *ptr2 = ((int *) (((char *) ptr) + 1)));
   = 0x12341235 😞
Agenda

- Malloc Overview
- Casting & Pointer Review
- Macros & Inline Functions
- Malloc Design
- Debugging & an Action Plan
Macros

#define  NAME  replacement-text

- Maps “name” to a definition or instruction.
- Macros are expanded by the preprocessor, i.e., before compile time.
- They’re faster than function calls.
- For malloc lab: use macros to give you quick (and reliable) access to header information — payload size, valid bit, pointers, etc.
Macros

- Useful for “magic number” constants – acts like a naïve search-and-replace
  - `#define ALIGNMENT 8`

- Useful for simple accesses and computations
  - Use parentheses for computations.
    - `#define multByTwoA(x) 2*x`
    - `#define multByTwoB(x) 2*(x)`

- `multByTwoA(5+1) = 2*5+1 = 11`
- `multByTwoB(5+1) = 2*(5+1) = 12`
Macros

- Useful for debugging
  - __FILE__ is the file name (%s)
  - __LINE__ is the line number (%d)
  - __func__ is the function it’s in (%s)

```c
#include <stdio.h>

int hello()
{
    printf("hello from function %s\n", __func__);  
}

int main()
{
    hello();
    printf("This is line %d\n", __LINE__);  
    printf("Belongs to function: %s\n", __func__);  
    printf("In filename: %s\n", __FILE__);  
}
```

Output:

```
hello from function hello
This is line 9.
Belongs to function: main
In filename: macros.c
```
Macros

- Useful for debugging: conditional printf statements

```c
// #define DEBUG

#ifdef DEBUG
#define dbg_printf(...) printf(__VA_ARGS__)
#else
#define dbg_printf(...) printf(__VA_ARGS__)
#endif
```
Inline Functions

- Alternative to macros: still more efficient than a function call, and easier to get right!
  
  ```
  #define max(A,B) ((A) > (B) ? (A) : (B))
  ```

- `vs.`
  ```
  inline int max(int a, int b) {
      return a > b ? a : b;
  }
  ```

- The compiler replaces each call to the function with the code for the function itself.
  (So, no stack setup, no call/ret.)

- Useful for small, frequently called functions.
Agenda

- Malloc Overview
- Casting & Pointer Review
- Macros & Inline Functions
- Malloc Design
- Debugging & an Action Plan
Maloc Design

- You have a ton of design decisions to make! 😊
- Thinking about fragmentation
- Method of managing free blocks
  - Implicit List
  - Explicit List
  - Segregated Free List
- Policy for finding free blocks
  - First fit
  - Next fit
  - Best fit
- Free-block insertion policy
- Coalescing (or not)
Fragmentation

- Internal fragmentation
  - Result of payload being smaller than block size.
    - Header & footer
    - Padding for alignment
  - Mostly unavoidable.
Fragmentation

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

```c
p1 = malloc(4)
p2 = malloc(5)
p3 = malloc(6)
free(p2)
p4 = malloc(6)
```

*Oops! (what would happen now?)*

- Some policies are better than others at minimizing external fragmentation.
Managing free blocks

- Implicit list
  - Uses block length to find the next block.
  - Connects *all* blocks (free and allocated).
  - All blocks have a 1-word header before the payload that tells you:
    - its size (so you know where to look for the next header) and
    - whether or not it’s allocated
  - You may also want a 1-word footer so that you can crawl the list in both directions to coalesce.
Managing free blocks

- Explicit list
  - A list of *free* blocks, each of which stores a pointer to the next free block.
  - Since only free blocks store this info, the pointers can be stored where the payload would be.
  - This allows you to search the free blocks much more quickly.
  - Requires an insertion policy.
Managing free blocks

- **Segregated free list**
  - Each size class has its own free list.
  - Finding an appropriate block is much faster (so next fit may become good enough); coalescing and reinsertion are harder.

![Diagram of segregated free list](image-url)
Finding free blocks

- **First fit**
  - Start from the beginning.
  - Find the first free block.
  - Linear time.

- **Next fit**
  - Search starting from where previous search finished.
  - Often faster than first fit.

- **Best fit**
  - Choose the free block closest in size to what you need.
  - Better memory utilization (less fragmentation), *but* it’s very slow to traverse the full list.

- **What if no blocks are large enough?**
  - Extend the heap
Insertion policy

- Where should free blocks go?
  - Blocks that have just been \texttt{free()}d.
  - “Leftovers” when allocating \textit{part} of a block.

- LIFO (Last In First Out)
  - Insert the free block at the beginning of the list.
  - Simple and constant time.
  - Studies suggest potentially worse fragmentation.

- Address-Ordered
  - Keep free blocks list sorted in address order.
  - Studies suggest better fragmentation.
  - Slower since you have to find where it belongs.
Coalescing policy

- Use the block size in the header to look left & right.

- **Implicit list:**
  - Write new size in the header of first block & footer of last block.

- **Explicit list:**
  - Must also relink the new block according to your insertion policy.

- **Segregated list:**
  - Must also use the new block size to figure out which bucket to put the new block in.
Agenda

- Malloc Overview
- Casting & Pointer Review
- Macros & Inline Functions
- Malloc Design
- Debugging & an Action Plan
Debugging

- Debugging malloc lab is hard!
  - rubber duck debugging
  - GDB
  - valgrind
  - mm_checkheap
**mm_checkheap**

- **mm_checkheap**
  - A consistency checker to check the correctness of your heap.
  - **Write it early** and update as needed.
  - What to check for? Anything that could go wrong!
    - address alignment
    - consistency of header & footer
    - whether free blocks are coalescing
    - consistency of linked list pointers
    - whether blocks are being placed in the right segregated list
    - ...
  - Focus on correctness, not efficiency.
  - Once you get it working, it should be *silent* and only output when your heap has messed up.
  - You can insert a call to it before & after functions to pin down exactly where things are going wrong.
  - **Do not request debugging help from a TA without a working checkheap.**
Suggested action plan

1. Start early — make the most use of empty office hours.
2. Keep consulting the handout (e.g. the “rules”) throughout your coding process.
3. Understand and implement a basic implicit list design.
4. Write your heap checker.
5. Come up with something faster/more memory efficient.
6. Implement it.
7. Debug it.
8. Git commit and/or submit.
Questions?

GOOD LUCK!!